Noname manuscript No.
(will be inserted by the editor)

Vacuity in Practice: Temporal Antecedent Failure

Shoham Ben-David · Fady Copty · Dana Fisman · Sitvanit Ruah

the date of receipt and acceptance should be inserted later

Abstract Different definitions of vacuity in temporal logic model checking have been suggested along the years. Examining them closely, however, reveals an interesting phenomenon. On the one hand, some of the definitions require high-complexity vacuity detection algorithms. On the other hand, studies in the literature report that not all vacuities detected in practical applications are considered a problem by the system verifier. 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 restrict our attention to practical usage of vacuity detection. 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 property. We show how vacuity information can be derived from the automaton built for the original property, and we introduce the notion of vacuity explanation. Our experiments demonstrate that this type of vacuity as well as its reasons can be computed with a negligible increase in the overall runtime.

S. Ben-David
The Hebrew University, Jerusalem, Israel.
E-mail: shohambd@gmail.com
Shoham Ben-David is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

F. Copty
IBM Systems and Technology Group, Mount Carmel, Haifa 31905, Israel.
E-mail: fadyc@il.ibm.com

D. Fisman
Weizmann Institute of Science and IBM Research, Mount Carmel, Haifa 31905, Israel.
E-mail: dana.fisman@gmail.com

S. Ruah
IBM Research, Mount Carmel, Haifa 31905, Israel.
E-mail: sitva.ruah@gmail.com

1 Introduction

Model checking ([26,54], c.f.[27]) is the procedure of deciding whether a given model satisfies a given property. One of the nice features of model checking is the ability to produce, when the property does not hold in the model, an execution path to demonstrate the failure. However, when the property is found to hold in the model, traditional model checking tools provide no further information. While satisfaction of the property 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 formulation of the property (the 'formula'), the model itself, or the environment. As a simple example consider the LTL formula $\varphi = \mathsf{G}(req \rightarrow \mathsf{F}(ack))$, 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. ([5,6]), who defined it as follows: a formula φ is vacuously satisfied in a model M iff it is satisfied in M, (i.e., $M \models \varphi$) and there exists a sub-formula ψ of φ that *does not affect* the truth value of φ in M. That is, there exists a sub-formula ψ of φ such that $M \models \varphi[\psi \leftarrow \psi']$ for any formula ψ' (where $\varphi[\psi \leftarrow \psi']$ denotes the formula obtained from φ by replacing the sub-formula ψ of φ with ψ'). For example, in the formula $\varphi = \mathsf{G}(req \rightarrow \mathsf{F}(ack))$, if *req* is never active in M, then the formula holds, no matter what the value of $\mathsf{F}(ack)$ is. Thus, the sub-formula $\mathsf{F}(ack)$ does not affect the truth value of φ 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 (see Section 7 for a discussion on related works). Armoni et al. in [2] defined *trace vacuity*, to deal with multiple occurrences of sub-formulas. Gurfinkel and Chechik in [34] showed that detecting trace vacuity is exponential for CTL, and double-exponential for CTL*. Bustan et al. in [18] 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 definitions should be adopted for practical applications. Note that RELTL is a simplified language defined for the purpose of the analysis done in [18]; Its language constructs however, are at the core of the languages used in practice today (PSL [36] and SVA [37]).

Chechik et al. in [20] report on an interesting phenomenon, which we observed independently as well. 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 $\varphi = \mathsf{G}(req \rightarrow \mathsf{F}(ack))$, 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 $\mathsf{F}(ack)$ always holds. According to the definition of vacuity, φ holds vacuously in the model (since *req* does not affect the truth value of φ 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 view of the discouraging complexity result of Bustan et al. for RELTL, and the observation of Chechik et al. discussed above, we propose a refined definition of vacuity, such that detection is almost "for free", and, according to feedback from users, vacuity *always* indicates real problems in the model, formula or environment. We work with temporal formulas for which the antecedent must occur earlier in time than the consequent. For such formulas, we define *temporal antecedent failure* (TAF), a vacuity that occurs when the antecedent part of the formula is never fulfilled in the model.

Our work is especially applicable for regular expression-based temporal logics, that are the major specification languages used in practice [9,29,36,37]. Such logics are verified using the construction of an automaton that represents the property to be checked. For the antecedent part of the property, the automaton constructed is a non-deterministic finite automaton (NFA). We introduce a special type of NFA, called a *forward* NFA (a variant of a position NFA [44,31]), that is designed to achieve an effortless derivation of vacuity information. We show how vacuity can be detected by asserting a single invariant condition on the forward NFA already built for the original formula. In addition, we use the same automaton to derive invariant conditions, providing an explanation for a vacuity result: we report exactly which of the events in the regular expression is responsible for the antecedent failure. Thus, unlike other methods that generate auxiliary vacuity formulas and automata, our method detects vacuity with no overhead on the state space.

Our method is implemented in the IBM model checking tools for more than a decade, and serves as the major vacuity detection algorithm of the tools. We present experimental results demonstrating that the overhead on the overall runtime is minimal. A survey conducted among users, confirms that TAF detected by the tool *always* indicates a crucial problem in the property, environment or block under verification, that must be fixed. Moreover, users report that most of the properties are found to suffer from TAF at least once during the development phase, when properties and driving environment are constructed, thus making TAF detection an indispensable debugging aid.

The rest of the paper is organized as follows. The next section gives preliminaries. Section 3 presents forward NFAs that are used for our efficient vacuity detection method. Section 4 defines temporal antecedent failure and its reasons in terms of regular expressions. Section 5 shows how to detect TAF and its reasons using model checking of forward NFAs. Section 6 present experimental results, and Section 7 discusses related work.

2 Preliminaries

2.1 Regular Expressions and Automata

We denote a letter from a given finite alphabet Σ by s. A finite/infinite *word* $s_0 s_1 \cdots$ is denoted by u, v, or w (possibly with subscripts). In the context of temporal logic, $\Sigma = 2^{AP}$ for a set AP of atomic propositions, and each word corresponds to a *computation* $s_0, s_1, \cdots$ over AP. The *concatenation* of u and v is denoted by uv. If u is infinite, then $uv = u$. The empty word is denoted by ϵ, so

that $w\epsilon = \epsilon w = w$. If $w = uv$ we say that u is a *prefix* of w, denoted $u \preceq w$, and that v is a *suffix* of w.

We denote the *length* of a word v by $|v|$. The empty word ϵ has length 0, a finite word $v = (s_0, s_1, s_2 \cdots s_n)$ has length $n+1$, and an infinite word has length ∞. 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 from zero). Let j and k denote integers such that $j \leq k$. We denote by $v^{j\cdot\cdot}$ 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. The *union* of U and V, denoted $U \cup V$, is the set $\{w \mid w \in U \text{ or } w \in V\}$. The *concatenation* of U and V, denoted UV, is the set $\{uv \mid u \in U,\ v \in V\}$. Let $V^0 = \{\epsilon\}$ and $V^k = VV^{k-1}$ for $k \geq 1$. The *Kleene-closure* of V, denoted V^*, is the set $V^* = \bigcup_{k<\omega} V^k$, where ω is the least infinite ordinal. The notation V^+ is used for the set $\bigcup_{0<k<\omega} V^k$. The infinite concatenation of a non-empty set V to itself is denoted V^ω.

Definition 1 (Regular Expressions (REs))

- Let $\mathbb{B}$ be a finite non-empty set, and let b be an element in $\mathbb{B}$. The set of *r*egular 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 r over $\mathbb{B}$, denoted $[\![r]\!]$, is a set of words over an alphabet Σ, which is recursively defined below. We use the function $\mathfrak{L} : \mathbb{B} \to 2^\Sigma$ to map an element of $\mathbb{B}$ to a set of letters from Σ, that is, $\mathfrak{L}(b) \subseteq \Sigma$.

1. $[\![b]\!] = \mathfrak{L}(b)$
2. $[\![r_1{\cdot}r_2]\!] = [\![r_1]\!][\![r_2]\!]$
3. $[\![r_1 \cup r_2]\!] = [\![r_1]\!] \cup [\![r_2]\!]$
4. $[\![r^*]\!] = [\![r]\!]^*$

In the standard definition of regular expressions, $\mathbb{B} = \Sigma$ and $\mathfrak{L}(b) = \{b\}$. In the context of temporal logic, $\mathbb{B}$ is the set of Boolean expressions over a set AP of atomic propositions, and $\Sigma = 2^{AP}$. In this context, $\mathfrak{L}(b) \subseteq \Sigma$ is the set of assignments to the propositions in AP for which the Boolean expression b holds.

Definition 2 (Non-deterministic Finite Automaton) A non-deterministic automaton (NFA) N is a tuple $N = (\Sigma, Q, Q_0, \delta, A)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $A \subseteq Q$ is the set of accepting states.

A run of N on a word $v = s_0, s_1, \cdots, s_n$ is a sequence of states $q_0 q_1 \ldots q_{n+1}$ such that $q_0 \in Q_0$, and for all $i \geq 0$, $(q_i, s_i, q_{i+1}) \in \delta$. The run is *accepting* if $q_{n+1} \in A$. The automaton N accepts the word v if there exists an accepting run of N on v. For every RE r, an NFA N_r can be built, that accepts exactly all the words in $[\![r]\!]$ ([35]).

2.2 Kripke Structures and Temporal Logic

Definition 3 (Kripke structures) 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, and $L : S \to 2^{AP}$ labels each state with the set of atomic propositions that are true in the state.

A computation π in a Kripke structure is a sequence of states $\pi = s_0, s_1, ...$ such that $s_0 \in I$ and $\forall i, (s_i, s_{i+1}) \in R$.

We sometimes look at the parallel composition of two Kripke structures, as defined below (adopted from [16]).

Definition 4 (Parallel Composition of Kripke Structures [16])
Let $K_1 = (S_1, I_1, R_1, L_1)$ and $K_2 = (S_2, I_2, R_2, L_2)$ be Kripke structures over the sets of atomic propositions AP_1 and AP_2 respectively. The parallel composition of K_1 and K_2, denoted $K_1 ||| K_2$ is the Kripke structure $K = (S, I, R, L)$ over $AP = AP_1 \cup AP_2$, defined as follows.

- $S = \{(s_1, s_2) \mid L_1(s_1) \cap AP_2 = L_2(s_2) \cap AP_1\}$.
- $I = (I_1 \times I_2) \cap S$.
- $R = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1 \text{ and } (s_2, t_2) \in R_2\}$.
- $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$.

Note that L is well defined since the definition of S guarantees that L_1 and L_2 agree on the common atomic propositions.

The logic *LTL* is a linear temporal logic [51]. LTL formulas describe languages over the alphabet 2^{AP}, and are constructed from a set AP of atomic propositions using the Boolean operators $\neg$ and $\wedge$, and the temporal operators X ("next time") and U ("until").

Definition 5 (Linear Temporal Logic (LTL)) Formulas of LTL are built from a set AP of atomic propositions:

- Every atomic proposition is an LTL formulas.
- If φ and ψ are LTL formulas then so are: • $\neg\varphi$ • $\varphi \wedge \psi$ • $\mathsf{X}\varphi$ • $[\varphi\ \mathsf{U}\ \psi]$

Additional operators are defined as syntactic sugaring of those above:
• $False = (\varphi \wedge \neg\varphi)$ • $True = \neg False$ • $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$
• $\varphi \rightarrow \psi = \neg\varphi \vee \psi$ • $\mathsf{F}\,\varphi = [True\ \mathsf{U}\ \varphi]$ • $\mathsf{G}\varphi = \neg\mathsf{F}\,\neg\varphi$

The semantics of LTL is defined with respect to computations over AP. For a computation $v \in \Sigma^\omega$, where $\Sigma = 2^{AP}$, the semantics is given as follows.

- $v \models p$ iff $p \in v^0$
- $v \models \neg\varphi$ iff $v \not\models \varphi$
- $v \models \varphi \wedge \psi$ iff $v \models \varphi$ and $v \models \psi$
- $v \models \mathsf{X}\varphi$ iff $v^{1\cdot\cdot} \models \varphi$
- $v \models [\varphi \mathsf{U} \psi]$ iff $\exists k \geq 0$ such that $v^{k\cdot\cdot} \models \psi$, and for every $0 \leq j < k, v^{j\cdot\cdot} \models \varphi$.

Given a Kripke structure M and an LTL formula φ, the model-checking problem is to decide whether all the computations of M satisfy φ. If this is the case, we denoted it by $M \models \varphi$.

2.3 Regular Expression based Temporal Logic

The IEEE standard temporal logics PSL [36, 29] and SVA [37, 19] use a combination of regular expressions and LTL formulas, known as *suffix implication* formulas (also

called *triggers* in the definition of RELTL [18]). A suffix implication formula is of the form $r \models\!\!> \varphi$ where r is a regular expression and φ is either an LTL formula or another suffix implication formula. Intuitively, this formula states that whenever a prefix of a given computation path matches the regular expression r, the suffix of that path must satisfy φ. The formal definition of suffix implication, taken from [36], is given below.

Definition 6 (Suffix Implication Formulas [36]) Let r be a regular expression and φ a temporal logic formula, both over a given set of atomic propositions AP. Let v be a word over 2^{AP}. Then the semantics of suffix implication is defined as follows.

$$v \models r \models\!\!> \varphi \iff \forall j < |v|, \text{ if } v^{0..j} \in [\![r]\!] \text{ then } v^{j+1..} \models \varphi$$

Example 1 The following formula

$$\psi = \{true^* \cdot req \cdot \neg ack^* \cdot ack \cdot \neg abort\} \models\!\!> (\neg retry \ \mathsf{U} \ req)$$

states that if the first *ack* following *req* is not *abort*ed one cycle after *ack*, then it must not be retried (signal *retry* should not hold at least until the next *req* holds).

Note that in the literature, the *overlapping suffix implication* (denoted $\mapsto$) is often used, where $v \models r \mapsto \varphi \iff \forall j < |v|$, if $v^{0..j} \in [\![r]\!]$ then $v^{j..} \models \varphi$. For our work, we use the *non-overlapping* operator ($\models\!\!>$) since we are concerned with consequents that occur strictly later than the precondition.

Note that the majority of formulas written in practice can be effectively transformed into a single suffix implication formula [9]. In particular, the common fragment of LTL and ACTL [43], all RCTL formulas [7] and all the safety formulas in the simple subset of PSL can be linearly translated into suffix implication form. Since the right hand side of the suffix implication operator can be any formula, liveness formulas can be supported in the same manner as well [14].

In the rest of the paper we assume the formula is given as a suffix implication, and we analyze temporal antecedent failure on its regular expression part. Note that the formula could have been written in suffix implication form originally, or transformed to this form by methods like those in [9,14].

3 Positions in RE and Forward NFAs

In order to efficiently detect vacuity, we need the automaton built for a formula to be of a special type. The standard construction of an NFA for a given RE r introduces states in the NFA that are not directly related to r. We use a variant of a *position* NFA [44,31], that keeps a one to one correspondence between states of the automaton and positions of r (see below). We use this correspondence for the purpose of detecting vacuity reasons, as discussed in Section 5. In this section we first define *position*s in RE (Section 3.1), allowing us to define prefixes of an RE and a partial order on positions. In Section 3.2 we define *function*s on positions, that determine what positions follow or precede a given one. Finally, in Section 3.3 we use the position functions to construct a forward NFA.

3.1 Positions in REs

An RE is called *linear* if no letter appears in it more than once [10]. An RE can be linearized by subscripting its letters with unique indexes [10, 44].

Example 2 Let r be the RE $\{a{\cdot}\{\{b^*{\cdot}c\}{\cup}\{d{\cdot}e^*\}\}{\cdot}c{\cdot}e\}$. It can be subscripted 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 a subscripted RE *position*s and save the term *letter*s for the deindexed positions. For example, 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. We use $pos(r)$ to denote the set of positions in a linearized regular expression r. For a position $x \in pos(r)$, we use $\ell(x)$ for the letter that appears in that position of r.
Based on positions, we now define *prefixes* of an RE r as follows.

Definition 7 (Prefixes in RE) Let r be an RE, and x a position in r. We denote by $r_{\lfloor x \rfloor}$ the prefix of r that ends with x, and by $r\lfloor_x$ the prefix of r that ends exactly before x. Formally,

1. $b_{\lfloor x \rfloor} = b$
2. $\{r_1{\cdot}r_2\}_{\lfloor x \rfloor} = \begin{cases} r_{1\,\lfloor x \rfloor} & \text{if } x \in pos(r_1) \\ r_1{\cdot}\{r_2\}_{\lfloor x \rfloor} & \text{otherwise} \end{cases}$
3. $r_1{\cup}r_{2\,\lfloor x \rfloor} = \begin{cases} r_{1\,\lfloor x \rfloor} & \text{if } x \in pos(r_1) \\ r_{2\,\lfloor x \rfloor} & \text{otherwise} \end{cases}$
4. $r^*{}_{\lfloor x \rfloor} = r_{\lfloor x \rfloor}$

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

Example 3 For r_1 defined in Example 2, we have $r_{1\,\lfloor e_5 \rfloor} = \{a_1{\cdot}d_4{\cdot}e_5\}$, $r_1\lfloor_{e_5} = \{a_1{\cdot}d_4\}$, $r_{1\,\lfloor c_6 \rfloor} = \{a_1{\cdot}\{\{b_2^*{\cdot}c_3\}{\cup}\{d_4{\cdot}e_5^*\}\}{\cdot}c_6\}$ and $r_1\lfloor_{c_6} = \{a_1{\cdot}\{\{b_2^*{\cdot}c_3\}{\cup}\{d_4{\cdot}e_5^*\}\}\}$.

The definition of prefixes of REs induces a partial order on positions of an RE r, as we define below.

Definition 8 (Partial Order on Positions) Let x, y be positions in r. We say that y *precedes* x in r, denoted $y \preceq\!\cdot\, x$, if $y \in pos(r_{\lfloor x \rfloor})$.

The $\preceq\!\cdot$ relation is reflexive, since $x \in pos(r_{\lfloor x \rfloor})$. Transitivity and anti-symmetry follow from Definition 7 by induction on the structure of r. We use $y \prec\!\cdot\, x$ to indicate that $y \preceq\!\cdot\, x$ and $y \neq x$.

Example 4 For r_1 defined in Example 2, the partial order is given by: $a_1 \prec\!\cdot\, b_2$, $b_2 \prec\!\cdot\, c_3$, $c_3 \prec\!\cdot\, c_6$, $c_6 \prec\!\cdot\, e_7$, $a_1 \prec\!\cdot\, d_4$, $d_4 \prec\!\cdot\, e_5$ and $e_5 \prec\!\cdot\, c_6$. Note that b_2 and c_3 are not comparable with d_4 and e_5, and thus the order on positions is only a partial one.

Example 5 For r_1 of Example 2, the partial order is given by: $a_1 \prec\!\cdot\, b_2$, $b_2 \prec\!\cdot\, c_3$, $c_3 \prec\!\cdot\, c_6$, $c_6 \prec\!\cdot\, e_7$, $a_1 \prec\!\cdot\, d_4$, $d_4 \prec\!\cdot\, e_5$ and $e_5 \prec\!\cdot\, c_6$. Note that b_2 and c_3 are not comparable with d_4 and e_5, and thus the order on positions is only a partial one.

3.2 Position Functions

A forward NFA for a given RE r is based on the positions of r. To ease its definition we provide several functions on the set of positions $pos(r)$, adapted from [1]: $\mathcal{F}(r)$

- $\mathcal{F}(\emptyset) = \emptyset$ - $\mathcal{F}(\lambda) = \emptyset$ - $\mathcal{F}(x) = \{x\}$ - $\mathcal{F}(r_1 \cup r_2) = \mathcal{F}(r_1) \cup \mathcal{F}(r_2)$ - $\mathcal{F}(n_1 \cdot r_2) = \mathcal{F}(n_1)$ - $\mathcal{F}(s_1 \cdot r_2) = \mathcal{F}(s_1) \cup \mathcal{F}(r_2)$ - $\mathcal{F}(r^*) = \mathcal{F}(r)$	- $\mathcal{N}(x, x) = \emptyset$ - $\mathcal{N}(r_1 \cup r_2, x) = \begin{cases} \mathcal{N}(r_1, x) & \text{if } x \in pos(r_1) \\ \mathcal{N}(r_2, x) & \text{if } x \in pos(r_2) \end{cases}$ - $\mathcal{N}(r_1 \cdot r_2, x) = \begin{cases} \mathcal{N}(r_1, x) & \text{if } x \in pos(r_1) \setminus \mathcal{L}(r_1) \\ \mathcal{N}(r_1, x) \cup \mathcal{F}(r_2) & \text{if } x \in \mathcal{L}(r_1) \\ \mathcal{N}(r_2, x) & \text{if } x \in pos(r_2) \end{cases}$ - $\mathcal{N}(r^*, x) = \begin{cases} \mathcal{N}(r, x) & \text{if } x \in pos(r) \setminus \mathcal{L}(r) \\ \mathcal{N}(r, x) \cup \mathcal{F}(r) & \text{if } x \in \mathcal{L}(r) \end{cases}$
- $\mathcal{L}(\emptyset) = \emptyset$ - $\mathcal{L}(\lambda) = \emptyset$ - $\mathcal{L}(x) = \{x\}$ - $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$ - $\mathcal{L}(r_1 \cdot n_2) = \mathcal{L}(n_2)$ - $\mathcal{L}(r_1 \cdot s_2) = \mathcal{L}(s_2) \cup \mathcal{L}(r_1)$ - $\mathcal{L}(r^*) = \mathcal{L}(r)$	- $\mathcal{P}(x, x) = \emptyset$ - $\mathcal{P}(r_1 \cup r_2, x) = \begin{cases} \mathcal{P}(r_1, x) & \text{if } x \in pos(r_1) \\ \mathcal{P}(r_2, x) & \text{if } x \in pos(r_2) \end{cases}$ - $\mathcal{P}(r_1 \cdot r_2, x) = \begin{cases} \mathcal{P}(r_2, x) & \text{if } x \in pos(r_2) \setminus \mathcal{F}(r_2) \\ \mathcal{P}(r_2, x) \cup \mathcal{L}(r_1) & \text{if } x \in \mathcal{F}(r_2) \\ \mathcal{P}(r_1, x) & \text{if } x \in pos(r_1) \end{cases}$ - $\mathcal{P}(r^*, x) = \begin{cases} \mathcal{P}(r, x) & \text{if } x \in pos(r) \setminus \mathcal{F}(r) \\ \mathcal{P}(r, x) \cup \mathcal{L}(r) & \text{if } x \in \mathcal{F}(r) \end{cases}$

Table 1 Position functions in regular expressions. $\mathcal{F}(r)$ gives the first positions of r, $\mathcal{L}(r)$ gives the last positions, $\mathcal{N}(r, x)$ are the positions immediately following position x in r and $\mathcal{P}(r, x)$ gives the positions that immediately precede x in r. We use r, r_1, r_2 for REs, s_1, s_2 for starred REs and n_1, n_2 for non-starred REs.

is the set of positions that match the *first* letter of some word in $[\![r]\!]$, $\mathcal{L}(r)$ is the set of positions that match the *last* letter of some word in $[\![r]\!]$, $\mathcal{N}(r, x)$ is the set of positions that immediately *follow* position x in a path through r and $\mathcal{P}(r, x)$ is the set of positions that immediately *precede* position x in a path through r.

Before we give the inductive definition of the position functions, we introduce two special regular expressions. The RE λ represents the empty word: $[\![\lambda]\!] = \epsilon$ and the RE $\emptyset$ represents the empty set of words: $[\![\emptyset]\!] = \emptyset$. In addition, we make use of the predicate $\mathcal{E}(r)$ that returns *true* when $\epsilon \in [\![r]\!]$. In such cases, we call r a *starred* RE.

Definition 9 (Starred RE) The predicate $\mathcal{E}(r)$ is recursively defined as follows: $\mathcal{E}(\emptyset) = \mathit{false}$; $\mathcal{E}(\lambda) = \mathit{true}$; $\mathcal{E}(b) = \mathit{false}$; $\mathcal{E}(r_1 \cdot r_2) = \mathcal{E}(r_1) \wedge \mathcal{E}(r_2)$; $\mathcal{E}(r_1 \cup r_2) = \mathcal{E}(r_1) \vee \mathcal{E}(r_2)$; and $\mathcal{E}(r^*) = \mathit{true}$. When $\mathcal{E}(r) = \mathit{true}$, we say that r is a *starred* RE.

The definition of the position functions, given in Table 1, uses the arbitrary REs r, r_1, r_2, the starred REs s_1, s_2 (where $\mathcal{E}(s_1) = \mathcal{E}(s_2) = \mathit{true}$) and the non-starred REs n_1, n_2 (where $\mathcal{E}(n_1) = \mathcal{E}(n_2) = \mathit{false}$). Note that $\mathcal{N}()$ and $\mathcal{P}()$ are not defined for $r = \emptyset$ or $r = \lambda$. This is because λ and $\emptyset$ have no positions.

Example 6 Consider again the RE $r_1 = \{a_1 \cdot \{\{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\}\} \cdot c_6 \cdot e_7\}$ from Example 2. We have that $\mathcal{F}(r_1) = a_1$, $\mathcal{L}(r_1) = e_7$, $\mathcal{N}(r_1, a_1) = \{b_2, c_3, d_4\}$ and $\mathcal{P}(r_1, c_6) = \{e_5, d_4, c_3\}$.

Definition 10 (Paths in REs) A sequence $x_1, x_2, ..., x_n$ of positions in a RE r is said to be a *path* in r if $x_1 \in \mathcal{F}(r)$, $x_n \in \mathcal{L}(r)$ and for every $1 \leq i < n$ we have that $x_{i+1} \in \mathcal{N}(r, x_i)$.

Example 7 For $r_1 = \{a_1 \cdot \{\{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\}\} \cdot c_6 \cdot e_7\}$, the sequences a_1, c_3, c_6, c_7 and a_1, d_4, e_5, c_6, e_7 are path examples. Note that a starred position may or may not be 'skipped' in a path.

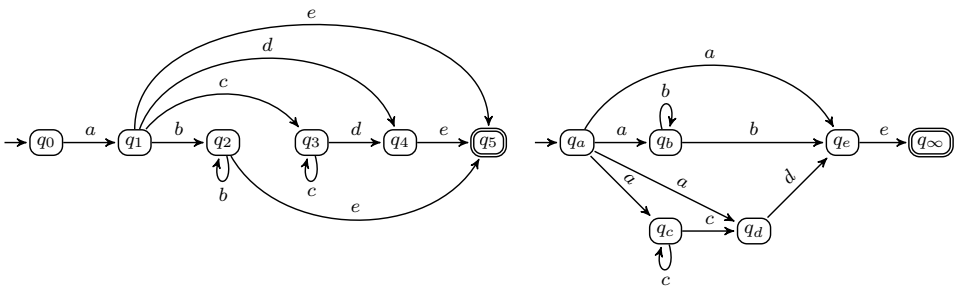

Fig. 1 On the left, an NFA for the RE $r_2 = \{a{\cdot}\{b^* \cup \{c^*{\cdot}d\}\}{\cdot}e\}$ as results form [10,31]. On the right, a forward NFA for the same RE.

3.3 Constructing a Forward NFA

When a position NFA [10,31,44] is constructed for a given linear RE r, a single state q_x is associated with every position x of r, and all the edges *entering* a state q_x are labeled with $\ell(x)$ – the letter in position x. A position NFA has the nice property that if the given RE is linear (without subscriptions), the resulting automaton is deterministic.

We use a variant of a position NFA, which we call a *forward* NFA. Like in [10], we associate a state with every position of r. However, for a state q_x, all the *outgoing* edges are labeled with $\ell(x)$ (rather than entering edges as in a position NFA). While this causes the loss of the deterministic property of a position NFA, it eases the detection of vacuity reasons, as discussed in Section 5.

Definition 11 (Constructing a Forward NFA) Let r be an RE over $\mathbb{B}$. We define $N_r = \langle \mathbb{B}, Q, Q_0, \delta, A \rangle$, where

- $Q = \{q_x \,|\, x \in pos(r)\} \cup \{q_\infty\}$
- $Q_0 = \{q_x \,|\, x \in \mathcal{F}(r)\}$ if $\mathcal{E}(r) = \mathit{false}$ and $Q_0 = \{q_x \,|\, x \in \mathcal{F}(r)\} \cup \{q_\infty\}$ otherwise
- $A = \{q_\infty\}$
- $\delta = \{(q_{x_1}, \ell(x_1), q_{x_2}) \mid x_2 \in \mathcal{N}(r, x_1)\} \cup \{(q_x, \ell(x), q_\infty) \mid x \in \mathcal{L}(r)\}$

The proof of correctness for this construction can be found in [8].

Example 8 Fig. 1 presents two NFAs accepting the RE $r_2 = \{a{\cdot}\{b^* \cup \{c^*{\cdot}d\}\}{\cdot}e\}$. On the left hand side, a position NFA due to [10,31] (the same NFA would be obtained from the algorithm of [44], after removing ϵ-transition and applying determinization). On the right, a forward NFA as described in Definition 11. Note that since r_2 is linear, the position NFA for it is deterministic. In the forward NFA the determinism is lost, but it has the property that all outgoing edges from a state q_x are labeled by $\ell(x)$.

In the context of model checking, an NFA should be a Kripke structure, and thus its transition relation must be total. For that we add another state q_{sink} to Q and the set of transitions

$$\{(q_x, \neg\ell(x), q_{sink}) \mid x \in pos(r)\} \cup \{(q_y, \mathit{true}, q_{sink}) \mid y \in \{\infty, sink\}\}.$$

That is, whenever we reach a state q_x on a computation, and $\ell(x)$ does not hold, we move to the sink state, in which we stay forever. For the totality of the transition relation, we move to the sink state also after visiting the accepting state q_∞.

In order to model check a suffix implication formula $\psi = r \mapsto \varphi$, the forward NFA N_r built for r, is composed with a Büchi automaton built for φ (see [17]). In Section 5 we use N_r to detect temporal antecedent failure of ψ, as well as its reasons.

4 Temporal Antecedent Failure (TAF)

Antecedent failure in propositional logic [4] occurs when a formula ψ is trivially valid because some pre-condition (antecedent) in ψ is not satisfiable in the model. We generalize this notion to *Temporal Antecedent Failure* (TAF), where some future requirement is expected to happen *after* a temporal pre-condition has occurred. For example, in the formula $\mathsf{G}(p \rightarrow \mathsf{X}q)$ we say that p is the temporal antecedent, and if it fails to hold in the model we declare it as a temporal antecedent failure. In the formula $\mathsf{G}(\mathsf{X}p \rightarrow q)$ however, p is not a temporal antecedent since it happens after what is supposed to be its "consequent". As discussed in section 2.3, we work with suffix implication formulas, of the form $r \mapsto \varphi$. In such formulas, the suffix implication operator ($\mapsto$) guarantees that the consequent happens after the antecedent. Moreover, since the antecedent is an RE, we use the partial order on positions to correlate to time. This is what makes suffix implication formulas especially convenient for identifying TAF.

Definition 12 (Temporal Antecedent Failure (TAF)) A regular expression r suffers from TAF in a model M, if for all finite words w in M we have that $w \notin [\![r]\!]$. A suffix implication formula of the form $r \mapsto \varphi$, is said to suffer from TAF in M if r suffers from TAF in M.

Note that φ plays no role in the decision whether $r \mapsto \varphi$ suffers from TAF in M. That is, TAF of $r \mapsto \varphi$ depends solely on r. In the rest of this section we define what are *reasons* for TAF, and what is considered a witness in cases where r does not suffer from TAF in M. The detection of TAF, using forward NFAs and model checking, is presented in Section 5.

4.1 TAF reasons

Let r be an RE that suffers from TAF in a model M. By Definition 12 we know that r is not satisfied by any prefix of a computation in M. Is there more information about r that might be be of interest to the user? Consider the following example.

Example 9 Let ψ be the formula from Example 1, for which the regular expression part is $r_3 = \{true^* \cdot req \cdot \neg ack^* \cdot ack \cdot \neg abort\}$. It suffers from TAF in a model M if there is never a sequence of req, followed by an ack after a finite number of cycles, that is not $abort$ed one cycle after ack. Note that there could be several reasons for r_3 never to hold in M. It could be the case that no reqs are ever given in M, or no acks are given. It could also be the case that in M, all reqs are always $abort$ed.

In Example 9, if no *req*s are ever given, then no *req* is ever followed by *ack*. Since one problem implies the other, we don't want to declare both as reasons. Intuitively, a TAF reason is a position that does not hold when expected, while one of its immediate predecessors does hold when expected.

Definition 13 (TAF Reasons) Let M be a model and r a linear RE. We say that a position x of r is a *reason* for TAF, if x is the first position on a path through r such that $r_{\lfloor x}$ suffers from TAF in M.

Note that a reason is defined also for cases where r as a whole does not suffer from TAF in M. In cases where r does suffer from TAF though, we claim that a reason always exists.

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.*

Proof By a simple induction on the structure of r.

By Definition 13, there can be more than one reason for a TAF result, since different positions can be the first to suffer from TAF on different paths through r. We discuss this in the example below.

Example 10 Let $r_1 = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\} \cdot c_6 \cdot e_7\}$, and let M be a model over the set $AP = \{a, b, c, d, e\}$. Assume that M consists of a single computation $w = s_0, s_1, s_2, s_3, s_4, s_5, s_6, \cdots$, where for $0 \leq i$, $s_i \in \Sigma^{AP}$. Let $s_0 = \{a\}, s_1 = \{b\}, s_2 = \{c\}, s_3 = \emptyset$ and $s_4 = \emptyset$. It is easy to see that r_1 suffers from TAF in M. Note that two positions are reasons for this TAF according to Definition 13: c_6 is the first to suffer from TAF on the path a_1, b_2, c_3, c_6, e_7, and d_4 is the first position to suffer from TAF on the path a_1, d_4, c_6, e_7.

Example 10 demonstrates two different types of reasons. If c_6 is a reason, it implies that r indeed suffers from TAF in M (whether d_4 suffers from TAF or not). This is not the case for d_4: it can be a "reason" for TAF by Definition 13, while r as a whole does not suffer from TAF in M. In Example 10, this could happen if we replaced s_3 and s_4 by $s_3 = \{c\}$ and $s_4 = \{e\}$. That is, the failure of d to hold in s_2 could have been "recovered" had c held in s_3. We thus differentiate between two types of reasons for antecedent failure. If the failure of the position can be "recovered", it is a *secondary* reason. Otherwise it is a *primary* reason.

Definition 14 (Primary and Secondary Reasons) Let r be an RE and x a position in r that is a reason for TAF in M. We say that x is a *primary reason* if for every position y such that $x \prec\cdot\, y$, we have that (1) $r_{\lfloor y}$ suffers from TAF in M, and (2) y is not a reason for TAF in M.
Otherwise, x is a *secondary reason.*

Examining Example 10 in light of Definition 14, we get that c_6 is a primary reason while d_4 is a secondary reason as expected. As another example consider the RE $r_3 = \{a \cdot b^* \cdot c\}$ and assume that, in our model, b and c never appear after a. Then both positions b and c are reasons, c is a primary reason while b is a secondary reason. Recall that we can have a secondary reason when r does not suffer from TAF at all. For example, consider r_3 above and a model where b never occurs after a, but c always holds.

We strengthen Proposition 1 to take into account *primary* reasons.

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

Proof By Proposition 1 there exists a reason for TAF of r in M. Let x be a maximal position (according to the partial order $\preceq_{\cdot}$) that is a reason. Then by Definition 14, x is a primary reason.

Note that there can be more than one primary reasons for TAF. For example, let $r_3 = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5\} \cdot c_6 \cdot e_7\}$ suffer from TAF in a model M, and suppose that c_3 and e_5 are the only reasons for that, then both are primary reasons since both are maximal according to $\preceq_{\cdot}$.

4.2 Witness for Non-TAF Satisfaction

The previous subsection discussed further information (reasons) that can be provided when r is found to suffer from TAF in M. When r does not suffer from TAF it is many times useful to provide a trace to prove this fact. In [5] Beer et al. suggested the term *witness* for a trace demonstrating that the formula does not hold vacuously.

In our case, intuitively, a witness trace should show that no position in r is a reason for TAF. Indeed, by Proposition 1, this is an indication that the formula does not suffer from TAF. Recall however that a "reason" might exist, while the r does not, in fact, suffer from TAF in M. It might be useful to present in addition a witness showing that no secondary reason exists in r.

We thus define two kinds of witnesses. A nonTAF-witness is a trace demonstrating that the r does not suffer from TAF in M. A nonReason-witness is a trace demonstrating that a given position is not a reason for TAF of r in M. A *full witness* is a set of traces demonstrating that no position is a reason in M. Formally,

Definition 15 (Witnesses) Let M be a model, r an RE and x a position in r.

- A computation w in a model M is a *nonTAF-witness* for r in M iff there exists a prefix $u \preceq w$ such that $u \in [\![r]\!]$.
- A computation w in a model M is a nonReason-witness for r in M with respect to x iff it is a nonTAF-witness and there exists a prefix $u \preceq w$ such that $u \in [\![r_{\not x}]\!]$.

A set $W \subseteq M$ is a *full witness for r in M* if for each $x \in pos(r)$, there exists $w \in W$ such that w is a nonReason-witness for r in M with respect to x.

5 Detecting Antecedent Failure and its Reasons

We show how TAF can be detected using model checking. Recall that for the verification of $r \Rightarrow \varphi$, a forward NFA N_r is constructed for r, that accepts exactly the words in $[\![r]\!]$. A key feature of our algorithm is that it does not add auxiliary automata, but rather, derives additional invariant formulas from N_r. We use $at(q)$ for a Boolean predicate stating that N_r is in its state named q. For a position x in r, let q_x be its corresponding state in N_r. We say that q_x is *never reached* in M if $M \,|||\, N_r \models \mathsf{G}(\neg at(q_x))$, and that q_x is *reachable* if $M \,|||\, N_r \not\models \mathsf{G}(\neg at(q_x))$. We say

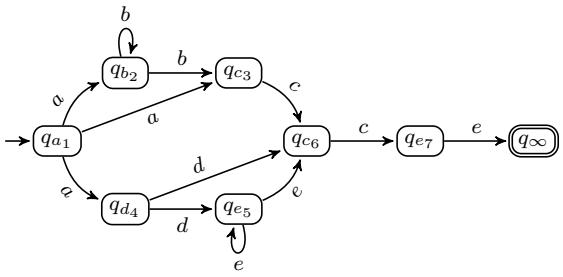

Fig. 2 A forward NFA N_{r_1} for $r_1 = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\} \cdot c_6 \cdot e_7\}$.

that x is *never exercised*, if whenever q_x is reached, $\ell(x)$ does not hold. That is, q_x is never exercised iff $M \,|||\, N_r \models \mathsf{G}\ (at(q_x) \rightarrow \neg \ell(x))$.

As a forward NFA, N_r has a single accepting state q_∞ (see Definition 11). If the Boolean predicate $at(q_\infty)$ holds, it means that the automaton N_r is at its final state. In order to check for TAF of r in a model M we model check the formula $\mathsf{G}(\neg at(q_\infty))$ on the composition of M and N_r.

Proposition 3 *Let M be a model and r an RE.*

$$r \textit{ suffers from TAF in } M \textit{ iff } M \,|||\, N_r \models \mathsf{G}(\neg at(q_\infty))$$

Proof By the construction of N_r, we know that a computation w in M reaches q_∞ if and only if it has a finite prefix that is in $[\![r]\!]$. Thus r suffers from TAF in M (no prefix of a computation of M belongs to $[\![r]\!]$) iff q_∞ is never reached.

5.1 Detecting TAF Reasons

Before we show how TAF reasons can be detected, let us examine the forward NFA N_{r_1} of Fig. 2, constructed for the RE $r_1 = \{a_1 \cdot \{b_2^* \cdot c_3\} \cup \{d_4 \cdot e_5^*\} \cdot c_6 \cdot e_7\}$. Note that a state q_x that corresponds to a position x, is reached *before* $\ell(x)$ is expected to appear. Note further, that all the transitions going out of q_x are labeled with $\ell(x)$. If $\ell(x)$ does not hold in the computation when N_r is in state q_x, the computation "falls off" the automaton (or, in the case of model checking, the computation is trapped in a "sink" state).

Let us examine the prefix $r_{\lfloor x \rfloor}$ of r. Note that for every position $y \in pos(r_{\lfloor x \rfloor})$, we have that $y \preceq\!\!\cdot\ x$. This is because by Definition 7, if position x appears on a branch of a $\cup$ operator, the other branch is cut off in the prefix. Note further that $x \in \mathcal{L}(r_{\lfloor x \rfloor})$ (see Table 1), and that x is the only final position of $r_{\lfloor x \rfloor}$. This is because x is not on a branch in $r_{\lfloor x \rfloor}$, and its letter is not starred (the star, if exists in r, is omitted in $r_{\lfloor x \rfloor}$ according to Definition 7). Let us now consider the forward NFA $N_{r_{\lfloor x \rfloor}}$. According to Definition 11, it has a transition $(q_x, \ell(x), q_\infty)$, and since x is the only position in $\mathcal{L}(r_{\lfloor x \rfloor})$, this transition is the only one leading to q_∞ in $N_{r_{\lfloor x \rfloor}}$. Based on the discussion above, we can state the following lemma.

Lemma 1 *$r_{\lfloor x \rfloor}$ suffers from TAF in M iff $M \,|||\, N_r \models \mathsf{G}\ (at(q_x) \rightarrow \neg \ell(x))$.*

Proof By Proposition 3, $r_{x\not}$ suffers from TAF in M iff $M \,|||\, N_{r_{x\not}} \models \mathsf{G}(\neg at(q_\infty))$. By the discussion above we know that the only way for q_∞ to be reached in $N_{r_{x\not}}$ is through a transition labeled $\ell(x)$ outgoing from the state q_x. Thus for $N_{r_{x\not}}$ we have that $M \,|||\, N_{r_{x\not}} \models \mathsf{G}(\neg at(q_\infty))$ iff $M \,|||\, N_{r_{x\not}} \models \mathsf{G}\ (at(q_x) \rightarrow \neg\ell(x))$ (that is, for q_∞ never to be reached, it must the case that whenever q_x is reached, its letter $\ell(x)$ does not hold). Note that the set of prefixes of computations in M that reach q_x in $N_{r_{x\not}}$, is exactly the same as those that reach q_x in N_r. We thus get that $M \,|||\, N_{r_{x\not}} \models \mathsf{G}\ (at(q_x) \rightarrow \neg\ell(x))$ iff $M \,|||\, N_r \models \mathsf{G}\ (at(q_x) \rightarrow \neg\ell(x))$.

Note that if q_x is never reached, we know that a problem occurred before position x. If q_x is reached but never exercised, it means that x is a reason for TAF.

Proposition 4 *Let M be a model, and r an RE that suffers from TAF in M. Let x be a position in r. Then, x is a* reason of TAF **iff** the following two conditions hold:

- $M \,|||\, N_r \not\models \mathsf{G}(\neg at(q_x))$
- $M \,|||\, N_r \models \mathsf{G}(at(q_x) \rightarrow \neg\ell(x))$

Proof By Definition 13, for x to be a reason, two conditions must hold: 1. $r_{x\not}$ should suffer from TAF in M, and 2. either $\mathcal{P}(r, x) = \emptyset$ or there exists a position $x' \in \mathcal{P}(r, x)$ such that $r_{x'\not}$ does not suffer from TAF in M. By Lemma 1, the first condition happens if and only if $M \,|||\, N_r \models \mathsf{G}(at(q_x) \rightarrow \neg\ell(x))$. It is left to be shown that the second condition of Definition 13 happens if and only if q_x is reachable, as this would entail the first item of our proposition. If $\mathcal{P}(r, x) = \emptyset$ then q_x is the initial state and is thus reachable. If there exists $x' \in \mathcal{P}(r, x)$ where $r_{x'\not}$ does not suffer from TAF, then by Lemma 1, $M \,|||\, N_r \not\models \mathsf{G}(at(q_{x'}) \rightarrow \neg\ell(x'))$, meaning that there exists a reachable state in M on which $at(q_{x'})$ and $\ell(x')$ hold together. Since x' is a predecessor of x, we get by the construction of N_r that one of the transitions outgoing from $q_{x'}$ (all of which are labeled with $\ell(x')$) leads to q_x. Therefore q_x is reachable. For the other direction, assume that q_x is reachable. This can happen in two ways. Either q_x is one of the initial states (and then $\mathcal{P}(r, x) = \emptyset$), or q_x is reached from a state q'_x, where x' is a predecessor of x. Since N_r is a forward NFA, all transitions from x' are labeled with $\ell(x')$. We thus know that $at(q_{x'}) \wedge \ell(x')$ hold in some reachable state of M. Thus $M \,|||\, N_r \not\models \mathsf{G}(at(q_{x'}) \rightarrow \neg\ell(x'))$, and by Lemma 1, $r_{x'\not}$ does not suffer from TAF in M.

By an additional formula we can also distinguish between primary and secondary reasons. This time the invariant formula states that there does not exist a successor of position x that is reachable.

Proposition 5 *Let M be a model, r an RE and x a position in r that is a reason of TAF. Then, x is a primary reason of TAF iff* $M \,|||\, N_r \models \mathsf{G} \bigwedge_{x \prec\cdot y} \neg at(q_y)$.

Proof $\Rightarrow$ Let x be a primary reason of TAF. By Definition 14, for every y such that $x \prec\cdot y$, we have that $r_{y\not}$ suffers from TAF in M and y is not a reason for TAF. Since $r_{y\not}$ suffers from TAF, we know by Lemma 1, that $M \,|||\, N_r \models \mathsf{G}\ (at(q_y) \rightarrow \neg\ell(y))$. Since y is not a reason, one of the conditions of Proposition 4 must not hold. Since the second condition holds ($M \,|||\, N_r \models \mathsf{G}\ (at(q_y) \rightarrow \neg\ell(y))$), it must be the case that the first condition does not hold. Thus $M \,|||\, N_r \models \mathsf{G}\ (\neg at(q_y))$ as required.

$\Leftarrow$ Let x be a reason for TAF and assume that for every y such that $x \prec\cdot\, y$ we have $M \,|||\, N_r \models \mathsf{G}(\neg at(q_y))$. Then by Proposition 4, every such y is not a reason, since the first condition of Proposition 4 does not hold. Note that if $\mathsf{G}(\neg at(q_y))$ holds then $\mathsf{G}(at(q_y)\rightarrow\neg\ell(y))$ also holds, since the precondition is always *false*. Thus by Lemma 1, we have that $r_{\lfloor y \rfloor}$ suffers from TAF in M. By Definition 14, x is a primary reason of TAF.

We emphasize again that all formulas that we use for detecting TAF are invariant formulas over the automaton already built for verifying the original property. They can thus enjoy efficient model checking algorithms, without the need to build additional automata.

5.2 Generating Witnesses

The generation of witnesses uses the model checker ability to generate counterexamples. In order to provide a nonTAF-witness we ask the model checker to find a counterexample to the claim that the accepting states of N_r are never visited. In order to provide a nonReason-witness for a given position x we ask the model checker to find a counterexample to the claim that in addition, whenever q_x is visited, $\ell(x)$ does not hold. The following proposition states this formally.

Proposition 6 *Let M be a model, r an RE , x a position in r and N_r a forward NFA for r.*

- *A counterexample for $\mathsf{G}(\neg at(q_\infty))$ in $M \,|||\, N_r$ is a nonTAF-witness for r in M.*
- *A counterexample for $\mathsf{G}(at(q_x)\rightarrow\neg\ell(x)) \vee \mathsf{G}(\neg at(q_\infty))$ in $M \,|||\, N_r$ is a nonReason-witness for r in M with respect to x.*

Proof

- Let w be a counterexample for $\mathsf{G}\,(\neg at(q_\infty))$ in $M \,|||\, N_r$. Thus w satisfies $\mathsf{F}\,(at(q_\infty))$. That is, there exists $k \geq 0$ such that $w^k \models at(q_\infty)$. Since N_r accepts r, we have that $w^{0..k} \in [\![r]\!]$. Therefore w is a nonTAF-witness for r in M.
- Let w be a counterexample for $\mathsf{G}\,(at(q_x)\rightarrow\neg\ell(x)) \vee \mathsf{G}\,(\neg at(q_\infty))$ in $M \,|||\, N_r$. Since w is a counterexample for $\mathsf{G}\,(\neg at(q_\infty))$ we know by the first item above that w is a nonTAF-witness for r in M. Since w is a counterexample for $\mathsf{G}\,(at(q_x)\rightarrow\neg\ell(x))$ we know that $w \models \mathsf{F}\,(at(q_x) \wedge \ell(x))$. Thus there exists $k \geq 0$ such that $w^k \models at(q_x)\wedge\ell(x)$. By the construction of N_r we have that $w^{0..k} \in [\![r\lfloor_x]\!]$ and $\ell(x) \in w^k$. We get that $w^{0..k+1} \in [\![r_{\lfloor x \rfloor}]\!]$ and thus w is a nonReason-witness for r in M.

5.3 Procedures for Detecting TAF, its Reasons and Generating Witnesses

Procedure **FindTafAndGenerateWitness**, presented in Fig. 3, receives as input 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 6.

Algorithm 1: FindTafAndGenerateWitness()

```
1  Input: model M; RE r;
2  Output: set PrimaryList; set SecondaryList; trace WitnessTrace;
3  PrimaryList := ∅;
4  SecondaryList := ∅;
5  FindTafReasons(M,r,∞,true,PrimaryList,SecondaryList,∅);
6  if PrimaryList = ∅ then
7  |  WitnessTrace := ProduceCounterExample(G¬at(q_∞));
8  end
```

Fig. 3 A procedure for finding all reasons and producing a witness.

Algorithm 2: FindTafReasons(model M,RE r, position x, Bool primary, set PrimaryList, set SecondaryList, set VisitedPos)

```
1  Bool unreachable, unexercised;
2  if x ∈ VisitedPos then
3  |  return;
4  end
5  VisitedPos := VisitedPos ∪ {x}
6  unreachable := M ⊨ G¬(at(q_x))
7  if unreachable then
8  |  foreach y ∈ P(r,x) do
9  |  |  FindTafReasons(M,r,y,primary,PrimaryList,SecondaryList,VisitedPos);
10 |  end
11 else
12 |  unexercised := M ⊨ G(at(q_x) → ¬ℓ(x))
13 |  if unexercised then
14 |  |  if primary then
15 |  |  |  PrimaryList := PrimaryList ∪ {x}
16 |  |  else
17 |  |  |  SecondaryList := SecondaryList ∪ {x}
18 |  |  end
19 |  end
20 |  foreach y ∈ P(r,x) do
21 |  |  FindTafReasons(M,r,y,false,PrimaryList,SecondaryList,VisitedPos);
22 |  end
23 end
```

Fig. 4 Finding Primary and Secondary Reasons.

Procedure **FindTafReasons**, given in Fig. 4, is a recursive procedure that receives as input 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 their names. In the initial call to this procedure the *primary* flag is turned on. The position sent in the initial call is ∞ which is a 'virtual' position representing the auxiliary state added in the construction of the forward NFA N_r (see section 3.3). We define $\ell(\infty) = false$.

The procedure first checks that the given position x was not visited before. If x was visited it returns (line 3), and 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 (by Proposition 4) x is not a reason. The procedure thus calls

itself recursively for each of x's immediate predecessors (lines 8-10). If position x is reachable, the procedure checks whether it is "exercised" — i.e. whether the corresponding letter holds when expected (line 12). If x is not exercised then (by Proposition 4) a reason is found. The position is inserted to either *PrimaryList* or *SecondaryList* according to the value of the *primary* flag (lines 14-18). The procedure proceeds with recursive calls to the immediate predecessors with the flag *primary* turned off (line 20-22). Proposition 7 below states the correctness of the procedure.

Proposition 7 *Let M be a model and r an RE. Let PrimaryList, SecondaryList and WitnessTrace be the output of the procedure* ***FindTafAndGenerateWitness*** *on the inputs r and M. Then*

1. *The formula suffer from TAF in M iff $PrimaryList \neq \emptyset$.*
2. *PrimaryList holds the set of positions that are a primary reason of TAF.*
3. *SecondaryList holds the set of positions that are a secondary reason of TAF.*
4. *If $PrimaryList = \emptyset$ then WitnessTrace holds a nonTAF-witness for r in M.*

Proof 1. Recall that the first call to the algorithm is with the position ∞ and *primary=true*. If r does not suffer from TAF, then q_∞ is both reachable and exercised (since $\neg(\ell(\infty)) = true$). Thus nothing is entered to PrimaryList, and future calls to **FindTafReasons** would have the primary flag turned off. For the other direction, if r does suffer from TAF then by Proposition 1 there exists a position that is a reason. By item (2) below, PrimaryList will not be empty.

2-3. Note that the procedure visits a position x (i.e. $x \in VisitedPos$) only if some $y \succ x$ was visited before. Note also that if a position is visited, then all of its immediate predecessors will be visited (from the calls in either line 9 or line 21). Thus by recursion all positions are eventually visited. Suppose x is a reason. Then by Proposition 4, x is reachable but not exercised. Thus when visited it will be detected as such (in lines 6,12) and so it will be added to either *PrimaryList* (line 15) or *SecondaryList* (line 17). If it is not a reason, it is either unreachable or reachable but exercised and so it will not be added to either list.
Suppose x is primary. Then by Proposition 5, all its successors $y \succ x$ are unreachable. Therefore it will be called from line 9 with $primary = true$, and thus it will be added to *PrimaryList*. If x is secondary, then there exists a position $y \succ x$ such that y is a reachable. By simple induction on the structure of the RE, it can be shown that exists a reachable position z on every path from x to ∞. Therefore, no matter which path was actually tracked by the procedure, x will be visited after a reachable z was. When z is visited *primary* is set to *false*. Once *primary* is *false* it remains that way. Thus, when visiting x it will be add to *SecondaryList*.

4. Follows directly from line 7 of **FindTafAndGenerateWitness** and the first item of Proposition 6.

6 Experience and Experimental Results

Our method is implemented in the IBM model checking tools for over a decade, and serves as the major vacuity detection algorithm of the tools. We discuss the user

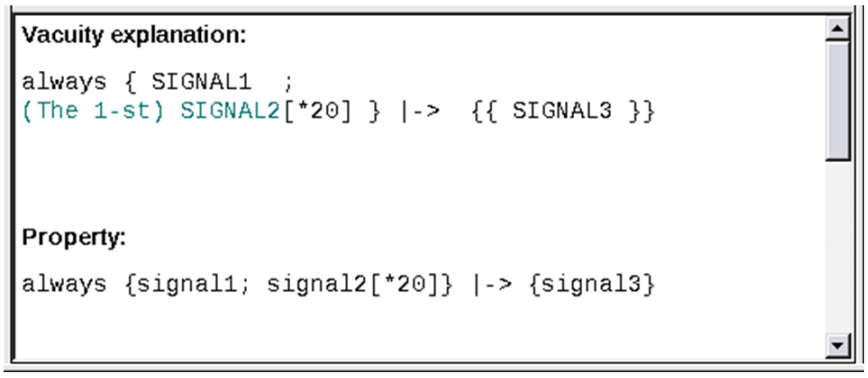

Fig. 5 A screen capture of vacuity explanation. The first occurrence SIGNAL2 (colored light blue) is the one that never occurs when expected.

experience with TAF detection below, and in Section 6.2 we present experimental results demonstrating that detecting reasons for TAF adds only a minor burden on the overall runtime.

6.1 User Experience

In the IBM model checker *RuleBase SixthSense Edition* (RBSXS), vacuity detection is activated by default in every run, while vacuity explanation (reasons) is optional and should be manually selected. Fig. 5 shows a screen capture from the tool, demonstrating a vacuity result with explanation. The formula (with names of signals omitted for confidentiality reasons), required that a sequence where SIGNAL1 holds in the initial state, followed by 20 consecutive states where SIGNAL2 holds, must be followed by SIGNAL3 being active in the last state of the sequence. This formula was found to be vacuous in the model (the 21 long prefix of the formula never holds); the vacuity explanation mechanism pointed to the problem: SIGNAL2 was never active one state after SIGNAL1.

When verifying a new design model, the model checking process consists of several phases. The first of those is the development phase, where properties and driving environment (possibly different for each property or a subset of properties) are being built. TAF detection is crucial in this phase, as an immediate feedback on the quality of the newly constructed environment and properties. A survey conducted among users of RBSXS, reveals that (1) the vast majority of the PSL properties verified are found to suffer from TAF at least once during the development phase, before the property and environment stabilize; (2) a TAF result *always* indicates a problem in the property or driving environment, and sometimes in the model itself; (3) a TAF result is never ignored and always investigated until fixed.

TAF detection is of a special importance when there is a need to reduce the size of the model under verification. While model checking technology is rapidly advancing and can be applied to larger and larger designs, it is commonly the case that the design block under verification is too large to fit into the model

checker as a whole. Thus, the verification engineer typically abstracts away some of the environment behaviors, in order to reach a manageable sized model. Such an abstraction might sometimes mistakenly eliminate important behaviors, causing a property to hold in the simplified model, only because the needed conditions are never given. TAF detection is a necessity in such cases.

6.2 Experimental Results

We conducted experiments to asses the overhead of detecting TAF reasons over regular TAF detection. The experiments included three real life hardware blocks, from a recent microprocessor design. The *Arbiter* is a component responsible of arbitrating between three different requesters that can each send several types of requests to a shared resource, and have a limit on the amount of requests not answered by the resource. The *Data Controller* is a component inside a larger unit, that is responsible for writing and reading data to a neighboring unit; it has to follow a transformation protocol and has to manage, reply to and arbitrate many requesters from its unit to the neighboring unit. The *Memory Controller* is a large design that controls access to a shared memory and follows a coherency model. It can serve a large number of requests and requesters, that vary in their types.

Table 2 presents our results of detecting vacuity and vacuity explanation using RBSXS, run on an AMD machine, with 64 cores, 4×1Ghz each, and 256 GB of RAM. We shortly explain how the tool works, to ease the understanding of the results. In RBSXS, a model checking problem, consisting of a model, its environment and a set of formulas to be verified, is taken through a sequence of reductions and transformations, that can either solve the property, or find and apply reductions to the existing model [3,47]. This transformation sequence is either user-defined, or selected by an expert system based on an internal heuristic. In our experiment we used the expert system on all of the properties. The run time reported in Table 2 is the total run time of the solving sequence of transformations; the solving engine (the right-most column of the table) is the last transformation that was used, the one that was able to solve the property.

Several solving engines are reported in the table. The *Boolean reduction* engines (BRN) [45,46] include common reduction techniques such as cone of influence, constant propagation and others. The *redundancy removal* engine (EQV) [13], uses equivalence checking techniques to detect and reduce pairs of sequentially equivalent signals. Note that while the above engines are used mainly in order to reduce the size of the model, they might sometimes solve the property. For example, while attempting constant propagation, the BRN engine might discover that the property is equal to a constant zero. The *bounded model checking* engine (BMC) consists of the original work of Biere et al. [11] that performs well when an error exists, combined with the work of Sheeran et al. [57], that is good for proving a formula correct. The *induction generalization* engine (IC3) is the one introduced by Bradley in [15], which is the best state-of-the-art proving engine known today. While other engines exist in the tool, they did not play a significant role in solving our problems, although some were used for reducing the model size.

The left-most column of the table provides the name of each block, and the size, in terms of state variables, of the original block. For each block we present four runs, on four different formulas, all of which are vacuously satisfied in the

Model Name & Size (Vars)	Vars ABR.	Prop. Len.	W. Vac. Reasons	WO. Vac. Reasons	Incr. due to Reasons	Solving Engine
Memory Cntr 29943	2993	2	288	184	104	IC3
	5230	5	800	721	79	BMC
	N/A	4	254	78	176	BRN
	10512	3	1138	1068	70	IC3
Arbiter 468	211	6	14	14	0	IC3
	176	3	13	13	0	IC3
	104	41	65	35	30	EQV
	304	3	30	25	5	IC3
Data Cntr 3901	401	21	53	49	4	BMC
	402	21	83	64	19	BMC
	N/A	3	79	27	52	BRN
	523	4	377	301	76	IC3

Table 2 Vacuity Explanation Results.

model. The second column of the table presents the size of the model (number of state variables) after Boolean reductions are performed. Note that the number of state variables after Boolean reduction differs between runs of the same block; this is because Boolean reductions depend heavily on the formula to be verified, as well as on the environment provided for the run. In two cases in the table, no 'after reduction' size is given and we specify N/A instead. In those cases, the first iteration of the Boolean reduction engines [45,46] was capable of solving the problem, and the run terminated already at that stage. The third column of the table gives the length of the property to be verified. The length of the property is of interest, since there is roughly a one-to-one correlation with the size of the generated automaton, and with the number of vacuity explanation formulas produced.

Columns 4 and 5 of the table are the important ones, listing the runtime (in seconds) of vacuity detection, with and without explanations. Column 6 calculates the difference in runtime between the runs. It can be seen that vacuity explanation adds only a small burden over the runtime of the model checking procedure. This is true also in cases where many vacuity explanation formulas are produced. The reason is that, as previously mentioned, the explanation formulas produced are defined over the automaton built for the original property, and are thus similar in nature. When using the IC3 as well as the BMC engine, the run performs a variant of a breadth first search, looking for a fix point in the combination of model and formulas. The vacuity explanation formulas are usually solved early during the run, since they are defined on parts of the automaton that are reached earlier in the search. A small cost is still paid for checking multiple formulas in the same run.

Note that the length of a property is not an indication for the complexity of its model checking procedure. For example, in the Data Cntr model, a property of length 21 takes much shorter to model check than one of length 4. The automaton representing the property, which is added to the model, is very small – it is linear in the size of the property, thus negligible compared to the size of the model. The complexity of the model checking procedure is primarily influenced by the complexity of the model under verification, which varies between runs of the same model. The complexity of the model depends mainly on the variables referenced

in the property (determining the *cone* of variables needed for verification) as well as the driving environment.

7 Related Results

Vacuity in model checking was introduced by Beer et al. in [5,6]. According to their definition, φ is said to be vacuous in a model M if $M \models \varphi$, and there exists a subformula ψ of φ that does not affect the value of φ in M. They showed how vacuity can be easily detected for formulas in a subset of ACTL. Different research directions in this area have been investigated, which we describe below.

7.1 Definitions and Languages

Kupferman and Vardi [41,42] refined the original definition, and extended the supported language to CTL*. They defined φ to be vacuous in a model M if $M \models \varphi$ and there exists an occurrence of a subformula ψ of φ such that $M \models \varphi[\psi \leftarrow \bot]$ (where $\bot$ is **false** if this occurrence of ψ in φ has a positive polarity, and **true** otherwise). This definition is known in the literature as *formula vacuity*. Dong et al. in [28] extended vacuity detection to the μ-calculus. Gurfinkel and Chechik in [34] proposed the extension of formula vacuity detection to existential CTL formulas that *fail* to hold in a model. They posed the vacuity detection problem as a multi-valued model checking problem.

Armoni et al. in [2] proposed a new definition of vacuity called *trace vacuity*. They claimed that when a subformula occurs in multiple polarities in φ, a problem may exist that cannot be detected by formula vacuity. For example, $\varphi = \mathsf{G}\,(p \vee \neg p)$ is a formula that should be found vacuous in every model, but replacing any of the occurrences of p with $\bot$ may make the formula fail in a model, thus declaring the formula non-vacuous. In the definition of [2], a fresh variable x is introduced, and all occurrences of a sub-formula ψ are simultaneously replaced by x in φ. Vacuity is declared if the new formula holds for all possible values of x, denoted $M \models \forall x \varphi[\psi \leftarrow x]$. Note that trace vacuity does not subsume formula vacuity. For example, the formula $\varphi = \mathsf{G}\,(p \rightarrow \mathsf{X}(p))$ is formula-vacuous in a model M where p is never active, but it is not trace-vacuous in M.

Gurfinkel and Chechik in [33] extended trace vacuity to the branching setting and called the enhanced notation *robust vacuity*. They showed that detecting robust vacuity is exponential for CTL, and double-exponential for CTL*. Bustan et al. in [18] showed that detecting trace vacuity for an extension of LTL with a regular layer (known as RELTL), involved an exponential blow-up in addition to the standard exponential blow-up for LTL model checking.

In [55], Samer and Veith propose another extension to vacuity definitions, called "Parameterized Vacuity". They claim that vacuity should be declared when φ is too weak for M. That is, when a strict strengthening of φ holds in M. For example, the formula $\mathsf{G}\,\mathsf{F}\,p$ should be declared vacuous in a model where $\mathsf{G}\,p$ holds. Since too many possible stronger formulas exist, they proposed that the user should be the one to list the strengthening formulas to be checked. In [56], Samer and Veith elaborate more on strengthening formulas, calling them "vacuity grounds". They

suggest to check all strengthening formulas of size bounded by some k. [56] gives also a useful comparison between formula, trace and robust vacuity.

Instead of extending the definitions and languages, resulting in more complex vacuity detection procedures, our work examines restrictions to the definition and language, that results in a very efficient vacuity detection algorithm. We note again that TAF is a type of *formula* vacuity.

7.2 Different Algorithms

Purandare and Somenzi in [52] showed how the information on formula vacuity for CTL can be gathered from intermediate results when performing symbolic model checking using BDDs. Dong et al. in [28] used a similar method for vacuity detection for the μ-calculus. Namjoshi in [49] suggested to examine deductive proofs [48, 50], generated by some tools when φ is satisfied in M. He called the type of vacuity that could be detected this way *proof vacuity*, and showed that if such a proof is given, the detection of vacuity is easier than using other methods. Simmonds et al. in [58] applied a similar idea in the context of bounded model checking (BMC [12]), where a model checking problem is translated into a SAT instance. They analyzed the resolution proofs generated by a SAT solver, searching for *trace* vacuity. They defined 3 different vacuity types that can be detected this way.

TAF falls under the definition of Namjoshi's proof vacuity, and can thus be detected using his method. Since Simmonds et al. search for *trace* vacuity, their method can be applied for detecting TAF only when the two definitions coincide.

7.3 Mutual Vacuity

Gurfinkel and Chechik in [34] introduced the notion of *mutual vacuity* (of formula vacuity), where $\perp$ is assigned to more than one sub-formula occurrence. Chockler et al. in [21] and later Purandare et al. in [53] showed that the formulas obtained when assigning $\perp$ to a subset of sub-formula occurrences, form a lattice. Chockler et al. then suggest the use of Lattice Automata [40] to detect such vacuities, while Purandare et al. propose an abstraction refinement algorithm for the task, strengthening the formula too much, and then using the counterexample to refine it.

While our work does not directly deal with mutual vacuity, we note that, if a vacuity reason is found in a primary position x, then the formula is "mutually vacuous" in all the position combinations that are greater than x. This is because if x is the reason, no position after x can have any influence on the satisfaction of the formula in the model. Thus assigning $\perp$ to any subset of them will result in a formula that is satisfied in the model.

7.4 Vacuity without a Model

Some recent works have considered checking vacuity of a set of formulas without the presence of a model. Chockler and Strichman [24, 25] suggest a series of checks

that can be applied to a set of formulas before model checking takes place. Thus, formulas can be found to contradict each other, or imply one another, in which case some of the model checking runs can be avoided. Fisman et al. [30] consider vacuity in the context of *property based design*, where a model is generated from given specifications. They define *inherent vacuity* to be the case where a formula is equivalent to a strengthening of itself according to some vacuity definition. For example, the formula $\varphi = \mathsf{F}\, p \vee \mathsf{X} p$ is equivalent to $\varphi[\mathsf{X} p \leftarrow \bot] = \mathsf{F}\, p$. [30] presents a framework for inherent vacuity, where different parameters such as vacuity type, polarity and system type are considered.

We note that TAF can occur without a model if an event in one of the primary positions is equivalent to **false**. In such a case, the formula is satisfied vacuously in every model, which comply with the definition of inherent vacuity.

7.5 Vacuity and Coverage

The relationship between coverage [22, 23] and vacuity has been examined by Kupferman [38] and Kupferman et al. [39]. They show that vacuity detection, coverage, and also fault tolerance can be presented using a theory of mutation. They define both the system and specification as a circuit, and a mutation is any change to the circuit. They show some properties of mutations such as duality, inverse and monotonicity.

This work is orthogonal to ours. We deal with vacuity only, where mutations are applied to the formula rather than to the model.

7.6 Other

As mentioned before, Chechik in et al. in [20] observe, like us, that not all vacuities detected according to existing definitions are considered a problem by the users. They propose to declare as vacuous formulas that depend only on the environment for their satisfaction. Namjoshi in [49] claims that the vacuity of formulas that are not *proof vacuous* is debatable. We claim that temporal antecedent failure is a type of vacuity that is never debatable, and propose to search for other types "on demand" only.

Finally, Große et al. in [32] search, like us, for the reasons of vacuity in antecedent debugging. Their paper aims to solve a problem encountered when using a special BMC method [59]. In this method, the behavior of the environment is defined as an antecedent of the BMC formula, and can thus get to be very complicated. The paper deals with the analysis of a contradicting *propositional* formula, which is the antecedent of the description of the model, in contrast to our work, that searches for reasons in a temporal formula.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.
2. R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection in linear temporal logic. In Proc. 15^{th} Intl. Conference on Computer Aided Verification (CAV'03), Lect. Notes in Comp. Sci.

3. J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *ICCD*, 2006.
4. D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Design Automation Conference*, pages 596–602, 1994.
5. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc.* 9^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1254, pages 279–290. Springer-Verlag, 1997.
6. 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.
7. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc.* 10^{th} *International Conference on Computer Aided Verification (CAV'98)*, LNCS 1427, pages 184–194. Springer-Verlag, 1998.
8. S. Ben-David, D. Fisman, and S. Ruah. Automata construction for regular expressions in model checking, June 2004. IBM research report H-0229.
9. 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.
10. G. Berry and R. Sethi. From regular expression to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.
11. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
12. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, 1999.
13. P. Bjesse and K. Claessen. Sat-based verification without state space traversal. In *FMCAD*, pages 372–389, 2000.
14. M. Boule and Z. Zilic. Efficient automata-based assertion-checker synthesis of SEREs for hardware emulation. In *ASP-DAC*, pages 324–329, 2007.
15. A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.
16. R. E. Bryant, P. Chauhan, E. M. Clarke, and A. Goel. A theory of consistency for modular synchronous systems. In *FMCAD*, pages 486–504, 2000.
17. D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical Report MCS05-04, The Weizmann Institute of Science, May 2005.
18. D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and Y. Vardi. Regular vacuity. In *13th International Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, volume 2725, pages 191–206, 2005.
19. E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny. *The Power of Assertions in SystemVerilog*. Springer, 2010.
20. M. Chechik, M. Gheorghiu, and A. Gurfinkel. Finding environment guarantees. In *FASE*, pages 352–367, 2007.
21. H. Chockler, A. Gurfinkel, and O. Strichman. Beyond vacuity: Towards the strongest passing formula. In *FMCAD*, pages 1–8, 2008.
22. H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for formal verification. *STTT*, 8(4-5):373–386, 2006.
23. H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. *Formal Methods in System Design*, 28(3):189–212, 2006.
24. H. Chockler and O. Strichman. Easier and more informative vacuity checks. In *MEMOCODE*, pages 189–198, 2007.
25. H. Chockler and O. Strichman. Before and after vacuity. *Formal Methods in System Design*, 34(1):37–58, 2009.
26. 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.
27. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.
28. Y. Dong, B. Saran-Starosta, C. Ramakrishnan, and S. A. Smolka. Vacuity checking in the modal mu-claculus. In *Proceeding of AMAST'02*, volume 2422, pages 147–162, September 2002.
29. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
30. D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Y. Vardi. A framework for inherent vacuity. In *Haifa Verification Conference*, pages 7–22, 2008.

31. V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1953.
32. D. Große, R. Wille, U. Kühne, and R. Drechsler. Contradictory antecedent debugging in bounded model checking. In *ACM Great Lakes Symposium on VLSI*, pages 173–176, 2009.
33. A. Gurfinkel and M. Chechik. Extending extended vacuity. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, pages 306–321, Austin, Texas, November 2004.
34. A. Gurfinkel and M. Chechik. How vacuous is vacuous? In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 451–466, Barcelona, Spain, March 2004.
35. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1979.
36. IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850™-2010.
37. IEEE Standard for SystemVerilog — unified hardware design, specification, and verification language, Annex F. IEEE Std 1800™-2009.
38. O. Kupferman. Sanity checks in formal verification. In *CONCUR*, pages 37–51, 2006.
39. O. Kupferman, W. Li, and S. A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *FMCAD*, pages 1–9, 2008.
40. O. Kupferman and Y. Lustig. Lattice automata. In *VMCAI*, pages 199–213, 2007.
41. O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 82–96, 1999.
42. O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, February 2003.
43. M. Maidl. The common fragment of CTL and LTL. In *IEEE Symposium on Foundations of Computer Science*, pages 643–652, 2000.
44. R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, EC-9(1):38–47, 1960.
45. H. Mony, J. Baumgartner, A. Mishchenko, and R. K. Brayton. Speculative reduction-based scalable redundancy identification. In *DATE*, pages 1674–1679, 2009.
46. H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. Exploiting suspected redundancy without proving it. In *DAC*, pages 463–466, 2005.
47. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. In *FMCAD*, pages 159–173, 2004.
48. K. S. Namjoshi. Certifying model checkers. In *CAV*, pages 2–13, 2001.
49. K. S. Namjoshi. An efficiently checkable, proof-based formulation of vacuity in model checking. In *CAV04*, pages 57–69, July 2004.
50. D. Peled, A. Pnueli, and L. D. Zuck. From falsification to verification. In *FSTTCS*, pages 292–304, 2001.
51. A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.
52. M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Computer Aided Verification (CAV02)*, pages 485–499, July 2002.
53. M. Purandare, T. Wahl, and D. Kroening. Strengthening properties using abstraction refinement. In *DATE*, pages 1692–1697, 2009.
54. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, 1982.
55. M. Samer and H. Veith. Parameterized vacuity. In *FMCAD*, pages 322–336, 2004.
56. M. Samer and H. Veith. On the notion of vacuous truth. In *LPAR*, pages 2–14, 2007.
57. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, pages 108–125, 2000.
58. J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik. Exploiting resolution proofs to speed up LTL vacuity detection for BMC. In *FMCAD*, pages 3–12, 2007.
59. K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. Cost-efficient block verification for a umts up-link chip-rate coprocessor. In *Design, Automation and Test in Europe Conference and Exposition (DATE'04)*, pages 162–167, February 2004.