Easier and More Informative Vacuity Checks

Hana Chockler
IBM Research Mount Carmel, Haifa 31905, Israel.
hanac@il.ibm.com

Ofer Strichman
Information Systems Engineering, Technion, Haifa 32000, Israel.
ofers@ie.technion.ac.il

Abstract

In formal verification, we verify that a system is correct with respect to a specification. Cases like antecedent failure can make a successful pass of the verification procedure meaningless. Vacuity detection can signal such "meaningless" passes of the specification, and indeed vacuity checks are now a standard component in many commercial model checkers.

We address two dimensions of vacuity: the computational effort and the information that is given to the user. As for the first dimension, we present several preliminary vacuity checks that can be done without the design itself, which implies that some information can be found with a significantly smaller effort. As for the second dimension, we present algorithms for deriving three types of information that are not provided by standard vacuity checks, assuming $M \models \varphi$ for a model M and property φ: a) behaviors that are possibly missing from M (or wrongly restricted by the environment) b) the largest subset of occurrences of literals in φ that can be replaced with **false** *simultaneously without falsifying φ in M, and finally c) the degree of* responsibility *of each occurrence of a literal in φ to its satisfaction in the model M, which can be seen as a fine-grain form of vacuity. The complexity of each of these problems is proven. Overall this extra information can lead to tighter specifications and more guidance for finding errors.*

1 Introduction

In *model checking* [CE81, QS81, LP85], 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 specification of this behavior, expressed in terms of a temporal logic formula or a finite automaton. Beyond being fully automatic, an additional attraction of model-checking tools is their ability to accompany a negative answer to the correctness query by 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 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 terminate with no further information to the user. Since a positive answer means that the system is correct with respect to the specification, this at first seems like a reasonable policy. Beer et al. raised the issue of suspecting the system of containing an error even if the model checking succeeds [BBER01]. The main justification of such suspicions are possible errors in the modeling of the system and a possible incompleteness in the specification.

Probably the most common method for systematically searching for such errors is *vacuity detection* [BBER01, AFF$^+$03, PS02, KV03, CG04b, CG04a, BFG$^+$05, Kup06], where cases like antecedent failure [BB94] make parts of the specification irrelevant to its satisfaction. For example, the specification $\varphi = G(req \rightarrow Fgrant)$ ("every request is eventually followed by a grant") is vacuously satisfied in a system in which req is always **false** (that is, requests are never sent). Thus, a specification might be satisfied for a reason unexpected by the user. Beer et al. formally defined vacuity and claimed that it is a serious problem: "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 the environment" [BBER01]. The definition of Beer et al. is based on the notion of subformulas that do not affect the satisfaction of the specification in the system. Consider a system M and a specification φ. A subformula ψ of φ *does not affect φ in M* iff ψ can be replaced with any formula θ

without changing the truth value of φ in M. A specification φ is *satisfied vacuously in* M if there exists ψ that does not affect φ in M. While this definition works both for the case where M satisfies φ and the case where it does not, in the former case there is no supplementary information such as a counterexample to support the result. Hence, it is more natural to worry about vacuous satisfaction in the event of a positive result. This is also the case that we focus on in this paper.

Kupferman and Vardi [KV03] suggested looking at *occurrences* of subformulas, which gives different information than the original definition. For example, given the formula $\varphi : \ p \wedge X(q \vee p)$ and a model M which satisfies it, it is possible that $M \models p \wedge X(q \vee \theta)$ where θ is arbitrary, but $M \not\models \theta \wedge X(q \vee \theta)$. Hence, focusing on occurrences can detect more cases of vacuous satisfaction. On the other hand, consider a valid property such as $\varphi = AG(p \vee \neg p)$. Looking at both occurrences of p separately leads to properties $AG(\theta \vee \neg p)$ and $AG(p \vee \theta)$, which can both be **false** in a system, and thus the vacuity check with respect to occurrences will not detect the problem with the property. Kupferman and Vardi showed that for vacuity of occurrences, it is enough to check the effect of replacing each occurrence separately with its $\perp$ value (i.e., **true** for subformulas with negative polarity, and **false** otherwise). The same result holds for checking vacuity with respect to all subformulas with the same polarity. This shift from focusing on subformulas to occurrences has implications on the complexity of the check as well. Vacuity detection of subformulas (according to the original definition) for CTL is co-NP-complete [KV03]. On the other hand, checking vacuity with respect to occurrences of subformulas is $|\varphi|$ times the complexity of model checking of φ in M, which is polynomial for CTL. Since model checking for LTL has higher complexity than for CTL, checking vacuity of subformulas is intractable for LTL as well. Chechik and Gurfinkel proved that checking vacuity with respect to all *occurrences* of *literals* in φ is equivalent to checking all occurrences of general subformulas [CG04b], and is computationally cheaper (as literals are a subset of all subformulas). Hence, unless otherwise stated in this paper, we concentrate on vacuity with respect to occurrences of literals.

Continuing the observation made by Beer et al., in the event of a vacuous pass, we might need to take one of the following actions to remedy the problem. We may need to rewrite the specification so that it captures the intended behavior more tightly. Alternatively, the system might be lacking some intended behaviors. These behaviors should be added to it in order to ensure "interesting" (non-vacuous) satisfaction. Another possibility is that the vacuous pass results from an over-constrained environment that blocks non-vacuous behaviors of the system. Since vacuous passes may hide real problems with the design, vacuity checking is now a part of almost every commercial model-checker.

In this work, we address two dimensions of vacuity: the computational effort and the information given to the user. Let Φ denote the set of properties we wish to verify with respect to the system M. As for computational effort, we propose several preliminary vacuity checks that can be done without M itself (rather they are done with respect to Φ), yet are sufficient to obtain some of the vacuity information[1]. The preliminary checks we consider are *redundancy* of properties and vacuity of properties *with respect to* Φ. For the redundancy problem, we show that finding a minimal subset of Φ's properties that is equivalent to Φ, and hence does not contain redundancies, is $\text{FP}^{\Sigma_2[\log n]}$-hard. For the latter problem, we present a procedure for tightening the set of specifications in Φ without considering M. We prove that vacuity with respect to Φ implies vacuity with respect to all systems that satisfy Φ, and thus the positive answer from this check saves a more costly vacuity check with respect to M.

As for the information given to the user, we consider several venues. First, in case a property is satisfied vacuously in M but not in Φ, we present the user an *interesting* witness that is missing from M (see Section 2 for the definition), which is beyond what a standard vacuity procedure outputs. Second, we address the problem of mutual vacuity, i.e., detecting vacuity with respect to several literal occurrences. Information about mutual vacuity is beneficial for creating tighter specifications. Consider, for example, a system M in which each state is labeled with an atomic proposition c and also with one of the atomic propositions a or b (but not both), and a specification $\varphi = \mathbf{G}(a \vee b \vee c)$ ("always either a or b or c"). It is easy to see that each literal separately does not affect φ in M; however, they cannot be replaced simultaneously with **false** without falsifying φ in M. It is also easy to see that the largest set of literals that can be replaced with **false** without affecting φ in M is $\{a, b\}$, thus resulting in the tightest specification $\mathbf{G}c$ for the system M. The problem of mutual vacuity was raised in [CG04b], where the authors describe a solution using multi-valued model checking. The results of [CG04b] include an algorithm for mutual vacuity that has an exponential complexity. In this work, we study the complexity of the mutual vacuity problem and show that finding the largest set of literals (not literal *occurrences*) that can be replaced with **false** simultaneously without affecting the specification in the system is hard in the class $\text{FP}^{FNP[\log n]}$ (and is complete in this class for CTL specifications). We also present an iterative

[1] A similar direction, of checking vacuity without any additional computational cost on top of model checking, is taken in [Nam04], where it is suggested to use the proof of correctness provided by the model checker FormalCheck to identify vacuous passes. Recently there has also been some work on deriving such information from resolution proofs corresponding to bounded model checking with depth equal to the completeness threshold [SDG06].

algorithm for finding this set.

In Section 5 we discuss the relationship between vacuity and *responsibility*. Responsibility is a notion that was introduced in [CH04]. It is a quantitative measure of causality as defined by Halpern and Pearl in [HP01]. Informally, responsibility measures how much a value of a variable affects the truth value of a Boolean formula. Using responsibility for refining the notion of vacuity is done similarly to the way it is done in coverage [CHK03]. For example, in the specification $\varphi = \mathbf{G}(a \vee b \vee c)$ and a system M as above, none of the literals is directly (or *counterfactually*) responsible for the value of φ in M. However, replacing a with **false** results in the specification $\mathbf{G}(b \vee c)$, that its value counterfactually depends on c. Counterfactual dependence directly corresponds to the notion of "affecting the truth value", on which vacuity is based. We discuss the possible insights and meaning of larger or smaller degree of responsibility of literal occurrences in the context of vacuity detection.

We conclude in Section 6 by reporting the results of applying some of the techniques presented in this paper to two real designs.

2 Definitions

We assume that specifications are given by LTL formulas and the verified systems are modeled by Kripke structures [Eme90]. It is shown in [VW94] that given an LTL formula φ, we can construct a nondeterministic Büchi automaton $\mathcal{B}_\varphi$ over the alphabet 2^{AP} such that $\mathcal{B}_\varphi$ accepts exactly all the words that satisfy φ. Formally, $\mathcal{L}(\mathcal{B}_\varphi) = \{\tau \in (2^{AP})^\omega : \tau \models \varphi\}$. We say that a Kripke structure M satisfies an LTL formula φ, denoted $M \models \varphi$, if $\mathcal{L}(M \times \mathcal{B}_{\neg\varphi}) = \emptyset$.

A temporal logic formula φ is in *positive normal form* if negations are applied only to atomic propositions. It is easy to see that using De-Morgan rules and similar rules for the temporal operators, an LTL formula can be translated to an equivalent formula in positive normal form of at most twice the length of the original formula.

A *literal* is either a propositional variable or its negation. Let φ be an LTL formula in positive normal form. Denote by $literals(\varphi)$ the set of literals appearing in φ in their polarity, e.g., for $\varphi : p \wedge X \neg q \vee p$, $literals(\varphi) = \{p, \neg q\}$. Denote by $lit\text{-}occur(\varphi)$ the multiset[2] of literals appearing in φ in their polarity, e.g., for φ as given earlier, $lit\text{-}occur(\varphi) = \{p, \neg q, p\}$. Note that each element $l \in lit\text{-}occur(\varphi)$ is implicitly associated with a specific occurrence of l in φ. For $l \in lit\text{-}occur(\varphi)$, denote by $\varphi[l \leftarrow \textbf{false}]$ the formula φ after substituting the occurrence of l with **false**. We will use the terms set and subset even when referring to multisets and sub-multisets, when the meaning is clear from the context.

In this paper we consider two notions of *vacuity*:

1. *Vacuity with respect to occurrences of literals*: φ is vacuously satisfied by M if there exists $l \in lit\text{-}occur(\varphi)$ such that $M \models \varphi[l \leftarrow \textbf{false}]$.
2. *Vacuity with respect to literals*: φ is vacuously satisfied by M if there exists $l \in literals(\varphi)$ such that $M \models \varphi[l' \leftarrow \textbf{false} | l' \in lit\text{-}occur(\varphi), l' = l]$.

As explained in the introduction, from vacuity as defined by the first definition we can derive vacuity information with respect to subformula occurrences.

We denote by $M \models_v \varphi$ the situation where φ is satisfied vacuously in M. For a system M that satisfies φ and a given literal $l \in lit\text{-}occur(\varphi)$, a path π of M is called *an interesting witness to the satisfaction of φ in M with respect to l* if $\pi \not\models \varphi[l \leftarrow \textbf{false}]$. This definition can be easily generalized to subsets of literal occurrences, literals, and subformulas in general.

Unless stated otherwise, throughout this paper we refer to LTL formulas in positive normal form, and our notion of vacuity is that of vacuity with respect to occurrences of literals.

3 Preliminary Checks

The fact that verification is performed over a set of properties Φ can be used for preliminary filtering and strengthening of properties, without model-checking them against the design. The computational effort of these checks is small and can save a lot of work in later stages, as well as improve the specification and design itself, as we explain below. We present two techniques of this sort, which we call *property redundancy checks* and *vacuity without design*. We then show that the set of properties Φ can also be used for giving the user a constructive feedback in case of vacuous satisfaction: in particular, a behavior that may be missing from the system M. In subsection 3.4 we integrate these techniques in a single algorithm. In the following when we refer to Φ as a formula rather than a set, the meaning is that we refer to a conjunction of all the properties in Φ, for example $\models \Phi$ is a shorthand for $\models \bigwedge_{\phi \in \Phi} \phi$.

3.1 Property redundancy checks

A specification φ is *redundant with respect to* Φ if $\bigwedge_{\phi \in \{\Phi \setminus \varphi\}} \phi \models \varphi$. Checking redundancy of one specification is expected to be easier in practice than model-checking, since the system defined by a conjunction of Φ's

[2] A multiset is a set in which elements can appear more than once. It is required in this case because there can be more than one occurrence of a literal in φ.

properties is relatively small. However, there can be more than one redundant formula, and they can be mutually dependent, e.g., while for $i \in \{1,2\}$ φ_i is redundant with respect to $\Phi \setminus \{\varphi_i\}$, neither is redundant with respect to $\Phi \setminus \{\varphi_1, \varphi_2\}$, hence, they cannot be removed together.

Ideally, we should search for the smallest subset Ψ of formulas that satisfy all the formulas in Φ, i.e., the smallest subset $\Psi \subseteq \Phi$ such that $\bigwedge_{\psi \in \Psi} \psi \models \Phi$. But this is a hard problem, as proven by the following theorem:

Theorem 3.1 *Finding the smallest $\Psi \subseteq \Phi$ such that $\bigwedge_{\psi \in \Psi} \psi \models \Phi$, is $FP^{\Sigma_2[\log n]}$-hard.*

Proof: By a simple reduction from the minimal unsatisfiable core problem, which is $FP^{\Sigma_2[\log n]}$-complete [Gup06]. Given an unsatisfiable CNF formula A, let A's clauses be Φ's formulas. Clearly the smallest subset Ψ of Φ's formulas that is equivalent to Φ, i.e., still contradictory, is equal to the minimal unsatisfiable core of A. □

One may argue that the number of properties in Φ is rather small in practice (say, several tens, or several hundred properties), which implies that in practice the minimal set can potentially be found in a reasonable amount of time. If this is not the case, however, it is still possible to try to remove them one at a time, settling for a non-optimal solution. Our implementation, in fact, does just that. In what follows, we denote a procedure that remove redundant properties (either an exact or an approximate one) by REMOVE-REDUNDANCY(Φ).

Although redundancy is a very simple test, our experiments with a real-life design (see Section 6) show that it can be very effective. In particular, while using an arbitrary order, it removed 9 out of the 17 properties that we tested.

3.2 Vacuity without design

The next step after removing redundant properties is to detect properties that are vacuous in *any* system that satisfies Φ's properties. Note that this technique can be applied before it is known whether the system satisfies all the properties in Φ, because the result of applying it is an equivalent set of properties: it cannot make a property that fails in M, pass.

Assume $M \models \Phi$. The fact that the behaviors allowed by Φ are a superset of the behaviors allowed in M, implies that

$$\forall_{\varphi \in \Phi}.\ \Phi \models_v \varphi \Rightarrow M \models_v \varphi, \tag{1}$$

where, recall, $\models_v$ stands for vacuous satisfaction. In other words, a property that is vacuous with respect to the conjunction of all the properties is also vacuous in the system M. This immediately suggests a procedure for early detection of vacuity, without a design, as can be seen in Algorithm 1.

Algorithm 1 A procedure for early detection of vacuity, without involving the design.

VACUITY-WITHOUT-DESIGN(Φ)

1: **for** each property $\varphi \in \Phi$ **do**
2: **for** each $l \in lit\text{-}occur(\varphi)$ **do**
3: **if** $\Phi \models \varphi[l \leftarrow$ **false**$]$ **then**
4: $\varphi \leftarrow \varphi[l \leftarrow$ **false**$]$

The order in which the properties are checked is immaterial, because the substitution in line 4 does not remove behaviors from Φ, or, formally:

Lemma 3.2 *For $\varphi \in \Phi$, if for $l \in lit\text{-}occur(\varphi)$ it holds that $\Phi \models \varphi[l \leftarrow$ **false**$]$, then $(\{\Phi \setminus \varphi\} \cup \varphi[l \leftarrow$ **false**$]) \equiv \Phi$ (where '$\equiv$' denotes language equivalence).*

Proof: Since l is a literal in a formula which is in positive normal form, it is always the case that $\varphi[l \leftarrow$ **false**$] \Rightarrow \varphi$. Thus, $\{\Phi \setminus \varphi \cup \varphi[l \leftarrow$ **false**$]\} \Rightarrow \Phi$, and this is true for all $\varphi \in \Phi$. Since $\Phi \models \varphi[l \leftarrow$ **false**$]$, the other direction is trivial. The bi-directional implication constitutes equivalence.

Note that since each replacement of a non-affecting literal with **false** is equivalence-preserving, we have that all replacements, no matter in which order they are done, leave us with an equivalent set of properties, and also that all non-affecting literals that are found in one order of replacements are found in any other order. □

The order of removal of non-affecting literal occurrences *in a single formula*, however, can affect the final result, as we show in Section 4 (the reason for the difference is that the formulas in Φ are conjoined, whereas subformulas in a single formula can have an arbitrary Boolean structure).

3.3 Informative vacuity

Let $\varphi \in \Phi$ be a property such that

$$\Phi \not\models_v \varphi \tag{2}$$

and

$$M \models_v \varphi. \tag{3}$$

Note that Formula (2) holds for every property after applying VACUITY-WITHOUT-DESIGN(). There can be two possible reasons for vacuous satisfaction of φ in M (Formula (3)): either the specification is not tight enough, or the system does not have all the behaviors it is expected to have. It is the user's responsibility to decide between the two. In the first case, replacing the original specification with a tighter

specification (obtained by replacing one literal occurrence with **false** and simplifying) solves the problem. In the second case, we would like to give the user an example of an *interesting* behavior that on the one hand satisfies the properties, and on the other hand is missing from M. An interesting witness that shows non-vacuous satisfaction of φ in Φ has exactly these properties: it satisfies Φ and represents an interesting behavior that is missing from M. Thus, such a counterexample can be part of the output of a vacuity check. Enhancing M with this behavior, either manually or automatically (the former is more realistic, since such a behavior probably reflects an error in the design that should be fixed manually), is what we refer to as *informative vacuity*. Note that this information is of higher quality for the user than just giving her a behavior from $\Phi \times \bar{M}$ (a behavior that satisfies the properties but is missing in M) because it represents an interesting behavior as defined in Section 2. Algorithm 2 summarizes these steps.

Algorithm 2 Checking the property, checking vacuity if the property holds, and reporting a missing behavior from the system M otherwise.

INFORMATIVE VACUITY(Φ, M)

1: **for** each property $\varphi \in \Phi$ **do**
2: **if** $M \not\models \varphi$ **then** Abort("*Counterexample found*");
3: **for** each $l \in lit\text{-}occur(\varphi)$ **do**
4: **if** $M \models \varphi[l \leftarrow \textbf{false}]$ **then**
5: Let ce be a c-example to $\Phi \models \varphi[l \leftarrow \textbf{false}]$;
6: Abort("*Do one of the following:*
7: *Change property:* $\varphi \leftarrow \varphi[l \leftarrow \textbf{false}]$*, or*
8: *Change system:* ce *is a missing behavior in* M");

3.3.1 Over-restrictive environment

Another potential benefit of this algorithm arises from the fact that in practice, verification can fail not only because of errors in the specification or the verified system, but also because of problems in the environment module. Such modules are developed by verification engineers for focusing on a specific component in a large system. An environment module models the behavior of the interface with the verified component and hence the rest of the system can be abstracted away. Since it restricts the allowed input sequences, if it is not implemented correctly, it can be over- or under-restrictive, or even both simultaneously.

Over-restricted environment may lead to false positives, because the module is not verified for some existing behaviors of the system. It may also lead to vacuous satisfaction, because some of the behaviors expected by the specification are blocked by the environment (and may contain errors, thus leading to false positives). Since false positives are not reported during the verification, vacuous satisfaction is the only indication of an over-restricted environment. An interesting witness reported by INFORMATIVE-VACUITY can then be used in order to find the combinations of values of input variables that were wrongly omitted from the environment.

Figure 1. The systems M_1 and M_2

3.4 Putting it all together

Putting together the checks that were defined in subsections 3.1 – 3.3, results in Algorithm 3.

Algorithm 3 A process of checking properties with preliminary 'cheap' vacuity checks, and feedback as to missing behaviors in the system in case of vacuous satisfaction.

PRELIMINARY-CHECKS(Φ, M)

1: REMOVE-REDUNDANCY(Φ);
2: VACUITY-WITHOUT-DESIGN(Φ);
3: INFORMATIVE-VACUITY(Φ, M);

Example 1: We now demonstrate the usefulness of the new information presented to the user by Algorithm 3. Let $\varphi = p\,\mathcal{U}\,(q\,\mathcal{U}\,r)$ and Φ be the set of properties $\{\varphi, p\}$. It is easy to see that Φ does not contain redundant properties and that φ is satisfied non-vacuously in Φ, hence the first two steps of PRELIMINARY-CHECKS pass without changing Φ. Consider the system M_1 in Figure 1. It is easy to see that M_1 satisfies φ vacuously. Indeed, the eventuality $q\,\mathcal{U}\,r$ is satisfied at the initial state, and thus M_1 also satisfies $\varphi[p \leftarrow \textbf{false}]$.

Algorithm 2, in line 5, can generate a counterexample such as $p \cdot q \cdot r^{\omega}$. This can help the user realize that the missing behavior is one in which the initial state does not satisfy q.

Now consider M_2, in which φ is satisfied vacuously due to the restrictions of the environment of M_2. Let $I = \{i\}$ be the set of input signals. In M_2, both initial states are labeled

with p, and their labeling with q is defined by the value of i: $init(q) = i$. After the initial state M_2 does not receive inputs from the environment. In an unrestricted environment, M_2 has two initial states, one of which is labeled with p and q, and another is labeled with p. It is easy to see that φ is satisfied non-vacuously. However, if the environment is restricted by setting $i =$ **true**, then M_2 has only one initial state that is labeled with p and q, and thus φ is satisfied vacuously. Either the specification, the system, or the environment can cause this vacuity. As in the earlier case, an interesting witness provided by INFORMATIVE-VACUITY can make it easier to track the problem.

4 Mutual Vacuity

For each system M and property φ that it satisfies vacuously, there exists a set S of subsets of literal occurrences of φ, such that

$$\forall s \in S.\ M \models \varphi[l \leftarrow \textbf{false} \mid l \in s]. \tag{4}$$

In other words, there are alternative subsets of $lit\text{-}occur(\varphi)$ that can be replaced with **false** simultaneously without falsifying φ in M.

If S is not empty, then the output given by standard vacuity checks (a list of literal occurrences that can be replaced with **false** separately) has two roles:

- It provides an alternative property, which is shorter and stronger than φ, yet is still satisfied by M.
- It provides insight as to why the formula is vacuous. This insight can help fixing either the formula, the system or both.

The problem is that when S contains more than a single subset with a single subformula, this output alone is not ideal: shorter and stronger formulas can be derived, and further insight can be provided. These improvements are the subject of this and the next sections.

We now focus on the problem of detecting vacuous satisfaction with respect to several literal occurrences, which was first defined by Chechik et al. in [CG04a].

Definition 4.1 (MAX-VACUOUS**)** *Given a temporal logic formula φ and a Kripke structure M that satisfies φ, let* MAX-VACUOUS (φ, M) *be the maximal number of literal occurrences in $lit\text{-}occur(\varphi)$ that can be replaced with* **false** *without affecting the truth value of φ in M.*

The output of a standard vacuity check cannot be used for solving this problem, because vacuity is not monotonic: there can be a situation where $\varphi[l_1 \leftarrow$ **false**$]$ and $\varphi[l_2 \leftarrow$ **false**$]$ are satisfied separately, but $\varphi[l_1 \leftarrow$ **false**$, l_2 \leftarrow$ **false**$]$ is not satisfied. So checking vacuity of each literal occurrence separately does not give us a set of literal occurrences that can be replaced with **false** simultaneously. Chechik and Gurfinkel describe an algorithm that solves MAX-VACUOUS in exponential time and is based on multi-valued model checking [CG04b].

We start with the lower bound for the complexity of MAX-VACUOUS. We define MAX-VACUOUS-LIT (φ, M) as the maximal number of *literals* (not literal occurrences) that can be replaced with **false** without falsifying φ in M. The following theorem states the lower bound for the complexity of MAX-VACUOUS-LIT.

Theorem 4.2 MAX-VACUOUS-LIT *is $FP^{NP[\log n]}$-hard.*

Proof: We prove hardness in $\text{FP}^{NP[\log n]}$ by a reduction from the problem CLIQUE-SIZE, which is known to be $\text{FP}^{NP[\log n]}$-complete [Pap84, Kre88]. CLIQUE-SIZE is the problem of determining the size of the largest clique of an input graph G. The reduction works as follows. Let $G = \langle V, E\rangle$ be a graph. The Kripke structure M_G has a single node with a self-loop. The atomic propositions are the nodes of G, and the single node of M is labeled with all of them. The formula ψ_G is $\psi_G = \bigwedge_{\langle v,w\rangle \notin E}(v \vee w)$.

The reduction is not complete yet, since the nodes of G with the branching degree $|V| - 1$ do not appear in ψ_G. It is easy to see that all nodes with the branching degree $|V| - 1$ are members of the largest clique in G. Indeed, assume the opposite. That is, there is a node v with the branching degree $|V| - 1$ that does not belong to the maximal clique C of G. Let $C \subseteq V$ be the set of nodes of the maximal clique of G. Then, the subgraph of G induced by the set of nodes $C \cup \{v\}$ is also a clique: all nodes in C are connected to each other, and v is connected to all of them. This in the contrary to the assumption that C is the largest clique.

In order to include the nodes with the branching degree $|V| - 1$, we add to ψ_G a tautology that includes all of these nodes. The resulting formula φ_G is $\varphi_G = \psi_G \wedge$ (**true** $\vee$ $\bigvee_{br(v)=|V|-1} v$), where $br(v)$ stands for the branching degree of v.

Clearly, φ_G is satisfied in G. Nodes with the branching degree $|V| - 1$ appear once in φ_G and clearly can be replaced with **false** without falsifying φ_G. Let us focus on the formula ψ_G. All variables appear in ψ_G in the positive polarity. It is easy to see that replacing a subset S of the variables of ψ_G with **false** does not falsify ψ_G in M iff there are edges in G between each pair of nodes in S. Let W be the set of nodes with the branching degree $|V| - 1$. Then, the size of the maximal clique in G is $|W|$ plus the size of the maximal subset S of nodes of ψ_G that can be replaced with **false** without falsifying ψ_G in M_G, which is exactly the size of the maximal subset of variables of φ_G that can be replaced with **false** without falsifying φ_G in M_G. Thus, CLIQUE-SIZE (G) =MAX-VACUOUS (φ_G, M_G). □

We conjecture that the same hardness result is also true for MAX-VACUOUS, and the proof is left for future work.

Since LTL model checking is exponential in the size of the formula, there is no matching upper bound. For CTL formulas, however, we are able to prove completeness. Formally, let MAX-VACUOUS-LIT$_{\text{CTL}}$ be the problem MAX-VACUOUS-LIT defined for φ in CTL.

Lemma 4.3 MAX-VACUOUS-LIT$_{\text{CTL}}$ *is* $FP^{NP[\log n]}$*-complete.*

Proof: We note that the formula φ_G constructed in the proof of Theorem 4.2 is propositional, hence it can be viewed as either a CTL or an LTL formula. Thus, it remains to prove membership in $\text{FP}^{NP[\log n]}$. The proof is by describing an algorithm in $\text{FP}^{NP[\log n]}$ for solving MAX-VACUOUS-LIT$_{\text{CTL}}$. The algorithm queries an oracle O_{L_c} for membership in the language L_c defined as follows:

$$L_c = \{\langle \varphi, M, i\rangle : \text{MAX-VACUOUS-LIT}_{\text{CTL}}(\varphi, M) \geq i\}. \quad (5)$$

In other words, $\langle \varphi, M, i\rangle \in L_c$ if $M \models \varphi$ and there exists a set of literals of φ of size i that can be replaced with **false** simultaneously so that the resulting formula φ' is still satisfied in M. It is easy to see that $L_c \in NP$. Indeed, given a set S of literals of φ of size i, model checking φ' is linear in the size of φ' (and hence also in the size of φ) and in the size of M. Given φ and M, the algorithm for solving MAX-VACUOUS-LIT$_{\text{CTL}}$ performs a binary search on the value of MAX-VACUOUS-LIT$_{\text{CTL}}(\varphi, M)$, each time dividing the range of possible values for MAX-VACUOUS-LIT$_{\text{CTL}}(\varphi, M)$ by 2 according to the answer of O_{L_c}. The number of possible candidates for MAX-VACUOUS-LIT$_{\text{CTL}}(\varphi, M)$ is bounded by $|\varphi|$, and thus the number of queries to O_{L_c} is at most $\lceil \log |\varphi| \rceil$. □

Based on Theorem 4.2, we can also prove several related complexity results.

Theorem 4.4 *Given a formula φ, a system M that satisfies φ, and an integer k, we have:*

1. *The decision problem of whether there exists a set of k literals of φ that can be replaced with* **false** *without falsifying φ in M is NP-hard.*

2. *The search problem of finding a set of literals of size k that can be replaced with* **false** *without falsifying φ in M (if such a set exists) is FNP-hard (where the class FNP is the class of search problems whose matching decision problems are in NP).*

3. *The problem* MAX-VACUOUS-SET-LIT *of computing the maximum set of literals that can be replaced with* **false** *without falsifying φ in M is $FP^{FNP[\log n]}$-hard.*

Proof: We observe that the reduction given in the proof of Theorem 4.2 can be used in all three cases. Indeed, there is a one-to-one correspondence between the literals that are assigned **false** without falsifying the specification and the nodes that form the clique. Since search problems of NP-complete decision problems are known to be FNP-complete, FNP-hardness of the problem of finding a set of literals of size k follows. For hardness of MAX-VACUOUS-SET-LIT in $\text{FP}^{FNP[\log n]}$ it is enough to observe that the oracle is FNP-complete (by the previous result), and thus there exists a generic reduction from a problem in $\text{FP}^{FNP[\log n]}$ to MAX-VACUOUS-SET-LIT. □

For formulas in CTL, it is easy to prove completeness results similar to the hardness results stated in Theorem 4.4: the proof of membership is similar to the proof of Lemma 4.3.

4.1 The parameterized complexity of MAX-VACUOUS-LIT

We note that the hardness of MAX-VACUOUS-LIT is determined by its dependence on the size of the formula, and not the size of the whole input. Since in most cases the size of the specification is much smaller than the size of the system, the hardness result might not mean that the problem is infeasible. The complexity of MAX-VACUOUS-LIT is more accurately described using parameterized complexity theory [DF95]. In parameterized complexity, there are two parameters that describe the size of the input, and the complexity of the problem depends on each one of these parameters differently.

The parameterized complexity class W[1] is defined as follows: $L \in W[1]$ if there exists a non-deterministic Turing machine M that accepts all words in L of the form $\langle x_1, x_2\rangle$, with $|x_1| = n$ and $|x_2| = k$ in an execution of length polynomial in n. The CLIQUE problem is complete in W[1], where n is the size of the graph, and k is the size of the clique (that is, the problem is "whether there exists a clique of size k in an input graph of size n") [DF95]. By a reduction from CLIQUE presented above, the decision problem of whether there exists a set of k literals of φ that can be replaced with **false** without falsifying φ in M is W[1]-complete, and therefore there exists an algorithm for solving it in time polynomial in n and exponential in k. Since k is bounded by the size of $|\varphi|$, the problem might be in fact feasible even for large systems, as long as the specifications are sufficiently small.

4.2 Solving MAX-VACUOUS-SET.

We now present an algorithm for solving MAX-VACUOUS-SET, the problem of finding the maximal subset of $lit\text{-}occur(\varphi)$ that can be replaced with

Figure 2. Converting a Büchi automaton to a conjunctive Büchi automaton.

Figure 3. (left) A Büchi automaton $\mathcal{CB}_{\neg\varphi}$ for $\varphi = pU(qUr)$. (right) The automaton $\mathcal{CB}_{\neg\varphi,S}$ for $S = \bar{p}, \bar{q}$.

false without falsifying φ in M. The algorithm can be easily changed to also solve the MAX-VACUOUS-SET-LIT problem that was defined earlier.

For a given formula φ, we begin with transforming $\mathcal{B}_{\neg\varphi}$ to an automaton $\mathcal{CB}_{\neg\varphi}$ that we call a *conjunctive* Büchi automaton, in which each transition is labeled with a set of literals (implicitly conjoined). The transformation is done by 1) replacing the propositional formulas labeling the transitions of $\mathcal{B}_{\neg\varphi}$ with their DNF representation, and then 2) replacing each transition with as many transitions as there are clauses in the DNF representation, and finally 3) labeling each such transition with the set of literals of the clause it represents (see an example in Figure 2).

Let S be a subset of the literal occurrences labeling the transitions of $\mathcal{CB}_{\neg\varphi}$ (i.e., a subset of $lit\text{-}occur(\neg\varphi)$). Let $\mathcal{CB}_{\neg\varphi,S}$ denote $\mathcal{CB}_{\neg\varphi}$ after erasing all the literal occurrences in S from $\mathcal{CB}_{\neg\varphi}$'s labels (technically we will keep the literals and only mark them as erased, since we will possibly unmark them in a later stage). Note that this substitution is likely to enlarge the language accepted by this automaton. Thus, while $\mathcal{L}(M \times \mathcal{CB}_{\neg\varphi})$ is empty, it is possible that $\mathcal{L}(M \times \mathcal{CB}_{\neg\varphi,S})$ is not. Algorithm 4 finds the maximal set S such that $\mathcal{L}(M \times \mathcal{CB}_{\neg\varphi,S})$ is still empty, and then reports the elements of S as the solution to the MAX-VACUOUS-SET problem.

An accepted trace in the product $M \times \mathcal{CB}_{\neg\varphi,S}$ indicates that S should be changed. Consider the projection of this trace to $\mathcal{CB}_{\neg\varphi,S}$, denoted π. Since $M \times \mathcal{CB}_{\neg\varphi}$ is empty, it means that there are literal occurrences in S that do not allow synchronization of π and $\mathcal{CB}_{\neg\varphi}$. Denote this set by $Removing\text{-}set_{\neg\varphi}(\pi, S)$. It is easy to find this set in a conjunctive Büchi: simply collect all the S literal occurrences labeling $\mathcal{CB}_{\neg\varphi,S}$ that disagree with π.

Algorithm 4 begins with assigning S the set $lit\text{-}occur(\neg\varphi)$. If there is a trace accepted by the product $M \times \mathcal{CB}_{\neg\varphi,S}$, it adds the removing set of its projection to the set of multisets Π. It then solves the following optimization problem in line 6:

Definition 4.5 (The Minimum-hitting-set problem) *Given a collection C of subsets of a finite set U, find a subset $u \subseteq U$ of minimal size such that u has a non-empty intersection with every element in C.*

In our case C is Π and U is $lit\text{-}occur(\neg\varphi)$[3]. Denote by minimum_hitting_set a procedure that solves the minimum-hitting set problem.

[3]Since C and U in our case are multisets, the literal occurrences should be renamed before solving the minimum-hitting-set problem.

Algorithm 4 Assuming $M \models \varphi$, MAX-VACUOUS-SET finds the largest number of literal occurrences that can be replaced in φ with **false** while still being satisfied by M.

MAX-VACUOUS-SET (M, φ)

1: $\Pi = \emptyset$;
2: $S = lit\text{-}occur(\neg\varphi)$;
3: **while** $S \neq \emptyset$ **do**
4: **if** $\mathcal{L}(M \times \mathcal{CB}_{\neg\varphi,S})$ is not empty **then**
5: For π a projection of a countertrace to $\mathcal{CB}_{\neg\varphi,S}$, add $Removing\text{-}set_{\neg\varphi}(\pi, S)$ to Π;
6: $S = lit\text{-}occur(\neg\varphi) \setminus$ minimum_hitting_set(Π);
7: **else**
8: Abort ("Replace φ with $\varphi[l \leftarrow$ **false**$|l \in S]$");
9: Abort ("φ is not vacuous");

The procedure minimum_hitting_set receives as input Π and returns the smallest set of literals that intersects all these sets, and hence eliminates all the traces in Π. S is then updated with the complement of this set, because these are the literals that should *not* be replaced with **true** in the next iteration, in line 4.

Example 2: Consider the property $\varphi = p\,\mathcal{U}\,(q\,\mathcal{U}\,r)$ and $\mathcal{CB}_{\neg\varphi}$ on the left of Figure 3 (since the propositions in $\neg\varphi$ are literals, there is no difference in this case between $\mathcal{B}_{\neg\varphi}$ and $\mathcal{CB}_{\neg\varphi}$). Now consider an iteration of the algorithm in which $S = \{\bar{p}, \bar{q}\}$. The drawing on the right of Figure 3 shows $\mathcal{CB}_{\neg\varphi,S}$, where the S literals appear as being erased. A possible countertrace to the check in line 4 is $\pi_1 = s_0 \cdot (pq\bar{r}) \cdot s_1 \cdot (pq\bar{r}) \cdot s_4 \cdot (pq\bar{r}) \cdot s_5 \cdot ((\bar{p}qr) \cdot s_6)^\omega$ (to be more precise, this is the projection of the countertrace to $\mathcal{CB}_{\neg\varphi,S}$). The removing set for π_1 is $\{\bar{p}, \bar{q}\}$ because removing either $\bar{p}$ or $\bar{q}$ from S eliminates this trace. Solving the minimum hitting set (assuming this is the only trace in Π) results in one of $\bar{p}$ or $\bar{q}$. Suppose the algorithm outputs the latter, and that this leads to another counterexample trace $\pi_2 = s_0 \cdot \bar{p}q\bar{r} \cdot s_3 \cdot (\bar{p}q\bar{r} \cdot s_6)^\omega$. The eliminating set for π_2 is $\{\bar{q}\}$, hence now $\Pi = \{\{\bar{p}, \bar{q}\}, \{\bar{q}\}\}$, and the minimum hitting set is $\bar{q}$. This demonstrates that S is not strictly decreasing in size at each iteration. This process continues until it finds S such that $\mathcal{L}(M \times \mathcal{CB}_{\neg\varphi,S})$ is empty.

4.2.1 Complexity of Algorithm 4

Transforming $\mathcal{B}_{\neg\varphi}$ to $\mathcal{CB}_{\neg\varphi}$ can increase the number of transitions exponentially in theory, but note that a) the exponential growth is in the size of $\neg\varphi$, not $\mathcal{B}_{\neg\varphi}$, hence the size of $\mathcal{CB}_{\neg\varphi}$ is still only a single exponent in $|\neg\varphi|$, like $\mathcal{B}_{\neg\varphi}$ itself, and b) in practice each propositional label of $\mathcal{B}_{\neg\varphi}$ is very short.

At each iteration Algorithm 4 computes a set S of literal occurrences that eliminates all countertraces seen so far. The next countertrace cannot be eliminated by the same set (due to line 4) hence the sets S monotonically increase in strength at each iteration. Thus, the number of iterations is bounded by the number of subsets of literal occurrences of φ, which is $2^{|\varphi|}$. The complexity of each iteration is the complexity of model checking plus the complexity of computing the minimum hitting set over the current set of traces (which is also bounded by $2^{|\varphi|}$). Minimum-hitting-set is a known NP-complete problem, and there exist algorithms that are exponential in the size of U and polynomial in the size of the sets in C for solving it. In our case U is the set $lit\text{-}occur(\neg\varphi)$ and hence expected to be small. Hence, since the dependence in the size of M is polynomial (because of model checking), the resulting overall complexity is $O(|M|2^{2|\varphi|})$. The minimum-hitting set problem can also be approximated within $1 + \ln|U|$ by a polynomial algorithm [Joh74], which leads to an approximation algorithm for solving MAX-VACUOUS-SET.

5 Vacuity Extended to Responsibility

Responsibility was formally defined in [CH04] based on the definition of causality by Halpern and Pearl [HP01]. The notion of responsibility extends the "all-or-nothing" approach to causal relations. Instead of saying that "either A is a cause of B or it is not", we can talk about *responsibility* of A for B, where responsibility is a number from 0 to 1. Responsibility 1 indicates that there is a counterfactual dependence between A and B, and responsibility 0 means that A has no influence on B. A number between 0 and 1 indicates some degree of responsibility of A in B.

The notion of responsibility was tied to model checking in [CHK03], where it is shown that coverage in model checking can be refined by responsibility. Informally, coverage is a "twin" of vacuity: in coverage we examine the role of atomic propositions in the satisfaction of the specification, and in vacuity we examine the role of subformulas. Similarly to the connection between responsibility and coverage, there also exists a connection between responsibility and vacuity. Formally, we introduce the following definition (which is similar to the definition of [CH04] adapted to vacuity):

Definition 5.1 *For a specification φ in positive normal form that is satisfied in M and a literal occurrence l of φ, we say that the* degree of responsibility $dr(l, \varphi, M)$ of l in the satisfaction of φ in M *is $1/(k+1)$ if there exists a subset S of k literal occurrences of φ such that $l \notin S$ and:*

1. *replacing all literal occurrences in S with* **false** *does not falsify φ in M;*
2. *replacing all literal occurrences in $S \cup \{l\}$ with* **false** *falsifies φ in M;*
3. *S is the smallest subset of literal occurrences that satisfies the above conditions.*

*If $M \models \varphi[l \leftarrow$ **false**$]$ and there is no such subset S, we say that $dr(l, \varphi, M) = 0$.*

In other words, k is the size of the minimal subset of literal occurrences of φ that need to be replaced with **false** in order to make the value of φ in M depend on l (if such subset exists). Note that if l affects φ in M, then $dr(l, \varphi, M) = 1$ (because the smallest S is of size 0).

The degree of responsibility of a literal occurrence in the satisfaction of φ in M can be viewed as a "fine-grain" vacuity. That is, for literals that do not affect the satisfaction of φ in M, it allows us to distinguish between those that play no role whatsoever in the satisfaction of φ in M and those that play some role but there is no direct counterfactual dependence between their value and the satisfaction of the specification.

Example 3: Consider a formula $\varphi_n : \bigvee_{i=1}^{n} \mathbf{AG}p_i$. For every $n > 1$, φ_n is satisfied vacuously in a system M in which all states are labeled with $\{p_1, \ldots, p_n\}$. The degree of responsibility of p_i for $1 \leq i \leq n$ in φ_n is $1/n$, because all other literals should be replaced with **false** in order to eliminate vacuity. The larger n is, the smaller is the degree of responsibility of each p_i in the satisfaction of φ_n in M. Thus, the degree of responsibility gives us a measure of redundancy in the specification. □

In the case of responsibility for coverage, it was shown in [CHK03] that computing the degree of responsibility of a literal l for the satisfaction of φ in M is $\text{FP}^{NP[\log n]}$-complete, by a reduction from CLIQUE-SIZE. Their proof can be easily adapted for the case of responsibility for vacuity.

We note that the degree of responsibility can be computed by a brute-force algorithm that checks the effect of replacing each subset of $lit\text{-}occur(\varphi)$ with **false** on the vacuous satisfaction of φ in M. The complexity of the brute-force algorithm is clearly exponential, but only in the size of φ, and not in the size of M. Thus, for small properties even the general problem can be feasible. We also note that similarly to the discussion in [CHK03], there are some special cases for which computing the degree of responsibility is polynomial in both the size of φ and in the size of M.

The connection between MAX-VACUOUS *and responsibility* is formulated in the following theorem.

Theorem 5.2 *Let φ be a specification satisfied vacuously in M, and let φ' be φ after the set of literal occurrences found by solving* MAX-VACUOUS-SET *has been replaced with* **false**. *Then the responsibility of all literal occurrences of φ' is* 1.

Proof: Assume the contrary. That is, there exists a literal occurrence l in φ' whose degree of responsibility in the satisfaction of φ' in M is less than 1. Then, l can be replaced with **false** without falsifying φ' in M. This means that all literals occurrences in MAX-VACUOUS-SET$(\varphi, M) \cup \{l\}$ can be replaced with **false** in φ without falsifying it in M, which contradicts the maximality of MAX-VACUOUS-SET(φ, M). □

Why, then, bother with responsibility if we can simply eliminate all literal occurrences with responsibility smaller than 1? Because as we indicated in the beginning of this section, the user might be interested in the insight it provides. First, as described in Example 3, responsibility provides a measure of redundancy in the specification. Second, responsibility helps to distinguish between literals that do not participate in the verification process at all (for example, literals that are present in the specification, but not in the system) and literals that can be eliminated, but can also be useful in the verification process. Consider, for example, the specification $\varphi = \mathbf{G}(a \vee b \vee c \vee d)$ and a system M in which all states are labeled with a, and in addition each state is labeled with either b or c (but not both). Clearly, d plays no role at all in verifying φ in M. With a mutual-vacuity analysis (applying MAX-VACUOUS-SET) we discover that the maximum set of literals that can be eliminated from φ is $\{b, c, d\}$, and thus the tightest specification is $\mathbf{G}a$. Responsibility gives more information: the degree of responsibility of d is 0, and the degree of responsibility of the other literals is 0.5 each. This means that each of the literals a, b, and c plays some role in the satisfaction of φ in M, in contrast to d.

6 Checking a Real-Life Example

We experimented with a real specification of a hardware block taken from the Prosyd project [Pro05] in order to assess the applicability of the methods suggested in this article.

The design is a real-life block consisting of a producer, a consumer, and a data receiver, as described in [Pro05]. The set of properties contains approximately 50 basic properties[4], and a set of environment restrictions. Building the Buchi automaton corresponding to the conjunction of all of them turned out to be too time-consuming in our experimental setting (we used the Wring [SB00] script, which builds Buchi automata from LTL formulas). We therefore broke the set of properties to three subsets, sacrificing thus the quality of the checks. Here we bring the results with the first subset, which includes the first 17 properties in the Prosyd document [Pro05].

It turns out that this subset of properties is very redundant. Nine out of the 17 properties were declared redundant by our implementation of REMOVE-REDUNDANCY. Our system checks each property (in the chronological order of the properties in the input file) and removes it if it is redundant with respect to those properties that were not yet removed. Hence, as was explained in Section 3.3.1, the result is sensitive to the order in which the properties are checked. In other words, it could be that a different order would reveal more redundant properties. The table in Figure 4 shows the properties in their original order. Those that are declared redundant by our system are marked in the middle column.

After removing the nine redundant properties, VACUITY-WITHOUT-DESIGN discovers only one vacuous property: property 2.4A can be replaced with the stronger property:

$$\text{G(error}\rightarrow\text{X}(\neg\text{error}\wedge\neg\text{rdy})).$$

We also ran VACUITY-WITHOUT-DESIGN without first running REMOVE-REDUNDANCY, to check to what degree the removed redundant properties were also vacuous. It turns out that only two such properties are vacuous: 2.4 and 2.6. Let us start with the latter, which is a simpler case. Property 2.6 can be replaced with:

$$(\neg\text{rdy U start}),$$

which, note, is the same as property 2.6A.

In Property 2.4, each of the literals other than 'error' can be replaced with false (individually), i.e., the following three stronger variations on 2.4 hold in any design that satisfies the other properties:

$$\text{G(error}\rightarrow\text{X}(\neg\text{error}\vee\neg\text{rdy}))$$
$$\text{G((error}\wedge\text{rdy)} \rightarrow\text{X}(\neg\text{rdy}))$$
$$\text{G((error}\wedge\text{rdy)} \rightarrow\text{X}(\neg\text{error})$$

This is where MAX-VACUOUS comes into the picture: we would like to know what is the maximal set of literals in this property that can simultaneously be replaced with **false**. In this case the number of subsets to try is small so we can find the exact answer, which is that we can replace the second and fourth literals with **false** simultaneously. Note that this yields exactly property 2.2C. Indeed, property 2.2C subsumes property 2.4 and hence makes it both redundant and vacuous. The same applies to property 2.6A which subsumes property 2.6.

[4]For some of the properties the document declares alternative formulations. We do not count these nor include them in our experiments.

#	R?	Property
2.1	Y	(¬start∧¬end∧¬rdy)U¬rst
2.1A		G(rst→(¬start∧¬end∧¬rdy))
2.2A		G(start→X(¬start))
2.2B		G(status_valid→X(¬status_valid))
2.2C	Y	G(error→X(¬error))
2.3	Y	G(end→X(¬end ∨(end∧start∧rdy)))
2.4	Y	G((error∧rdy) →X(¬error∨¬rdy))
2.4A		G((error∧rdy) →X(¬error∧¬rdy))
2.5	Y	¬end U start
2.6	Y	(¬rdy U start)∨G(¬rdy))
2.6A		¬rdy U start
2.7	Y	G(start→(rdy U(rdy∧(end∨stop∨ error))))
2.7A		G(start→(rdy U(end∨stop∨error)))
2.8		G((end∨stop∨error)→X(¬rdy U start))
2.9		G((end∨stop∨error)→rdy)
2.9A	Y	G(end→rdy)
2.9B	Y	G(stop→rdy)

Figure 4. The first 17 properties in Prosyd, more than half of which declared redundant (denoted by 'Y') by our system.

7 Conclusions and Future Work

We addressed two dimensions of vacuity: the computational effort and the information that is given to the user. As for the former, we proposed several preliminary vacuity checks that can be done without the design. As our experiments with real designs have shown, they are strong enough to detect many problems (e.g., they detected that more than half of the properties in the design specification we examined are redundant). We also showed how information gathered in this process can be helpful to the user in pointing to missing interesting behavior from the design (typically due to over-restrictive environment). A usability study is required in order to assess the helpfulness of this information in an industrial setting – a study yet to be performed. We also studied various ways to gain information more refined than standard vacuity - including the problem of mutual vacuity and the extension of vacuity to responsibility. We formally defined various problems associated with the above issues and established corresponding complexity results.

References

[Acc] Accelera Property Specification Language: Reference Manual – Version 1.1. http://www.eda-stds.org/vfv/docs/psl_lrm-1.1.pdf.

[AFF+03] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi. Enhanced vacuity detection in linear temporal logic. In *Proc. of CAV*, LNCS 2725:368–380, 2003.

[BB94] D. Beaty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. of DAC*, pp. 596–602. IEEE Computer Society, 1994.

[BBER01] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. *Formal Methods in System Design*, 18(2):141–162, 2001.

[BFG+05] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. of CHARME*, LNCS 3725:191–206, 2005.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pp. 1–12, Stanford, 1962.

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

[CG04a] M. Chechik and A. Gurfinkel. Extending extended vacuity. In *Proc. of FMCAD*, LNCS 3312, 2004.

[CG04b] M. Chechik and A. Gurfinkel. How vacuous is vacuous? In *Proc. of TACAS*, LNCS 2988:451–466, 2004.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. DAC*, pp. 427–432. IEEE Computer Society, 1995.

[CH04] H. Chockler and J.Y. Halpern. Responsibility and blame: a structural-model approach. *(AIR*, 22:93–115, 2004.

[CHK03] H. Chockler, J. Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification? CoRR cs.LO/0312036, 2003.

[DF95] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness ii: On completeness for w[1]. *TCS*, 141(1,2):109–131, 1995.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, B(16): 997–1072. MIT press, 1990.

[Gup06] A. Gupta. *Learning Abstractions for Model Checking*. PhD thesis, CMU, 2006.

[HP01] J.Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach — part 1: Causes. In *Proc. of UAI*, pp. 194–202, 2001. Morgan Kaufmann.

[Joh74] D.S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[Kre88] M.W. Krentel. The complexity of optimization problems. *Journal of the CSS*, 36:490–509, 1988.

[Kup06] O. Kupferman. Sanity checks in formal verification. In *Proc. CONCUR*, LNCS 4137:37–51, 2006.

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

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

[Nam04] K. S. Namjoshi. An efficiently checkable, proof-based formulation of vacuity in model checking. In *Proc. of CAV*, LNCS 3114, pp. 57–69, 2004.

[Pap84] C.H. Papadimitriou. The complexity of unique solutions. *Journal of ACM*, 31:492–500, 1984.

[Pro05] Prosyd: Property-Based System Design. "Property-by-Example" Guide: a Handbook of PSL Examples. http://www.prosyd.org/, 2005.

[PS02] M. Purandare and F. Somenzi. Vacuum cleaning ctl formulae. In *Proc. CAV*, LNCS 2404:485-499, 2002.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Progr.*, LNCS 137:337–351, 1981.

[SB00] F. Somenzi and R. Bloem. EfficientBüchi automata from LTL formulae. In *Proceedings of 12th CAV*, LNCS 1855:248–263, 2000.

[SDG06] J. Simmonds, J. Davies, and A. Gurfinkel. VAQTREE: Efficient Vacuity Detection for Bounded Model Checking. Tool Presentation in *FM*, 2006.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.