CSSL: A Logic for Specifying Conditional Scenarios

Shoham Ben-David
University of Toronto
Toronto, Canada
shoham@cs.toronto.edu

Marsha Chechik
University of Toronto
Toronto, Canada
chechik@cs.toronto.edu

Arie Gurfinkel
SEI/CMU
Pittsburgh, USA
arie@cmu.edu

Sebastian Uchitel
University of Buenos Aires, Argentina
Imperial College London, UK
suchitel@dc.uba.ar

ABSTRACT

Scenarios and use cases are popular means of describing the intended system behaviour. They support a variety of features and, notably, allow for two different interpretations: existential and universal. These modalities allow a progressive shift from examples to general rules about the expected system behaviour. The combination of modalities in a scenario-based specification poses technical challenges when automated reasoning is to be provided. In particular, the use of conditional existential scenarios, of which use cases with preconditions are a common example, require reasoning in branching time. Yet, formally grounded approaches to requirements engineering and industrial verification approaches shy away from branching-time logics due to their relatively unintuitive semantics.

In this paper, we define an extension of an (industry standard) linear-time logic with sufficient branching expressiveness to allow capturing conditional existential statements. The resulting logic, called CSSL (Conditional Scenario Specification Language), has an efficient model-checking procedure. It supports reasoning about heterogeneous requirements specifications that include universal and existential statements in the form of use cases and conditional existential scenarios, and other sequence chart variants, in addition to general (linear) liveness and safety properties. We report on two industrial case studies in which the logic was used to specify and verify scenarios and properties.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Design,Verification

1. INTRODUCTION

Scenarios, use case and user stories are popular means of describing intended system behaviour [20, 31, 17, 1, 26]. They provide an intuitive framework for describing *examples* of how the system-to-be, its environment and users are expected to interact.

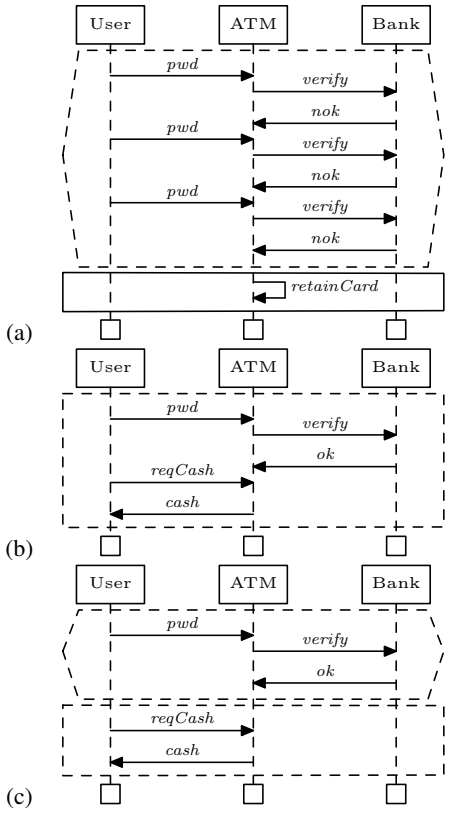

Figure 1: Several ATM scenarios: (a) P_2: a universal conditional scenario; (b) P_3: an existential scenario; (c) P_4: an existential conditional scenario.

Languages for formally describing scenario-based specifications have extended preliminary and widespread notations such as the ITU 1992 standard Message Sequence Charts (MSCs) [16] to allow for a variety of features. One important extension, embodied in Live Sequence Charts (LSCs) [9], is the possibility of distinguishing between existential and universal scenarios to allow a progressive shift from examples to general rules about the expected system behaviour [12], within one specification framework.

The combination of modalities in a scenario-based specification poses technical challenges when automated reasoning is to be provided. In particular, the use of conditional existential scenarios, of which use cases [17], when used with preconditions, are a notable example, require reasoning in branching time. Yet, formally grounded approaches to requirements engineering [30, 33] (and

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

Name (Type)	Description	Logical Expression	Exp. Res.
P_1	A non-authenticated user may never	LTL: $G(\text{logout} \rightarrow (\neg\text{cash } U\ (\text{verify} \wedge X(\text{ok}))))$	T
(U)	withdraw cash	PSL, CSSL: $G(\text{logout} \rightarrow (\neg\text{cash } U\ \text{verify} \cdot \text{ok}))$	T
P_2	Three consecutive unsuccessful login	LTL: $G(\text{pwd} \rightarrow X(\text{verify} \rightarrow X(\text{nok} \rightarrow X(\text{pwd} \rightarrow ... \rightarrow X(\text{nok} \rightarrow X\text{retainCard}))))$	T
(UC)	attempts must lead to retaining the user's card	PSL, CSSL: $G((\text{pwd} \cdot \text{verify} \cdot \text{nok})^3 \mapsto \text{nok} \cdot \text{retainCard})$	T
P_3	It is possible for the user to be authen-	LTL: $G(\text{pwd} \rightarrow X(\text{verify} \rightarrow X(\text{ok} \rightarrow X(\text{reqCash} \rightarrow G(\neg\text{cash})))))$	F
(E)	ticated, and then request and obtain cash	PSL: $G((\text{pwd} \cdot \text{verify} \cdot \text{ok} \cdot \text{reqCash}) \mapsto G(\neg\text{cash}))$	F
		CSSL: $\mathit{true} \mapsto (\text{pwd} \cdot \text{verify} \cdot \text{ok} \cdot \text{reqCash} \cdot \text{cash})$	T
P_4	Whenever a user successfully logs	LTL, PSL : N/A	N/A
(EC)	into an ATM, it should be possible for her to request and obtain cash	CSSL: $G((\text{pwd} \cdot \text{verify} \cdot \text{ok}) \mapsto X(\text{reqCash} \cdot \text{cash}))$	T
P_5	After login, it should be possible to	LTL, PSL : N/A	N/A
(EC)	withdraw money before changing the pin	CSSL: $G((\text{pwd} \cdot \text{verify} \cdot \text{ok}) \mapsto (\neg\text{changePin } U\ (\text{reqCash} \cdot \text{cash})))$	T
P_6	After login, it should be possible to	LTL, PSL : N/A	N/A
(EC)	change the pin before withdrawing money	CSSL: $G((\text{pwd} \cdot \text{verify} \cdot \text{ok}) \mapsto (\neg(\text{reqCash} \cdot \text{cash})\ U\ \text{changePin}))$	T
P_7	A legitimate user should not be limited	LTL, PSL : N/A	N/A
(EC)	in the number of operations she is allowed	CSSL: $G((\text{pwd} \cdot \text{verify} \cdot \text{ok}) \mapsto (G\ \neg\text{logout}) \wedge (GF(\text{reqCash} \vee \ldots \vee \text{changePin})))$	T

Table 1: Several properties of the ATM system. The column "Type" lists the scenario type: universal conditional (UC), existential conditional (EC), existential (E) or universal(U).

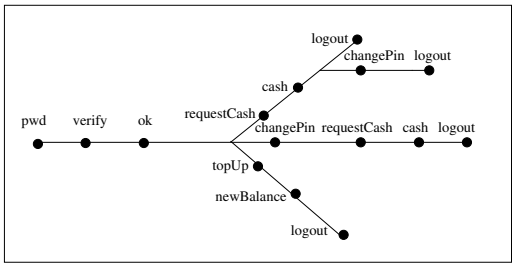

Figure 2: A collection of behaviors of an ATM system.

industrial verification researchers [35]) shy away from branching-time logics due to their relatively unintuitive semantics.

In this paper, we define an extension of a linear-time logic with sufficient branching expressiveness to allow expressing conditional existential statements. The resulting logic, called CSSL (Conditional Scenario Specification Language), has an efficient model-checking procedure and therefore supports reasoning about heterogeneous requirements specifications that include universal and existential statements in the form of use cases and conditional existential scenarios, LSCs, MSCs, and other sequence chart variants, in addition to general (linear) liveness and safety properties.

Conditional Existential and Universal Scenarios. An existential scenario specifies behavior that the system is expected to exhibit, among others which are yet to be specified or are specified elsewhere. For example, an existential scenario P_3 of the ATM system is depicted in Figure 1(b) using a notation in the style of MSCs. The interpretation of the scenario is: "it is possible for the user to be authenticated, and then request and obtain cash". We denote that the scenario is to be interpreted existentially using a dashed line around the sequence chart.

Conditional scenarios are, loosely, statements of the form "if x then y", where x and y are scenarios. For instance, a conditional universal scenario P_2 is depicted using an LSC notation in Figure 1(a). The top part, enclosed in a dashed bracket, is a conditional, called a *prefix* (or, in the LSC terminology, a *pre-chart*). The bottom part, enclosed in a solid rectangle, is the *suffix* (or, in the LSC terminology, a *main chart*).

Conditional universal scenarios are interpreted as follows: whenever the prefix holds, then the suffix *must follow*. Hence, the scenario of Figure 1(a) states that when three consecutive authentication failures occur, the ATM must retain the user's card.

Conditional existential scenarios, although syntactically very similar to their universal counterparts, are interpreted differently. In Figure 1(c), we depict an conditional existential scenario P_4 in the notation style of eTS [29]. The suffix is enclosed in a dashed rectangle to distinguish it from universal conditional scenarios. The interpretation of conditional existential scenarios is that when the prefix occurs, then the suffix should be *possible*. Thus, the conditional existential scenario of Figure 1(c) states that whenever a user successfully logs into an ATM, it *should be possible* for her to request and obtain cash.

Unlike conditional *universal* statements, conditional existential statements do not forbid behaviour. That is, there can be an execution of the system where the prefix is true but the suffix is not. For example, it is possible to have an execution of the ATM system where a user successfully logs in and then tops up her mobile phone rather than withdrawing cash. However, conditional existential statements require that every time the prefix holds, *some* continuation where the suffix holds must exist. This semantics, albeit with a different syntax, is frequently found in the form of use cases with a preconditions: [17] When the precondition holds, it is expected (but not mandatory) that the user be able to "execute" the use case.

Thus, conditional existential statements refer to the possibilities or branches of behaviour that are available. When the current state of the system is that the user has logged in, a number of possibilities may exist, but obtaining cash must be one of them (see Figure 2). Table 1 shows three other existential conditional scenarios, P_5, P_6 and P_7, capturing possible interactions between the user and the ATM system, once she logs in. Several other examples of existential conditional scenarios are described in the TV controller case study in Section 5.1.

Reasoning about System Behaviour. Automated reasoning about specifications is an important software engineering task that can help identify problems early and thus result in cheaper fixes. Reasoning requires representing statements about system behaviour in some formal language – a logic. Temporal logics have been used to reason about software system behaviour extensively. Specifically, *linear* temporal logics such as LTL [23] and its more recent extension, Property Specification Language (PSL) [14, 10], which adds

regular expressions, are strongly favoured for describing requirements [30, 33] as they have a relatively intuitive semantics.

For example, properties P_1-P_3 are expressed in LTL as shown in Table 1: P_1 is a (non-conditional) universal statement , P_2 a conditional universal statement and P_3 is a non-conditional existential statement. Note that to verify that a system satisfies P_3 described as a scenario in Figure 1(b), it is necessary to check that the LTL formalization of P_3 does not hold. A *counterexample* to the property acts as a witness of the existential statement. We indicate this by putting "F" as the expected result (Exp. Res.) in Table 1. In addition, note that for P_2 (and other properties), we assumed that events occur consecutively. Relaxing this assumption would result in a more complex LTL formula (with lots of "until"s).

Table 1 also lists properties $P_1 - P_3$ in PSL. These show that PSL can be more compact and more convenient for capturing sequences of events, commonly found in scenarios. The properties use a familiar regular expression syntax, with "$\cdot$" expressing concatenation and "x^n" or "x^*" – finite repetition of string x. For example, in the PSL expression of property P_2 in Table 1, the entire string of `pwd`, `verify` and `nok` repeats three times. The PSL formalization of P_3 uses the *suffix implication operator* ($\mapsto$) which captures the intended semantics of conditional universal scenarios: when the left hand side holds, the right hand side must follow.

Conditional existential scenarios, on the other hand, cannot be expressed using linear logics, since their combination of universal and existential fragments requires branching (and has, in fact, been formalized using CTL and CTL* [28]).

CSSL: Conditional Scenario Specification Language. In this paper, we define and study an extension of PSL with sufficient branching expressiveness to allow us to capture conditional existential statements by enriching PSL with a top-level branching operator called "branching suffix implication", and denoted by a symbol $\mapsto$. The resulting logic, called CSSL, allows us to compose "true" linear fragments specifying prefix and suffix. For example, the prefix of the existential conditional scenario P_4 in Table 1 is expressed as $(\text{pwd}\cdot\text{verify}\cdot\text{ok})$, i.e., a regular expression consisting of concatenation of three individual events. Its suffix is $X(\text{reqCash} \cdot \text{cash})$, describing that in the next state, `reqCash` should happen, followed by `cash`. Thus, the entire scenario is required "globally" and is expressed in CSSL as $P_4 = G((\text{pwd} \cdot \text{verify} \cdot \text{ok}) \mapsto X(\text{reqCash} \cdot \text{cash}))$. CSSL expressions for scenarios P_5, P_6, and P_7 are shown in Table 1 as well. Moreover, while only the negation of the existential property P_3 can be expressed in PSL or LTL, the property can be easily expressed in CSSL, with the prefix *true* (see Table 1).

The resulting logic is strictly more expressive than PSL and is in fact a fragment of CTL* extended with regular expressions. Yet, unlike the full CTL*, this language has an efficient model-checking procedure. CSSL is not the only tractable fragment of CTL*. Linear tractable fragments include LTL and PSL (which, as we said earlier, are not expressive enough for conditional existential statements). A branching logic CTL is also tractable; yet it is not an option because it does not extend PSL (specifically, its regular expressions) and does not support expressing sequences of events commonly present in scenarios. Our approach contributes a tractable linear logic with regular expressions and limited branching.

Contributions. This paper makes the following contributions:

- We introduce a new operator, called *branching suffix implication*, extending PSL, and show that this operator is useful for expressing existential conditional scenarios. Therefore, heterogeneous specifications that include use cases, sequence chart variants and general (linear) safety and liveness properties can now be expressed in the same "linear with a bit of branching" language CSSL.
- We show that CSSL is not more computationally complex than PSL. We then develop a model-checking algorithm for CSSL, implemented on top of NuSMV [7].
- We report on our experience using CSSL to specify and verify properties of two industrial examples.

The rest of the paper is organized as follows: In Section 2, we give the technical background; introduce the new logic CSSL in Section 3 and, in Section 4, describe how to model-check this logic, including our implementation on top of NuSMV. We report on two case studies in Section 5. After comparing our approach to related work in Section 6, we conclude in Section 7 with a summary and an outline of future research directions.

2. BACKGROUND

In this section, we present the necessary definitions and notation.

Models and Properties.

DEFINITION 1 (KRIPKE STRUCTURE). *Let AP be a finite set of atomic propositions. A* Kripke structure *K over AP is a tuple $K = (S, I, R, L)$, where S is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a* total *transition relation, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$, and L is a labeling function $L : S \to 2^{AP}$ that labels each state with the set of variables true in that state.*

We say that $\pi = s_0, s_1, ...$ is a *path* in K iff $s_0 \in I$ and $\forall i$, $(s_i, s_{i+1}) \in R$. We denote by π^k the suffix of π starting at s_k.

DEFINITION 2 (LINEAR TEMPORAL LOGIC (LTL) [23]). *Given a finite set AP of atomic propositions, formulas of LTL are recursively defined as follows:*

- *Every atomic proposition is an LTL formula.*
- *If φ and ψ are LTL formulas, then so are $\neg\varphi$, $\varphi \wedge \psi$, $X\varphi$ and $[\varphi U \psi]$,*

where X and U stand for the "next" and the "until" operators, respectively.

Additional operators "future" and "globally" are defined as syntactic sugar using the ones above:

$$F\varphi = [\textit{true}\ U\varphi] \qquad G\varphi = \neg F \neg \varphi$$

The formal semantics of an LTL formula is defined with respect to an infinite path $\pi = s_0, s_1, ...$ of a given Kripke structure $K = (S, I, R, L)$, where $s_0 \in I$:

$$\begin{array}{lll}
\pi^i \models p & \text{iff} & p \in L(s_i) \\
\pi^i \models \neg\varphi & \text{iff} & \pi^i \not\models \varphi \\
\pi^i \models \varphi \wedge \psi & \text{iff} & (\pi^i \models \varphi) \ \wedge \ (\pi^i \models \psi) \\
\pi^i \models X\varphi & \text{iff} & \pi^{i+1} \models \varphi \\
\pi^i \models [\varphi U \psi] & \text{iff} & \exists k \geq i \text{ s.t. } (\pi^k \models \psi) \wedge \\
 & & (\forall i \leq j < k \centerdot \pi^j \models \varphi)
\end{array}$$

A formula φ holds in a Kripke structure K, denoted $K \models \varphi$, iff it holds in every path of K.

DEFINITION 3 (COMPUTATION TREE LOGIC (CTL) [8]). *Given a finite set AP of atomic propositions, CTL formulas are recursively defined as follows:*

- *Every atomic proposition is a CTL formula.*

- *If φ and ψ are CTL formulas then so are $\neg\varphi$, $\varphi \wedge \psi$, $AX\varphi$, $EX\varphi$, $A[\varphi U \psi]$ and $E[\varphi U \psi]$,*

where AX and EX are "forall next" and "exist next" operators, and AU and EU are "forall until" and "exists until" operators, respectively.

Additional operators, "forall/exists future" and "forall/exists globally", are defined as syntactic sugar over the ones above:

$$AF\varphi = A[\textit{true}\ U\varphi] \qquad EF\varphi = E[\textit{true}\ U\varphi]$$
$$AG\varphi = \neg E[\textit{true}\ U\neg\varphi] \qquad EG\varphi = \neg A[\textit{true}\ U\neg\varphi]$$

The formal semantics of a CTL formula is defined with respect a Kripke structure $K = (S, I, R, L)$.The notation $K, s \models \varphi$ means that the formula φ is true in state s of the Kripke structure K.

$$\begin{array}{lll}
K, s \models p & \text{iff} & p \in L(s) \\
K, s \models \neg\varphi & \text{iff} & K, s \not\models \varphi \\
K, s \models \varphi \wedge \psi & \text{iff} & (K, s \models \varphi)\ \wedge\ (K, s \models \psi) \\
K, s_0 \models\ AXp & \text{iff} & \text{for all paths } (s_0, s_1, ...)\,.\, K, s_1 \models p \\
K, s_0 \models\ EXp & \text{iff} & \text{for some path } (s_0, s_1, ...)\,.\, K, s_1 \models p \\
K, s_0 \models A[\varphi U \psi] & \text{iff} & \text{for all paths } (s_0, s_1, ...). \\
 & & (\exists i \geq 0\,.\, K, s_i \models \psi) \wedge \\
 & & (\forall 0 \leq j < i\,.\, K, s_j \models \varphi) \\
K, s_0 \models E[\varphi U \psi] & \text{iff} & \text{for some path } (s_0, s_1, ...). \\
 & & (\exists i \geq 0\,.\, K, s_i \models \psi) \wedge \\
 & & (\exists 0 \leq j < i\,.\, K, s_j \models \varphi)
\end{array}$$

We say that a CTL formula φ holds in a Kripke structure $K = (S, I, R, L)$ iff $\forall s_i \in I\,.\, K, s_i \models \varphi$.

DEFINITION 4 (FAIR COMPUTATION). *Let b be Boolean expression over AP and K be a Kripke structure. We say that a computation path π of K is* fair *with respect to the fairness constraint b iff the LTL formula $GF(b)$ holds on π.*

We write $K, s \models_{fair} \varphi$ for fair *satisfaction of φ by K and s that is defined as $\models$ above with all "paths" replaced by "fair paths".*

Automata and Computations.

DEFINITION 5 (AUTOMATON). *An* automaton *A is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where: Q is a finite set of states, Σ is a finite set of symbols (the* alphabet*), $\delta \subseteq Q \times \Sigma \times Q$ is a* total *transition relation , $Q_0 \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final (accepting) states.*

A *run* r of an automaton on an input word $w = a_0, a_1, ...$ is a (finite or infinite) sequence of states $q_0, q_1, ...$, where $\forall i, q_i \in Q$ such that $q_0 \in Q_0$, $(q_i, a_i, q_{i+1}) \in \delta$.

We use two kinds of automata: non-deterministic finite automata (NFA) and non-deterministic Büchi automata (NBA). If A is an NFA, runs over it are finite. We say that a run $r = q_0, q_1, ..., q_n$ is *accepting* iff $q_n \in F$. Thus, an NFA accepts finite words over Σ. The universe of finite words is denoted by Σ^*.

Runs of an NBA A are infinite. For a run r, we define *Inf*(r) to be the set of states that occur infinitely often in r. We say that r is *accepting* iff *Inf*$(r) \cap F \neq \emptyset$. Thus, NBA automata accept infinite words over Σ. The universe of infinite words is denoted by Σ^ω.

A word $w \in \Sigma^* \cup \Sigma^\omega$ is *accepted* by A iff there exists as accepting run for it. The language of A, denoted $\mathcal{L}(\mathcal{A})$, is the set of all words accepted by A.

DEFINITION 6 (FUSION AUTOMATON [10]). *Let $A = (Q_A, \Sigma_A, \delta_A, Q^0_A, F_A)$ be an NFA and $B = (Q_B, \Sigma_B, \delta_B, Q^0_B, F_B)$ be an NFA or an NBA. The* fusion automaton *of A and B is an automaton $A \circ B = (Q_A \cup Q_B, \Sigma_A \cup \Sigma_B, \delta_{A\circ B}, Q^0_A, F_B)$, such that*

$$\delta_{A\circ B} = \delta_A \cup \delta_B \cup \{(q, a, q') \mid \exists f \in F_A\,.\,(q, a, f) \in \delta_A \wedge \exists i \in Q^0_B\,.\,(i, a, q') \in \delta_B\}\,.$$

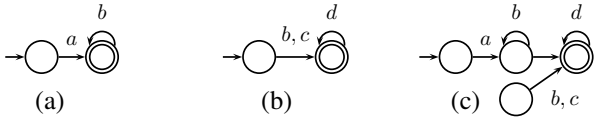

Figure 3: Automata for: (a) $a \cdot b^*$, (b) $(b + c) \cdot d^*$, and (c) the fusion $(a \cdot b^*) \circ ((b + c) \cdot d^*)$.

The fusion automaton $A \circ B$ has the initial states of A, and the final states of B. The transitions to a final state of A are fused with the transitions from an initial state of B. Note that while A is an NFA, B can be either an NFA or an NBA, and the resulting fusion automaton $A \circ B$ is of the same type as B. An example of a fusion automaton over NFAs in Figure 3(a) and Figure 3(b) is shown in Figure 3(c).

3. CONDITIONAL SCENARIO SPECIFICATION LANGUAGE (CSSL)

In this section, we define the syntax and semantics of Conditional Scenario Specification Language (CSSL). CSSL extends PSL with a *branching suffix implication* (BSI). We first review PSL and then describe syntax and semantics of the new branching operator, illustrating it on properties of our running example.

3.1 Property Specification Language (PSL)

PSL adds regular expressions to LTL via the operator $\mapsto$, known as the *suffix implication operator*. The formula $r \mapsto \varphi$ (read r "suffix-implies" φ) requires that *if* there exists a non-empty prefix of the path satisfying r *then* the suffix starting at the last letter of the prefix should satisfy φ.

DEFINITION 7 (SEREs). Sequentially Extended Regular Expressions *(SEREs) of PSL are built from a set AP of atomic propositions, defined recursively as follows:*

- *Every Boolean expression over AP is a SERE.*
- *If r and s are SEREs, then so are r^*, $r \cdot s$, $r \circ s$, $r \cup s$, and $r \cap s$,*

where r^* is the Kleene star operator, $r \cdot s$ is the concatenation of two SEREs, $r \circ s$ is fusion – the concatenation of two SEREs with an overlap of one state, $r \cup s$ is the union and $r \cap s$ is the intersection of two SEREs, respectively.

We omit SERE operators that can be defined from the ones above.

Let $K = (S, I, R, L)$ be a Kripke structure. The *language* of SERE over K is a set of finite paths, defined inductively as follows:

$$\begin{array}{lcl}
\mathcal{L}(b) & = & \{s \mid s \in S\ \wedge\ b \in L(s)\} \\
\mathcal{L}(r^*) & = & \{\varepsilon\} \cup \{\pi \mid \exists n \geq 1\,.\,\pi = \pi_1 \cdot \pi_2 \cdot ... \cdot \pi_n \wedge \\
 & & \forall 1 \leq i \leq n\,.\,\pi_i \in \mathcal{L}(r)\} \\
\mathcal{L}(r \cdot s) & = & \{\pi_1 \cdot \pi_2 \mid \pi_1 \in \mathcal{L}(r) \wedge \pi_2 \in \mathcal{L}(s)\} \\
\mathcal{L}(r \circ s) & = & \{\pi_1 \cdot s \cdot \pi_2 \mid \pi_1 \cdot s \in \mathcal{L}(r) \wedge s \cdot \pi_2 \in \mathcal{L}(s)\} \\
\mathcal{L}(r \cup s) & = & \mathcal{L}(r) \cup \mathcal{L}(s) \\
\mathcal{L}(r \cap s) & = & \mathcal{L}(r) \cap \mathcal{L}(s)
\end{array}$$

DEFINITION 8 (PSL). *PSL formulas are defined inductively:*

- *Every Boolean expression over AP is a PSL formula.*

- *If φ and ψ are PSL formulas and r a SERE, then the following are PSL formulas: $\neg\varphi$, $\varphi \wedge \psi$, $\varphi U \psi$, $X\varphi$, $r \mapsto \varphi$, and $r \diamond\!\!\rightarrow \varphi$.*

The *suffix implication operator* $r \mapsto \varphi$ states that whenever a prefix of the path matches a SERE r, then the suffix of the path (with one state overlapping) must satisfy φ.

The *suffix conjunction operator* $r \diamond\!\!\rightarrow \varphi$ holds iff there exists some prefix of the path that matches r and the (overlapping) suffix satisfies φ. Note that the suffix conjunction operator is dual to the suffix implication: $\neg(r \mapsto \varphi) = r \diamond\!\!\rightarrow \neg\varphi$.

The semantics of PSL formulas is defined with respect to infinite paths in a Kripke structure $K = (S, I, R, L)$. The semantics of a Boolean expression b and the formulas $\neg\varphi$, $\varphi \wedge \psi$, $\varphi\ U\ \psi$ and $X\varphi$ are the same as those for LTL. Let $\pi = s_0, s_1, ...$ be a path in K. We define semantics of the two suffix operators below:

$$\pi^i \models r \mapsto \varphi \quad \text{iff} \quad \forall j \,.\, (s_i \cdot \cdots \cdot s_j \in \mathcal{L}(r) \Rightarrow \pi^j \models \varphi)$$
$$\pi^i \models r \diamond\!\!\rightarrow \varphi \quad \text{iff} \quad \exists j \,.\, (s_i \cdot \cdots \cdot s_j \in \mathcal{L}(r) \wedge \pi^j \models \varphi)$$

Several PSL properties are shown in Table 1. Note that in some formulas $r \mapsto \psi$, we use a regular expression r' as ψ. This should be read as an abbreviation for $r' \diamond\!\!\rightarrow \mathit{true}$. For property P_1, neither the suffix implication nor the suffix conjunction operator is being used, and so the expression of the property is the same as LTL. Properties P_2 and P_3 use the suffix implication operator.

3.2 Branching Suffix Implication (BSI)

We now enrich PSL with a branching construct $\rightarrowtail$, called "branching suffix implication" (BSI). Intuitively, $p \rightarrowtail \psi$, where p is a regular expression and ψ a PSL formula, is satisfied by a Kripke structure K iff every finite path in K that satisfies p can be extended with a suffix (in K) that satisfies ψ.

DEFINITION 9 (BSI SATISFACTION). *Let r be a regular expression, ψ be a PSL formula, and $K = (S, I, R, L)$ be a Kripke structure. Then, $K, s \models (r \rightarrowtail \psi)$ iff for all paths $\pi = (s_0, \ldots, s_n)$ in K s.t. $\pi \models r$, there exists a suffix $\pi' = (s'_0, \ldots, s'_k)$ s.t. $\pi \cdot \pi'$ is a path in K and $\pi' \models \psi$.*

We say $K \models r \rightarrowtail \psi$ iff $\forall s_0 \in I \,.\, K, s_0 \models (r \rightarrowtail \psi)$.

Additional operators "next" and "globally" are defined as syntactic sugar using the BSI operator:

$$X(r \rightarrowtail \psi) = \mathit{true} \cdot r \rightarrowtail \psi \quad G(r \rightarrowtail \psi) = \mathit{true}^* \cdot r \rightarrowtail \psi$$

The resulting language, which includes PSL together with BSI, is called CSSL.

Table 1 lists several properties that use the new BSI operator (corresponding to $P_3 - P_7$). Note that most of these use the operator "globally" introduced above.

Consider the CSSL expression of property P_3. With prefix *true*, it easily allows us to specify the existential scenario as a property which is intended to hold in the system. Properties $P_3 - P_7$ illustrate that the suffix for a BSI operator can be an arbitrary LTL formula. Moreover, this suffix is finite for properties $P_3 - P_6$ and infinite for property P_7.

Since CSSL subsumes PSL, all scenarios in Table 1, corresponding to all four scenario types, are expressible in this new language.

4. MODEL-CHECKING CSSL

In this section, we discuss how to model-check CSSL properties. Specifically, we show how to reduce model-checking of a CSSL property with a BSI operator, of the form $\varphi = r \rightarrowtail \psi$, to the CTL model checking, via the construction of an auxiliary automaton.

We give the construction and prove its correctness in Section 4.1 and analyze its complexity in Section 4.2. In Section 4.3, we discuss the implementation of our method in NuSMV [7], and illustrate it on the ATM example. In the rest of this section, when we say a CSSL formula, we always mean a formula like φ above.

4.1 From CSSL to CTL

Let $K = (S, I, R, L)$ be a Kripke structure and $\varphi = r \rightarrowtail \psi$ be a CSSL specification. Without loss of generality, we assume that $\epsilon \notin \mathcal{L}(r)$, where ϵ is the empty string. Otherwise, we simply extend K with a dummy initial state and modify both r and ψ to skip that state.

In order to model check whether φ holds in K ($K \models \varphi$), we build an automaton A_φ and a CTL formula P_φ such that

$$K \models \varphi \iff K \times A_\varphi \models P_\varphi.$$

In order to reduce a CSSL formula $\varphi = r \rightarrowtail \psi$ to CTL, we first build the automaton A_φ (see below) and then use it to derive the CTL formula P_φ. Depending on ψ, the reduction is either *finite* or *infinite*, If ψ holds after a finite number of steps, i.e., its satisfaction is detected by an NFA, we call it a *finite* formula. For example, as in the CSSL expression of property P_5. Otherwise, it needs an NBA, which makes it an *infinite* formula, e.g., as in the CSSL expression of property P_7, ¬login should hold forever.

Note that our *finite* and *infinite* classification is based on the witnesses, and is dual to the standard notions of *safety* and *liveness* that are based on counterexamples. A safety formula, e.g., Gp, is one for which a counterexample is finite. A liveness formula, e.g., Fp, is one for which the counterexample is infinite. On the other hand, we say that Gp is *infinite* because its witness is infinite, and, similarly, Fp is *finite*.

4.1.1 Finite Suffix

Given a formula $\varphi = r \rightarrowtail \psi$, where ψ is finite, our goal is to build an automaton A_φ and a CTL formula $P_{\varphi_{\text{fin}}}$.

We first build two NFAs: A_r that accepts $\mathcal{L}(r)$, and A_ψ that accepts ψ. Both automata are built using standard methods [13, 36]. We then build A_φ – a *fusion* automaton (see Definition 6) of A_r and A_ψ.

For $P_{\varphi_{\text{fin}}}$, we introduce three new atomic propositions.

- *in_*A_ψ is true exactly when A_φ is in one of the states that originated from A_ψ;
- *at_*F_r is true exactly when A_φ is in a final state of A_r;
- *at_*F_ψ is true exactly when A_φ is in a final state of A_ψ.

The CTL formula $P_{\varphi_{\text{fin}}}$ is defined as follows:

$$P_{\varphi_{\text{fin}}} = AG((EX\ \textit{at\_}F_\varphi) \Rightarrow EX(\textit{in\_}A_\psi \wedge EF\ \textit{at\_}F_\psi)) \quad (1)$$

The prefix of the formula, EX *at_*F_φ, identifies those states s on the execution paths that have a one-step (EX) continuation to an accepting state of A_r. Recall that in the fusion automaton A_φ, we added transitions from one state before final in A_r to one state after initial in A_ψ. We thus require that s has a one-step continuation that 'lands' in a state of A_ψ. We further require that one of the branches that land in A_ψ have a continuation leading to the final state of A_ψ (denoted *at_*F_ψ), which is where ψ is accepted.

The above construction allows us to perform model-checking of CSSL formulas using standard CTL model-checking tools.

Correctness. To show correctness of the above construction, we need to show that $K \models \varphi \iff K \times A_\varphi \models P_{\varphi_{\text{fin}}}$.

Proof: Let $\varphi = r \mapsto \psi$ be a CSSL formula and let A_r and A_ψ be the automata for the prefix r and the (finite) suffix ψ, respectively. Let A_φ be the fusion automaton of A_r and A_ψ, and let P be as defined by (1).

Assume $K \times A_\varphi \not\models P$. We show that $K \not\models r \mapsto \psi$. From the assumption, let $(s_0, a_0) \cdots (s_n, a_n)$ be the counterexample to the AG part of P. Then, the path $\pi = s_0, \ldots, s_n$ satisfies r since A_r has an accepting run $a_0, \ldots, a_{n+1}$ on π. Note that the run has one more state than letters in the input. Assume that $s_n \not\models EX\mathit{in\_A_\psi}$. Then, by construction of the fusion automaton, the path $s_0, \ldots, s_n$ cannot be continued to satisfy the CSSL formula. Assume $s_n \models EX\mathit{in\_A_\psi}$ and $s_n \not\models EX(\mathit{in\_A_\psi} \wedge EF\mathit{at\_F_\psi})$. Then, there is no continuation of the path $s_0, \ldots, s_n$ that satisfies the suffix. This completes the proof.

For the other direction, assume $K \not\models r \mapsto \psi$. Then there exists a path $s_0, \ldots, s_n$ in K that satisfies r and cannot be continued to satisfy ψ (i.e., a counterexample to $r \mapsto \psi$). Thus, there exists an execution $(s_0, a_0), \ldots, (s_n, a_n), (s_{n+1}, a_{n+1})$ of $K \times A_\varphi$ such that a_{n+1} is an accepting state of A_r. Assume by contradiction that there exists a continuation $(s_n, b_1), \ldots, (s_{n+m}, b_m)$ such that b_m is an accepting state of A_ψ. Then the path $s_0, \ldots, s_n, \ldots, s_{n+m-1}$ satisfies $r \mapsto \psi$; hence, $s_0 \ldots, s_n$ is not a counterexample. This contradicts our assumption that $K \not\models r \mapsto \psi$. □

4.1.2 *Infinite Suffix*

Let $\varphi = p \mapsto \psi$, where ψ is infinite. A_ψ is now an NBA and so is the fusion automaton A_φ. That is, for a computation path to be accepted, a final state of A_φ should be visited infinitely often. We reduce the infinite case to *fair* CTL. The acceptance condition of A_φ is translated into a fairness constraint. Note that using *at_F*$_\psi$ as a fairness constraint is too strong: it would mean that only paths where *at_F*$_\psi$ appears infinitely often are checked; a bug, if it exists, would be masked. Instead, we introduce another atomic proposition, *in_A*$_r$, which is true exactly when A_φ is in one of the states that originated from A_r, and define *in_A*$_r \vee$ *at_F*$_\psi$ to be the fairness constraint.

The (fair) CTL formula for this case is given below:

$$P_{\varphi_{\text{inf}}} = AG((EX\ \mathit{at\_F_\varphi}) \Rightarrow EX(\mathit{in\_A_\psi} \wedge EG\ \top)) \quad (2)$$

As in the finite case, we first identify the states s on the execution paths that have a one-step (EX) continuation to an accepting state of A_r. We then require that each s has a one-step continuation that 'lands' in a state of A_ψ. At this point, we only require that one of the branches that continue to A_ψ has an infinite continuation. This is sufficient, since the fairness constraint ensures that infinite paths left in the model (that can no longer visit states from A_r) are those on which a final state of A_ψ is visited infinitely often. Thus, ψ holds on those paths.

Correctness. To show correctness of the above construction, we need to show that $K \models \varphi \iff K \times A_\varphi \models_{\text{fair}} P_{\varphi_{\text{inf}}}$.

Proof: Assume $K \times A_\varphi \not\models P\varphi_{\text{inf}}$. Then there exists a path $(s_0, a_0) \cdots (s_n, a_n)$ in $K \times A_\varphi$ (under the fairness constraint) such that $(s_n, a_n) \models (EX\ \mathit{at\_F_\varphi})$ but $(s_n, a_n) \not\models EX(\mathit{in\_A_\psi} \wedge EG\ \top)$. The path $\pi = s_0, \ldots, s_n$ in K satisfies r since A_r has an accepting run $a_0, \ldots, a_{n+1}$ on π. We have to show that s_n has no continuation satisfying ψ. Assume by contradiction that a continuation $\pi^n = s_n, s_{n+1}, \ldots$ such that $\pi^n \models \psi$ does exist in K. By the construction of the fusion automaton, there exists a path $(s_n, a_n), (s_{n+1}, a_{n+1}), \ldots$ in $K \times A_\varphi$ such that $a_n, a_{n+1}, \ldots$ is a run of A_ψ that accepts ψ. This means that $a_n, a_{n+1}, \ldots$ visits a final state of A_ψ infinitely often, and thus it is a *fair* computation. We have found a fair path, starting at (s_n, a_n) on which $EX(\mathit{in\_A_\psi} \wedge EG\ \top)$ holds, contradicting our assumption.

For the other direction, assume $K \not\models r \mapsto \psi$. Then there exists a path $s_0, \ldots, s_n$ in K that satisfies r and cannot be continued to satisfy ψ. Thus, there exists an execution path $\pi = (s_0, a_0), \ldots, (s_n, a_n), (s_{n+1}, a_{n+1})$ of $K \times A_\varphi$ such that a_{n+1} is an accepting state of A_r. Note that π can be continued with $(s_{n+2}, a_{n+2}), (s_{n+3}, a_{n+3}), \ldots$, where $a_{n+2}, a_{n+3}, \ldots$ are states of A_r (because of the totality of the transition relation). Note further that this path is *fair* (it stays in states from A_r forever). Thus, state (s_n, a_n) exists in a fair path of $K \times A_\varphi$. We know that $(s_n, a_n) \models (EX\ \mathit{at\_F_\varphi})$ (along π). We have to show that $(s_n, a_n) \not\models EX(\mathit{in\_A_\psi} \wedge EG\ \top)$. Assume by contradiction that it does. Then from (s_n, a_n) there is a continuation $\pi' = (s'_{n+1}, a'_{n+1}), (s'_{n+2}, a'_{n+2}), (s'_{n+3}, a'_{n+3}), \ldots$ that enters the automaton A_ψ, and is infinite. By the fairness constraint, π' must be *fair*, and since it is out of A_r, it must visit an accepting state of A_ψ infinitely often. Thus, $s_n, s'_{n+1}, , s'_{n+2}, s'_{n+3} \ldots$ is a path in K that satisfies ψ, which contradicts our assumption that such a path does not exist. □

4.2 Complexity

The complexity of CTL model checking is linear in the size of the model and the formula [8]. Model-checking complexity of a CSSL formula $\varphi = r \mapsto \psi$ on a Kripke structure K is $O(|K| \times |A_\varphi| \times |P_\varphi|)$. Note that the size of P_φ is constant and does not depend of φ. The size of A_φ is $O(|A_r| + |A_\psi|)$, where $|A_r| = O(|r|)$. The size of A_ψ depends on ψ. In the general case, we need to build a Büchi automaton for the LTL formula ψ, which is of size $2^{|\psi|}$. The complexity of model checking of CSSL is thus $O(|K| \times |r| \times 2^{|\psi|})$, which is the model-checking complexity of the original PSL.

Yet, for some PSL formulas, the size of the automaton A_ψ can be of size $|\psi|$. For such cases, model-checking complexity of CSSL is $O(|K| \times |r| \times |\psi|)$, i.e., if model-checking of ψ is efficient, then so is model-checking of the resulting CSSL (and PSL) formula.

Thus, we are able to enrich PSL with a branching operator without incurring an additional model-checking cost.

4.3 Implementation and Illustration

We have implemented the translation of a CSSL formula $\varphi = r \mapsto \psi$ on top of the model checker NuSMV.

Let SM_r be the state machine for the prefix r and SM_ψ – the state machine for the suffix. We illustrate the construction using two properties specified in Table 1: P_5, for the finite suffix case, and P_7 for the infinite suffix.

We start with property P_5, where

$$\begin{aligned} r &= \mathit{true}^* \cdot \text{pwd} \cdot \text{verify} \cdot \text{ok} \\ \psi &= (\neg\text{changePin}\ U\ (\text{reqCash} \wedge X(\text{cash}))) \end{aligned}$$

The code for SM_r in the language of NuSMV, is shown in Figure 4, lines 3-11. It describes a simple non-deterministic state machine, with its initial state set by the `init` statement, and transitions determined by a `case` statement. `case`*s* are evaluated top to bottom. When the first condition becomes *true*, its corresponding assignment is executed, determining the `next` state. The language allows specification of non-deterministic transitions, captured in a set. These mean that the machine may non-deterministically

```
1: VAR SMr : {1,2,3,4,accept,sink};
2: ASSIGN
3:   init(SMr) := {1,2};
4:   next(SMr) :=
5:    case
6:      SMr = 1              : {1,2};
7:      SMr = 2 & pwd        : 3;
8:      SMr = 3 & verify     : 4;
9:      SMr = 4 & ok         : accept;
10:     TRUE                 : sink;
11:   esac;
12:
13: DEFINE SMrCanAccept := (SMr = 4 & ok);
14:
15: VAR SMpsi : {wait,1,2,3,accept,sink};
16: ASSIGN
17:    init(SMpsi) := wait;
18:    next(SMpsi) :=
19:    case
20:      SMpsi = wait & !SMrCanAccept : wait;
21:      SMpsi = wait & reqCash       :{wait,3};
22:      SMpsi = wait & !changePin    :{wait,1,2};
23:      SMpsi = 1 & !changePin       :{1,2};
24:      SMpsi = 2 &  reqCash         : 3;
25:      SMpsi = 3 &  cash            : accept;
26:      TRUE                         : sink;
27:    esac;
28:
29: DEFINE In_SMpsi := !(SMpsi in {wait,sink});
30:
31: SPEC AG(EX(SMr=accept) ->
32:        EX (In_SMpsi & EF (SMpsi = accept)))
```

Figure 4: NuSMV code for checking property P_5.

choose an element of the set as its next-state value and its initial value.

The main challenge in the implementation is the *fusion* of SM_r and SM_ψ, since SM_ψ starts working in the initial state of the system while we need it to start operating only when SM_r has reached an accepting state. To solve this problem, we add a `wait` state to SM_ψ, in which it starts and stays until SM_r has (almost) reached an accepting state. At that point, SM_ψ is triggered and starts its intended work.

On line 13, we `DEFINE SMrCanAccept` as a "going to accept" signal, used to trigger the suffix automaton SM_ψ. We then implement SM_ψ in lines 15-27. While SM_ψ can leave the `wait` state in lines 21 and 22, we allow it to stay in `wait` as one of the nondeterministic options. This is done to accommodate cases where SM_r reaches the accepting state more than once on a given path, and thus SM_ψ should be triggered more than once as well.

Using SM_r and SM_ψ, and the intermediate definition of the signal `In_SMpsi`, we specify the property to be checked as shown on lines 31-32 in Figure 4. This model is verifiable by NuSMV.

We now consider the property P_7 in Table 1, where

$$\begin{aligned} r &= \mathit{true}^* \cdot \text{pwd} \cdot \text{verify} \cdot \text{ok} \\ \psi &= (G\ \neg\text{logout}) \wedge (GF(\text{reqCash} \vee \ldots \vee \text{changePin})) \end{aligned}$$

Here, the suffix is infinite. In order to encode an NBA automaton to represent it, we need to use NuSMV's fairness mechanism.

The state machine for the prefix r is the same as in Figure 4. The state machine for the suffix ψ is given in Figure 5, lines 1-13. We then make additional definition in line 15 and add a fairness constraint (line 17). This ensures that no legal path is omitted for the wrong reason. We can now define the specification as shown in lines 19-20, and the resulting code is again verifiable by NuSMV.

```
1: DEFINE Operation: (reqCash | ... | changePin);
2: VAR SMpsiInf : {wait,1,2,accept,sink};
3: ASSIGN
4:    init(SMpsiInf) := wait;
5:    next(SMpsiInf) :=
6:    case
7:      SMpsiInf = wait & !SMrCanAccept     : wait;
8:      SMpsiInf = wait &  SMrCanAccept     :{wait,1};
9:      SMpsiInf = 1 & !logout              :{1,2};
10:     SMpsiInf = 2 & Operation & !logout  :accept;
11:     SMpsiInf = accept & !logout         : 1;
12:     TRUE                                :sink;
13:   esac;
14:
15: DEFINE In_SMpsiInf := !(SMpsiInf in {wait,sink});
16:
17: FAIRNESS (SMpsiInf in {wait,accept})
18:
19: SPEC AG (EX(SMr=accept)
20:          -> EX (In_SMpsiInf & EG True))
```

Figure 5: NuSMV code for checking property P_7.

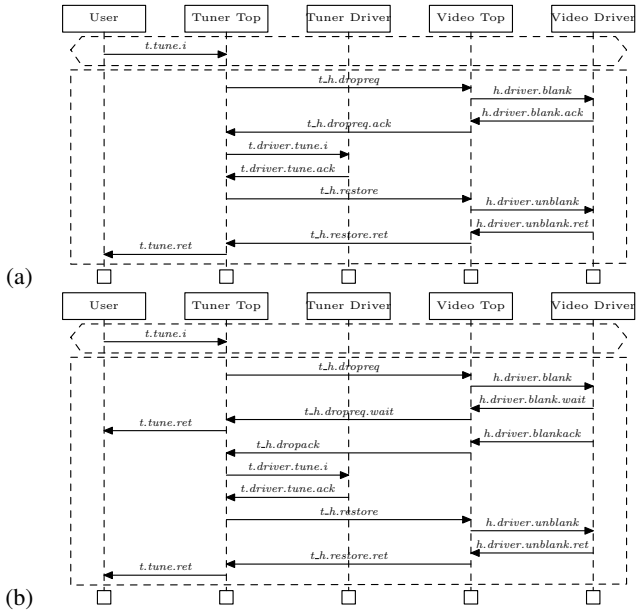

Figure 6: Two scenarios of the TV system.

5. CASE STUDIES

In this section, we present our experience specifying and verifying CSSL properties for two realistic examples. These examples and the underlying tool support have been successfully evaluated by the ESEC/FSE artifact evaluation committee.

5.1 Philips TV Controller Protocol

We describe the application of CSSL to the specification and verification of an industrial protocol for a product family of Philips television sets [34]. The product family supports sets with multiple tuners and video output devices that can be configured by the television user to display several signals in different configurations. The protocol is concerned with controlling the signal path in a TV to avoid visual artefacts. To do so, it coordinates between four main component types: video and tuner tops which perform horizontal communication to route the signal path, and video and tuner drivers that perform vertical communication between their corresponding top and physical device.

The key property that the protocol must ensure is that the signal

Name	Expression	Result
TV_1	G(t_h.restore.ret $\Rightarrow$ ($\bigwedge_{i \in Freq} \neg$t.driver.tune.i U (t_h.dropreq.ack $\vee$ t_h.dropack)))	Yes
TV_2	*true** · t.tune.i $\mapsto$ (t_h.dropreq · h.driver.blank · h.driver.blank.ack · t_h.dropreq.ack · t.driver.tune.i · t.driver.tune.ack · t_h.restore · h.driver.unblank · h.driver.unblank.ret · t_h.restore.ret · t.tune.ret)	No
TV_3	*true** · t.tune.i $\mapsto$ (t_h.dropreq · h.driver.blank · h.driver.blank.wait · t_h.dropreq.wait · (t.tune.ret · h.driver.blankack | h.driver.blankack · t.tune.ret) · t_h.dropack · t.driver.tune.i · t.driver.tune.ack · t_h.restore · h.driver.unblank · h.driver.unblank.ret · t_h.restore.ret · t_h.dropack.ret · h.driver.blankack.ret)	No
TV_4	*true** · t.tune.i $\mapsto$ (t_h.dropreq · h.driver.blank · h.driver.blank.ack · t_h.dropreq.ack · t.driver.tune.i · t.driver.tune.wait · (t.tune.ret · t.driver.tuneack | t.driver.tuneack · t.tune.ret) · t_h.restore · h.driver.unblank · h.driver.unblank.ret · t_h.restore.ret · t.driver.tuneack.ret)	No
TV_2'	*See Section 5.1*	Yes
TV_3'	*See Section 5.1*	Yes
TV_4'	*See Section 5.1*	Yes
TV_5	(*true** · t_h.dropreq.wait · $\neg$(t_h.dropack $\vee$ t.tune.ret $\vee$ t.tune.i)*) $\mapsto$ (t.tune.ret · t.tune.i)	Yes
TV_6	(*true** · t.driver.tune.wait · $\neg$(t.driver.tuneack $\vee$ t.tune.ret $\vee$ t.tune.i)*) $\mapsto$ ($\neg$t.driver.tuneack U t.tune.i)	Yes
TV_7	G(t_h.dropreq.ack $\Rightarrow$ X(t_h.dropack $\Rightarrow$ ($\neg$t_h.dropreq U t_h.restore.ret)))	Yes
TV_8	(*true** · t_h.dropreq.wait · $\neg$(t_h.dropack $\vee$ t.tune.ret $\vee$ t.tune.i)* · t.tune.i) $\mapsto$ (*true* · t.tune.ret · t.driver.tune.i · t.driver.tuneack · t_h.restore · t_h.restore.ret · t_h.dropack.ret)	Yes
TV_9	(*true** · t.tune.i) $\mapsto$ (($\neg$t.tune.j $\wedge$ $\neg$t.driver.tune.j)* · t.driver.tune.i)	Yes

Table 2: Properties of the TV Controller Protocol.

frequency on the wire connected to a video device is only changed by the tuner driver when the video device has been blanked. We formalize it as a CSSL property TV_1 later in this section. First, we look at some simple scenarios involving this protocol.

The simplest scenario, which consists of a TV set with one tuner and one video device, is when the user changes the channel on which the tuner top receives a `t.tune.i` request, where i is the frequency of the channel to be displayed. Then the tuner top requests the video top to blank the screen (`t_h.dropreq`). The video top requests the video driver to blank the video device (`h.driver.blank`), receives an acknowledgment (`h.driver.blank.ack`) and then acknowledges the tuner top's request (`t_h.dropreq.ack`). The tuner top then requests the tuner driver to change its signal to frequency i (`t.driver.tune.i`) and once it receives the acknowledgment (`t.driver.tune.ack`), requests the video top, which, in turn, requests the video driver to unblank the screen and display the image being sent over the wire (`t_h.restore`, `h.driver.unblank`, `h.driver.unblank.ret`, `t_h.restore.ret`). This behaviour is specified as an existential triggered scenario shown in Figure 6(a) and formalized in CSSL as property TV_2. This and other properties of the TV system are shown in Table 2.

Part of the complexity of the protocol is that requests from tops to drivers may not be acknowledged immediately. For instance, the video driver may respond to a blanking request from a video top with a `h.driver.blank.wait` event rather than with `h.driver.blank.ack`. In this case, the protocol returns and waits for an upcall, `h.driver.blankack` from the video driver to the video top, informing it that the video has finally been blanked. The upcall is sent to the tuner top (`t_h.dropack`) which proceeds as in the previous scenario to change the wire frequency and then acknowledges the upcall (`t_h.dropack.ret`) to the video top which, in turn, acknowledges it to the video driver (`h.driver.blankack.ret`). This behavior is specified in Figure 6(b) and formalized as a CSSL property TV_3. Events `t.tune.ret` and `h.driver.blankack` can occur in any order.

The tuner driver may not acknowledge straight away, responding with `t.driver.tune.wait` rather than `t.driver.tune.ack` to a request of changing the signal's frequency. In this case, the protocol returns and awaits an upcall from the tuner driver (`t.driver.tuneack`). When the upcall occurs, the video is unblanked and the upcall acknowledged (`t.driver.tuneack.ret`). Events `t.tune.ret` and `t.driver.tuneack` can occur in this scenario in any order, as captured in property TV_4 (see Table 2).

A key aspect of this protocol, which can be traced to a responsiveness requirement, is that the protocol must process user tuning requests while the TV set is in intermediate states (i.e., those where it is waiting for an upcall from the tuner device or video device). Such properties can be expressed as existential triggered scenarios and formalized in CSSL as TV_5 and TV_6, respectively.

Another important aspect of the protocol, which can be traced to efficiency constraints, is that it must not request blanking the screen if it is already blanked. This is a safety property, formalized in CSSL as TV_7. TV_7 together with TV_5 and TV_6 are inconsistent with the first three existential scenarios, $TV_2 - TV_4$. The latter require that if users request tuning the TV set, then the protocol must be able to do so in three different ways (depending on whether devices acknowledge or upcall). Each of these scenarios requests that it be possible to blank the video device. TV_5 requires user tune requests to be processed while the video screen is blanked; hence, in conjunction with any of $TV_2 - TV_4$, the protocol must able to process the user's request by (among other things) blanking the screen. However, this would violate property TV_7.

Scenarios $TV_2 - TV_4$ are too strong, since the trigger part of the scenario (or the prefix of the CSSL formula) does not account for the possible states in which the TV set may be in when receiving a `t.tune.i` command. They require the behaviour (the main chart of the scenario or the suffix of the CSSL formula) that should only be required when the protocol is idle (i.e., when the frequency being displayed on the video device is that of the last `t.tune` command). This can be formalized by changing the prefix in $TV_2 - TV_4$ to be

true* · t.tune.i · ($\bigwedge_{j \in Freq} \neg$t.tune.j)* · t.driver.tune.i · ($\bigwedge_{j \in Freq} \neg$t.tune.j)* · (t.driver.tune.ack $\vee$ t.driver.tuneack) · ($\bigwedge_{j \in Freq} \neg$t.tune.j)* · t.tune.i

resulting in new properties, $TV_2' - TV_4'$, respectively.

There are many properties that are relevant to the protocol in this case study. Due to space restrictions, we mention two final conditional existential statements related to the responsiveness of requests to tune the TV set. The first one, called TV_8, strengthens TV_6 by exemplifying one way in which `t.tune` must be processed when it is received while waiting for the screen to be blanked. The other, called TV_9, states that whenever the tuner is requested to tune on frequency i (`t.tune.i`), then it must be possible for the next frequency change on the wire to be to frequency i.

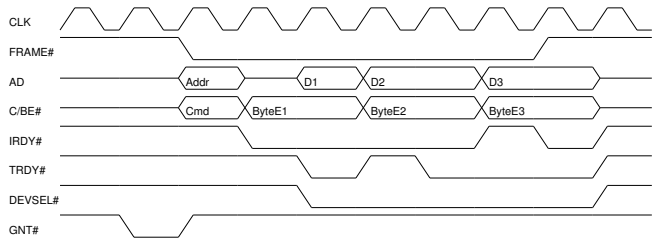

Figure 7: An example of a PCI transaction.

Finally, we now formalize the main safety property TV_1, discussed in the beginning of this section. `t_h.dropreq.ack` and `t_h.dropack` are sent from the video top to the tuner top as are the events that indicate that the video device has been blanked, and `t_h.restore.ret` (sent from the video top to the tuner top) is the only event that indicates that the video device has been unblanked.

In order to verify properties $TV_1 - TV_9$ and $TV_2' - TV_4'$, we took a sample model of the TV-tuner, expressed in FSP [21] and originally written by Magee and van Ommering [34] (the latter is one of the original protocol developers). Using a simple LTSA-to-SMV converter, we converted it into the format of NuSMV. Then each of the properties has been encoded in CTL using the process described in Section 4. Table 2 shows the result of verifying all of the properties formalized above against the NuSMV model of the protocol. The table shows that properties $TV_2 - TV_4$ fail, whereas others hold. All properties took under a second to verify, so we do not list the exact running times. While the protocol model is relatively complex, it is quite small by NuSMV's standards, and thus the analysis took a very short time.

We chose to start our presentation with properties that do not hold in the model. While it is commonplace to present, as use cases or scenarios, examples of behaviours that a system is expected to exhibit, often the preconditions or triggers of such scenarios are too weak. When this happens, the suffix behaviour is exhibited in cases when, in fact, it should not. Conditions for existential statements can be complicated to get right, as we have experienced during this case study, and we certainly welcomed having mechanized model-checking support to help us debug our specs. In fact, they underwent several iterations before they took their current, correct, form. Furthermore, our current model-checking approach converts the CSSL property to (fair) CTL, which means that counterexamples that NuSMV produced were very incomplete (see a discussion in Section 7).

5.2 PCI

Our second case study demonstrates the usefulness of the branching-suffix implication operator and thus the resulting CSSL logic in a different setting, that of hardware verification.

The *Peripheral Component Interconnect* (PCI) [27] is a local bus protocol commonly used for the communication between hardware devices (such as network cards, modems and USB ports) inside a computer. While multiple devices may be connected to a single PCI bus, we describe here a typical transaction between one initiator (or "master") and one target device. All signals of the PCI protocol are 'active low', that is, are asserted when dropped from 1 to 0.

A transaction on a PCI bus starts with the initiator asserting its FRAME#, and at the same time driving the address of the target device on the address/data (A/D) bus, as well as the command (the type of transaction) on the C/BE bus. The cycle in which FRAME# is asserted is called the *address-phase*. The selected device, once recognizing its address, asserts the DEVSEL# signal. In a write transaction, the initiator, when ready, drives the data to be transferred on the A/D bus and asserts its IRDY# signal. This is the beginning of a *data-phase*. When the target is ready to read the data, it asserts its TRDY# signal; until then, the IRDY# must remain active. Whenever both TRDY# and IRDY# are asserted together, the data held in the A/D bus is assumed to be transferred, and this is the end of a data-phase. In a read transaction, the target device is the one responsible to drive the data on the A/D bus, asserting TRDY# when the data is ready for transfer. A transaction may have multiple data-phases, and each one may last one cycle (if IRDY# and TRDY# are active together in consecutive cycles) or more if one or more of IRDY#, TRDY# are inactive for some cycles. The last data-phase is indicated by the initiator with FRAME# de-asserting when IRDY# is being asserted. One cycle after the last data-phase ends, IRDY#, DEVSEL# and TRDY# must all be de-asserted. Figure 7 demonstrates a simple PCI transaction with 3 data phases. Another target-driven signal (which does not appear in Figure 7) is the STOP# signal. It is used by the target to indicate that it cannot complete the transaction. If STOP# is asserted, with or without the assertion of TRDY#, it means that the target requests the initiator to end the current transaction and "retry" it later.

In order to verify a PCI target device, we model the behavior of the *initiator device*, mainly that of FRAME#, IRDY# and the C/BE bus, that control the command of the transactions. For simplicity, we model only bit 0 (C/BE[0]) of this bus, which determines whether the transaction is a read (0) or a write (1). For the verification to be thorough, our model of the initiator allows all legal behaviors to occur.

We concentrate on checking the dependency between the assertion of TRDY# and IRDY#. The PCI specification indicates that each of the initiator and target is free to assert its corresponding RDY# signal when ready. In read transactions, however, the initiator is allowed to wait until the target asserts TRDY# before asserting its IRDY# (so it can have more than one cycle to process the data). If both the initiator and the target condition the assertion of their RDY# signals on the assertion of the other's, this results in a deadlock. The above case was a real problem found in verification of the PCI component performed by an industrial partner.

Such a problem can be identified by checking the following CSSL property: "If when FRAME# is asserted (address-phase) C/BE[0]=0 (it is a read transaction), then at the beginning of every data-phase that is not retried, there exists at least one continuation where TRDY# is asserted before IRDY#." It is formalized as follows (F stands for FRAME#, D for DEVSEL#, S for STOP#, I and T for IRDY and TRDY, respectively):

$$(\mathit{true}^* \cdot \mathsf{F} \cdot (\neg\mathsf{F} \wedge (\mathsf{C/BE[0]{=}0})) \cdot (\neg\mathsf{F})^* \cdot (\neg\mathsf{F} \wedge \neg\mathsf{D} \wedge \mathsf{S} \wedge ((\mathsf{T} \wedge \mathsf{I}) \vee (\neg\mathsf{T} \wedge \neg\mathsf{I}))) \mapsto X((\mathsf{I} \wedge \mathsf{S})\ U\ \neg\mathsf{T}).$$

Such a property failed when checked on the original model, exposing an error identified after the built PCI bus was used in the field.

It is common knowledge that verification engineers naturally think about universal properties, which is one of the reasons why industrial standards are based on linear temporal logics. This case study demonstrates that universal properties are not sufficient, since there are bugs that require some branching to be detected. CSSL adds just the right amount of branching needed.

6. RELATED WORK

Scenarios. Our work is concerned with reasoning about the system behaviour in the presence of universal and existential statements, and, in particular, of conditional existential statements which are commonplace in scenario and use-case specifications. Hence, our work is related to a large body of work in the area of

scenario-based specifications which were originally conceived as existential, i.e., example based, specifications.

The ITU Message Sequence Chart standard [16] and, subsequently, the UML Sequence Diagrams are widespread notations for documenting scenarios which in their original formulations do not provide mechanisms for distinguishing between existential and universal statements and, in practice, have been interpreted existentially. Many variants of these languages have been proposed. The first to support multiple modalities is LSC [9]. However, LSCs do not support conditional existential scenarios. Although the original formulation of existential LSCs included a prechart (a prefix) P and a main chart (a suffix) M, their semantics is equivalent to a non-conditional existential scenario that concatenates both charts (i.e., $P \cdot M$). Later versions of LSCs, e.g., [3], drop the existential prechart. Indeed, formalization of LSC semantics can be given in linear temporal logic. CSSL is capable of supporting a more expressive language than LSCs, one that includes proper conditional existential scenarios.

Sibay et al. [28] present a language with conditional existential and conditional universal scenarios. The language semantics is provided in terms of CTL, and verification is done through construction of Modal Transition Systems (MTS) [19], model merging, and MTS weak alphabet refinement [5]. The language is restricted to two temporal patterns: a triggered existential statement of the form $AG(\mathit{trigger} \Rightarrow \bigwedge_w EXw)$, where w ranges over the language of the main chart, and a triggered universal statement of the form $AG(\mathit{trigger} \Rightarrow \bigwedge_w EXw \wedge AX \bigvee_w w)$, where w ranges of the language of the main chart. The languages of the trigger and main chart are defined by a finite partial orderings of events. CSSL can express properties not expressible by the scenario language of [28].

Arguably the most widespread form of conditional existential statements are use cases [17], when they are used with preconditions, as advocated in popular software engineering practitioners' literature [25]. The importance of documenting and reasoning about such kinds of statements can be inferred from popular development methods such as RIP [18], and has been argued for from multiple perspectives such as supporting "what-if" elaboration of requirements specifications [22] and the progressive shift from existential statements, in the form of examples and use-cases, to universal statements in the form of declarative properties [12]. This motivates the need for a logic with the expressiveness of CSSL.

Branching properties. It is often assumed that linear properties are easy to express and, thus, linear logics are widely used to specify requirements [30, 33]. On the other hand, branching is seen as confusing and often avoided, and, thus, there are very few approaches which use branching logics. Yet some forms of branching are essential. For example, an industrial case study [24] showed that properties that are natural in the early design phases (especially when some information is uncertain) are branching. The authors of the study informally proposed the same branching pattern that underlies CSSL. We are unaware of other approaches to extend existing logics with "limited" branching.

Logic and model-checking. The early 2000's saw an activity, led by the IEEE Accellera committee, to define an industrial temporal logic language. In addition to PSL [14], it yielded the Standard for SystemVerilog Assertions (SVA) [15]. Both PSL and SVA are linear languages with regular expressions. They are supported by major CAD vendors (see [4]). The use of regular expressions improves both the expressive power and usability of these languages.

However, neither of these languages can express widespread branching properties like existential conditional statements. While PSL's standard defines an "optional branching extension", it is essentially CTL, which does not allow the use of linear temporal operators or regular expressions.

CTL* is a very expressive language which allows combination of linear and branching temporal operators (but not regular expressions) and is more than adequate for capturing scenarios. However, not only is specifying properties in CTL* highly non-intuitive, we are not aware of any model-checkers for it.

The model-checking method introduced in this paper combines auxiliary automata with temporal logic specifications, where the propositions used in the specification are derived from the automata states. This was inspired by the work described in [2], where automata are embedded into regular expression-based specifications. In [2], this approach was used to gain reusability as well as to reduce the overall size of the automata built. In this paper, it gives us a clean way to implement a fusion automaton, and thus to add branching capabilities to an otherwise linear language.

7. CONCLUSION AND FUTURE WORK

Software engineers use a variety of languages to specify the intended behaviour of a software system. These include scenarios, use cases, declarative global properties, as well as local operational properties such as preconditions for actions. This heterogeneity is due to the different contexts in which such languages are used (initial elicitation and elaboration stages, communication with stakeholders, final complete specification, communication with designers, etc.); however, it poses technical challenges when such specifications need to be analyzed automatically. In particular, the use of conditional existential scenarios (or use cases) require reasoning in branching time while grounded approaches to behaviour specification and verification shy away from branching-time logics due to their relatively unintuitive semantics.

In this paper, we introduced an extension to PSL which gives "just enough branching" to be able to capture existential conditional scenarios without an increase in the model-checking complexity. The resulting language, called CSSL, can be used for formalizing and reasoning about heterogeneous specification that include use-cases, variants of sequence chart notations and general (linear) safety and liveness properties. We showed how to do model-checking of CSSL properties using NuSMV and illustrated the usefulness of CSSL on two real-life examples.

Like all branching properties, CSSL has a fundamental problem with counterexamples produced when a property fails. Since a CSSL formula $r \mapsto \psi$ consists of a universal and an existential part, when it fails, the counterexample is provided only for the universal part, demonstrating a path on which r holds. To demonstrate failure of ψ, we would have to produce the entire set of continuations of this path. This makes debugging difficult, especially if ψ is a long formula. In the future, we are interested in creating an automated support for debugging failed CSSL properties.

Our work on formal reasoning of scenario languages is not limited to verification, when the model of the system is already given. Scenarios are often used as a specification language for software, as a prototyping mechanism, and as a way to synthesize an operational model [32]. In this context, efficient algorithms for *satisfiability*, *synthesis* and *vacuity detection* are a must [5]. Satisfiability allows us to determine whether a set of CSSL properties is consistent, i.e., whether it is possible to build a model in which all of these properties hold. Vacuity [11] enables us to check whether a formula holds "for the wrong reason", i.e., vacuously. This can happen when a universal prefix of conditional scenarios is false, and thus the property is always true. It would also be interesting not only to tell that a property is vacuous but also to determine the cause of this vacuity [11]. Finally, given a (consistent) set of CSSL

properties, it would be very desirable to synthesize a preliminary operational model which is guaranteed to satisfy them.

Synthesis, satisfiability and vacuity checking algorithms exist for CTL*, so we have a clear upper bound on the complexity of the corresponding CSSL procedures. However, we are yet to determine whether algorithms with better lower bounds for CSSL exist.

Finally, PSL allows us to specify safety properties of the system-to-be, thus providing an *over-approximation* of its behaviour. Our new operator, BSI, allows us to express existential and conditional existential scenarios, describing *under-approximations* of the behaviour of the system-to-be. We are interested in studying whether CSSL can effectively replace μ-calculus as the language characterizing properties of partial behavioural models, synthesized from properties and scenarios [6, 32].

Acknowledgements. We thank the anonymous ESEC/FSE referees for their suggestions for improving the paper and Winnie Lam for her help with the tool support. The work was partially funded by ERC PBM-FIMBSE, CONICET, UBACyT X021, PIP112-200801-00955KA4, PICT-PAE 37279, NSERC and Ontario Postgraduate Fellowship.

8. REFERENCES

[1] I. F. Alexander and N. Maiden, editors. *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*. Wiley, 2004.

[2] S. Ben-David, D. Fisman, and S. Ruah. "Embedding Finite Automata within Regular Expressions". *Theor. Comput. Sci.*, 404(3):202–218, 2008.

[3] Y. Bontemps, P. Heymans, and P.-Y. Schobbens. "From Live Sequence Charts to State Machines and Back: A Guided Tour". *IEEE TSE*, 31(12):999–1014, 2005.

[4] D. Borionne, M. Liu, P. Ostier, and L. Fesquet. "PSL-based Online Monitoring of Digital Systems". In *Proc. of FDL'05*, pages 465–479, 2005.

[5] G. Brunet, M. Chechik, D. Fischbein, N. D'Ippolito, and S. Uchitel. "Weak Alphabet Merging of Partial Behaviour Models". *ACM TOSEM*, 2011. To appear.

[6] M. Chechik, A. Gurfinkel, S. Uchitel, and S. Ben-David. "Raising Level of Abstraction with Partial Models: A Vision". In *Proc. of NSF/MSR Wrksp. on Usable Verification*, 2010.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and O. Tacchella. "NuSMV 2: An Opensource Tool for Symbolic Model Checking". In *Proceedings of CAV'02*, pages 359–364. Springer, 2002.

[8] E.M. Clarke and E.A. Emerson. "Design and synthesis of synchronization skeletons using Branching Time Temporal Logic". In *Proc. Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[9] W. Damm and D. Harel. "LSCs: Breathing Life into Message Sequence Charts". *FMSD*, 19:45–80, 2001.

[10] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[11] A. Gurfinkel and M. Chechik. "How Vacuous Is Vacuous?". In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 451–466, March 2004.

[12] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., 2003.

[13] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[14] IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850™-2010.

[15] IEEE Standard for SystemVerilog, Annex F. IEEE Std 1800™-2009.

[16] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1992.

[17] I. Jacobson. *Object-Oriented Software Engineering: a Use Case driven Approach*. Addison–Wesley, 1995.

[18] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Development Process*. Addison-Wesley, 1999.

[19] K. G. Larsen and B. Thomsen. "A Modal Process Logic". In *Proc. of LICS*, pages 203–210, July 1988.

[20] S. Leue and T. Systä, editors. *Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*. Springer, 2005.

[21] J. Magee and J. Kramer. *"Concurrency - State Models and Java Programs"*. John Wiley, 1999.

[22] N. Maiden and S. Robertson. "Developing Use Cases and Scenarios in the Requirements Process". In *Proc. of ICSE'05*, pages 561–570, 2005.

[23] A. Pnueli. "The Temporal Logic of Programs". In *Proc. of IEEE Symp. on Foundation of Comp. Sci.*, pages 46–57, 1977.

[24] A. Post, I. Menzel, and A. Podelski. "Applying Restricted English Grammar on Automotive Requirements – Does it Work? A Case Study". In *Proc. of REFSQ'11*, volume 6606 of *LNCS*, pages 166–180. Springer, 2011.

[25] R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 7th edition, 2010.

[26] K. Schwaber. *Agile Project Management With Scrum*. Microsoft Press, 2004.

[27] T. Shanley and D. Anderson. *PCI System Architecture*. Mindshare, Inc., 4th edition, 1999.

[28] G. Sibay, V. Braberman, S. Uchitel, and J. Kramer. "Synthesizing Modal Transition Systems from Triggered Scenarios", March 2010. submitted for journal publication.

[29] G. Sibay, S. Uchitel, and V. Braberman. "Existential Live Sequence Charts Revisited". In *Proceedings of ICSE'08*, pages 41–50, 2008.

[30] R. Smith, G. Avrunin, L. Clarke, and L. Osterweil. "PROPEL: An Approach Supporting Property Elucidation". In *Proceedings of ICSE'02*, pages 11–21, 2002.

[31] A. G. Sutcliffe, N.A.M. Maiden, S. Minocha, and D. Manuel. "Supporting Scenario-Based Requirements Engineering". *IEEE TSE*, 24:1072–1088, 1998.

[32] S. Uchitel, G. Brunet, and M. Chechik. "Behavioural Model Synthesis from Properties and Scenarios". *IEEE TSE*, 3(35):384–406, 2009.

[33] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[34] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. "The KOALA Component Model for Consumer Electronics Software". *IEEE Computer*, 33(3):78–85, 2000.

[35] M.Y. Vardi. "Branching vs. Linear Time: Final Showdown". In *Proc. of TACAS'01*, pages 1–22, 2001.

[36] M.Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)". In *Proc. of LICS'86*, pages 332–344, 1986.