

# Temporal Antecedent Failure: Refining Vacuity

Shoham Ben-David<sup>1</sup> Dana Fisman<sup>2,3</sup> Sitvanit Ruah<sup>3</sup>

<sup>1</sup> David R. Cheriton School of Computer Science University of Waterloo

<sup>2</sup> School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel.

<sup>3</sup> IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel.

**Abstract.** We re-examine vacuity in temporal logic model checking. We note two disturbing phenomena in recent results in this area. The first indicates that not all vacuities detected in practical applications are considered a problem by the system verifier. The second shows that vacuity detection for certain logics can be very complex and time consuming. This brings vacuity detection into an undesirable situation where the user of the model checking tool may find herself waiting a long time for results that are of no interest for her.

In this paper we define *Temporal Antecedent Failure*, an extension of antecedent failure to temporal logic, which refines the notion of vacuity. According to our experience, this type of vacuity *always* indicates a problem in the model, environment or formula. On top, detection of this vacuity is extremely easy to achieve. We base our definition and algorithm on regular expressions, that have become the major temporal logic specification in practical applications.

Keywords: Model checking, Temporal logic, Vacuity, Regular expressions, PSL, SVA, Antecedent failure

## 1 Introduction

Model checking ([15, 28], c.f.[16]) is the procedure of deciding whether a given model satisfies a given formula. One of the nice features of model checking is the ability to produce, when the formula does not hold in the model, an execution path demonstrating the failure. However, when the formula is found to hold in the model, traditional model checking tools provide no further information. While satisfaction of the formula in the model should usually be considered “good news”, it is many times the case that the positive answer was caused by some error in the formula, model or environment. As a simple example consider the LTL formula  $\varphi = \mathbf{G}(req \rightarrow \mathbf{F}(ack))$ , stating that every request *req* must eventually be acknowledged by *ack*. This formula holds in a model in which *req* is never active. In this case we say that the formula is *vacuously* satisfied in the model.

Vacuity in temporal logic model checking was introduced by Beer et al ([3, 4]), who defined it as follows: a formula  $\varphi$  is vacuously satisfied in a model  $M$  iff it is satisfied in  $M$ , (i.e.,  $M \models \varphi$ ) and there exists a subformula  $\psi$  of  $\varphi$  that *does not affect* the truth value of  $\varphi$  in  $M$ . That is, there exists a subformula  $\psi$  of  $\varphi$  such that  $M \models \varphi[\psi \leftarrow \psi']$

for any formula  $\psi'$ .<sup>1</sup> For example, in the formula  $\varphi = \mathbf{G}(req \rightarrow \mathbf{F}(ack))$ , if  $req$  is never active in  $M$ , then the formula holds, no matter what the value of  $\mathbf{F}(ack)$  is. Thus, the subformula  $\mathbf{F}(ack)$  does not affect the truth value of  $\varphi$  in  $M$ . Note that this definition ignores the question of the *reason* for a vacuous satisfaction. In our example the reason for vacuity is the failure of  $req$  to ever hold in the model.

Since the introduction of vacuity, research in this area concentrated mainly on extending the languages and methods to which vacuity could be applied [24, 27, 17, 26, 20, 30]. Armoni et al. in [1] defined a new type of vacuity, which they called *trace vacuity*. Chechik and Gurfinkel in [19] showed that detecting trace vacuity is exponential for CTL, and double-exponential for CTL\*. Bustan et al. in [13] showed that detecting trace vacuity for RELTL (an extension of LTL with a regular layer), involves an exponential blow-up in addition to the standard exponential blow-up for LTL model checking. They suggested that weaker vacuity definition should be adopted for practical applications.

In a recent paper [14] Chechik et al. present an interesting observation. They note that in many cases, vacuities detected according to existing definitions are not considered a problem by the verifier of the system. For example, consider again the formula  $\varphi = \mathbf{G}(req \rightarrow \mathbf{F}(ack))$ , and assume the system is designed in such a way that  $ack$  is given whenever the system is ready. Thus, if the system behaves correctly, it infinitely often gets to a “ready” state, and the formula  $\mathbf{F}(ack)$  always holds. According to the definition of vacuity,  $\varphi$  holds vacuously in the model (since  $req$  does not affect the truth value of  $\varphi$  in the model), although no real problem exists. This phenomenon, coupled with the high complexity results reported for some logics, brings vacuity detection into an undesirable situation where the user of the model checking tool may find herself waiting a long time for results that are of no interest for her.

In this paper we approach the problem by providing a refined definition of vacuity, such that detection is almost “for free”, and, according to feedback from users, vacuity *always* indicate real problems. We define *temporal antecedent failure* (TAF), a type of vacuity that occurs when some *pre-condition* in the formula is never fulfilled in the model. The definition of TAF is semantic and is therefore unlikely to suffer from typical problems with syntactic definitions as shown in Armoni et al. [1].

We work with formulas given in terms of regular expressions, that are the major specification method used in practice [8, 18, 23, 22]. We show how vacuity can be detected by asserting an *invariant* condition on the automaton already built for the original formula. We further use these invariant conditions to determine the *reason* for vacuity, when detected. Vacuity and its reasons are therefore detected with almost no overhead on the state space.

Note that the notion of *vacuity reasons*, formally defined in section 3.3, is different from *vacuity causes* defined by Samer and Veith in [29]. Samer and Veith introduce *weak vacuity* which occurs when a sub-formula can be replaced by a stronger sub-formula without changing the truth value (where a formula  $\varphi$  is stronger than  $\psi$  if  $\varphi$  implies  $\psi$ ). Since the set of candidates for stronger formulas is large and not all stronger formulas potentially provide a trivial reason for satisfaction of the original formula they ask the user to provide a set of stronger formulas which they term *causes*

---

<sup>1</sup> We use  $\varphi[\psi \leftarrow \psi']$  to denote the formula obtained from  $\varphi$  by replacing the subformula  $\psi$  of  $\varphi$  with  $\psi'$ .

according to which *weak vacuity* will be sought for. In contrast, in our work a *reason* is a Boolean expression appearing at the original formula which is responsible for the detected vacuity.

We note that our approach resembles that of the original vacuity work of Beer et al. [3], who considered *antecedent failure* [2] as their motivation. While our definition of TAF generalizes antecedent failure in the temporal direction, their definition of vacuity generalizes antecedent failure even in propositional logic — where the dimension of time is absent. We further note that the algorithm provided in [3] concentrated mainly on TAF. Thus our experience of vacuities being 100% interesting, comply with their report of vacuities “... always point to a real problem in either the design or its specification or environment” [3].

The rest of the paper is organized as follows. The next section gives preliminaries. Section 3 defines temporal antecedent failure and its reasons, and shows that formulas that may suffer from TAF can be represented by regular expressions. Section 4 gives an efficient algorithm to detect TAF and its reasons. Section 5 concludes the paper and discusses related work. Due to lack of space, all proofs are omitted. They can be found in the full version of the paper [9].

## 2 Preliminaries

### 2.1 Notations

We denote a letter from a given alphabet  $\Sigma$  by  $\ell$ , and an empty, finite, or infinite word from  $\Sigma$  by  $u$ ,  $v$ , or  $w$  (possibly with subscripts). The *concatenation* of  $u$  and  $v$  is denoted by  $uv$ . If  $u$  is infinite, then  $uv = u$ . The empty word is denoted by  $\epsilon$ , so that  $w\epsilon = \epsilon w = w$ . If  $w = uv$  we say that  $u$  is a *prefix* of  $w$ , denoted  $u \preceq w$ , that  $v$  is a *suffix* of  $w$ , and that  $w$  is an *extension* of  $u$ , denoted  $w \succeq u$ .

We denote the *length* of a word  $v$  by  $|v|$ . The empty word  $\epsilon$  has length 0, a finite word  $v = (\ell_0\ell_1\ell_2 \cdots \ell_n)$  has length  $n + 1$ , and an infinite word has length  $\infty$ . Let  $i$  denote a non-negative integer. For  $i < |v|$  we use  $v^i$  to denote the  $(i + 1)^{th}$  letter of  $v$  (since counting of letters starts at zero), and  $v^{-i}$  to denote the  $(i + 1)^{th}$  letter of  $v$  counting backwards. That is if  $v = (\ell_n \dots \ell_2\ell_1\ell_0)$  then  $v^{-i} = \ell^i$ . Let  $j$  and  $k$  denote integers such that  $j \leq k$ . We denote by  $v^{j..}$  the suffix of  $v$  starting at  $v^j$  and by  $v^{j..k}$  the subword of  $v$  starting at  $v^j$  and ending at  $v^k$ .

We denote a set of finite/infinite words by  $U$ ,  $V$  or  $W$  and refer to them as *properties*. The *concatenation* of  $U$  and  $V$ , denoted  $UV$ , is the set  $\{uv \mid u \in U, v \in V\}$ . Define  $V^0 = \{\epsilon\}$  and  $V^k = VV^{k-1}$  for  $k \geq 1$ . The *\*-closure* of  $V$ , denoted  $V^*$ , is the set  $V^* = \bigcup_{k < \omega} V^k$ . The notation  $V^+$  is used for the set  $\bigcup_{0 < k < \omega} V^k$ . The infinite concatenation of  $V$  to itself is denoted  $V^\omega$ .

In this work we follow the linear time framework. Thus, we assume the underlying temporal logic is linear and we regard a model  $M$  as an  $\omega$ -regular property (that is, we regard  $M$  as a set of infinite words that is generated by an automaton).

### 2.2 Temporal logic

Since the underlying temporal logic is linear, the models satisfying a temporal formula are words (finite or infinite). A temporal logic formula may refer to the past, to the

future or to both. We use the term *past* formulas for formulas that do not refer to the strict future. We use the term *future* formulas for formulas that do not refer to the strict past. We use the term *present* formula for formulas that do not refer to the strict future or to the strict past. The set of models satisfying a past formula is composed of finite non-empty words. The set of models satisfying future formulas is composed of infinite words. The set of models satisfying a present formula is composed of single letters. The set of models satisfying an arbitrary formula is composed of pairs  $\langle w, i \rangle$  where  $w$  is an infinite word and  $i$  is a non-negative integer. The prefix  $w^{0..i}$  represents the past and present and the suffix  $w^{i..}$  represents the present and future.

Below we give the semantics of the commonly used operators:  $\mathbf{X}$  (next),  $\overleftarrow{\mathbf{X}}$  (previous),  $\mathbf{G}$  (globally),  $\overleftarrow{\mathbf{G}}$  (historically),  $\mathbf{F}$  (eventually),  $\overleftarrow{\mathbf{F}}$  (once),  $\mathbf{U}$  (until), and  $\overleftarrow{\mathbf{U}}$  (since). Let  $\mathbb{P}$  be a given set of atomic propositions. We identify the set of present formulas  $\mathbb{B}$  with the set  $2^{2^{\mathbb{P}}}$  of subsets of valuation to the atomic propositions  $\mathbb{P}$ . We use *true* and *false* to denote the elements  $2^{\mathbb{P}}$  and  $\emptyset$  of  $\mathbb{B}$ , respectively. Let  $b$  be a present formula and  $\varphi$  an arbitrary formula.

- $\langle w, i \rangle \models b \iff w^i \in b.$
- $\langle w, i \rangle \models \mathbf{X}\varphi \iff \langle w, i + 1 \rangle \models \varphi.$
- $\langle w, i \rangle \models \overleftarrow{\mathbf{X}}\varphi \iff i > 0 \text{ and } \langle w, i - 1 \rangle \models \varphi.$
- $\langle w, i \rangle \models \mathbf{G}\varphi \iff \forall j \geq i, \langle w, j \rangle \models \varphi.$
- $\langle w, i \rangle \models \overleftarrow{\mathbf{G}}\varphi \iff \forall j \leq i, \langle w, j \rangle \models \varphi.$
- $\langle w, i \rangle \models \mathbf{F}\varphi \iff \exists j \geq i, \langle w, j \rangle \models \varphi.$
- $\langle w, i \rangle \models \overleftarrow{\mathbf{F}}\varphi \iff \exists j \leq i, \langle w, j \rangle \models \varphi.$
- $\langle w, i \rangle \models \varphi\mathbf{U}\psi \iff \exists k \geq i, \langle w, k \rangle \models \psi \text{ and } \forall i \leq j < k, \langle w, j \rangle \models \varphi.$
- $\langle w, i \rangle \models \varphi\overleftarrow{\mathbf{U}}\psi \iff \exists k \leq i, \langle w, k \rangle \models \psi \text{ and } \forall i \geq j > k, \langle w, j \rangle \models \varphi.$

A formula  $\varphi$  is *initially true* on a word  $w$  iff  $w, 0 \models \varphi$ . Given a temporal logic formula  $\varphi$  we use  $\llbracket \varphi \rrbracket$  to denote the set of words initially satisfying  $\varphi$ . That is,  $\llbracket \varphi \rrbracket = \{w : \langle w, 0 \rangle \models \varphi\}$ . For a past formula  $\varphi$ , we use  $\llbracket \varphi \rrbracket$  to denote the set of finite words satisfying  $\varphi$  at their last letter. That is,  $\llbracket \varphi \rrbracket = \{w : w \text{ is finite and } \langle w, |w| - 1 \rangle \models \varphi\}$ .

### 2.3 Regular expressions

**Definition 1 (Regular Expressions (RE s)).**

- Let  $\mathbb{B}$  be a finite non-empty set, and let  $b$  be an element in  $\mathbb{B}$ . The set of regular expressions (REs) over  $\mathbb{B}$  is recursively defined as follows:  $r, = b \mid r \cdot r \mid r \cup r \mid r^*$
- The language of a regular expression over  $\mathbb{B}$  with respect to  $\mathcal{L}(\cdot)$  is recursively defined as follows, where  $b \in \mathbb{B}$ , and  $r_1$  and  $r_2$  are REs.

1.  $\llbracket b \rrbracket = \mathcal{L}(b)$
2.  $\llbracket r_1 \cdot r_2 \rrbracket = \llbracket r_1 \rrbracket \llbracket r_2 \rrbracket$
3.  $\llbracket r_1 \cup r_2 \rrbracket = \llbracket r_1 \rrbracket \cup \llbracket r_2 \rrbracket$
4.  $\llbracket r^* \rrbracket = \llbracket r \rrbracket^*$

In standard definition of regular expressions  $\mathcal{L}(b) = \{b\}$  in the context of temporal logic  $\mathbb{B}$  is a set of present formulas over a given set of atomic propositions  $\mathbb{P}$  and  $\mathcal{L}(b) \subseteq 2^{2^{\mathbb{P}}}$  is the set of assignments to the propositions  $\mathbb{P}$  for which  $b$  holds.

## 2.4 Positions in Regular Expressions and Related Partial Orders

An RE is called *linear* if no letter appears in it more than once [10]. Any RE can be linearized by subscripting its letters with unique indexes.

*Example 1.* Let  $r_1$  be the RE  $\{a \cdot \{b^* \cdot c\} \cup \{d \cdot e^*\} \cdot c \cdot e\}$ . It can be subscriptized to the linear RE  $r'_1 = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\} \cdot c_6 \cdot e_7\}$ .

We call the letters of the subscriptized RE *positions* and save the term *letters* for the deindexed positions. That is, the positions of  $r'_1$  are  $a_1, b_2, c_3, d_4, e_5, c_6$  and  $e_7$  and its letters are  $a, b, c, d$  and  $e$  — the same as the letters of  $r_1$ . The set of positions of an RE  $r$  is denoted  $\text{pos}(r)$ . We can define a partial order between positions of a given RE. The definition of the partial order makes use of the following functions:

**Definition 2 (Position Functions).** Let  $r$  be a linear RE, and let  $\llbracket r \rrbracket$  be defined as in Definition 1 where for a position  $b$ ,  $\mathcal{L}(b)$  is  $\{b\}$ .

- $\mathcal{F}(r)$  - the set of positions that match the first letter of some word in  $\llbracket r \rrbracket$ .  
Formally,  $\mathcal{F}(r) = \{x \in \text{pos}(r) \mid \exists v \in \text{pos}(r)^* \text{ s.t. } xv \in \llbracket r \rrbracket\}$ .
- $\mathcal{L}(r)$  - the set of positions that match the last letter of some word in  $\llbracket r \rrbracket$ .  
Formally,  $\mathcal{L}(r) = \{x \in \text{pos}(r) \mid \exists v \in \text{pos}(r)^* \text{ s.t. } vx \in \llbracket r \rrbracket\}$ .
- $\mathcal{N}(r, x)$  - the set of positions that can follow position  $x$  in a path through  $r$ .  
Formally,  $\mathcal{N}(r, x) = \{y \in \text{pos}(r) \mid \exists u, v \in \text{pos}(r)^* \text{ s.t. } uxyv \in \llbracket r \rrbracket\}$ .
- $\mathcal{P}(r, x)$  - the set of positions that can precede position  $x$  in a path through  $r$ .  
Formally,  $\mathcal{P}(r, x) = \{y \in \text{pos}(r) \mid \exists u, v \in \text{pos}(r)^* \text{ s.t. } uyxv \in \llbracket r \rrbracket\}$ .

An inductive definition of these functions appears in [6] and is repeated in the full version of the paper [9].

The transitive closure of  $\mathcal{N}(\cdot)$  induces a partial order  $\prec$  on the set of positions so that,  $\mathcal{F}(r)$  positions are the minimal positions,  $\mathcal{L}(r)$  are the maximal positions, if  $y \in \mathcal{N}(r, x)$  then  $y \succ x$ , and if  $z \succ y$  and  $y \succ x$  then  $z \succ x$ .

*Example 2.* For  $r'_1$  defined in Example 1 we have  $\mathcal{F}(r'_1) = \{a_1\}$ ,  $\mathcal{L}(r'_1) = \{e_7\}$ . The partial order is the transitive closure of the relation given by:  $a_1 \prec b_2, b_2 \prec c_3, c_3 \prec c_6, c_6 \prec e_7, a_1 \prec d_4, d_4 \prec e_5$  and  $e_5 \prec c_6$ .

Given an RE  $r$  and a position  $x$  we denote by  $r \rfloor_x$  the “prefix” of  $r$  that ends with  $x$ . Similarly, we denote by  $r \lfloor_x$  the “prefix” of  $r$  that ends exactly before  $x$ . Formally,

$$\begin{array}{ll}
 1. b \rfloor_x = b & 1. b \lfloor_x = \epsilon \\
 2. r_1 \cdot r_2 \rfloor_x = \begin{cases} r_1 \rfloor_x & \text{if } x \in r_1 \\ r_1 \cdot r_2 \rfloor_x & \text{otherwise} \end{cases} & 2. r_1 \cdot r_2 \lfloor_x = \begin{cases} r_1 \lfloor_x & \text{if } x \in r_1 \\ r_1 \cdot r_2 \lfloor_x & \text{otherwise} \end{cases} \\
 3. r_1 \cup r_2 \rfloor_x = \begin{cases} r_1 \rfloor_x & \text{if } x \in r_1 \\ r_2 \rfloor_x & \text{otherwise} \end{cases} & 3. r_1 \cup r_2 \lfloor_x = \begin{cases} r_1 \lfloor_x & \text{if } x \in r_1 \\ r_2 \lfloor_x & \text{otherwise} \end{cases} \\
 4. r^* \rfloor_x = r \rfloor_x & 4. r^* \lfloor_x = r \lfloor_x
 \end{array}$$

*Example 3.* For  $r'_1$  defined in Example 1, we have  $r'_1 \rfloor_{e_5} = \{a_1 \cdot d_4 \cdot e_5\}$ ,  $r'_1 \rfloor_{e_7} = \{a_1 \cdot d_4\}$ ,  $r'_1 \rfloor_{c_6} = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\} \cdot c_6\}$  and  $r'_1 \rfloor_{c_6} = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\}\}$ .

### 3 Temporal Antecedent Failure (TAF) and its Reasons

Antecedent failure in propositional logic [2] occurs when the formula is trivially valid because the pre-condition (antecedent) of the formula is not satisfiable in the model. We generalize this notion to *Temporal Antecedent Failure* (TAF), where some future requirement depends on a temporal pre-condition. We start by giving a definition of TAF in terms of past and future formulas. In section 3.2 we show that any formula that may suffer from TAF can be expressed using regular expressions. When a temporal pre-condition fails to hold, there could be different reasons for that. We define TAF reasons in section 3.3.

#### 3.1 Temporal Antecedent Failure

In propositional logic a formula suffers from antecedent failure if it is of the form  $(A \rightarrow B)$  and  $A$  is not satisfiable. Intuitively, the generalization to temporal logic is that a formula suffers from *temporal antecedent failure* if it is of the form  $\mathbf{G}(\Phi \rightarrow \Psi)$  where  $\Phi$  is a past formula representing a condition,  $\Psi$  is a future formula representing a requirement, and the condition  $\Phi$  never holds. To capture this intuition, we present the following two definitions.

**Definition 3 (Temporal Implication Form).** A formula has a temporal implication form iff it can be expressed in the form  $\mathbf{G}(\Phi \rightarrow \Psi)$  where  $\Phi$  is a past formula and  $\Psi$  is a future formula.

We note that  $\Psi$  is not required to refer to the strict future, that is, it may also refer to the present. We also note that this allows  $\Psi$  to be the present formula *false* or another unsatisfiable formula. In such a case  $\Psi$  does not carry a requirement, but rather states that  $\Phi$  is disallowed. Therefore,  $\Phi$  is not really an antecedent and the notion of antecedent failure does not apply. We therefore assume  $\Psi$  is satisfiable.

**Definition 4 (Temporal Antecedent Failure).** Let  $M$  be a model,  $\Phi$  a past formula and  $\Psi$  a future formula. We say that the formula  $\varphi = \mathbf{G}(\Phi \rightarrow \Psi)$  suffers from TAF of  $\Phi$  in model  $M$  iff  $M \models \varphi$  and for all words  $w$  in  $M$ , and for all  $i < |w|$  we have  $\langle w, i \rangle \not\models \Phi$ .

We say that a *property* has a temporal implication form if it can be expressed by a formula that has a temporal implication form. We say that a *property* suffers from temporal antecedent failure in  $M$  if it can be expressed by a formula that suffers from temporal antecedent failure in  $M$ . In the sequel we use antecedent failure to mean temporal antecedent failure. Also, when  $M$  is understood from the context we sometimes omit it.

#### 3.2 Moving to Regular Expressions

We examine formulas in temporal implication form in view of the IEEE standard temporal logic PSL [18, 23]. A major feature of PSL is the *suffix implication* operator, which makes use of regular expressions. A suffix implication formula is of the form  $r \mapsto \varphi$

where  $r$  is a regular expression and  $\varphi$  is a PSL formula. Intuitively, this formula states that whenever a prefix of a given path matches the regular expression  $r$ , the suffix of that path should satisfy the requirement  $\varphi$ .

*Example 4.* The following formula

$$\psi = \{true^* \cdot req \cdot \neg ack^* \cdot ack \cdot \neg abort\} \mapsto (\neg retry \text{ U } req)$$

states that if the first  $ack$  following  $req$  is not *aborted* one cycle after  $ack$ , then it must not be retried (signal  $retry$  should not hold at least until the next  $req$  holds). The formal definition of suffix implication taken from [23], is given below.

**Definition 5 (Suffix Implication Formulas [23]).** *Let  $r$  be a regular expression and  $\varphi$  a temporal logic formula, both over a given set of atomic propositions  $\mathbb{P}$ . Let  $v$  be a word over  $2^{\mathbb{P}}$ . Then*

$$v \models r \mapsto \varphi \iff \forall j < |v|, \text{ if } v^{0..j} \in \llbracket r \rrbracket \text{ then } \langle v, j \rangle \models \varphi$$

The following claim connects formulas in temporal implication form to suffix implication formulas.

**Claim 1.** *An omega regular property has a temporal implication form iff it can be expressed in the form  $r \mapsto \Psi$  where  $r$  is a regular expression and  $\Psi$  is a future formula.*

We note that although PSL includes other language constructs, suffix implication is the major one and in fact, other temporal formulas can be translated into suffix implication [8, 11].<sup>2</sup> Note further that a property may be expressed by two different formulas in suffix implication form. For example, the formula  $\{a \cdot b\} \mapsto (\text{F } c)$  is equivalent to the formula  $\{a\} \mapsto \text{X}(\neg b \vee (b \wedge \text{F } c))$ . In order to get maximum information regarding TAF it is beneficial to take a form in which the left-hand-side of suffix implication is maximal in terms of the partial order between RES. Indeed, the procedure in [8] yields a formula in which the RE at the left-hand-side is maximal.<sup>3</sup> In the rest of the paper we shall assume the formula is given as a suffix implication.

The following claim provides a characterization of TAF in terms of suffix implication formulas.

**Claim 2.** *A formula of the form  $r \mapsto \Psi$  where  $r$  is a regular expression and  $\Psi$  is a future formula suffers from TAF of  $r$  in model  $M$  iff for all words  $w$  in  $M$ , and for all prefixes  $u \preceq w$  we have  $u \notin \llbracket r \rrbracket$ .*

<sup>2</sup> As elaborated in [8], the majority of formulas written in practice can be effectively transformed into suffix implication form. In particular, the common fragment of LTL and ACTL, all RCTL formulas and all the safety formulas in the simple subset of PSL can be linearly translated into suffix implication form. Clearly, since the left hand side of the suffix implication operator can be any formula, we can manage in the same manner liveness formulas as well [11].

<sup>3</sup> The procedure in [8] yields a formula of the form  $\{r\} \mapsto \text{false}$ . As explained earlier, for TAF detection we assume the right-hand-side is not a contradiction. However, it is easy to transform the result of [8] to one in which the right-hand-side is not a contradiction.

*Example 5.* Consider the formula  $\psi$  given in Example 4, for which  $r = \{true^* \cdot req \cdot \neg ack^* \cdot ack \cdot \neg abort\}$ . The formula  $\psi$  would suffer from TAF in a model  $M$  where no word ever satisfies  $r$ . (That is, there is never a sequence of  $req$ , followed by an  $ack$  after a finite number of cycles, and then  $\neg abort$  one cycle after  $ack$ .) Note that there could be several reasons for this  $r$  never to hold on any word. It could be that no  $reqs$  are ever given in  $M$ , or no  $acks$  are given. It could also be the case that in our model, all  $reqs$  are always *aborted*. In the next section we define what it means to be a *reason* for TAF.

Note that  $\Psi$  plays no role in the decision of whether  $r \mapsto \Psi$  suffers from TAF in a given model  $M$ . Thus, if  $r \mapsto \Psi$  suffers from TAF of  $r$  in  $M$  then for any other formula  $\Psi'$ , the formula  $r \mapsto \Psi'$  also suffers from TAF of  $r$  in  $M$ . Therefore, from now on we say that  $r$  suffers from TAF in  $M$  without specifying the formula.

### 3.3 Defining TAF reasons

Let  $r$  be an RE that suffers from TAF in a model  $M$ . We would like to find the reason for this. For example if  $r = \{true^* \cdot a \cdot b \cdot c \cdot c^* \cdot d\}$  and  $b$  never holds after an  $a$ , we would like to say that  $b$  is the reason for TAF. Note that in addition, it might be the case that  $d$  never holds after  $c$ , but we still view  $b$  as the reason. That is, our intuitive view of the reason for TAF is the earliest letter that does not hold when expected. Having said that, we note that sometimes the earliest letter that does not hold when expected, is actually *not* the reason. To see why, consider the following example.

*Example 6.* Let  $r'_1 = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\} \cdot c_6 \cdot e_7\}$  be the RE defined in Example 1 and assume its letters are mutually exclusive (that is, if one holds the others do not). Let  $M$  be a model such that  $a_1$  holds at the first cycle,  $b_2$  holds at the second cycle,  $c_3$  at the third cycle and  $c_6$  never holds. Thus,  $c_6$  is a reason for TAF, although there exists an earlier position,  $d_4$ , that also does not hold when expected.

The formal definition of TAF reasons follows. Intuitively, a TAF reason is a position that does not hold when expected although one of its immediate predecessors does hold when expected.

**Definition 6 (Temporal Antecedent Failure Reason).** *Let  $M$  be a model and  $r$  a linear RE, such that  $r$  suffers from TAF in  $M$ . A position  $x$  is a reason for TAF of  $r$  in  $M$  iff  $r_{\cdot x}$  suffers from TAF in  $M$  and there exists a position  $x' \in \mathcal{P}(r, x)$  such that  $r_{\cdot x'}$  does not suffer from TAF in  $M$ .*

The following proposition assures that if a model  $M$  suffers from TAF with respect to some RE  $r$  then indeed (at least) one of the positions in  $r$  is a reason for TAF of  $r$  in  $M$ .

**Proposition 1.** *Let  $M$  be a model and  $r$  an RE, such that  $r$  suffers from TAF in  $M$ . Then there exists a position  $x$  in  $r$  such that  $x$  is a reason for TAF of  $r$  in  $M$ .*

Consider again the RE  $r_1$  and the model  $M$  of Example 6. We note that according to Definition 6 the position  $d_4$  is also a reason for TAF. However,  $c_6$  is a more fundamental reason: the fact that  $c_6$  does not hold when expected implies that for any  $\Psi$  the formula

$r \mapsto \Psi$  holds. The fact that  $d_4$  does not hold when expected does not imply that, as it could be that  $b_2, c_3, c_6, e_7$  do hold when expected and thus in the formula  $r \mapsto \Psi$ , the requirement  $\Psi$  must be fulfilled. We thus differentiate between two types of reasons for antecedent failure. If the failure of the position is “recovered” by another position, it is a *secondary* reason. Otherwise it is a *primary* reason.

**Definition 7 (Primary and Secondary Reasons).** *Let  $M$  be a model and  $r$  a linear RE. Given position  $x$  is a reason of antecedent failure of  $r$  in  $M$ , we say that  $x$  is a primary reason if for all  $x' \succ x$  we have that  $r_{x'}$  suffers from TAF in  $M$  and  $x'$  is not a reason for TAF of  $r$  in  $M$ . Otherwise we say that  $x$  is a secondary reason.*

One can check that these definitions indeed give us that in Example 6 both  $d_4$  and  $c_6$  are reasons, and  $c_6$  is a primary reason while  $d_4$  is a secondary reason. There are no other reasons for TAF of  $r'_1$  in this example. As another example consider the RE  $r_2 = \{true^* \cdot a \cdot b^* \cdot c\}$  and assume the model is such that  $a$  holds only on the 11<sup>th</sup> cycle and neither  $b$  nor  $c$  hold on the 12<sup>th</sup> cycle. Then both  $b$  and  $c$  are reasons, and  $c$  is a primary reason while  $b$  is a secondary reason. There are no other reasons for TAF of  $r_2$  in this example.

The following proposition strengthens Proposition 1 by asserting that if  $r$  suffers from TAF in  $M$  then there exists a *primary* reason for this.

**Proposition 2.** *Let  $M$  be a model and  $r$  an RE, such that  $r$  suffers from TAF in  $M$ . Then there exists a position  $x$  in  $r$  such that  $x$  is a primary reason for TAF of  $r$  in  $M$ .*

## 4 Implementation

### 4.1 Detecting Antecedent Failure and its Reasons

Below we provide the implementation of detection of antecedent failure and its reasons, for suffix implication formulas. A key feature of our algorithm is that it does not add auxiliary automata, but rather adds invariant checks on the automata built for model checking the formula. Thus, before providing our algorithm we recall several facts regarding the model checking of suffix implication formulas, using the automata theoretic approach.

The first fact we build upon is that for any regular expression there exists a natural NFA accepting the same language. An NFA  $N$  is said to be *natural* for a linear RE  $r$  if every state of  $N$  (except possibly a trapping state) is associated with a unique position in  $r$ . A construction for a linear NFA for a given RE  $r$  can be found in [6] and is repeated in the full version of the paper [9]. This NFA also satisfies that all the outgoing edges from a given state have the same label, and the label is the set of valuations satisfying the letter associated with the state.

The following proposition [12, Proposition 3.5] states that this NFA can be used for both recognizing a given RE and failing to recognize it.

**Proposition 3 ([12]).** *Let  $r$  be an RE. There exists an NFA  $N_r$  with  $O(|r|)$  states such that  $N_r$  has a trapping non-accepting state  $q_{bad}$  and for every word  $w$  over  $\Sigma$ ,*

1. there exists an accepting run of  $N_r$  on  $w$  iff there exists a non-empty prefix  $u \preceq w$  such that  $u \in \llbracket r \rrbracket$ .
2. every run of  $N_r$  on  $w$  reaches  $q_{bad}$  iff for every non-empty prefix  $u \preceq w$  we have  $u \notin \llbracket r \rrbracket$ .

We refer to such an NFA as a *trapping* NFA. The proof of the following proposition from [12, Proposition 3.15] asserts that efficient model checking a formula of the form  $r \mapsto \varphi$  can be done by “concatenating” the trapping NFA for  $r$  with the Büchi automaton for  $\varphi$ .

**Proposition 4 ([12]).** *Let  $r$  be an RE, and  $\varphi$  a formula. If there exists a weak (terminal) Büchi automaton for  $\neg\varphi$  with state set  $S$ , then there exists a weak (terminal) Büchi automaton for  $\neg(r \mapsto \varphi)$  with at most  $O(|r| + |S|)$  states.*

We are now ready for the implementation. The following proposition states that antecedent failure of  $r \mapsto \varphi$  with respect to  $r$  in  $M$  can be detected by checking the invariant property stating that the trapping NFA of  $r$  does not reach an accepting state on a parallel composition of the model with the trapping NFA of  $r$ . We represent the model  $M$  and the NFA for  $r$  symbolically using discrete transition systems (DTSs) over finite and infinite words. The definition of a DTS, the transition from an NFA to a DTS and their parallel composition, denoted  $\parallel$  are defined in [7] and repeated in the full version of the paper [9]. In the sequel we use  $\mathcal{D}_M$  for the DTS representation of  $M$ . We use  $\mathcal{D}_r$  to denote the DTS representation of the trapping NFA for  $r$  and  $\mathcal{A}_r$  to denote the set of accepting states of  $\mathcal{D}_r$ . For a position  $x$  of  $r$  we use  $at\_x$  to denote a present formula stating that  $\mathcal{D}_r$  is in the state corresponding to position  $x$ . We use  $\ell(x)$  to denote the letter on the outgoing edges from state satisfying  $at\_x$ . Thus  $\ell(x)$  is the letter obtained from position  $x$  after removing the added subscript.

**Proposition 5.** *Let  $M$  be a model and  $r$  an RE.*

$$r \text{ suffers from TAF in } M \text{ iff } \mathcal{D}_M \parallel \mathcal{D}_r \models \mathbf{AG} \neg \mathcal{A}_r$$

Proposition 6 below states the invariants that provide a detection of a TAF reason. The first invariant checks that the position is reachable, implying that an immediate predecessor did hold when expected. The second invariant holds if the position does not hold when expected. Proposition 7 below provides the invariant that distinguishes between primary and secondary reasons: it checks whether there exists a (not necessarily immediate) successor that is reachable.

**Proposition 6.** *Let  $M$  be a model,  $r$  an RE such that  $M$  suffers from antecedent failure of  $r$ . Let  $x$  be a position in  $r$ . Then,  $x$  is a reason of antecedent failure in  $\varphi$  iff the following two conditions hold:*

- $\mathcal{D}_M \parallel \mathcal{D}_r \not\models \mathbf{AG} \neg at\_x$  //  $x$  is reachable.
- $\mathcal{D}_M \parallel \mathcal{D}_r \models \mathbf{AG} (at\_x \rightarrow \neg \ell(x))$  //  $x$  is not exercised.

**Proposition 7.** *Let  $M$  be a model,  $r$  an RE and  $x$  a position in  $r$  that is a reason of antecedent failure. Then,  $x$  is a primary reason of antecedent failure iff  $\mathcal{D}_M \parallel \mathcal{D}_r \models \mathbf{AG} \bigwedge_{y \succ x} \neg at\_y$ . Otherwise  $x$  is a secondary reason.*

It is important to note that all the detection “tests” are phrased as a set of invariants over the parallel execution of the given model with the NFA for the given formula. Thus, they can run on the fly over the same model (without adding any automata) and therefore the cost of adding them to the verification is small.

## 4.2 Witness Generation

The previous sections discussed detection of temporal antecedent failure. If a formula is valid on a model  $M$ , then using Proposition 5 we can check whether it suffers from TAF. If the formula suffers from TAF we seek for the reason for TAF, if it does not suffer from TAF we would like to convince the user of this as well. In [5] Beer et al. suggested the term *witness* for a trace convincing of the fact the formula does not hold vacuously.

In our case, intuitively, a witness should show that no position is a reason for TAF. Indeed, by Proposition 1, this is an indication that the formula does not suffer from TAF. Recall however that the existence of a secondary reason is not an indication for TAF. However, users may want to get a witness to the fact that a position is not a secondary reason as well.

We thus define two kinds of witnesses. A nonTAF-witness is a trace convincing of the fact that the model does not suffer from TAF with respect to the given RE. A nonReason-witness is a trace convincing of the fact that a given position is not a reason for TAF of  $r$  in  $M$ . A *full witness* is a set of traces contradicting the fact that there exists a position which is a reason for TAF of  $r$  in  $M$ . Formally,

**Definition 8 (Witnesses).** *Let  $M$  be a model,  $r$  an RE and  $x$  a position in  $r$ .*

- A run  $\beta$  of a DTS  $\mathcal{D}_M$  is a nonTAF-witness for  $r$  in  $M$  iff there exists a prefix  $\alpha$  of  $\beta$  such that  $L(\alpha) \in \llbracket r \rrbracket$
- A run  $\beta$  of a DTS  $\mathcal{D}_M$  is a nonReason-witness for  $r$  in  $M$  with respect to  $x$  iff it is a nonTAF-witness and there exists a prefix  $\alpha$  of  $\beta$  such that  $L(\alpha) \in \llbracket r_x \rrbracket$ .

A set  $S \subseteq L(\mathcal{D}_M)$  is a full witness for  $r$  in  $\mathcal{D}_M$  iff for each  $x \in \text{pos}(r)$ , there exists  $\beta \in S$  such that  $\beta$  is a nonReason-witness for  $r$  in  $\mathcal{D}_M$  with respect to  $x$ .

The generation of witnesses uses the model-checking ability to generate counter examples. In order to provide a nonTAF-witness we ask the model checker to find a counter example to the fact that the accepting states of the NFA for the given RE are never visited. In order to provide a nonReason-witness for a given position  $x$  we ask the model checker to find a counter example to the fact that in addition, whenever the state associated with  $x$  is visited, the corresponding letter does not hold. The following proposition states this formally.

**Proposition 8.** *Let  $M$  be a model,  $r$  an RE and  $x$  a position in  $r$ .*

- A counter example for  $\mathbf{G}(\neg \mathcal{A}_r)$  in  $\mathcal{D}_M \parallel \mathcal{D}_r$  is a nonTAF-witness for  $r$  in  $\mathcal{D}_M$ .
- A counter example for  $\mathbf{G}(\text{at}_-(x) \rightarrow \neg \ell(x)) \wedge \mathbf{G}(\neg \mathcal{A}_r)$  in  $\mathcal{D}_M \parallel \mathcal{D}_r$  is a nonReason-witness for  $r$  in  $\mathcal{D}_M$  with respect to  $x$ .

### 4.3 Procedure for Detecting TAF, its Reasons and Generating Witnesses

Procedure **FindTAFAndGenerateWitness** described below, receives as input a DTS representation  $\mathcal{D}_M$  of a model  $M$  and an RE  $r$ . If  $M$  suffers from TAF of  $r$  the procedure returns the reasons for TAF (both primary and secondary). If  $M$  does not suffer from TAF of  $r$  the procedure returns a nonTAF-witness. The procedure does not generate nonReason-witness. It is easy to augment it to produce a nonReason-witness for a given position  $x$  (or for all positions  $x$ ), by adding additional invariant checks as described in Proposition 8.

---

**Algorithm 1:** Find TAF and Generate Witness()

---

```

1 Input: DTS  $\mathcal{D}_M$ ; RE  $r$ ;
2 Output: set PrimaryList; set SecondaryList; trace WitnessTrace;
3 PrimaryList :=  $\emptyset$ ;
4 SecondaryList :=  $\emptyset$ ;
5 FindTAFReasons( $\mathcal{D}_M, r, x_\infty, true, PrimaryList, SecondaryList, \emptyset$ );
6 if  $PrimaryList = \emptyset$  then
7   | WitnessTrace := ProduceCounterExample( $AG \neg at_{x_\infty}$ );
8 end

```

---

The procedure **FindTAFReasons** is a recursive procedure that receives as input a DTS representation  $\mathcal{D}_M$  of a model  $M$ , an RE  $r$ , a position  $x$ , a flag *primary* indicating whether a primary or secondary reason is searched for, and three sets *PrimaryList*, *SecondaryList* and *VisitedPos* to save sets of positions as implied by the name of the set. At the initial call to the procedure the *primary* flag is turned on. The position sent at the initial call is  $x_\infty$  which is a position added in the construction of the natural NFA (see section A.2 of the full version). This position is added as the last position in order to create a unique last position and simplify the procedure. That is, if  $r$  is a natural RE the procedure works on  $r \cdot x_\infty$ .

The procedure first checks that the given position  $x$  was not visited before. If it was visited it returns (line 3) otherwise it updates the set of visited positions accordingly (line 5). It then checks whether the position is reachable (line 6). If it is unreachable then (according to Proposition 5) it is not a reason. It thus calls recursively to each of  $x$ 's immediate predecessors (lines 20-21). If it is reachable it checks whether it is "exercised" — i.e. whether the corresponding letter holds when expected (line 8). If it is not exercised then (by Proposition 5) a reason is found. The position is inserted to either *PrimaryList* or *SecondaryList* according to the value of the *primary* flag (lines 10-13). The procedure proceeds with recursive calls to the immediate predecessor with the flag *primary* turned off (line 16-17). Proposition 9 below states the correctness of the procedure.

**Proposition 9.** *Let  $\mathcal{D}_M$  be a DTS representation of a model  $M$  and  $r$  an RE. Let *PrimaryList*, *SecondaryList* and *WitnessTrace* be the output of the procedure **FindTAFAndGenerateWitness** on the input  $r$  and  $\mathcal{D}_M$ . Then*

1. *The formula does not suffer from TAF in  $M$  iff  $PrimaryList = \emptyset$ .*
2.  *$PrimaryList$  holds the set of positions which are a primary reason of TAF.*

3. *SecondaryList* holds the set of positions which are a secondary reason of TAF.
4. If  $\text{PrimaryList} = \emptyset$  then *WitnessTrace* holds a nonTAF-witness for  $r$  in  $M$ .

---

**Algorithm 2:** FindTAFReasons(FDS  $\mathcal{D}_M$ , RE  $r$ , position  $x$ , Bool primary, set PrimaryList, set SecondaryList, set VisitedPos)

---

```

1 Bool unreachable, unexercised;
2 if  $x \in \text{VisitedPos}$  then
3   | return;
4 end
5 VisitedPos := VisitedPos  $\cup \{x\}$ 
6 unreachable :=  $\mathcal{D}_M \models \text{AG} \neg(at\_x)$ 
7 if unreachable == false then
8   | unexercised :=  $\mathcal{D}_M \models \text{AG}(at\_x \rightarrow \neg \ell(x))$ 
9   | if unexercised then
10    | if primary then
11      | PrimaryList := PrimaryList  $\cup \{x\}$ 
12    | else
13      | SecondaryList := SecondaryList  $\cup \{x\}$ 
14    | end
15  | end
16  | foreach  $y \in \mathcal{P}(r, x)$  do
17    | FindTAFReasons( $\mathcal{D}_M, r, y, false, \text{PrimaryList}, \text{SecondaryList}, \text{VisitedPos}$ );
18  | end
19 else
20  | foreach  $y \in \mathcal{P}(r, x)$  do
21    | FindTAFReasons( $\mathcal{D}_M, r, y, primary, \text{PrimaryList}, \text{SecondaryList}, \text{VisitedPos}$ );
22  | end
23 end

```

---

## 5 Discussion

Several definitions of vacuity exist in the literature. The commonly used one is that of Beer et al. [3] cited in the introduction, saying that  $\varphi$  is vacuous in a model  $M$  if there exists a sub-formula  $\psi$  of  $\varphi$  that *does not affect* the truth value of  $\varphi$  in  $M$ . We note that TAF refines vacuity in the sense that if  $\varphi$  suffers from TAF in  $M$  it is also vacuous in  $M$  according to Beer et al. To see this, let  $\varphi = (r \mapsto \Psi)$ , and assume that  $M \models r \mapsto \Psi$  and  $r$  suffers from TAF in  $M$ . Thus, the full condition  $r$  never occurs in  $M$ , and therefore for any formula  $\Psi'$ ,  $M \models r \mapsto \Psi'$ . Thus  $\Psi$  does not affect the truth value of  $\varphi$  in  $M$ .

Our method detects TAF by asserting a single invariance (safety) formula, derived from the automaton already built for model checking of the original formula. Thus, it has two advantages over existing methods. First, only *one* formula needs to be checked, as opposed to several formulas — one for each proposition — in other methods [24, 1]. Second, it avoids building another automaton for the vacuity formula. The actual effort needed to detect TAF depends on the model checking method used in practice.

In the worst case it amounts to another model checking run of an invariance formula. However, when the reachable state space is computed, as done by BDD-based model checkers such as SMV [25], TAF is detected by intersecting the BDD of the invariance formula with that of the reachable states. This intersection is easily performed, and never takes more than a few seconds, even for very large models. TAF is therefore detected with almost no overhead on the model checking process.

TAF detection is implemented in the IBM model checking tool RuleBase [21] and serves as the major vacuity detection algorithm of the tool (applied to all formulas that can be translated into suffix implication form, which forms the vast majority of formulas written in practice [8]). Unlike evidence regarding other types of vacuity, experience shows that temporal antecedent failure always indicates a real problem in the model, environment or formula under verification. Evidence in the other direction are less clear. Although non-TAF vacuities are many times considered non-problem by the users, this is not always the case. We thus propose a two button approach. Button 1 (the default), would check for TAF that, on the one hand is guaranteed to be important and on the other hand can be detected efficiently. If the user is interested in a more comprehensive check, with the risk of it producing non-real problems and taking a longer time, she can then choose button 2, that would perform the desired comprehensive check.

## References

1. R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection in linear temporal logic. volume 2725, pages 368–380, Boulder, CO, USA, July 2003. Springer-Verlag.
2. D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Design Automation Conference*, pages 596–602, 1994.
3. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc. 9<sup>th</sup> International Conference on Computer Aided Verification (CAV)*, LNCS 1254, pages 279–290. Springer-Verlag, 1997.
4. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.
5. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc. 10<sup>th</sup> International Conference on Computer Aided Verification (CAV'98)*, LNCS 1427, pages 184–194. Springer-Verlag, 1998.
6. S. Ben-David, D. Fisman, and S. Ruah. Automata construction for regular expressions in model checking, June 2004. IBM research report H-0229.
7. S. Ben-David, D. Fisman, and S. Ruah. Embedding finite automata within regular expressions. In 1st International Symposium on Leveraging Applications of Formal Methods . Springer-Verlag, November 2004.
8. S. Ben-David, D. Fisman, and S. Ruah. The safety simple subset. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *First International Haifa Verification Conference*, volume 3875 of *Lecture Notes in Computer Science*, pages 14–29. Springer, November 2005.
9. S. Ben-David, D. Fisman, and S. Ruah. Temporal antecedent failure: Refining vacuity, September 2007. IBM research report H-0252.
10. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.
11. M. Boule and Z. Zilic. Efficient automata-based assertion-checker synthesis of psl properties. In *Workshop on High LevelDesign, Validation, and Test(HLDVTO6)*, 2006.

12. D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical Report MCS05-04, The Weizmann Institute of Science, May 2005.
13. D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Y. Vardi. Regular vacuity. In *CHARME05*, October 2005.
14. M. Chechik, A. Gurfinkel, and M. Gheorghiu. Finding environmental guarantees. In *FASE*, March 2007.
15. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
16. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.
17. Y. Dong, B. saran Starosta, C. Ramakrishnan, and S. A. Smolka. Vacuity checking in the modal mu-calculus. In *In Proc. AMAST'02*, volume 2422, pages 147–162, 2002.
18. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
19. A. Gurfinkel and M. Chechik. Extending extended vacuity. In *FMCAD'04*, pages 306–321, Austin, Texas, November 2004.
20. A. Gurfinkel and M. Chechik. How vacuous is vacuous? In *(TACAS'04)*, pages 451–466, Barcelona, Spain, March 2004.
21. IBM's Model Checker RuleBase. [http://www.haifa.il.ibm.com/projects/verification/rb\\_home\\_page/index.html](http://www.haifa.il.ibm.com/projects/verification/rb_home_page/index.html).
22. Annex E of IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800<sup>TM</sup>-2005.
23. IEEE Standard for Property Specification Language (PSL). IEEE Std 1850<sup>TM</sup>-2005.
24. O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, February 2003.
25. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
26. K. S. Namjoshi. An efficiently checkable, proof-based formulation of vacuity in model checking. In *In Proceeding of CAV04*, pages 57–69, July 2004.
27. M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. pages 485–499, July 2002.
28. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *5th International Symposium on Programming*, 1982.
29. M. Samer and H. Veith. Parametrized vacuity. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, pages 322–336, November 2004.
30. R. Tzoref and O. Grumberg. Automatic refinement and vacuity detection for symbolic trajectory evaluation. In *CAV*, pages 190–204, 2006.