Speeding Up Model Checking of Timed-Models by Combining Scenario Specialization and Live Component Analysis *

Víctor Braberman[1], Diego Garbervestky[1], Nicolás Kicillof[2], Daniel Monteverde[1,3], and Alfredo Olivero[3,4]

[1] DC, FCEyN, UBA - Argentina. {vbraber,diegog}@dc.uba.ar
[2] Microsoft - USA. nicok@microsoft.com
[3] INTEC, UADE - Argentina. {damonteverde,aolivero}@uade.edu.ar
[4] ECyT, UNSAM - Argentina.

Abstract. The common practice for verifying properties described as event occurrence patterns is to translate them into observer state machines. The resulting observer is then composed with (the components of) the system under analysis in order to verify a reachability property. Live Component Analysis is a "cone of influence" abstraction technique aiming at mitigating state explosion by detecting, at each observer location, which components are actually relevant for model checking purposes. Interestingly enough, the more locations the observer has, the more precise the relevance analysis becomes. This work proposes the formal underpinnings of a method to safely leverage this fact when properties are stated as event patterns (scenarios). That is, we present a sound and complete method of property manipulation based on specializing and complementing scenarios. The application of this method is illustrated on two case studies of distributed real-time system designs, showing dramatic improvements in the verification phase, even in situations where verification of the original scenario was unfeasible.

1 Introduction

The use of observers to express properties for the automatic verification of models of reactive software is common-place (e.g., [2, 1, 12], etc). This is specially the case when requirements are heavily based on event occurrences, since using logical formalisms tends to be cumbersome, if possible at all [12]. In most cases, (automaton) observers are either hand-written or generated from some high-level property specification notation (for example, event patterns). Once defined, observers are composed with the components of the System Under Analysis (SUA), and a reachability property is verified using a model checker.

Live Component Analysis (LCA) is a special case of cone-of-influence abstraction [13] that works by instructing a model checker to ignore the state of some

* Research partially supported by CONICET and the projects UBACyT X021, ANPCyT PICTO-CRUP 31352 and ANPCyT PICT 32440

components at the observer-location level [10, 11]. Supressing irrelevant activity mitigates state space explosion and has a –sometimes dramatically– positive impact on the performance of verification tools in terms of time, size and counterexample length. The fact that the technique works at the observer-location level implies that the more locations the observer has, the more precise the analysis becomes and that precision may in turn positively impact the verification phase.

In this article we describe a human-in-the-loop method to safely obtain detailed versions of the original observers, which may imply dramatic improvements in time and space during verification.

Of course, allowing a Verification Engineer (VE) straightforwardly manipulate the observer to exploit this phenomenon is a risky and awkward business, since validity of verifications might be jeopardized (i.e., will verification results have some formal link with the verification of original properties?).

Actually, our approach consists in enabling a VE to modify a high level representation of the original verification goal. To do this, we propose theoretical and practical tools to soundly manipulate scenarios created in the specification phase, instead of the observer automata obtained by translating them.

We call this manipulation *specialization*. It essentially amounts to adding constraints and/or dividing the goal into cases. To the best of our knowledge, this sort of manipulation, although typical in logical and type-theoretic frameworks, is novel for scenario-based notations. More concretely, the method is based on the observation that the chances for a more effective optimization are greatly improved by formulating guesses about the shape of violating traces. The informal notion of "guessing" is formalized as scenario specialization.

To formalize and validate our method, we choose VTS [12] (a visual scenario notation) as the formalism to express properties. Its minimality and expressive power enable the definition of all the required notions for checking soundness of manipulations. VTS comes in two flavors: **existential scenarios** enabling the expression of pattern of events (essentially, a partial ordered set of events), which usually stand for negative scenarios and, not surprisingly, are checked by converting them into an observer automata; and **conditional scenarios** allowing to express universal properties requiring that, whenever a pattern (*the antecedent*) is matched in a trace, it must be also possible to match at least one of the (*consequent*) patterns. For the sake of simplicity, the method will be illustrated on verification goals given as existential scenarios. Nevertheless, conditional scenarios will also play a key role to articulate, manipulate and check the completeness assumption underlying the specialization of an existential goal. The ability to feature consequents non-trivially intertwined with the antecedent pattern (i.e., referring to events that should happen before, in between, and/or after events of the antecedent) is a key feature for automatically expressing completeness assumptions. According to our case studies, those completeness provisos usually may be verified efficiently against (the model of) the SUA by further applying conservative abstractions on scenarios and the SUA.

Contributions: we have pointed out that adding detail to property observer automata may speed up model checking phase when SUA and observer are preprocessed by LCA. Thus, this paper presents the formal underpinnings and validation for a novel, sound and complete user-intervention approach in the verification of models featuring real-time components. More concretely, we show how to leverage that phenomenon as follows:

- We equip VTS with the notion of *specialization*. The presentation of this concept is compactly based on morphisms.
- We introduce the formal link between specialization and satisfiability of scenarios, allowing their sound deductive manipulation.
- We also show how to check that manipulations are safe by building and checking a conditional scenario. That is, we show that specialization may be efficiently checked to be complete w.r.t. the original verification goal for the SUA.
- We build proof-of-concept tools and illustrate the ideas on case studies.

The article is structured as follows: we start by describing the running case studies and provide a novel and concise presentation of VTS based on morphisms. Then, we introduce the notion of specialization and its related results. In the following section we explain the method, combined with tools and results, and illustrate it with applications to case studies. Finally, we relate this work to other similar efforts, discuss treats to validity and draw conclusions.

2 Motivating Case Study

The following case study illustrates both the use of observers to model safety properties and the role of Live Component Analysis. `MinePump` is a timed automata [4] model of a design of a fault-detection mechanism for a distributed mine-drainage controller [8].

We are interested in the behavior of the subsystem responsible of detecting failures in sensors. A watchdog task (wd) periodically checks the availability of a water-level-sensing device (hlw) by sending a request and extracting acknowledgments that were received and queued during the previous cycle (by another sporadic task, $ackh$). When the watchdog finds the queue empty, it registers a fault condition in a shared memory ($alarmadded$), which is periodically read ($alarmget$) and forwarded ($alarmsent$) by a proxy task ($proxy$) through a net (net) to a remote console ($display$) featuring a latching mechanism ($latch$).

We want to analyze whether the following condition is possible *"the remote operator is informed of a failure of the water sensor too late"*. The automaton in Fig. 1 (a) captures a trace violating the requirement, while Fig. 1 (b) shows a more detailed observer which makes explicit the control flow in the chain of parallel activities. The verification consists in checking whether location 2 (respectively 7) is reachable when composed with the SUA.

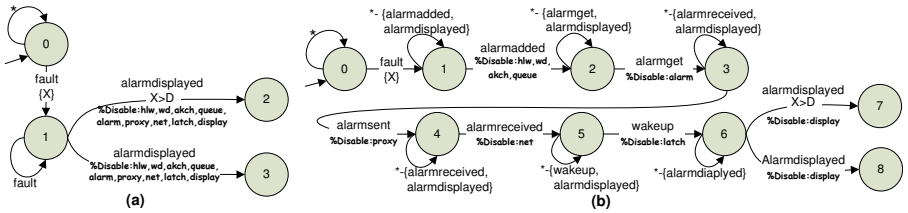


Fig. 1: Observers: (a) Original Version (b) Detailed Version

2.1 Live Component Analysis

Live Component Analysis (LCA) has been conceived to tackle state explosion in a verification setting pursuing the detection of design flaws in a set of concurrent timed-activities [10]. ObsSlice [11] is a tool implementing LCA.

When fed with a network of timed automata consisting of a SUA and an observer, ObsSlice statically discovers *for each observer location* a set of modeling elements (automata and/or clocks) that can be safely ignored (disabled) without compromising the validity of arbitrary TCTL [3] formulas stated over the observer (i.e., an exact reduction method w.r.t. branching-time analysis). The effect of ObsSlice on observers is conveyed by "%Disable" annotations (Fig. 1).

The result of running ObsSlice is a transformed network of timed automata that avoids irrelevant behavior while being equivalent to the input network, up to branching-time observation (i.e., preserves TCTL satisfaction over the composed system, including reachability). For instance, the observer of Fig. 1 (b) composed with the SUA reaches location 7 if and only if the same location is reachable in the corresponding transformed network of timed automata computed by ObsSlice.

For the observer in Fig. 1 (a), ObsSlice localizes the property by declaring globally relevant nine of the twenty timed automata involved in the complete model and deactivates components when the observer enters trap locations. If this localization is not performed verification turns out to be practically unfeasible. On the other hand, on the observer in Fig. 1 (b) the tool discovers that, after the *alarmadded* event is matched, the sensor (hlw), watchdog (wd), acknowledgment handler ($ackh$), and the queue that accumulates those acks ($queue$), are no longer relevant since their behavior would have no impact in future evolution of the observed flow. We can see in the observer the incremental disabling of components that ObsSlice safely performs after each location. We will see, later in this article, that the intuitively better deactivation profile for the more detailed version actually translates into a performance gain in the model checking phase.

2.2 Describing properties using *VTS*

In VTS, labels associated to points stand for *event occurrences*, arrows stand for *precedence* and a negated label over an arrow means *absence*. The high-level counterpart of the observer shown in Fig.1 (a) is the existential VTS scenario

Fig. 2: `MinePump`: (a) Original Property (b) Specialized Version

in Fig.2 (a). Thus, we are talking about a *fault* event for which the following *alarmdisplayed* happens later than D time units. In this case, the relationship between the scenario and the observer obtained seems to be fairly simple, but the gap widens considerably for more complex requirements [12]. On the other hand, the scenario in Fig.2 (b) makes explicit the control flow in the chain of parallel activities.

The article focuses, in general terms, on two basic questions: which relationship holds between Fig.2 (a) and Fig.2 (b)? and, more importantly, how can it be known that verifying (the scenario in) Fig.2 (b) is equivalent to verifying Fig.2 (a).

3 Visual Timed Scenarios

VTS is a scenario-based notation designed to graphically define predicates over complex control or interaction flows. In VTS, there are two kinds of scenarios: existential and conditional ones. In what follows we revisit the definition presented in [12] by compactly reintroducing notions using morphisms, and extending it to include scenario specialization.

3.1 Existential Scenarios

An existential VTS scenario is basically an annotated strict partial order of relevant events, denoting a (possibly infinite) set of matching time-stamped traces. Existential VTS is used to state a simple though relevant family of questions of the form "Is there a potential run that matches this generic scenario?". When interpreted as negative scenarios, as is the case in this work, these questions can express infringements of safety or progress requirements. They turn out to be decidable, by translating a scenario into a Timed Automata (observer) that recognizes matching runs [12]. This automaton is composed with the SUA to check whether a violating execution is reachable by using available model checking tools for Timed Automata like Kronos [7] and Uppaal [6].

The basic elements of the graphical notation are points connected by lines and arrows. Points are labeled with sets of events, where the point stands for an occurrence of one of the events during execution. An arrow between two points indicates that an occurrence of the source point precedes an occurrence of the target. Thus, the occurrences of two points that are neither directly nor indirectly connected, may appear in any order in a trace. A dashed line linking two points specifies that they must represent different event instances, but their relative order is irrelevant. Both arrows and dashed lines may have associated

time restrictions and forbidden events between their ends. VTS can also identify the **beginning** of an execution (depicted with a big full circle).

Definition 1 (Scenario). *A* scenario *is a tuple* $\langle \Sigma, P, \ell, \not\equiv, <, \gamma, \delta \rangle$*, where* Σ *is a finite set of events;* P *is a set of points;* $\ell : P \to 2^{\Sigma}$ *is a function that labels each point with a set of events;* $\not\equiv \subseteq P \times P$ *is an irreflexive relation among points (separation);* $< \subseteq (P \uplus \{\mathbf{0}\} \times P)$ *is a precedence relation between points –asymmetric and irreflexive– (*$\mathbf{0}$ *represents the beginning of execution);* $\gamma : (< \cup \not\equiv) \to 2^{\Sigma}$ *assigns to each pair of related points the set of events forbidden between them; and* $\delta : (< \cup \not\equiv) \to \Phi$ *assigns a restriction (expressed as a logical combinations of interval time constraints*[5]*) for the time elapsed between each pair of related points.*

The formal semantics of VTS is given by assigning to each scenario a set of traces that satisfy it based on the notion of *matching* [12]. A matching is a mapping from event occurrences in a time-stamped trace to the scenario points, such that all constraints imposed by the scenario are satisfied between those event occurrences. For example, the scenario in Fig.2 (a) expresses a predicate that is true for a run if and only if it contains a *fault* followed by the next *alarmdisplayed* in more than D time units. The arrow indicates that *fault* occurs before *alarmdisplayed*. The negated label on it states that no other occurrence of *alarmdisplayed* is found in between. The temporal constraint *"the distance between both events is greater than D"*, is expressed by attaching a "$> D$" to the arrow.

In the scenario of Fig.2 (b) we used an abbreviation for a frequent idiom: certain point represents the **next** occurrence of *alarmadded* after *fault*. The abbreviation is a second (open) arrow and is equivalent to adding *alarmadded* as a forbidden event on the arrow. As a counterpart, VTS also includes an abbreviation for **previous**, when the forbidden event coincides with the source one (see Fig. 4). See [12] for more information on VTS features.

In [12] it is shown how to build a Tableau observer $\mathcal{T}_{\mathcal{S}}$ that recognizes all traces matching a scenario $\mathcal{S}$, so that model checking $\mathcal{A} \vDash \mathcal{S}$ is equivalent to model checking a reachability formula on the parallel composition of $\mathcal{A}$ with $\mathcal{T}_{\mathcal{S}}$.

Definition 2 (Scenario Morphism). *Given two scenarios* $\mathcal{S}_1$*,* $\mathcal{S}_2$ *(assuming a common universe of labels), and* f *a total function between* P_1 *and* P_2 *we say that* f *is a morphism from* $\mathcal{S}_1$ *to* $\mathcal{S}_2$ *(denoted* $f : \mathcal{S}_1 \to \mathcal{S}_2$*) iff for all* $\mathsf{p}, \mathsf{q} \in P_1$*:*

M1: $\ell_2(f(\mathsf{p})) \subseteq \ell_1(\mathsf{p})$

M2: $\gamma_1(\mathsf{p}, \mathsf{q}) \subseteq \gamma_2(f(\mathsf{p}), f(\mathsf{q}))$
$\gamma_1(\mathbf{0}, \mathsf{q}) \subseteq \gamma_2(\mathbf{0}, f(\mathsf{q}))$

M3: *if* $\mathsf{p} <_1 \mathsf{q}$ *then* $f(\mathsf{p}) <_2 f(\mathsf{q})$

M4: *if* $\mathsf{p} \not\equiv_1 \mathsf{q}$ *then* $f(\mathsf{p}) \not\equiv_2 f(\mathsf{q})$

M5: $\delta_2(f(\mathsf{p}), f(\mathsf{q})) \subseteq \delta_1(\mathsf{p}, \mathsf{q})$
$\delta_2(\mathbf{0}, f(\mathsf{q})) \subseteq \delta_1(\mathbf{0}, \mathsf{q})$

3.2 Conditional Scenarios

Conditional scenarios (CSs) [12] are useful to state that whenever a matching is encountered in a given trace for a scenario (the *antecedent*), the same matching

[5] When the scenario is translated into a *TA* for model checking purposes, Φ must be restricted to real intervals with integer bounds.

must be extensible to cover at least one scenario in the set of the so called *consequent scenarios*. This notion is actually the scenario counterpart of the propositional scheme $A \implies (C_1 \vee \ldots \vee C_n)$ where scenario A is interconnected with each scenario C_i.

Although conditional scenarios were meant to enable verification engineers (VE) to express verification goals, in this work conditional scenarios are used as a means to automatically state, prove, or refute the expected formal link between the original goal (existential scenario) and the manipulations done by the VE. Technically speaking, the antecedent is a common substructure of all consequents enabling complex relationship between points in antecedent and consequents.

Syntactically, a conditional scenario is a scenario that plays the role of the antecedent embedded into consequents. A set-theoretic notion of inclusion, called specialization, is introduced in [12]. In this article, we redefine all those notions in terms of morphisms. That is, the embedding of antecedents into consequents is now presented as a set of morphisms.

Actually, due to decidability concerns, we require morphisms to be such that all points in the codomain but not in the image are reachable from a point in the image by following **next** and **previous** arrows [12]. That is, we say that an injective morphism $f : \mathcal{S}_1 \to \mathcal{S}_2$ is matching-kernel iff $R(f(P_1)) = P_2$ where $R(X)$ is an operator that provides every point reachable from $X \subseteq P_2$ by following **next** and **previous** arrows in $\mathcal{S}_2$. We note such morphism as $\twoheadrightarrow$.

Definition 3 (Conditional Scenario). *Given an* antecedent *scenario* $\mathcal{S}_0$ *and an indexed set of* consequent *scenarios* $\mathcal{S}_i$ *with the respective matching-kernel morphisms* $f_i : \mathcal{S}_0 \twoheadrightarrow \mathcal{S}_i$*, we call* $\mathcal{C} = \langle \mathcal{S}_0, \{(\mathcal{S}_i, f_i)\}_{i=1\ldots k} \rangle$ *a Conditional Scenario.*

In the graphical notation, consequent elements (points and arrows) are depicted in gray and numbered in order to distinguish between different consequents. The antecedent image under the morphisms is highlighted in black. Fig. 3 shows a CS expressing that between a fault occurs and an alarm is displayed, the intermediate chain of events shown in the consequent is required to happen.

Fig. 3: CS stating the chain of events that is supposed to happen

Model checking of a CS $\mathcal{C} = \langle \mathcal{S}_0, \{(\mathcal{S}_i, f_i)\}_{i=1\ldots k} \rangle$ is done by building a set of existential scenarios that stand for the possible violations of $\mathcal{C}$. More precisely, there is an algorithm that generates a set of existential scenarios $\mathcal{N}_j$ (called *negative scenarios*) and morphisms $n_j : \mathcal{S}_0 \twoheadrightarrow \mathcal{N}_j$ for $j = 1 \ldots n$ such that: **(a)** $\langle \mathcal{S}_0, \{(\mathcal{S}_i, f_i)\}_{i=1\ldots k} \cup \{(\mathcal{N}_j, n_j)\}_{j=1\ldots n} \rangle$ is a *tautological* CS (i.e., if $\mathcal{S}_0$ is matched then necessarily one of the consequents is also matched), and **(b)** there is no way to extend a matching of a $\mathcal{N}_j$ to match any of the $\mathcal{S}_i$ (i.e., $\mathcal{N}_j$ reveals a violation of $\mathcal{C}$). Then, it is not difficult to see that, if each $\mathcal{N}_j$ were checked

unfeasible in a timed automata $\mathcal{A}$ (for instance, using conservative abstractions), then $\mathcal{A} \vDash \mathcal{C}$ would necessarily hold. Interestingly enough, this model-checking approach explicitly generates the consequents potentially missing in a CS.

4 Scenario Specialization and Optimization Method

In this section we introduce a scenario specialization notion based on morphisms and formally present some of the intuitive definitions from previous sections. Given two existential scenarios $\mathcal{S}_1$ and $\mathcal{S}_2$, we say that $\mathcal{S}_2$ specializes $\mathcal{S}_1$ and, conversely, $\mathcal{S}_1$ generalizes $\mathcal{S}_2$ (denoted $\mathcal{S}_2 <: \mathcal{S}_1$) when there exists a morphism $m : \mathcal{S}_1 \to \mathcal{S}_2$. This relation establishes that $\mathcal{S}_1$ is embedded into $\mathcal{S}_2$ if the latter features more constrains (this is analogous to a logical subsumption).

Interestingly enough, the formal semantics of existential scenarios can then be defined using specialization (a syntactic notion) as follows. A time-stamped sequence of events (trace σ) produced by timed models like timed automata can be regarded as a totally ordered scenario $\mathcal{S}_\sigma$[6]. Thus, under this interpretation, the existence of a morphism (specialization) coincides with the notion of matching in [12] (i.e., $\sigma \vDash \mathcal{S}$ iff $\mathcal{S}_\sigma <: \mathcal{S}$). We say that a timed automata satisfies a scenario ($\mathcal{A} \vDash \mathcal{S}$) iff $\mathcal{A}$ exhibits at least one time-divergent trace that, regarded as a scenario, specializes $\mathcal{S}$.

Note also that if $\mathcal{S}_2 <: \mathcal{S}_1$ and $\mathcal{A} \vDash \mathcal{S}_2$, then $\mathcal{A} \vDash \mathcal{S}_1$ (by composing the witness morphisms). In general it is not true that if $\mathcal{A} \vDash \mathcal{S}_1$, then necessarily $\mathcal{A} \vDash \mathcal{S}_2$ holds. However, as we will see later, we can check whether $\mathcal{A} \vDash \mathcal{S}_1$ implies $\mathcal{A} \vDash \mathcal{S}_2$ by checking exhaustiveness using conditional scenarios.

Additionally, the formal semantics of CS can also be compactly defined in terms of morphisms. A trace σ satisfies a CS $\mathcal{C}$, ($\sigma \vDash \mathcal{C}$) iff for every morphism $m : \mathcal{S}_0 \to \mathcal{S}_\sigma$ there exists $m_i : \mathcal{S}_i \to \mathcal{S}_\sigma$, for some $i \in \{1..k\}$, such that $m = m_i \circ f_i$.

In the next section, we will introduce the optimization method using the previously presented notions. However, is worth mentioning that [9] shows how specialization can be extended to conditional scenarios. Specialization works contravariantly on the antecedent (i.e, weakening it) and covariantly on consequents (i.e., strengthening them). We believe that those results generalize the following optimization method, which will be explored as future work.

4.1 Optimization Method

Method 1 is a sketch of the procedure to safely verify properties using scenario specialization. It starts with a verification goal given as an existential scenario and a SUA. Now we assume that the original verification goal consumes excessive time and space resources, so the method can be used to mitigate state space explosion. In what follows, $\texttt{Verify}(SUA, \mathcal{S})$ denotes modelchecking a reachability property on the result of applying LCA to SUA and the tableau of $\mathcal{S}$ (i.e., $\mathcal{T}_\mathcal{S}$). VE

[6] $\mathcal{S}_\sigma$ has a single-labeled point per event occurrence in the trace, plus the end point, and explicitly features separation and time distance constraints between every pair of points as well as forbidding all events that do not actually occur in between.

Algorithm 1 Specialization based method

Input: SUA: TA network; $\mathcal{G}$: VTS existential scenario
Output: $\mathcal{G}$_Matchable?: boolean

1: Provide $\mathcal{G}'_1, \ldots, \mathcal{G}'_n$ such that $\mathcal{G}'_i <: \mathcal{G}$ $(m_i : \mathcal{G} \to \mathcal{G}'_i)$
2: **for** each $\mathcal{G}'_i$ **do**
3: $R_i :=$ `Verify`$(SUA, \mathcal{G}'_i)$ // better precision and better computation time
4: **end for**
5: **if** $\bigvee(R_i = \text{true})$ **then**
6: $\mathcal{G}$_Matchable? := true
7: **else**
8: $\mathcal{P} := \langle \mathcal{G}, \{(\mathcal{G}'_i, m_i)\}_{i=1\ldots n} \rangle$
9: $\mathcal{N} = \{\mathcal{N}_i\}_{i=1\ldots k} := GenNegative(\mathcal{P})$
10: **for** each $\mathcal{N}_i \in \mathcal{N}$ **do**
11: $R'_i :=$ `Verify`$(SUA', \mathcal{N}'_i)$ // $\mathcal{N}_i <: \mathcal{N}'_i$ and SUA' is an abstraction of SUA
12: **end for**
13: **if** $\bigwedge(R'_i = \text{false})$ **then**
14: $\mathcal{G}$_Matchable? := false
15: **else**
16: **for** each $\mathcal{N}'_i$ such that $R'_i = \text{true}$ **do**
17: $R''_i :=$ `Verify`$(SUA, \mathcal{N}'_i)$
18: **end for**
19: $\mathcal{G}$_Matchable? := $\bigvee(R''_i = \text{true})$
20: **end if**
21: **end if**
22: **return** $\mathcal{G}$_Matchable?

provides a set of specializations (line 1). When LCA is applied to them, precision is improved and verification is hopefully sped up. Note that specializations can be checked in parallel. Then there are two possible courses of actions depending on the result. If a match is found (line 5) then, since specialization matching implies the original scenario is matchable also (line 6), we are done. Else, it would be necessary to check exhaustiveness of specialization to safely conclude the original goal is non-matchable. That is automatically expressed as satisfiability of a CS $\mathcal{P}$ made up of the original goal as antecedent and the specializations as consequents (line 8). Fig. 3 and Fig. 5a show this construction for the `MinePump` and `RemoteSensing` examples, respectively. Note that, if the conditional scenario were true, we would know that the original scenario is actually non-matchable since the conditional scenario tells us that, whenever we can find a match for the original scenario, one of the specializations is bound to happen (and we have already discarded that in previous steps).

The set of negative scenarios needed to model check a CS can be automatically derived from it (line 9) corresponding to the exhaustiveness claim. They are also methodologically valuable intermediate artifacts, since they explicitly reveal and document the engineer's belief on either scenarios that cannot happen, or scenarios that do not exercise worst cases for the original verification goal.

When checking the infeasibility of negative scenarios, it is important to note that the SUA may be abstracted and a negative scenario may be generalized (line 11). The reason is simple: if generalized versions of negative scenarios are proven to be non-matchable over an abstract version of the SUA, we can safely conclude that the conditional scenario has no way to be violated and thus

the exhaustiveness proviso holds. In this context, typical abstractions include getting rid of temporal constraints on negative scenarios and applying convex-hull approximation when model checking against SUA (see case studies). These strategies have proven very effective in practice. In the worst case, if a negative scenario cannot be discarded (i.e, conservative abstractions and/or generalizations show it may be potentially matched), it is actually checked as it is (line 17) to determine whether matchings were spurious (due to aggressive abstractions) or it is actually a case where the original goal is matched.

Notice that it may not be necessary to examine every specialization (line 2) in the loop. As soon as one produces $R_i = \text{true}$, the method terminates returning $\mathcal{G}$_Matchable? = true. Something similar happens with the loop in line 16.

5 Case Studies analysis and verification results

Here we apply our method to two case studies: `MinePump` and `RemoteSensing` that will be explained afterwards. In Tables 1 and 2 we compare the time and memory consumed by model checking tools to verify an original and a specialized scenario, for the normal and the sliced versions of the SUA. We use UPPAAL 4.0.2 as a model checker running on a Pentium IV 3Ghz, 2GB RAM, LINUX 2.6.3, and OBSSLICE as the LCA. UPPAAL was run using different search orders (BFS, DFS, RND (i.e., random DFS)), state space reductions (none, conservative, aggressive) and state space representations (DBMs, compact data structure, and convex hull abstraction). We decided to show experiments for each example using the most effective parameter setting for the model checker (BFS for `MinePump`; DFS and Random DFS for `RemoteSensing`). When using BFS, we also provide the comparison in terms of the shortest traces and the memory consumed to generate them (line Shortest Trace in both tables). OBSSLICE execution times are not reported, being negligible w.r.t. verification times (less than five seconds). Moreover the results of the LCA technique can be reused provided VE only alter timing constraints in subsequent verifications.

In `MinePump` (described in Sec. 2), we focus our analysis on the scenario in which the alarm may go off after its deadline D (Fig. 2(a)). As Table 1 shows slicing according to the original scenario is of little help in the verification phase, if we eliminate by hand timed components that interact neither with the property nor with the subsystem under analysis. The specialization proposed in Fig. 2(b) provides more detail, stating the existence of a sequence of events between $fault$ and *alarmdisplayed* that includes *alarmadded*, *alarmget*, *alarmset*, *alarmreceived* and *wakeup*. This helps OBSSLICE do a more precise job deactivating components; which is significantly reflected in time, space and size of counterexamples during the model checking phase. Note that in case D=3202 the specialized version is not matchable. Thus, Fig. 3 expresses specialization exhaustiveness as a CS (line 8 of method) required to be checked in order to ensure that the result is sound. Exhaustiveness of the specialization can be verified using the LCA on generalizations of each negative scenario obtained by eliminating timing constraints (line 11). Actually, five negative scenarios were

Default verif. settings DBM + BFS			Original Scenario Fig. 2 (a)				Specialized Scenario Fig. 2 (b)			
			Normal		Sliced		Normal		Sliced	
+settings	D	match?	time	mem.	time	mem.	time	mem.	time	mem.
–	3201	yes	345.45	93.3	342.63	91.6	342.42	88.1	34.96	27.4
–	3202	no	1293.63	167.1	1271.83	163.3	1225.03	154.1	887.09	120.8
Shortest	3201	yes	677.33	208.7	686.85	207.6	727.67	208.0	52.98	49.0
Trace	length / mem		321 st / 91 Kb		322 st / 91 Kb		322 st / 94 Kb		175 st / 50 Kb	

Table 1: `MinePump` Verification time in seconds, and memory in Mbytes

Default verif. settings: DBM		Original Scenario Fig. 4				Specialized Scenario Fig. 6			
		Normal		Sliced		Normal		Sliced	
+ settings	match?	time	mem.	time	mem.	time	mem.	time	mem.
DFS	yes	–	O/M	1892.03	187.8	–	O/M	55.58	28.8
RND best	yes	–	O/M	178.26	19.5	–	O/M	6.53	8.2
RND worst	yes	–	O/M	4476.29	110.2	–	O/M	71.81	43.1
RND aver.	yes	–	O/M	1658.90	82.5	–	O/M	35.16	19.6
Shortest	yes	–	O/M	O/T	–	–	O/M	21401.00	579.7
Trace (CDS)	length / mem	–	–	–	–	–	–	157steps. / 84Kb	

Table 2: `RemoteSensing` Verification time in seconds (O/T: >11hs), and memory in MBytes (O/M: Out of Memory)

generated, not shown in the article due to space limitations. The verification of exhaustiveness can be done using convex hull over-approximation and takes about 10 secs.

The other case study, called `RemoteSensing`, is a system consisting of a central node and two remote sensors [12]. Sensors regularly sample a pair of environmental variables $V1$ and $V2$, and store their values in shared memory. Signals are periodically sent to the sensors by threads running in the central node. Each sensor runs a thread dedicated to handle these messages by reading the last stored value from shared memory and sending it back to the central node. The latter pairs up the readings of each end-to-end task (i.e., the pipeline of activities that starts with a request of the corresponding variable and ends sending the sampled value to the pairing component for processing). A freshness requirement violation for a single data flow is informally stated as *"value of variable $V1$ used by the pairing component has been sampled in a too distant time instant"*.

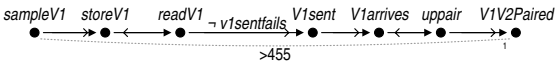
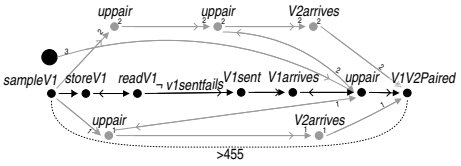


Fig. 4: Existential VTS scenario: Freshness Violation

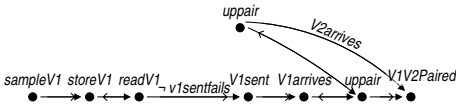
Fig. 4 shows an existential scenario that matches whenever two signals conveying values for $V1$ and $V2$ are paired and the time distance between the

sampling of $V1$ and the pairing instant is greater than 455 time units. Note the use of the **previous** notation in order to express that there is no other $V1arrives$-event after a particular $V1arrives$ and before an *uppair*-event.

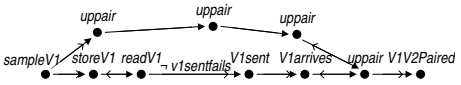
This example shows that specialization can be also used as a tool for a "proof by case" approach to model checking. In `RemoteSensing`, the VE provides additional information indicating all different ways an *uppair* event can occur before the occurrence of *uppair* included in Fig. 4 and the next occurrence of $V2arrives$ relative to the same scenario. Fig. 5a shows a CS that expresses exhaustiveness of specialization provided for this example.



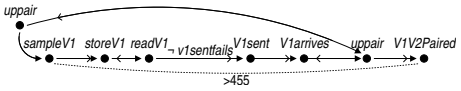
(a) CS stating the expected 3 possible alternatives for previous *uppair* event and its next $V2arrives$ event



(b) Neg. scenario: $V2arrives$ does not occur



(c) Neg. scenario: there are 3 *uppairs* during a candidate end-to-end job



(d) Neg. scenario: previous *uppair* occurs before the sampling

Fig. 5: CS for the specialization and its respective negative scenarios

In the original verification goal of `RemoteSensing`, ObsSlice (see Table 2) is key to make verification feasible. It detects: (a) the irrelevance of sampler 2, (b) the irrelevance of sampler 1's behavior as soon as its values are read ($ReadV1$); and (c) that the rest of the modeling components are no longer relevant once the pairing component is triggered (i.e. the *uppair* event).

Two key observations provide the rationale for the specializations presented: i) the signal rate in data flow for $V1$ is slower than that for $V2$, ii) once the first signal carrying $V2$ (after the previous *uppair*) is registered by the pairing latch unit, the activity of that pipeline is no longer relevant and the occurrence of the pairing event only depends upon tracked behavior for the matched job carrying $V1$.

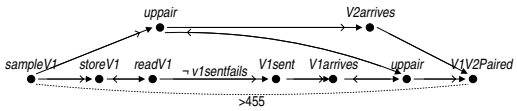


Fig. 6: Remote Sensing: the specialization used in verification

It turns out that it is worth making explicit this scenario fragment of expected behavior (see Fig. 6) in order to help ObsSlice detect the predicted irrelevance of activities of data flow for $V2$ after $V2arrives$, thus dramatically improving model checker performance w.r.t. bug finding on the model sliced in terms of the original scenario (line 3 of method). Of course, this guess can be formally stated, and efficiently and soundly analyzed[7] as the CS in Fig. 5a. Figures 5b, 5c and 5d, show negative scenarios that were discarded as infeasible in a few secs. using overapproximation by convex hull (line 11). They are actually generalizations obtained from negative scenarios generated by the tool (line 9) by eliminating the timing constraint.

6 Related Work

The closest related work on automatic syntactic preprocessing of timed models are the clock-reduction technique for TAs presented in [14], and the static-guard analysis to find location-dependent maximum constants based on relevant guards [5]. Similarly to ObsSlice here, these techniques examine timed-components at a syntactic level to derive reductions that preserve the branching-time structure. In [14], like in ObsSlice, there is also a limited use of timing information (clocks are variables) to keep the preprocessing as light as possible. However, ObsSlice includes an "activity calculus" that can be applied to a SUA given as a parallel composition (i.e., not-already composed). Contrary to both techniques, which are indeed orthogonal to that of ObsSlice, our optimization essentially implies the deactivation of irrelevant components (not just clocks) during the on-the-fly verification step. Several approaches leverage the existence of partial information on expected/relevant behavior either provided by humans or mined by tools (e.g., test purposes [15, 18], etc.).

Several formally-stated interaction-based notations can be found in the literature suitable for describing systems using triggered scenarios or for specifying temporal properties (e.g., [16, 19], etc.).

We chose VTS as our base notation since some of its features make it suitable for defining and implementing conditional scenario specialization. More precisely: (a) VTS has a simple formal syntax and descriptive semantics based on partial orders and morphisms, (b) VTS verification algorithms are based on Timed Automata, for which tool support is readily available, and (c) VTS CSs can be automatically completed to a tautological form (a key feature for the exhaustiveness check). Further discussion of the related literature for scenario notations can be found in [12], while related work on automatic syntactic preprocessing of timed models in [10, 11].

We are not aware of any scenario-based notation equipped with a set of deductive rules. In [17] a fragment of LSCs is shown to translatable into temporal logics which would enable some sort of deductive manipulation. Unlike their

[7] Note that exhaustiveness check is not actually required in this particular example because Fig. 6 is actually matchable.

approach, our manipulations may be performed by the VE at the syntactic level not at a semantic layer like temporal logics or automata.

A syntactic notion of refinement for Timed MSCs is presented in [20]. Since the authors pursue a development process based on refining charts, these notions distinguish whether the timing constraint is an assumption about the environment or a constraint on process performance. Our specialization concept is not meant to change the level of abstraction as refinements do.

To our knowledge, the combination of live component analysis with scenarios, and the application of the latter to guide and improve the former constitutes an innovative approach. In some hard-to-check models, *VEs* would probably give up the pretension to provide exhaustive sets of specializations that are believed to exhibit the worst-case scenario for the verification goal. Thus, it is reasonable to compare our techniques with scenario-control ones (e.g., [15]). Firstly, our combination of specialization with LCA is not just focused on constraining model exploration by means of parallel composition, but it was devised as a way to conservatively optimize model checking. Secondly, events added in a specialization are actually merged in a single scenario with the property to be checked. Finally, our method generates a set of scenarios that, together with the ones checked, provide complete coverage of the original goal. Even assuming that exhaustive model checking is out of discussion, we believe that explicit description of discarded scenarios is a valuable tool.

7 Conclusions and Ongoing Work

We introduce an human-driven, exact optimization technique based on manipulation and checking of scenarios that dramatically reduces back-end model-checking effort by improving the positive effects of LCA.

Driven by the need to provide a neat framework for the technique, we define the first steps towards equipping the scenario notation VTS (a visual formalism to express, manipulate and model check event-based requirements for real-time systems) with a rigorous theory of syntactic relationships (specializations) between scenarios. One of the main steps in the method consists in specializing the original scenario into one or more cases, to achieve higher static detection of irrelevant activity in the model. VTS equipped with the notion of specialization, as defined in this paper, is well suited for this combination: case specialization is a neatly defined concept, and its exhaustiveness of coverage (i.e., are these all possible cases?) can be expressed as a conditional scenario. Thus, our method provides warnings whenever a set of specializations is potentially pruning state space and thus hiding bugs. Moreover, it identifies the scenarios not being considered which can be in turn checked for infeasibility. In theory, larger observers might contribute to larger state spaces, thus a more expensive verification process. However, this is not bound to happen and we plan to characterize those cases. In addition, results of this paper show that, combining specialization with LCA may actually provide a valuable approach specially when the VE has a good hint of candidate counterexamples.

References

1. L. Aceto, A. Burgueño, and K. G. Larsen. Model checking via reachability testing for timed automata. In *Proceedings of 4th TACAS '98*, pages 263–280, 1998.
2. B. Alpern and F. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. Programming Lang. and Systems*, 11(1):147–167, 1989.
3. R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
4. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
5. G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static guard analysis in timed automata verification. In *Proc. of 9th TACAS '03*, pages 254–270, 2003.
6. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Proceedings of the International Conference on Hybrid Systems*, pages 232–243. Springer Verlag, 1995.
7. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proceedings of the 10th International Conference CAV '98*, volume 1427 of *LNCS*, pages 546–550. Springer Verlag, 1998.
8. V. Braberman. *Modeling and Checking Real-Time Systems Designs*. PhD thesis, FCEyN. Univ. de Buenos Aires, 2000.
9. V. Braberman, D. Garbervetsky, N. Kicillof, D. Monteverde, and A. Olivero. Specializing scenarios. Technical report, DC. FCEN. UBA. http://www.dc.uba.ar/people/exclusivos/vbraber, 2008.
10. V. Braberman, D. Garbervetsky, and A. Olivero. Improving the verification of timed systems using influence information. In *Proceedings of the 8th International Conference TACAS '02*, volume 2280 of *LNCS*, pages 21–36. Springer Verlag, 2002.
11. V. Braberman, D. Garbervetsky, and A. Olivero. ObsSlice: A timed automata slicer based on observers. In *Proc. of Int. Conf. 16th CAV '04*, 2004.
12. V. Braberman, N. Kicillof, and A. Olivero. A scenario-matching approach to the description and model checking of real-time properties. *IEEE Transactions on Software Engineering*, 31(12):1028–1041, 2005.
13. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.
14. C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of IEEE RTSS '96*. IEEE Computer Soc. Press, 1996.
15. W. Grieskamp, N. Kicillof, and N. Tillmann. Action machines: A framework for encoding and composing partial behaviors. *Int. Jour. SEKE*, 16(5):705–726, 2006.
16. D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *Proceedings of the 10th IEEE/ACM International Symposium MASCOTS '02*, pages 193–202. IEEE Computer Society, 2002.
17. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *Proceedings of 11th TACAS'05*, volume 3440 of *LNCS*. Springer-Verlag, 2005.
18. Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes: Adapting the notion of specification to testing. In *Proceedings of the 16th IEEE Int. Conf. ASE '01*. IEEE Computer Society, 2001.
19. B. Sengupta and R. Cleaveland. Triggered message sequence charts. In *SIGSOFT FSE*, pages 167–176, 2002.
20. T. Zheng, F. Khendek, and B. Parreaux. Refining timed mscs. In *SDL Forum*, volume 2708 of *LNCS*, pages 234–250, 2003.