Enhanced Vacuity Detection in Linear Temporal Logic

Roy Armoni[1], Limor Fix[1], Alon Flaisher[1,2], Orna Grumberg[2], Nir Piterman[1], Andreas Tiemeyer[1], and Moshe Y. Vardi[4⋆]

[1] Intel Design Center, Haifa.
(roy.armoni,limor.fix,alon.flaisher)@intel.com
(nir.piterman,andreas.tiemeyer)@intel.com
[2] Technion, Israel Institute of Technology. orna@cs.technion.ac.il
[3] Rice University. vardi@cs.rice.edu

Abstract. One of the advantages of temporal-logic model-checking tools is their ability to accompany a negative answer to a correctness query with a counterexample to the satisfaction of the specification in the system. On the other hand, when the answer to the correctness query is positive, most model-checking tools provide no witness for the satisfaction of the specification. In the last few years there has been growing awareness of the importance of suspecting the system or the specification of containing an error also in cases where model checking succeeds. In particular, several works have recently focused on the detection of the *vacuous satisfaction* of temporal logic specifications. For example, when verifying a system with respect to the specification $\varphi = G(req \rightarrow F grant)$ ("every request is eventually followed by a grant"), we say that φ is satisfied vacuously in systems in which requests are never sent. Current works have focused on detecting vacuity with respect to subformula occurrences. In this work we investigate vacuity detection with respect to subformulas with multiple occurrences.
The generality of our framework requires us to re-examine the basic intuition underlying the concept of vacuity, which until now has been defined as sensitivity with respect to syntactic perturbation. We study sensitivity with respect to semantic perturbation, which we model by universal propositional quantification. We show that this yields a hierarchy of vacuity notions. We argue that the right notion is that of vacuity defined with respect to traces. We then provide an algorithm for vacuity detection and discuss pragmatic aspects.

1 Introduction

Temporal logics, which are modal logics geared towards the description of the temporal ordering of events, have been adopted as a powerful tool for specifying and verifying concurrent systems [Pnu77]. One of the most significant developments in this area is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [CE81,CES86,LP85,QS81,VW86]. This derives its significance both from the fact that many synchronization and communication protocols can be modeled as finite-state systems, as well as from the great ease of use of fully algorithmic methods. In

⋆ Supported in part by NSF grants CCR-9988322, IIS-9908435, IIS-9978135, and EIA-0086264, by BSF grant 9800096, and by a grant from the Intel Corporation.

temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior (for an in-depth survey, see [CGP99]).

Beyond being fully-automatic, an additional attraction of model-checking tools is their ability to accompany a negative answer to the correctness query with a counterexample to the satisfaction of the specification in the system. Thus, together with a negative answer, the model checker returns some erroneous execution of the system. These counterexamples are very important and can be essential in detecting subtle errors in complex designs [CGMZ95]. On the other hand, when the answer to the correctness query is positive, most model-checking tools provide no witness for the satisfaction of the specification in the system. Since a positive answer means that the system is correct with respect to the specification, this may, a priori, seem like a reasonable policy. In the last few years, however, industrial practitioners have become increasingly aware of the importance of checking the validity of a positive result of model checking. The main justification for suspecting the validity of a positive result is the possibility of errors in the modeling of the system or of the desired behavior, i.e., the specification.

Early work on "suspecting a positive answer" concerns the fact that temporal logic formulas can suffer from *antecedent failure* [BB94]. For example, in verifying a system with respect to the CTL specification $\varphi = AG(req \rightarrow AF\,grant)$ ("every request is eventually followed by a grant"), one should distinguish between *vacuous satisfaction* of φ, which is immediate in systems in which requests are never sent, and non-vacuous satisfaction, in systems where requests are sometimes sent. Evidently, vacuous satisfaction suggests some unexpected properties of the system, namely the absence of behaviors in which the antecedent of φ is satisfied.

Several years of practical experience in formal verification have convinced the verification group at the IBM Haifa Research Laboratory that vacuity is a serious problem [BBER97]. To quote from [BBER97]: "Our experience has shown that typically 20% of specifications pass vacuously during the first formal-verification runs of a new hardware design, and that vacuous passes always point to a real problem in either the design or its specification or environment." The usefulness of vacuity analysis is also demonstrated via several case studies in [PS02]. Often, it is possible to detect vacuity easily by checking the system with respect to hand-written formulas that ensure the satisfaction of the preconditions in the specification [BB94,PP95]. To the best of our knowledge, this rather unsystematic approach is the prevailing one in the industry for dealing with vacuous satisfaction. For example, the FormalCheck tool [Kur98] uses "sanity checks", which include a search for triggering conditions that are never enabled.

These observations led Beer et al. to develop a method for automatic testing of vacuity [BBER97]. Vacuity is defined as follows: a formula φ is satisfied in a system M vacuously if it is satisfied in M, but some subformula ψ of φ does not *affect* φ in M, which means that M also satisfies $\varphi\,[\psi \leftarrow \psi']$ for all formulas ψ' (here, $\varphi\,[\psi \leftarrow \psi']$ denotes the result of substituting ψ' for ψ in φ). Beer et al. proposed testing vacuity by means of *witness formulas*. Formally, we say that a formula φ' is a *witness formula* for the specification φ if a system M satisfies φ non-vacuously iff M satisfies both φ and φ'. In the example above, it is not hard to see that a system satisfies φ non-vacuously iff it also satisfies $EF\mathit{req}$. In general, however, the generation of witness formulas is not trivial, especially when we are interested in other types of vacuity passes, which are more complex than antecedent failure. While [BBER97] nicely set the basis for a methodology for detecting vacuity in temporal-logic specifications, the particular method described in [BBER97] is quite limited.

A general method for detection of vacuity for specifications in CTL* (and hence also LTL, which was not handled by [BBER97]) was presented in [KV99,KV03]. The key idea there is a general method for generating witness formulas. It is shown in [KV03] that instead of replacing a subformula ψ by all subformulas ψ', it suffices to replace it by either $\mathbf{true}$ or $\mathbf{false}$ depending on whether ψ occurs in φ with negative polarity (i.e., under an odd number of negations) or positive polarity (i.e., under an even number of negations). Thus, vacuity checking amounts to model checking witness formulas with respect to all (or some) of the subformulas of the specification φ. It is important to note that the method in [KV03] is for vacuity with respect to subformula occurrences. The key feature of occurrences is that a subformula occurrence has a *pure* polarity (exclusively negative or positive). In fact, it is shown in [KV03] that the method is not applicable to subformulas with mixed polarity (both negative and positive occurrences).

Recent experience with industrial-strength property-specification languages such as ForSpec [AFF$^+$02] suggests that the restriction to subformula occurrences of pure polarity is not negligible. ForSpec, which is a linear-time language, is significantly richer syntactically (and semantically) than LTL. It has a rich set of arithmetical and Boolean operators. As a result, even subformula occurrences may not have pure polarity, e.g., in the formulas $p \oplus q$ ($\oplus$ denotes exclusive or). While we can rewrite $p \oplus q$ as $(p \wedge \neg q) \vee (\neg p \wedge q)$, it forces the user to think of every subformula occurrence of mixed polarity as two distinct occurrences, which is rather unnatural. Also, a subformula may occur in the specification multiple times, so it need not have a pure polarity even if each occurrence has a pure polarity. For example, if the LTL formula $G(p \rightarrow p)$ holds in a system M then we'd expect it to hold vacuously with respect to the subformula p (which has a mixed polarity), though not necessarily with respect to either occurrence of p, because both formulas $G(\mathbf{true} \rightarrow p)$ and $G(p \rightarrow \mathbf{false})$ may fail in M. (Surely, the fact that $G(\mathbf{true} \rightarrow \mathbf{false})$ fails in M should not entail that $G(p \rightarrow p)$ holds in M non-vacuously.) Our goal is to remove the restriction in [KV03] to subformula occurrences of pure polarity. To keep things simple, we stick to LTL and consider vacuity with respect to subformulas, rather than with respect to subformula occurrences. We comment on the extension of our framework to ForSpec at the end of the paper.

The generality of our framework requires us to re-examine the basic intuition underlying the concept of vacuity, which is that a formula φ is satisfied in a system M vacuously if it is satisfied in M but some subformula ψ of φ does not *affect* φ in M. It is less clear, however, what does "does not affect" mean. Intuitively, it means that we can "perturb" ψ without affecting the truth of φ in M. Both [BBER97] and [KV03] consider only syntactic perturbation, but no justification is offered for this decision. We argue that another notion to consider is that of semantic perturbation, where the *truth value* of ψ in M is perturbed arbitrarily. The first part of the paper is an examination in depth of this approach. We model arbitrary semantic perturbation by a universal quantifier, which in turn is open to two interpretations (cf. [Kup95]). It turns out that we get two notions of "does not affect" (and therefore also of vacuity), depending on whether universal quantification is interpreted with respect to the system M or with respect to its set of computations. We refer to these two semantics as "structure semantics" and "trace semantics". Interestingly, the original, syntactic, notion of perturbation falls between the two semantic notions.

We argue then that trace semantics is the preferred one for vacuity checking. Structure semantics is simply too weak, yielding vacuity too easily. Formula semantics is more discriminating, but it is not robust, depending too much on the syntax of the language. In addition, these two semantics yield notions of vacuity that are computationally intractable. In contrast, trace semantics is not only intuitive and robust, but it can be checked easily by a model checker.

In the final part of the paper we address several pragmatic aspects of vacuity checking. We first discuss whether vacuity should be checked with respect to subformulas or subformula occurrences and argue that both checks are necessary. We then discuss how the number of vacuity checks can be minimized. We also discuss how vacuity results should be reported to the user. Finally, we describe our experience of implementing vacuity checking in the context of a ForSpec-based model checker.

A version with full proofs can be downloaded from the authors' homepages.

2 Preliminaries

LTL. Formulas of LTL are built from a set AP of atomic propositions using the usual Boolean operators and the temporal operators X ("next time") and U ("until"). Given a set AP , an LTL formula is:

- $\mathbf{true}$, $\mathbf{false}$, p for $p \in AP$.
- $\neg\psi$, $\psi \wedge \varphi$, $X\psi$, or $\psi U \varphi$, where ψ and φ are LTL formulas.

We define satisfaction of LTL formulas with respect to computations of Kripke structures. A Kripke structure is $M = \langle AP, S, S_0, R, L\rangle$ where AP is the set of atomic propositions, S is a set of states, S_0 is a set of initial states, $R \subseteq S \times S$ is a total transition relation, and $L : AP \to 2^S$ assigns to each atomic proposition the set of states in which it holds. A *computation* is a sequence of states $\pi = s_0, s_1, \ldots$ such that $s_0 \in S_0$ and forall $i \geq 0$ we have $(s_i, s_{i+1}) \in R$. We denote the set of computations of M by $\mathcal{T}(M)$ and the suffix $s_j, s_{j+1}, \ldots$ of π by π^j.

The semantics of LTL is defined with respect to computations and locations. We denote $M, \pi, i \models \varphi$ when the LTL formula φ holds in the computation π at location $i \geq 0$. A computation π satisfies an LTL formula φ, denoted $\pi \models \varphi$ if $\pi, 0 \models \varphi$. The structure M satisfies φ, denoted $M \models \varphi$ if for every computation $\pi \in \mathcal{T}(M)$ we have $\pi \models \varphi$. For a full definition of the semantics of LTL we refer the reader to [Eme90].

An occurrence of formula ψ of φ is of *positive polarity* in φ if it is in the scope of an even number of negations, and of *negative polarity* otherwise. The polarity of a subformula is defined by the polarity of its occurrences as follows. Formula ψ is of *positive polarity* if all occurrences of ψ in φ are of positive polarity, of *negative polarity* if all occurrences of ψ in φ are of negative polarity, of *pure polarity* if it is either of positive or negative polarity, and of *mixed polarity* otherwise.

Given a formula φ and a subformula of pure polarity ψ we denote by $\varphi\,[\psi \leftarrow \perp]$ the formula obtained from φ by replacing ψ by $\mathbf{true}$ ($\mathbf{false}$) if ψ is of negative (positive) polarity.

UQLTL. The logic UQLTL augments LTL with universal quantification over *propositional variables*. Let X be a set of propositional variables. The syntax of UQLTL is as follows. If φ is an LTL formula over the set of atomic propositions $AP \cup X$, then $\forall X \varphi$ is a UQLTL formula. E.g., $\forall x\, G\,(x \to p)$ is a legal UQLTL formula, while $G\, \forall x\,(x \to p)$ is not. UQLTL is a subset of *Quantified Propositional Temporal Logic* [SVW85], where the free variables are quantified universally. We use x to denote a propositional variable. A *closed* formula is a formula with no free propositional variables.

We now define two semantics for UQLTL. In *structure semantics* a propositional variable is bound to a subset of the states of the Kripke structure. In *trace semantics* a propositional variable is bound to a subset of the locations on the trace.

Let M be a Kripke structure with a set of states S, let $\pi \in \mathcal{T}(M)$, and let X be a set of propositional variables. A *structure assignment* $\sigma : X \to 2^S$ maps every propositional variable $x \in X$ to a set of states in S. We use s_i to denote the ith state along π, and φ to denote UQLTL formulas.

Definition 1 (Structure Semantics). *The relation $\models_s$ is defined inductively as follows:*

- $M, \pi, i, \sigma \models_s x$ *iff* $s_i \in \sigma(x)$.
- $M, \pi, i, \sigma \models_s \forall x \varphi$ *iff* $M, \pi, i, \sigma[x \leftarrow S'] \models_s \varphi$ *for every* $S' \subseteq S$.
- *For other formulas,* $M, \pi, i, \sigma \models_s$ *is defined as in LTL.*

We now define the trace semantics for UQLTL. Let X be a set of propositional variables. A *trace assignment* $\alpha : X \to 2^{\mathbf{N}}$ maps a propositional variable $x \in X$ to a set of natural numbers (points on a path).

Definition 2 (Trace Semantics). *The relation $\models_t$ is defined inductively as follows:*

- $M, \pi, i, \alpha \models_t x$ *iff* $i \in \alpha(x)$.
- $M, \pi, i, \alpha \models_t \forall x \varphi$ *iff* $M, \pi, i, \alpha[x \leftarrow N'] \models_t \varphi$ *for every* $N' \subseteq \mathbf{N}$.
- *For other formulas,* $M, \pi, i, \sigma \models_t$ *is defined as in LTL.*

A closed UQLTL formula φ is *structure satisfied* at point i of trace $\pi \in \mathcal{T}(M)$, denoted $M, \pi, i \models_s \varphi$, iff $M, \pi, i, \sigma \models_s \varphi$ for some σ (choice is not relevant since φ is closed). A closed UQLTL formula φ is structure satisfied in structure M, denoted $M \models_s \varphi$, iff $M, \pi, 0 \models_s \varphi$ for every trace $\pi \in \mathcal{T}(M)$. *Trace satisfaction* is defined similarly for a trace and for structure, and is denoted by $\models_t$.

Trace semantics is stronger than structure semantics in the following sense.

Theorem 1. *Given a structure M and a UQLTL formula φ, $M \models_t \varphi$ implies $M \models_s \varphi$. The reverse implication does not hold.*

The proof resembles the proofs in [Kup95] for the dual logic EQCTL. Kupferman shows that a structure might not satisfy a formula, while its computation tree does. Indeed, a trace assignment can assign a variable different values when the computation visits the same state of M. We observe that for LTL formulas both semantics are identical. That is, if φ is an LTL formula, then $M \models_s \varphi$ iff $M \models_t \varphi$. We use $\models$ to denote the satisfaction of LTL formulas, rather than $\models_s$ or $\models_t$.

3 Alternative Definitions of Vacuity

Let ψ be a subformula of φ. We give three definitions of when ψ *does not affect* φ, and compare them. We refer to the definition of [BBER97] as *formula vacuity*. We give two new definitions, *trace vacuity* and *structure vacuity*, according to trace and formula semantics. We are only interested in the cases where φ is satisfied in the structure.

Intuitively, ψ does not affect φ in M if we can perturb ψ without affecting the truth of φ in M. In previous work, syntactic perturbation was allowed. Using UQLTL we formalize the concept of semantic perturbation. Instead of changing ψ syntactically, we directly change the set of points in a structure or on a trace in which it holds.

Definition 3. *Let φ be a formula satisfied in M and let ψ be a subformula of φ.*

- ψ does not affect$_f$ φ *in M iff for every LTL formula ξ, $M \models \varphi[\psi \leftarrow \xi]$* [BBER97].
- ψ does not affect$_s$ φ *in M iff $M \models_s \forall x \varphi[\psi \leftarrow x]$.*
- ψ does not affect$_t$ φ *in M iff $M \models_t \forall x \varphi[\psi \leftarrow x]$.*

We say that ψ *affects$_f$* φ in M iff it is not the case that ψ does not affect$_f$ φ in M. We say that φ is *formula vacuous* in M, if there exists a subformula ψ such that ψ does not affect$_f$ φ. We define *affects$_s$*, *affects$_t$*, *structure vacuity* and *trace vacuity* similarly. Notice that we do not restrict a subformula to occur once and it can be of mixed polarity.

The three semantics form a hierarchy. Structure semantics is the weakest and trace semantics the strongest. Formally, for an LTL formula φ we have the following.

Lemma 1.
- *If ψ does not affect$_t$ φ in M, then ψ does not affect$_f$ φ in M as well.*
- *If ψ does not affect$_f$ φ in M, then ψ does not affect$_s$ φ in M as well.*
- *The reverse implications do not hold.*

Which is the most appropriate definition for practical applications? We show that structure and formula vacuity are sensitive to changes in the design that do not relate to the formula. Consider the formula $\varphi = p \rightarrow Xp$ and models M_1 and M_2 in Figure 1. In M_2 we add a proposition q whose behavior is independent of p's behavior. We would not like formulas that relate to p to change their truth value or their vacuity. Both M_1 and its extension M_2 satisfy φ and φ relates only to p. While p does not affect$_f$ φ in M_1, it does affect$_f$ φ in M_2 (and similarly for affects$_s$). Indeed, the formula $\varphi[p \leftarrow q] = q \rightarrow Xq$ does not hold in M_2. Note that in both models p affects$_t$ φ.

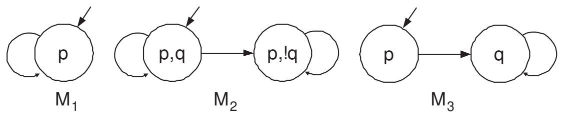

Fig. 1. Changes in the design and dependance on syntax.

Formula vacuity is also *sensitive to the specification language*. That is, a formula passing vacuously might pass unvacuously once the specification language is extended.

Consider the Kripke structure M_3 in Figure 1 and the formula $\varphi = Xq \rightarrow XXq$. For the single trace $\pi \in \mathcal{T}(M_3)$, it holds that $\pi^2 = \pi^1$. Thus, every (future) LTL formula is either true along every suffix of π^1, or is false along every such suffix. Hence, subformula q does not affect$_f$ φ. We get an opposite result if the specification language is LTL with X^{-1} [LPZ85]. Formally, for ψ in LTL, $M, \pi, i \models X^{-1}(\psi)$ iff $i > 0$ and $M, \pi, i-1 \models \psi$. Clearly, for every model M we have $M, \pi, 0 \not\models X^{-1}(p)$. In the example, $M_3 \not\models \varphi \left[q \leftarrow X^{-1}(p)\right]$ since $M_3, \pi, i \models X^{-1}(p)$ iff $i = 1$, thus q affects$_f \varphi$.

To summarize, trace vacuity is preferable since it is not sensitive to changes in the design (unlike structure and formula vacuity) and it is independent of the specification language (unlike formula vacuity). In addition as we show in Section 4, trace vacuity is the only notion of vacuity for which an efficient decision procedure is known to exist.

We claim that if subformulas are restricted to pure polarity, all the definitions of vacuity coincide. In such a case the algorithm proposed in [KV03], to replace the subformula ψ by $\perp$ is adequate for vacuity detection according to all three definitions.

Theorem 2. *If ψ is of pure polarity in φ then the following are equivalent.*

1. $M, \pi, i \models \varphi\,[\psi \leftarrow \perp]$
2. $M, \pi, i \models_s \forall x\, \varphi\,[\psi \leftarrow x]$
3. *for every formula ξ we have* $M, \pi, i \models \varphi\,[\psi \leftarrow \xi]$
4. $M, \pi, i \models_t \forall x\, \varphi\,[\psi \leftarrow x]$

4 Algorithm and Complexity

We give now algorithms for checking vacuity according to the different definitions. We show that the algorithm of [KV03], which replaces a subformula by either $\mathbf{true}$ or $\mathbf{false}$ (according to its polarity), cannot be applied to subformulas of mixed polarity. We then study structure and trace vacuity. Decision of formula vacuity remains open.

We show that the algorithm of [KV03] cannot be applied to subformulas of mixed polarity. Consider the Kripke structure M_2 in Figure 1 and the formula $\varphi = p \rightarrow Xp$. Clearly, $M_2 \not\models_s \forall x \varphi\,[p \leftarrow x]$ (with the structure assignment $\sigma(x)$ including only the initial state), $M_2 \not\models \varphi\,[p \leftarrow q]$, and $M_2 \not\models_t \forall x \varphi\,[p \leftarrow x]$ (with the trace assignment $\alpha(x) = \{0\}$). Hence, p affects φ according to all three definitions. On the other hand, $M \models \varphi\,[p \leftarrow \mathbf{false}]$ and $M \models \varphi\,[p \leftarrow \mathbf{true}]$. We conclude that the algorithm of [KV03] cannot be applied to subformulas of mixed polarity.

We now solve trace vacuity. For a formula φ and a model $M = \langle AP, S, S_0, R, L\rangle$ where $M \models \varphi$, we check whether ψ affects$_t$ φ in M by model checking the UQLTL formula $\varphi' = \forall x\, \varphi\,[\psi \leftarrow x]$ on M. Subformula ψ does not affect$_t$ φ iff $M \models \varphi'$. The algorithm in Figure 2 detects if ψ affects$_t$ φ in M. The structure M' guesses at every step the right assignment for the propositional variable x. Choosing a path in M' determines the trace assignment of x. Formally, we have the following.

Theorem 3. [VW94] *Given a structure M and an LTL formula φ, we can model check φ over M in time linear in the size of M and exponential in φ and in space polylogarithmic in the size of M and quadratic in the length of φ.*

1. Compute the polarity of ψ in φ.
2. If ψ is of pure polarity, model check $M \models \varphi[\psi \leftarrow \bot]$.
3. Otherwise, construct $M' = \langle AP \cup \{x\}, S \times 2^x, S_0 \times 2^x, R', L\rangle$, where for every $X_1, X_2 \subseteq 2^x$ and $s_1, s_2 \in S$ we have $(s_1 \times X_1, s_2 \times X_2) \in R'$ iff $(s_1, s_2) \in R$.
4. Model check $M' \models \varphi[\psi \leftarrow x]$.

Fig. 2. Algorithm for Checking if ψ Affects$_t$ φ

Corollary 1. *Given a structure M and an LTL formula φ such that $M \models \varphi$, we can decide whether subformula ψ affects$_t$ φ in time linear in the size of M and exponential in φ and in space polylogarithmic in the size of M and quadratic in the length of φ.*

Recall that in symbolic model checking, the modified structure M' is not twice the size of M but rather includes just one additional variable. In order to check whether φ is trace vacuous we have to check whether there exists a subformula ψ of φ such that ψ does not affect$_t$ φ. Given a set of subformulas $\{\psi_1, \ldots, \psi_n\}$ we can check whether one of these subformulas does not affect$_t$ φ by iterating the above algorithm n times. The number of subformulas of φ is proportional to the size of φ.

Theorem 4. *Given a structure M and an LTL formula φ such that $M \models \varphi$. We can check whether φ is trace vacuous in M in time $O(|\varphi| \cdot C_M(\varphi))$ where $C_M(\varphi)$ is the complexity of model checking φ over M.*

Unlike trace vacuity, there does not exist an efficient algorithm for structure vacuity. We claim that deciding does not affect$_s$ is co-NP-complete in the structure. Notice, that co-NP-complete in the structure is much worse than PSPACE-complete in the formula. Indeed, the size of the formula is negligible when compared to the size of the model. Co-NP-completeness of structure vacuity renders it completely impractical.

Lemma 2 (Deciding does not affect$_s$). *For φ in LTL, a subformula ψ of φ and a structure M, the problem of deciding whether ψ does not affect$_s$ φ in M is co-NP-complete with respect to the structure M.*

The complexity of deciding affects$_f$ is unclear. For subformulas of pure polarity (or occurrences of subformulas) the algorithm of [KV03] is correct. We have found neither a lower bound nor an upper bound for deciding affects$_f$ in the case of mixed polarity.

5 Pragmatic Aspects

Display of Results. When applying vacuity detection in an industrial setting there are two options. We can either give the user a simple yes/no answer, or we can accompany a positive answer (vacuity) with a witness formula. Where ψ does not affect φ we supply $\varphi\,[\psi \leftarrow x]$ (or $\varphi\,[\psi \leftarrow \bot]$ where ψ is of pure polarity) as our witness to the vacuity of φ. When we replace a subformula by a constant, we propagate the constants upwards[4].

[4] I.e. if in subformula $\theta = \psi_1 \wedge \psi_2$ we replace ψ_1 by $\mathbf{false}$, then θ becomes $\mathbf{false}$ and we continue propagating this value above θ.

$$\begin{array}{c}
\text{active} := \text{en} \wedge \neg \text{in} ; \qquad\qquad \text{rdy_active} := \neg\ \text{rdy_out} \wedge \neg\ \text{active} ; \\
\text{bsy_active} := \neg\ \text{bsy_out} \wedge \neg\ \text{active}; \\
\text{active_inactive} := \text{rdy_active} \wedge \neg\ \text{bsy_active} ; \\
\text{two_consecutive} := G[(\text{reset} \wedge \text{active_inactive}\ ) \rightarrow X \neg \text{active_inactive}]; \\
\text{two_consecutive}[\text{active_inactive}_2 \leftarrow \bot] := G \neg(\ \text{reset} \wedge \text{active_inactive}); \\
\text{two_consecutive}[\text{en}_2 \leftarrow \bot] := G(\ \text{reset} \wedge \text{active_inactive}) \rightarrow X \neg(\neg\ \text{rdy_out} \wedge \neg\ \text{bsy_active}\ );
\end{array}$$

Fig. 3. Vacuous pass

Previous works suggested that the users of vacuity detection are interested in simple yes / no answers. That is, whether the property is vacuous or not. Suppose that ψ does not affect φ. It follows that if ψ' is a subformula of ψ then ψ' does not affect φ as well. In order to get a yes / no answer only the minimal subformulas (atomic propositions) of φ have to be checked [BBER97,KV03]. When the goal is to give the user feedback on the source of detected vacuity, it is often more useful to check non-minimal subformulas.

Consider for example the formula *two_consecutive* in Figure 3. This is an example of a formula that passed vacuously in one of our designs. The reason for the vacuous pass is that one of the signals in *active_inactive* was set to **false** by a wrong environmental assumption. The formula two_consecutive $[\text{active\_inactive}_2 \leftarrow \bot]$ is the witness to the fact that the second occurrence of *active_inactive* does not affect *two_consecutive*. From this witness it is straightforward to understand what is wrong with the formula. The formula two_consecutive $[\text{en}_2 \leftarrow \bot]$ is the witness associated with the occurrence of the proposition *en* under the second occurrence of *rdy_active* (after constant propagation). Clearly, this report is much less legible. Thus, it is preferable to check vacuity of non-minimal subformulas and subformula occurrences.

If we consider the formula as represented by a tree (rather than DAG – directed acyclic graph) then the number of leaves (propositions) is proportional to the number of nodes (subformulas). We apply our algorithm from top to bottom. We check whether the maximal subformulas affect the formula. If a subformula does not affect, there is no need to continue checking below it. If a subformula does affect, we continue and check its subformulas. In the worst case, when all the subformulas affect the formula, the number of model checker runs in order to give the most intuitive counter example is double the size of the minimal set (number of propositions). The yes / no view vs. the intuitive witness view offer a clear tradeoff between minimal number of model checker runs (in the worst case) and giving the user the most helpful information. We believe that the user should be given the most comprehensive witness. In our implementation we check whether **all** subformulas and occurrences of subformulas affect the formula.

Occurrences vs. Subformulas. We introduced an algorithm that determines if a subformula with multiple occurrences affects a formula. We now give examples in which checking a subformula is more intuitive, and examples in which checking an occurrence is more intuitive. We conclude that a vacuity detection algorithm has to check both.

The following example demonstrates why it is reasonable to check if a subformula affects a formula. Let $\varphi = G(p \rightarrow p)$. Intuitively, p does not affect φ, since every expression (or variable) implies itself. Indeed, according to all the definitions p does not affect φ, regardless of the model. However, every occurrence of p may affect φ. Indeed, both $Gp = \varphi\,[p_1 \leftarrow \bot]$ and $G\neg p = \varphi\,[p_2 \leftarrow \bot]$ may fail (here, p_i denotes the ith occurrence of p).

The following example demonstrates why it is reasonable to check if an occurrence affects a formula. Let $\varphi = p \wedge G(q \rightarrow p)$. Assume q is always $\mathbf{false}$ in model M. Clearly, the second occurrence of p does not affect φ in M. However, the subformula p does affect φ in M. Every assignment that gives x the value $\mathbf{false}$ at time 0 would falsify the formula $\varphi\,[p \leftarrow x]$. Recall the formula *two_consecutive* in Figure 3. The vacuous pass in this case is only with respect to occurrences and not to subformulas.

We believe that a *thorough vacuity-detection* algorithm should detect both subformulas and occurrences that do not affect the examined formula. It is up to the user to decide which vacuity alerts to ignore.

Minimizing the number of checks. We choose to check whether all subformulas and all occurrences of subformulas affect the formula. Applying this policy in practice may result in many runs of the model checker and may be impractical. We now show how to reduce the number of subformulas and occurrences for which we check vacuity by analyzing the structure of the formula syntactically.

As mentioned, once we know that ψ does not affect φ, there is no point in checking subformulas of ψ. If ψ affects φ we have to check also the subformulas of ψ. We show that in some cases for ψ' a subformula of ψ we have ψ' affects φ iff ψ affects φ. In these cases there is no need to check direct subformulas of ψ also when ψ affects φ.

Suppose the formula φ is satisfied in M. Consider an occurrence θ_1 of the subformula $\theta = \psi_1 \wedge \psi_2$ of φ. We show that if θ_1 is of positive polarity then ψ_i affects φ iff θ_1 affects φ for $i = 1, 2$. As mentioned, θ_1 does not affect φ implies ψ_i does not affect φ for $i = 1, 2$. Suppose θ_1 affects φ. Then $M \not\models \varphi\,[\theta_1 \leftarrow \mathbf{false}]$. However, $\varphi\,[\psi_i \leftarrow \mathbf{false}] = \varphi\,[\theta_1 \leftarrow \mathbf{false}]$. It follows that $M \not\models \varphi\,[\psi_i \leftarrow \mathbf{false}]$ and that ψ_i affects φ. In the case that θ_1 is of negative (or mixed) polarity the above argument is incorrect. Consider the formula $\varphi = \neg(\psi_1 \wedge \psi_2)$ and a model where ψ_1 never holds. It is straightforward to see that $\psi_1 \wedge \psi_2$ affects φ while ψ_2 does not affect φ.

It follows that we can analyze φ syntactically and identify occurrences θ_1 such that θ_1 affects φ iff the subformulas of θ_1 affect φ. In these cases, it is sufficient to model check $\forall x \varphi\,[\theta_1 \leftarrow x]$. Below the immediate subformulas of θ_1 we have to continue with the same analysis. For example, if $\theta = (\psi_1 \vee \psi_2) \wedge (\psi_3 \wedge \psi_4)$ is of positive polarity and θ affects φ we can ignore $(\psi_1 \vee \psi_2)$, $(\psi_3 \wedge \psi_4)$, ψ_3, and ψ_4. We do have to check ψ_1 and ψ_2. In Table 1 we list the operators under which we can apply such elimination. In the polarity column we list the polarities under which the elimination scheme applies to the operator. In the operands column we list the operands that we do not have to check. We stress that below the immediate operands we have to continue applying the analysis.

The analysis that leads to the above table is quite simple. Using a richer set of operators one must use similar reasoning to extend the table. We distinguish between pure and mixed polarity. The above table is true for occurrences. Mixed polarity is only introduced when the specification language includes operators with no polarity (e.g. $\oplus$, $\leftrightarrow$). In order to apply a similar elimination to subformulas with multiple occurrences one has to take into account the polarities of all occurrences and the operator under

Operator	Polarity	Operands
$\wedge$	+	all
$\vee$	-	all
$\neg$	pure / mixed	all

Operator	Polarity	Operands
X	pure / mixed	all
U	pure	second
G	pure	all
F	pure	all

Table 1. Operators for which checks can be avoided

which every occurrence appears. However, suppose that the subformula $\theta = f(\psi_1, \psi_2)$ occurs more than once but ψ_1 and ψ_2 occur only under θ. In this case, once we check whether θ affects φ, the elimination scheme can be applied to ψ_1 and ψ_2.

Implementation and Methodology. We implemented the above algorithms in one of Intel's formal verification environments. We use the language ForSpec [AFF$^+$02] with the BDD-based model checker Forecast [FKZ$^+$00] and the SAT-based bounded model checker Thunder [CFF$^+$01]. The users can decide whether they want thorough vacuity detection or just to specify which subformulas / occurrences to check. In the case of thorough vacuity detection, for every subformula and every occurrence (according to the elimination scheme above) we create one witness formula. The vacuity algorithm amounts to model checking each of the witnesses. Both model checkers are equipped with a mechanism that allows model checking of many properties simultaneously.

The current methodology of using vacuity is applying thorough vacuity on every specification. The users prove that the property holds in the model; then, vacuity of the formula is checked. If applying thorough vacuity is not possible (due to capacity problems), the users try to identify the important subformulas and check these subformulas manually. In our experience, vacuity checks proved to be effective mostly when the pruning and assumptions used in order to enable model checking removed some important part of the model, thus rendering the specification vacuously true. In many examples vacuity checking pointed out to such problems. We also have cases where vacuity pointed out redundant parts in the specification.

In Table 2 we include some experimental results. We used real-life examples from processor designs. We include these results in order to give the flavor of the performance of vacuity checking. Each line in the table corresponds to one property. Some properties are the conjunction of a few assertions. In such a case, every assertion is checked separately (both in model checking and vacuity detection). For each property we report on 4 different experiments. The first column specifies the number of witness formulas sent to the model checker for vacuity detection. The number in parentheses indicates the number of non-affecting subformulas / occurrences. The block titled Forecast reports on three experiments. The column titled *MC* reports on the results of model checking the property itself. The column titled *Vacuity* reports on the results of model checking all the witness formulas for all the assertions. Finally, the column titled *Combined* reports on the results of model checking all the witnesses with all the assertions. In each column we specify the time (in seconds) and space (BDD nodes) required by Forecast. The symbol ! indicates that Forcast timed out. Recall that in vacuity detection we hope that all the witness formulas do not pass in the model. As bounded model checking is especially adequate for falsification, we prove the correctness of the property using Forecast and falsify the witness formulas using Thunder. Witness formulas that Thunder was unable to falsify can be checked manually using Forecast. The last column reports on the results of model checking all the witness formulas for all the assertions using Thunder. We write the time (in seconds) required by Thunder, and ! in case that Thunder did not terminate in 8 hours. In the case that Thunder did not terminate we report (in brackets) the bound up to which the formulas were checked[5]. We ran the examples on a Intel(R) Pentium™ 4 2.2GHz processor running Linux with 2GByte memory. Notice that some of these examples pass vacuously.

Property	♯ Checks	Forecast						Thunder
		MC		Vacuity		Combined		
		Time	Nodes	Time	Nodes	Time	Nodes	
check_internal_sig	5(1)	1936	3910K	2051	2679K	3185	5858K	2.28(0)
lsd_indication	17(5)	1699	2150K	2265	2566K	1986	3483K	!(5)[40]
directive	4(0)	611	1120K	16132	4945K	7355	8943K	25(0)
forbidden_start	2(0)	532	549K	1859	4064K	2422	4274K	22(0)
nested_start	22 (13)	737	1294K	11017	6153K	10942	6153K	!(18)[70]
pilot_session	129(?[6])	5429	3895K	67126!	25366K	66157!	20586K	!(16)[60]
new_code	31(1)	1265	2455K	1765	2853K	3097	3932	!(1)[50]

Table 2. Experimental results

6 Summary and Future Work

We investigated vacuity detection with respect to subformulas with multiple occurrences. We re-examined the basic intuition underlying the concept of vacuity, which until now has been defined as sensitivity with respect to syntactic perturbation. We studied sensitivity with respect to semantic perturbation, which we modeled by universal propositional quantification. We showed that this yields a hierarchy of vacuity notions. We argued that the right notion is that of vacuity defined with respect to traces, described an algorithm for vacuity detection, and discussed pragmatic aspects.

We were motivated by the need to extend vacuity detection to industrial-strength property-specification languages such as ForSpec [AFF$^+$02]. ForSpec is significantly richer syntactically and semantically than LTL. Our vacuity-detection algorithm for subformulas of mixed polarity can handle ForSpec's rich set of arithmetical and Boolean operators. ForSpec's semantic richness is the result of its *regular layer*, which includes regular events and formulas constructed from regular events. The extension of vacuity detection to ForSpec's regular layer will be described in a future paper.

[5] Note that in the case of *lsd_indication* and *new_code* the partial answer is in fact the final answer, as can be seen from the run of Forecast

[6] For this property, we do not know the number of non affecting subformulas / occurrences. There are 16 non affecting subformulas / occurrences up to bound 60.

7 Acknowledgments

We thank R. Fraer, D. Lichten, and K. Sajid for their contribution and useful remarks.

References

[AFF+02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *8th TACAS*, LNCS 2280, 296–211, 2002. Springer.

[BB94] D. Beaty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *31st DAC*, IEEE Computer Society, 1994.

[BBER97] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *9th CAV*, LNCS 1254, 279–290, 1997. Full version in *FMSD*, 18 (2): 141–162, 2001.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *WLP*, LNCS 131, 52–71. 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2):244–263, 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *13th CAV*, 2001.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *32nd DAC*, 1995.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Eme90] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, chapter 16. Elsevier, MIT press, 1990.

[FKZ+00] R. Fraer, G. Kamhi, B. Ziv, M. Vardi, and L. Fix. Prioritized traversal: efficient reachability analysis for verication and falsification. In *12th CAV*, LNCS 1855, 2000.

[Kup95] O. Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. In *8th CAV*, LNCS 939, 1995.

[Kur98] R.P. Kurshan. *FormalCheck User's Manual*. Cadence Design, Inc., 1998.

[KV99] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In *10th CHARME*, LCNS 170, 82–96. Springer-Verlag, 1999.

[KV03] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th POPL*, 97–107, 1985.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, LNCS 193, 196–218, 1985. Springer-Verlag.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th FOCS*, 46–57, 1977.

[PP95] B. Plessier and C. Pixley. Formal verification of a commercial serial bus interface. In *14th IEEE Conf. on Computers and Comm.*, 378–382, 1995.

[PS02] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *14th CAV*, LNCS 2404, 485–499. Springer-Verlag, 2002.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, LNCS 137, 1981.

[SVW85] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. In *10th ICALP*, LNCS 194, 1985.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st LICS*, 332–344, 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *IC*, 115(1), 1994.