Learning from Vacuously Satisfiable Scenario-Based Specifications*

Dalal Alrajeh[1], Jeff Kramer[1], Alessandra Russo[1], and Sebastian Uchitel[1,2]

[1] Department of Computing, Imperial College London, UK
[2] Departamento de Computaciòn, FCEyN, UBA

Abstract. Scenarios and use cases are popular means for supporting requirements elicitation and elaboration. They provide examples of how the system-to-be and its environment can interact. However, such descriptions, when large, are cumbersome to reason about, particularly when they include conditional features such as scenario triggers and use case preconditions. One problem is that they are susceptible to being satisfied vacuously: a system that does not exhibit a scenario's trigger or a use case's precondition, need not provide the behaviour described by the scenario or use case. Vacuously satisfiable scenarios often indicate that the specification is partial and provide an opportunity for further elicitation. They may also indicate conflicting boundary conditions. In this paper we propose a systematic, semi-automated approach for detecting vacuously satisfiable scenarios (using model checking) and computing the scenarios needed to avoid vacuity (using machine learning).

1 Introduction

Scenarios, use cases and story boards are popular means for supporting requirements engineering activities. They illustrate examples of how the software-to-be and its environment should and should not interact. They are commonly used as an intuitive, semi-formal language for describing behaviour at a functional level.

A common form for providing examples of behaviour is through conditional statements. Use cases [1] support existential conditional statements such as "once an appropriate user ID and passwords has been obtained, a homeowner *can* access the surveillance cameras placed throughout the house from any remote location via the internet" [21]. Live Sequence Charts [14] support universal conditional statements such as "the controller should probe the thermometer for a temperature value every 100 milliseconds, and if the result is more that 60 degrees, it *should* deactivate the heater and send a warning to the console". Some languages support both existential and universal conditional scenarios [24].

Conditional scenarios with different modalities are useful. They provide support for "what-if" elaboration of requirements specifications [1], and the progressive shift from existential statements, in the form of examples and use-cases, to universal statements in the form of declarative properties. Each conditional scenario

* We acknowledge financial support for this work from ERC project PBM - FIMBSE (No. 204853).

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 377–393, 2012.

constitutes only a partial description of the system's intended behaviour. Hence, typically many of them are used in conjunction along with other behaviour descriptions such as system goals [10]. The emergent behaviour of such rich descriptions can be complex to reason about, hindering validation, and resulting frequently in specifications that are incomplete or contradictory.

One particular issue that conditional scenarios have is that they are liable to being satisfied vacuously; a system can be constructed so that it satisfies the conditional scenarios by never satisfying the condition. For instance, a system in which the homeowner is never given a user password vacuously satisfies the use case described above. This problem, commonly referred to as *antecedent failure* [8] in temporal specifications, is often an indication that the specification is partial and hence provides an opportunity for elicitation; it is clear that the stakeholder's intention is that "the system should provide the user with an id and password", and if it does, then the user can access the installed surveillance cameras. In addition, vacuously satisfiable specifications can have pernicious effects, concealing conflicting behaviour which is important to explore. For example, consider two scenarios extracted from the mine pump example in [16]: "once the methane sensors detect that the methane level is critical, *then* the pump controller must send a signal to the pump to be switched off" and "once the water sensors detect that the water level is above the high-threshold, *then* the pump controller must send a signal to the pump to be switched on". These scenarios are consistent as a system in which water sensors never detect high water and methane levels vacuously satisfies both scenarios. However, if these two levels were to occur, then the scenarios provide contradictory information of what the controller must do.

In this paper we describe an approach that not only detects vacuously satisfiable conditional scenarios but also provides automated support for learning new scenarios that ensure the conditions, i.e. triggers, are satisfied. More specifically, the approach takes as input a set of scenarios formalised as triggered existential and universal scenarios [24] and consists of two main phases. The first involves (*i*) synthesising a Modal Transition System from the scenarios, representing all possible implementations that satisfy them and (*ii*) performing a vacuity check, using a model checker, against a scenario's trigger. If the vacuity check is positive, the model checker produces examples of how the system-to-be could satisfy the trigger, i.e. non-vacuity witnesses [13]. In the second phase, (*iii*) an engineer classifies the examples as either positive or negative, i.e. ones that should be accepted or not in the final implementation, and then (*iv*), together with the given scenarios, inputs them into an inductive logic programming learning tool to compute new triggered scenarios which, if added to the existing scenarios, guarantee that they are no longer vacuously satisfiable. This process is repeated for each given triggered scenario, producing in the end a scenario-based specification that is not vacuously satisfiable. Figure 1 outlines the proposed framework.

Although the integrated use of model checking and ILP has been previously applied to other software engineering tasks, such as goal operationalisation [2] and zeno behaviour elimination [3], the current application introduces a

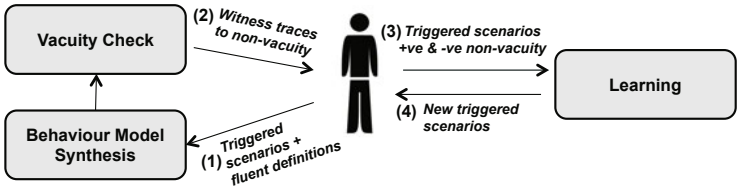

Fig. 1. Overview of the proposed framework

number of new technical challenges not present previously: the need to model and reason about partial behaviour, branching time and alphabet scoping of learned expressions. We elaborate further on these issues in Section 6.

The rest of this paper is organised as follows. We describe a motivating example in Section 2 and the necessary background in Section 3. Section 4 presents the main approach. Section 5 illustrates the results obtained by applying the approach to two case studies. We discuss related work in Section 6 and conclude in Section 7.

2 Motivating Example

Consider a simplified version of the mobile phone system described in [17]. The system is composed of six participants: a user, cover, display screen, speaker, chip and the environment. A phone user can open and close the phone cover, switch the phone on and off, answer and end calls and talk. The chip can detect incoming calls from the environment and the cover opening and closing. It can also initialise the phone settings and send requests to display the caller ID on the screen and to the speaker to start and stop ringing.

Suppose the engineer elicits the two scenarios shown in Figure 2 using a universal and existential triggered scenario notation, respectively. The universal scenario *Receive* informally states that "once an incoming call is detected (*incomeCall*), the phone rings (*startRing*) and the caller id is displayed on the screen (*displayCaller* and *setDisplay*) subsequently". The existential scenario *Phone* specifies the requirement "once an incoming call is detected (*incomeCall*) and the user opens the cover (*open* followed by *coverOpened*), the user may talk (*talk*)". Both scenarios are composed of two parts; a trigger (shown in a hexagon) and a triggered sequence (shown in a box; solid in universal and dashed in existential).

One problem with the specified scenarios is that although they describe what the system must or can do when the system exhibits the triggers, they do not state what it is required to do otherwise. For instance, they do not say when the system can exhibit an incoming call nor what the system can do between the occurrence of an incoming call and the user opening the phone cover. Because this specification is only partial, any implementation of the system in which an

incoming call is never allowed to occur is a valid implementation of the *Receive* scenario (See Figure 2.a). We refer to triggered scenarios which may result in a system that never exhibits the trigger as *vacuously satisfiable scenarios*.

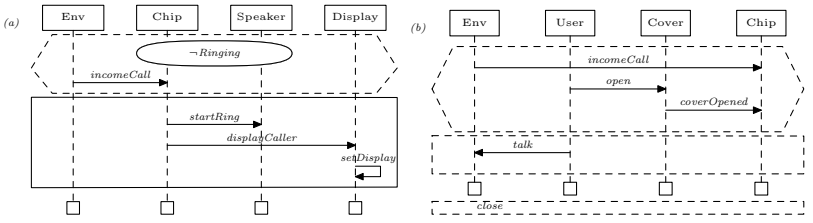

Fig. 2. Mobile phone system scenarios for (a) *Receive* and (b) *Phone*

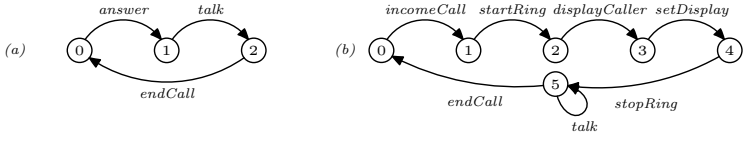

Fig. 3. Implementation that (a) vacuously satisfies *Receive* and *Phone* scenarios, and (b) satisfies the *Receive* scenario non-vacuously but *Phone* vacuously

Feedback about vacuously satisfiable scenarios may help engineers in recognising further behaviour which should be required or proscribed by any derived implementation. By informing an engineer about possible implementations in which an incoming call never occurs, the engineer could provide further examples of what the system behaviour may, must or cannot include. For instance, an engineer could provide a trace showing that incoming calls occurs after the phone is switched on and initialised, i.e. *switchOn*, *initialise*, *incomeCall*, or a negative trace where an incoming call occurs after the phone starts ringing, i.e. *startRing*, *incomeCall*. From such traces, it can be inferred that an incoming call may be triggered when the phone is initialised, or not ringing as shown in Figure 4.

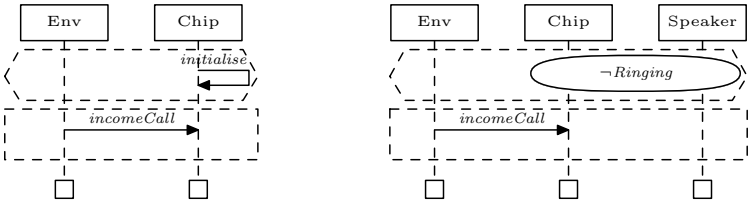

Fig. 4. New triggered scenarios to avoid vacuously satisfying the *Receive* scenario

In this paper, we show how model checking and inductive logic programming provide automated support for detecting vacuously satisfiable scenarios and the computing new scenarios that avoids vacuity, such as those shown in Figure 4.

3 Background

3.1 Triggered Scenarios

Triggered scenarios are sequence charts that represent interactions between the system's agents. Graphically, a triggered scenario comprises several vertical lines labelled by names representing agents' lifeline. Time is assumed to flow downward. Annotated arrows between these lines correspond to synchronous messages which represent instantaneous events on which both objects synchronise.

A triggered scenario consists of three parts; a trigger that is surrounded by a dashed hexagon, a main chart that is surrounded by a rectangular frame and a scope. The trigger is the condition that activates the main chart. It can include event messages as well as properties (depicted in rounded boxes). A property may be associated with one or more agent instances. It is a boolean combination of propositional atoms and their negations, expressed in Fluents Linear Temporal Logic (discussed later), that are expected to be true or false at that point in the system. A main chart can only contain messages. Event messages and properties are associated with ordered locations along the agents' lifelines. A universal Triggered Scenario (uTS) forces the occurrence of the main chart (depicted in a solid rectangular frame) after every occurrence of the trigger. An existential Triggered Scenario (eTS) asserts that it is possible to perform the main chart after every occurrence of the trigger but not necessarily, i.e. alternative behaviour after the trigger is allowed. The purpose of the scope is to restrict the occurrence of certain messages. Events appearing in a triggered scenario are by default within its scope. Further events can be included in the scope by adding them to the *restricts* set depicted in a dotted frame below the scenario's main chart. We refer to events in the scope as *observed* events. Any non-observed event can occur interleaved without restriction.

Triggered scenarios are interpreted over execution trees. An uTS (resp. eTS) is satisfied in an execution tree if at any node of the tree where the trigger is satisfied, *every* (resp. *at least one*) outgoing branch satisfies the main chart.

3.2 Fluent Linear Temporal Logic

Fluent Linear Temporal Logic (FLTL) is a linear temporal logic of fluents [12]. A fluent is a propositional atom defined by a set I_f of initiating events, a set T_f of terminating events and an initial truth value either *true* ($\mathbf{tt}$) or *false* ($\mathbf{ff}$). Given a set of event labels *Act*, we write $f = \langle I_f, T_f, Init \rangle$ as a shorthand for a fluent definition, where $I_f \subseteq Act$, $T_f \subseteq Act$, $I_f \cap T_f = \emptyset$ and $Init \in \{\mathbf{tt}, \mathbf{ff}\}$. We use $\dot{a}$ as a shorthand for a fluent defined as $\langle a, Act \backslash \{a\}, \mathbf{ff} \rangle$.

Returning to our running example, the fluents *Opened*, *Ringing* and *Calling*, meaning the cover is open, the phone is ringing and there is an incoming call, can be respectively defined in FLTL as follows.

```
Opened =<coverOpened, coverClosed, ff>
Ringing =<startRing, stopRing, ff>
Calling =<incomeCall, endCall, ff>
```

Given a set of fluents F, FLTL formulae are constructed using standard boolean connectives and temporal operators X (next), U (strong until), F (eventually) and G (always). The satisfaction of FLTL formulae is defined with respect to traces, i.e. sequences of events over a given alphabet *Act*. Given a trace $\sigma = a_1, a_2, ...$ over *Act* and fluent definitions D, a fluent is said to be true in σ at position i with respect to D if and only if,

- f is defined initially true and $\forall j \in \mathcal{N}.\ ((0 < j \leq i) \rightarrow a_j \notin T_f)$;
- $(\exists j \in \mathcal{N}.\ (j \leq i) \wedge (a_j \in I_f)) \wedge (\forall k \in \mathcal{N}.((j < k \leq i) \rightarrow a_k \notin T_f))$.

In other words, a fluent f holds if and only if it is initially true or an initiating event for f has occurred and no terminating event has occurred since.

3.3 Modal Transition Systems

A Modal Transition System (MTS) is used to formalise a partial model of the system's behaviour [19]. It extends Labelled Transition Systems (LTSs), a widely used formalism for describing and reasoning about system behaviour, by distinguishing between transitions that are required, proscribed and unknown, i.e. transitions for which it is not possible, based on current available knowledge, to guarantee that they will be admissible or prohibited.

Definition 1 (MTS and LTS). *A Modal Transition System is a tuple* $M = (Q, Act, \Delta^r, \Delta^p, q_0)$ *where* Q *is a finite set of states, Act is a set of event labels, called the alphabet,* $\Delta^r \subseteq Q \times Act \times Q$ *is a required transition relation and* $\Delta^p \subseteq Q \times Act \times Q$ *is a possible transition relation where* $\Delta^r \subseteq \Delta^p$ *and* q_0 *is the initial state. A transition that is possible but not required is called a maybe transition. An MTS where all possible transitions are required is called a Labelled Transition System, written* (Q, Act, Δ, q_0).

An MTS M is said to have a required transition on a, denoted $q \xrightarrow{a}_r q'$, if $(q, a, q') \in \Delta^r$. Similarly, M is said to have a maybe transition on a, denoted $q \xrightarrow{a}_m q'$, if $(q, a, q') \in \Delta^p - \Delta^r$. Figure 5 shows an example MTS for the mobile phone system, with the alphabet *Act*= {*open*, *close*, *incomeCall*, *coverOpened*, *coverClosed*, *setDisplay*, *displayCaller*, *startRing*, *answer*, *talk*}, where maybe transitions are denoted with a question mark following the label. Figure 3 shows two LTSs for the same system. Note that the numbered nodes are used for reference and do not designate a particular state.

A trace $\sigma = a_1, a_2, ...$, where $a_i \in Act$, is said to be required in an MTS M if there exists in M a sequence of states such that $q_0 \xrightarrow{a_1}_r q_1 \xrightarrow{a_2}_r q_2 ....$ It is said to be possible if there exists in M a sequence of states such that $q_0 \xrightarrow{a_1}_p q_1 \xrightarrow{a_2}_p q_2 ...$, with at least one transition relation that is in $\Delta^p - \Delta^r$.

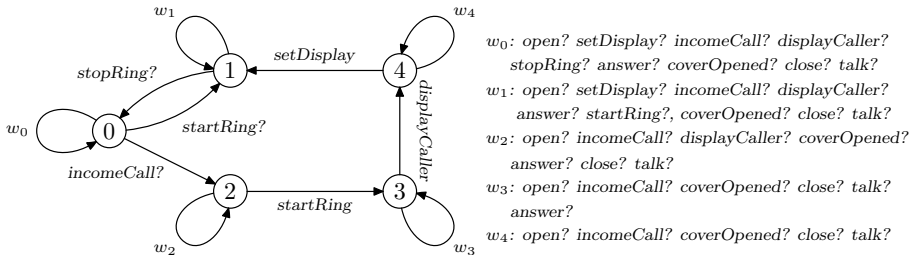

Fig. 5. An MTS synthesised from *Receive* scenario

Given two MTSs N and M, N is said to refine M if N preserves all of the required and proscribed transitions of M [19]. An LTS that refines an MTS M, i.e. an implementation, is a complete description of the system up to the alphabet of M. For example, the LTS shown in Figure 3.b is an implementation of the MTS given in Figure 5. *Merging* MTSs is the process of combining what is known from each MTS. In other words, it is the construction of a new MTS that includes all the required behaviour from each MTS but none of the prohibited ones. An MTS can be synthesised automatically from a safety property ϕ expressed in FLTL [25] and triggered scenarios *TS* [24] that characterises all implementations satisfying ϕ under a 3-valued interpretation on FLTL and *TS*, respectively.

4 Approach

As illustrated in Figure 1, the approach comprises two main phases. The first takes as input a set of fluent definitions and universal and existential triggered scenarios and uses model checking to verify if any of the existing triggered scenarios are vacuously satisfiable by some system implementations. If this is the case, the model checker provides non-vacuity witnesses. In the second phase, after an engineer classifies the non-vacuity witnesses into positive and negative examples, these are used to compute new triggered scenarios that ensure that the existing scenario is satisfied non-vacuously.

4.1 Checking Vacuity of Triggered Scenarios

We first define the term *vacuously satisfiable triggered scenario*, and then discuss the MTS construction, vacuity checks and witness generation.

Definition 2 (Vacuously Satisfiable Triggered Scenario). *Let S be a triggered scenario with trigger P, main chart C and scope Θ. Let M be an MTS that characterises all LTSs that satisfy S. The scenario S is said to be vacuously satisfiable in M, if there exists at least one LTS implementation I of M such that for all traces in I, restricted to the scope Θ, the trigger P is never satisfied.*

For instance, the triggered scenario *Receive*, shown in Figure 2.a, is vacuously satisfiable since there exists at least one implementation (e.g. the LTS in Figure 3.a) of the MTS synthesised from the scenario (shown in Figure 5) where the trigger is never satisfied.

The first step in detecting vacuity involves automatically synthesising an MTS that characterises all LTSs that satisfy the given set of triggered scenarios. The synthesis is done on a per triggered scenario basis, following the technique described in [24]. Once constructed, the generated MTSs are then merged. If the merge is successful, then the resulting MTS describes all implementations that satisfy all triggered scenarios. If it is unsuccessful then this indicates that the scenarios are inconsistent and hence do not have an implementation.

The second step comprises performing a vacuity check on the MTS resulting from the merge against a property that informally says: *it is always the case that a scenario's trigger does not hold*. The property can be expressed formally in FLTL and automatically constructed from the scenario's trigger. We refer to this property as the *negated trigger property* of a triggered scenario. For instance, the negated trigger property for the uTS *Receive* in Figure 2 is

$$\mathsf{G}\neg(\neg \mathit{Ringing} \wedge \dot{\mathit{incomeCall}}) \tag{1}$$

Model checking an MTS against a property is akin to checking the property against every LTS implementation that it describes. The result can be one of three values: all, none, and some, or more formally, true, false or undefined.

When checking for vacuity, if the result of model checking an MTS against a negated trigger property is true, then every trace in every implementation of the MTS satisfies the property, i.e. the trigger of the scenario under analysis never occurs. This entails that any implementation that satisfies the available specification vacuously satisfies the triggered scenario. This is an undesirable situation as it is not possible to extend the specification to avoid vacuity, and hence it must be revised. If the verification returns false, every implementation of the MTS has a trace that violates the property, i.e. in which the trigger occurs. Hence the triggered scenario is not vacuously satisfiable, so the specification need not be augmented for this particular scenario.

If the result of the verification is undefined, this means that there are some implementations that satisfy the concerned scenario vacuously and others that satisfy it non-vacuously. The purpose of the second phase of this approach is to automatically learn triggered scenarios that will prune out all implementations of the MTS that vacuously satisfy the concerned triggered scenario. However, for such learning to occur, examples of how the system-to-be may trigger the scenario under analysis are needed. In cases where the result is either false or undefined, a counterexample is given. In the former case, the counterexample is a trace that violates the property and can be exhibited by all LTS implementations. In the latter case, the counterexample is a trace that violates the property and can be exhibited by at least one LTS implementation. Our interest lies in

the latter case where the model checker provides an example of how some implementations can achieve the scenario's trigger. This trace, leading to the trigger, is taken as a non-vacuity witness for that triggered scenario.

Returning to our running example, verifying the MTS generated from the scenario *Receive* and *Phone* against the property (1) using the MTSA model checker [11] gives the following violation:

```
Trace to property violation in Never_Trigger_Receive:
     incomeCall       Calling
No. MobilePhone+ does not satisfy Never_Trigger_Receive
```

The trace produced by the MTSA is the shortest trace in an implementation of the MTS that violates the negated trigger property. In particular, the above means that there exists some implementations of the mobile phone system in which the phone is not ringing and there is an incoming call, i.e. where the trigger of scenario *Receive* is reachable (e.g. Figure 3.b).

Once the model checker detects a non-vacuity witness, this is shown to the engineer for validation. The engineer might indicate that the trace is *positive*, i.e. should be required in all implementations, or *negative*, i.e. should be proscribed in all implementation. In the former case, the trace is given to the learning phase. In the latter case, the engineer is expected to produce at least one positive non-vacuity witness which satisfies the trigger. Positive witnesses can be automatically generated from the model checker.

4.2 Learning Triggered Scenarios

The input to this phase is a set of triggered scenarios, fluent definitions and positive and negative non-vacuity witnesses. The output is a set of triggered scenarios, called a *required*, that ensure that a trigger is required by at least one positive witness trace in every implementation of the system.

Definition 3 (Required Scenarios). *Let TS be a set of triggered scenario, D a set of fluent definitions and $\Sigma^+ \cup \Sigma^-$ a set of positive and negative traces consistent with TS. Then a set of triggered scenarios S is said to be required of TS with respect to traces in $\Sigma^+ \cup \Sigma^-$ if the MTS synthesised from $TS \cup S$ requires each trace in Σ^+ but none of the traces in Σ^-.*

To compute new triggered scenarios, we use an Inductive Logic Programming (ILP) approach described in [22]. ILP is a machine learning technique for computing a new solution H that explains a given set E of examples with respect to a given (partial) background knowledge B [20]. Within the context of our problem, the background comprises the given set of triggered scenarios and fluent definitions whereas the positive and negative traces constitute the examples. A solution is a set of required scenarios requiring the positive non-vacuity witnesses, but none of the negative ones.

To perform the learning task, the input is encoded into Prolog. We have defined a sound translation (based on an extension of that given in [4]) that maps triggered scenarios and fluent definitions into an Event Calculus (EC) [15] program.

The program makes use of new predicates such as *required*, *maybe*, *reachable*, *trigger_satisfied*. The encoding of a triggered scenario TS results in a number of Prolog rules, one for each event appearing in the main chart of TS. Each rule defines the predicate *trigger_satisfied(e,T_m, S)*, where e is the event appearing in the main chart, T_m is the time variable associated with the location m at which the trigger is satisfied, and S is a trace variable[1]. The body of this rule contains $happens(e, T\_l, S)$ atoms for each event e at location $l < m$ in TS, and a conjunction of literals $(not)\ holds\_at(f_i, T\_l, S)$ for each fluent $(\neg)f_i$ that appears in a property $(\neg)f_1 \wedge \ldots \wedge (\neg)f_n$ at location $l < m$ in TS[2]. The order of the time variables in EC respects the location ordering in TS. Constraints over the type of triggering rule, i.e. existential or universal, are also defined according to the semantics in [24]. The scope of any scenario to be learned is constrained to be a subset of the events appearing in the positive non-vacuity witness. The main charts are encoded to ensure the soundness of the resulting program and consistency of learned hypotheses.

The solution search space is governed by a language bias which defines the syntactic structure of plausible solutions. To learn triggered scenarios, we define a language bias to capture rules with triggers as conditions and triggered events as its consequents. The language bias is also set to compute sequences of events leading to the main chart of the given scenarios so that all computed scenarios are consistent with the existing ones. For every positive example, the learning tries to construct a solution H which explains why a certain sequence of events must be required either existentially or universally, within a given scope. It then performs a generalisation step in which it tries to weaken the conditions to cover required occurrences of the triggered sequence in other traces. This generalisation can be controlled by providing several positive and negative non-vacuity witnesses.

The learning succeeds in computing a solution if there is at least one event occurrence in a positive non-vacuity witness that is not required by an existing triggered scenario. If several possible required scenarios exists, these will be given as output and it is the engineer's task to select the appropriate ones from those available. The number of triggered scenarios learned can be influenced by a number of factors including the number of events in the scope of the scenario to be learned, their occurrences in the example traces and the number of given negative traces. All produced scenarios are guaranteed to be consistent with the existing specification and the traces provided, as stated in Theorem 1 below.

Theorem 1. *[Soundness of Learning] Let TS be a set of triggered scenarios, D a set of fluent definitions and $\Sigma^+ \cup \Sigma^-$ a set of positive and negative traces consistent with TS. Let $\Pi = B \cup E$ be the EC encoding of TS, D and $\Sigma^+ \cup \Sigma^-$ into background knowledge B and examples E. If H is a solution to E with respect to B, then the set of learned triggered scenarios T, where T is the triggered scenarios corresponding to H, are required of TS w.r.t. traces in $\Sigma^+ \cup \Sigma^-$.*

[1] In Prolog, variables (resp. constants) start with a capital (resp. lowercase) letter.

[2] The notation $(\neg)\phi$ is a shorthand for ϕ or $\neg\phi$. A similar interpretation is used for $(not)\phi$.

The proof is by contradiction. In brief, it assumes that a trace $\sigma+ = e_1, \ldots, e_n$ is not required in the MTS synthesised from $TS \cup T$. Then it goes to show that this results in a program Π that does not contain any rule that requires the occurrence of some event e_i in the trace σ^+. Given that this leads to a contradiction as H is a set of rules that requires the occurrence of each event in σ^+, then $\sigma+ = e_1, \ldots, e_n$ is shown to be a required trace. The proof for σ^- is done in a similar fashion. As a corollary of the above theorem, when the traces are examples of positive and negative non-vacuity traces to triggers in *TS*, the learned triggered scenarios will guarantee that each positive non-vacuity trace is required in every implementation but none of the negative non-vacuity traces.

The choice of which learned scenarios to include may have an impact on later iterations. For instance, selecting a universal scenario over an existential one might imply that an incoming call is the only observed event when there is no incoming call detected and the phone is not ringing, for a given scope. It is obvious that selecting such an interpretation would prevent the occurrence of any behaviour other than that which is depicted in the main chart of the learned scenario within that scope. Therefore, we found that it is often preferable to select existential scenarios over universal at early stages of the elaboration process.

In our running example, the ILP tool computed two alternative required extensions as solutions; an existential and a universal. The learned existential triggered scenario (shown in Figure 6) states that whenever the user is not engaged in a call and the phone is not ringing then it is possible to accept an incoming call. The scope is restricted to the event appearing in the scenario. The universal contained the same trigger, main chart and scope.

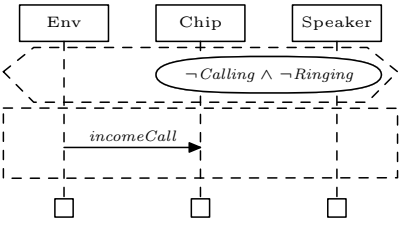

Fig. 6. A learned existential triggered scenario *IncomingCalls*

Once the engineer has made a selection, the learned scenario is added to the initial set. Then the newly synthesised MTS (which is a refinement of the original) is verified against the negated trigger property of another triggered scenario. If the model checker returns false for all negated trigger properties of the available scenario then this marks the end of the elaboration task with respect to the concerned trigger. If however, the model checkers returns undefined, then the process is repeated again with respect to the new non-vacuity witness trace.

Note that the encoding of the specification and the computation are hidden from the engineer. In fact, the engineer only needs to provide the learning system with the triggered scenarios, witnesses and fluent definitions and it will automatically propose a set of required scenarios with respect to the witnesses.

5 Case Studies

We report on the results obtained from two case studies, the Philips television set configuration from [23] and the air traffic control system in [9]. These were chosen because they have been used as case studies in much of the literature for which an elaborated scenario specification exists.

For the Philips configuration set, the specification contained existential triggered scenario from [23]. For the air traffic control system, it included a set of universal live sequence charts from [9]. All available scenarios were produced by third parties. We extracted a subset of the scenarios that constituted the main behaviour requirements provided, i.e. sunny day, normal behaviour. The aim of the case studies was (i) to investigate the capability of the approach in identifying the partiality of the given specification (ii) to verify that the learned triggered scenarios resulted in implementations that non-vacuously satisfied the given scenarios and finally (iii) to ensure that the learned scenarios were relevant to the domain at hand. The latter was achieved by comparing the learned scenarios with the available specification.

5.1 Philips Television Set Configuration

This case study is on a protocol used in a product family of Philips television sets. It include multiple tuners and video output devices that can be configured by a user. The protocol is concerned with controlling the signal path to avoid visual artefacts appearing on video outputs when a tuner is changing frequency.

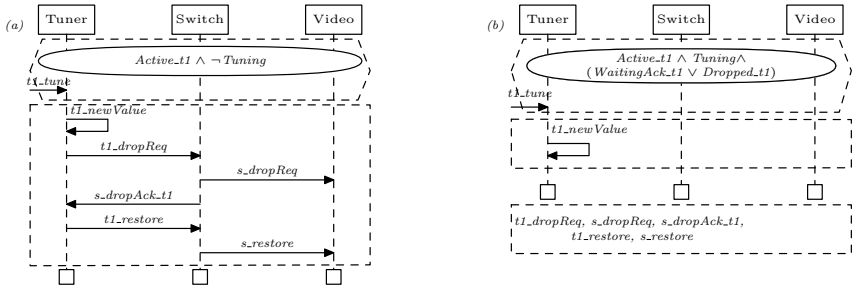

Fig. 7. Triggered scenarios: (a) *Tuning_t1_Active_t1* (b) *NestedTuning_t1_Active_t1*

We discuss here the results obtained by applying our approach to two existential triggered scenarios; *Tuning_t1_Active_t1* and *NestedTuning_t1_Active_t1* shown in Figure 7. The fluents appearing in the triggers are defined as follows.

```
Active_t1 =<set_Active_t1, set_Active_t2, tt>
Tuning_t1 =<t1_tune, {s_restore, set_Active_t1, set_Active_t2}, ff>
WaitingAck_t1 =<t1_dropReq, s_dropAck_t1, ff>
Dropped_t1=<s_dropAck_t1, t1_restore, ff>
```

A vacuity check was performed for the scenario *Tuning_t1_Active_t1* first by checking the system MTS resulting from the merge of the scenarios' MTSs against the following negated trigger property.

$$\mathsf{G}\neg((Active\_t1 \wedge \neg Tuning\_t1) \ \wedge \ (\ \mathsf{X}\ t1\_\dot{t}une)) \tag{2}$$

The model checker produced the shortest non-vacuity witness, i.e. *t1_tune*. Based on the description given in [23], we provided the system with a negative non-vacuity trace where a *t1_tune* events occurs when tuner 2 is active instead. From these traces, the learning produced two plausible triggered scenarios for the event *t1_tune*, one existential and one universal, requiring *t1_tune* event to happen when tuner 1 is active and not tuning. Choosing the universal scenario implies that a *t1_tune* must be observed every time the trigger is satisfied. We selected an existential interpretation to allow exploration of other behaviour (see Figure 8.a). The approach was also used to check for the vacuous satisfiability of *NestedTuning_t1_Active_t1* (Figure 7.b). Our application resulted in a single existential triggered scenario shown in Figure 8.b. The learned scenarios were added to the initial specification. Verifying the new MTS against the negated trigger properties showed that both triggered scenarios *Tuning_t1_Active_t1* and *NestedTuning_t1_Active_t1* were non-vacuously satisfied in all implementations.

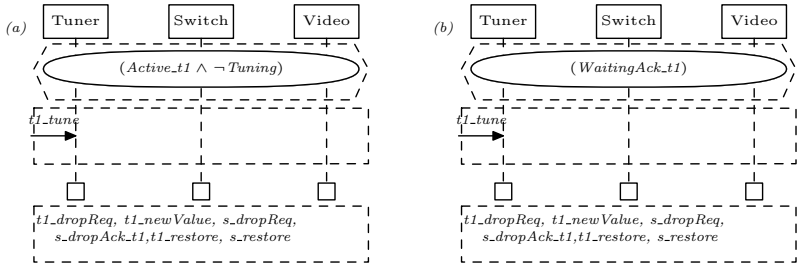

Fig. 8. Learned existential triggered scenarios (a) *TuneAllowed_t1_Active_NotTuning* (b) *TuneAllowed_WaitingAck_t1*

During the analysis phase, the approach also helped in detecting negative non-vacuity witnesses. For example, the analysis showed that the specification permitted behaviour in which the tuner sent a request to drop the signal without the user requesting a tune signal, and another in which a nested tune occurred outside the 'storing regions', i.e. when *Waiting_Ack_t1* and *Dropped_t1* are both false. With the identification of positive non-vacuity traces the learning ensured

that the learned scenarios did not require the occurrence of such events under such conditions. The final set of scenarios produced using our proposed method were validated against those generated by running the existing protocol in [23].

5.2 Air Traffic Control System

The Center-TRACON Automation System (CTAS) is a system for controlling and managing air traffic flow at major terminal areas to reduce travel delays and improve safety. The communication between the CTAS components is managed by the Communication Manager (CM) component which stores all interactions in a database and sends any required information to the requesting components. Among the CTAS requirements is that every client using weather data should be notified of any weather update. The scenarios in Figure 9 are universal triggered scenarios reproduced from [9] regarding the successful and failed update of new weather information.

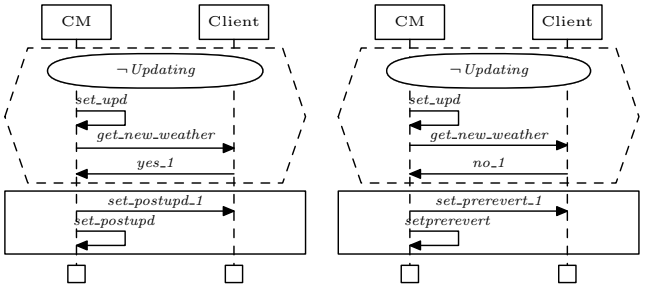

Fig. 9. Successful and failed update universal triggered scenarios

An application of our approach to this problem resulted in a total of six universal triggered scenarios and a single existential one. The set of uTSs computed were in fact the same triggered scenarios given in [9]. An excerpt is shown in Figure 10. Our approach also computed the existential scenario depicted in Figure 10 for setting the weather cycle status to "pre-updating" which was not present in the specification but is necessary to start the update process.

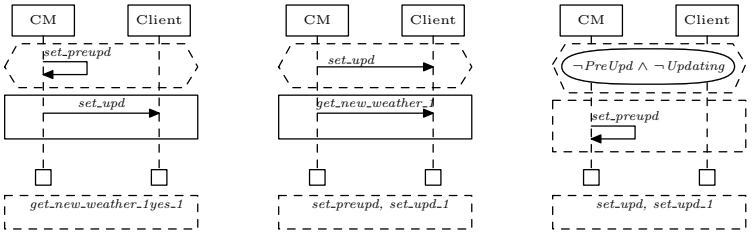

Fig. 10. CTAS learned universal and existential triggered scenarios

6 Discussion and Related Work

Although this paper discusses learning from vacuously satisfiable scenarios, the approach can be generalised for other forms of conditional scenarios (e.g. LSCs) and conditional statements (e.g. goals and requirements [10]). The exact definition of the learning task could be customised to the specific problem at hand.

There has been much research on providing automated support for elaborating scenario-based specifications [1,26]. However, much of the existing work is either informal, deals with message sequence charts or does not address the problem of vacuity introduced by conditional scenarios.

To the best of our knowledge, there is no prior work on applying learning algorithms to compute triggered scenarios. However, using model checking to detect vacuously satisfiable specifications has been the subject of several research efforts e.g. [7,18,13]. The work in [13] for instance presents a technique for detecting vacuity in temporal properties expressed in $\mathcal{X}$CTL. They use a multi-valued model checking algorithm to determine which subformulas in a given expression are vacuously satisfied in a model. Our approach is similar in that we use model checking algorithms to detect non-vacuity, and to produce a non-vacuity witness. However, in addition to the type of specifications used, our work differs in that it computes possible ways to avoid non-vacuity.

In our previous work, we combined the use of model checking and ILP to provide automated support for *different* software engineering tasks. In [2], a complete set of operational requirements in the form of preconditions and trigger conditions are iteratively learned from goal models. In [3], model checking and ILP are used to infer the missing conditions required to guarantee that a discrete time goal-based specification only admits non-zeno behaviours. Although we use here the same techniques, i.e. model checking and ILP, as in [2,3] to solve different software engineering tasks, their application to the problem of detecting vacuity and learning new triggered scenarios has posed three new main challenges: partial behaviour models, branching time and scoping of learned expressions. These points are elaborated below.

The problem addressed in this paper requires the ability to reason about universal and existential statements (both [2] and [3] deals only with universal statements). This means that traditional 2-valued semantic domains for these specifications are inadequate and a partial behaviour formalism such as MTS is required. As a consequence, the logic programming language is extended with new predicates (e.g. *required* and *maybe*). In addition, the use of triggered existential scenarios introduces statements that have a branching time semantics (in both [2,3] learning is only defined over properties with linear time). For this, the logic programming language has been extended to formalise hypothetical paths that branch from particular positions in a trace. The scenario language used in this paper supports scoping each scenario with an alphabet. (both [2,3] consider statements to have the same scope). This entails that the learning procedure must not only consider the scope of given scenarios (i.e. axioms for ensuring that the satisfiability notion with respect to a given scope are required) but also learned scenarios must include the scope for which they are intended. In [4] we

have presented preliminary work on the application of ILP in the context of MTS models. The focus there is on learning safety properties to requires some possible transitions from given traces. In this paper we build on the formalisation of MTSs in the logic programs and extend it to represent statements with a branching semantics and scoping which are not considered in [4]. Finally, note the work in [6] addresses the problem of learning operational requirements as in [2,5] but without the use of model checking.

7 Conclusion and Future Work

This paper presents a novel tool-supported approach for the elaboration of partial, conditional scenario-based specifications. In particular, we show how model checking can be used for identifying vacuously satisfiable triggered scenarios and how inductive logic programming can support the computation of new triggered scenarios needed to avoid such vacuity.

As part of this work and future work, we intend to investigate alternative methods for learning scopes of triggered scenarios. We also aim to extend the approach to resolve inconsistencies in the specification by providing support for detecting which parts of the specifications are the cause of inconsistency (building upon results in [13]) and learning possible revisions to the triggered scenarios necessary to resolve inconsistencies.

References

1. Alexander, I., Maiden, N.: Scenarios, stories, use cases: through the systems development life-cycle. Wiley (2004)
2. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: Learning operational requirements from goal models. In: Proc. of 31st ICSE, pp. 265–275 (2009)
3. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: Deriving non-zeno behaviour models from goal models using ILP. J. of FAC 22(3-4), 217–241 (2010)
4. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: An inductive approach for modal transition system refinement. In: Tech. Comm. of 27th ICLP, pp. 106–116 (2011)
5. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Extracting requirements from scenarios with ILP. In: Proc. of 16th Intl. Conf. on ILP, pp. 63–77 (2006)
6. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Using abduction and induction for operational requirements elaboration. J. of Applied Log. 7(3), 275–288 (2009)
7. Armoni, R., Fix, L., Flaisher, A., Grumberg, O., Piterman, N., Tiemeyer, A., Vardi, M.Y.: Enhanced Vacuity Detection in Linear Temporal Logic. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 368–380. Springer, Heidelberg (2003)
8. Beatty, D.L., Bryant, R.E.: Formally verifying a microprocessor using a simulation methodology. In: Proc. of 31st DAC, pp. 596–602 (1994)
9. Bontemps, Y.: Relating Inter-Agent and Intra-Agent Specifications: The Case of Live Sequence Charts. PhD thesis, Faculties Universitaires Notre-Dame de la Paix, Namur Institut dInformatique, Belgium (2005)
10. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Comp. Program. 20(1), 3–50 (1993)

11. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: The Modal Transition System Analyser. In: Proc. of 23rd Intl. Conf. on ASE, pp. 475–476 (2008)
12. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: Proc. 11th ACM SIGSOFT FSE, pp. 257–266 (2003)
13. Gurfinkel, A., Chechik, M.: Extending Extended Vacuity. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 306–321. Springer, Heidelberg (2004)
14. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer-Verlag New York, Inc. (2003)
15. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. New Generation Comp. 4(1), 67–95 (1986)
16. Kramer, J., Magee, J., Sloman, M.: Conic: An integrated approach to distributed computer control systems. In: IEE Proc., Part E 130 (1983)
17. Kugler, H.-J., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal Logic for Scenario-Based Specifications. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 445–460. Springer, Heidelberg (2005)
18. Kupferman, O.: Sanity checks in formal verification. In: Conc. Theory, pp. 37–51 (2006)
19. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proc. of 3rd Annual Symp. on Log. in Comp. Science, pp. 203 –210 (1988)
20. Muggleton, S.H.: Inverse Entailment and Progol. New Generation Comp., Special Issue on ILP 13(3-4), 245–286 (1995)
21. Pressman, R.S.: Software Engineering: A Practitioner's Approach, 7th edn. McGraw-Hill Higher Education (2010)
22. Ray, O.: Nonmonotonic abductive inductive learning. J. of Applied Log. 7(3), 329–340 (2009)
23. Sibay, G.: The Philips television set case study, `http://sourceforge.net/projects/mtsa/files/mtsa/CaseStudies/`
24. Sibay, G., Uchitel, S., Braberman, V.: Existential live sequence charts revisited. In: Proc. of 30th ICSE, pp. 41–50 (2008)
25. Uchitel, S., Brunet, G., Chechik, M.: Behaviour model synthesis from properties and scenarios. In: Proc. of 29th Intl. Conf. on Softw. Eng., pp. 34–43 (2007)
26. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proc. of the 22nd ICSE, pp. 314–323 (2000)