Vacuity in Testing

Thomas Ball[1] and Orna Kupferman[2]

[1] Microsoft Research, tball@microsoft.com
[2] Hebrew University, orna@cs.huji.ac.il

Abstract. In recent years, we see a growing awareness to the importance of assessing the quality of specifications. In the context of model checking, this can be done by analyzing the effect of applying mutations to the specification or the system. If the system satisfies the mutated specification, we know that some elements of the specification do not play a role in its satisfaction, thus the specification is satisfied in some *vacuous* way. If the mutated system satisfies the specification, we know that some elements of the system are not *covered* by the specification. Coverage in model checking has been adopted from the area of testing, where coverage information is crucial in measuring the exhaustiveness of test suites. It is now time for model checking to pay back, and let testing enjoy the rich theory and applications of vacuity. We define and study vacuous satisfaction in the context of testing, and demonstrate how vacuity analysis can lead to better specifications and test suites.

1 Introduction

The realization that hardware and software systems can, and often do, have bugs brought with it two approaches for reasoning about the correctness of systems. In *testing*, we execute the system on input sequences and make sure its behavior meets our expectation. In *model checking*, we formally prove that the system satisfies its specification.

Each input sequence for the system induces a different execution, and a system is correct if it behaves as required for all possible input sequences. Checking all the executions of a system is an infeasible task. Testing can be viewed as a heuristic in which the execution of only some input sequences is checked [4]. It is therefore crucial to measure the exhaustiveness of the input sequences that are checked. Indeed, there has been an extensive research in the testing verification community on *coverage metrics*, which provide such a measure. Coverage metrics are used in order to monitor progress of the verification process, estimate whether more input sequences are needed, and direct simulation towards unexplored areas of the system. Coverage metrics today play an important role in the system validation effort [22]. For a survey on the variety of metrics that are used in testing, see [13, 20, 23].

Since testing suites are typically not exhaustive, the verification community welcomed the idea of model checking, where the system is formally proven to be correct with respect to all input sequences [12]. In the last few years, however, there has been growing awareness to the importance of suspecting the system and the specification of containing an error also in the case model checking succeeds. The main justification of such suspects are possible errors in the (often not simple) modeling of the system or of the behavior.

Early work on "suspecting a positive answer" concerns the fact that temporal logic formulas can suffer from antecedent failure [2]. For example, verifying

a system with respect to the specification $\varphi = AG(req \rightarrow AF\,grant)$ ("every request is eventually followed by a grant"), one should distinguish between satisfaction of φ in systems in which requests are never sent, and satisfaction in which φ's precondition is sometimes satisfied. Evidently, the first type of satisfaction suggests some unexpected properties of the system, namely the absence of behaviors in which the precondition was expected to be satisfied.

In [3], Beer et al. suggested a first formal treatment of vacuity. As described there, vacuity 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 environment" [3]. The definition of vacuity according to [3] is based on the notion of subformulas that do not affect the satisfaction of the specification. Consider a model M satisfying a specification φ. A subformula ψ of φ *does not affect* (the satisfaction of) φ in M if M also satisfies all formulas obtained by modifying ψ arbitrarily. In the example above, the subformula *grant* does not affect φ in a model with no requests. Now, M satisfies φ vacuously if φ has a subformula that does not affect φ in M. A general method for vacuity definition and detection was presented in [17] and the problem was further studied in [1, 6, 7]. It is shown in these papers that for temporal logics such as LTL, the problem of vacuity detection is in PSPACE — not harder than model checking.

When the system is proven to be correct, and vacuity has been checked too, there is still a question of how complete the specification is, and whether it really covers all the behaviors of the system. It is not clear how to check completeness of the specification. Indeed, specifications are written manually, and their completeness depends entirely on the competence of the person who writes them. The motivation for a completeness check is clear: an erroneous behavior of the system can escape the verification efforts if this behavior is not captured by the specification. In fact, it is likely that a behavior not captured by the specification also escapes the attention of the designer, who is often the one to provide the specification.

Measuring the exhaustiveness of a specification in model checking ("do more properties need to be checked?") has a similar flavor as measuring the exhaustiveness of a test suite in testing ("do more input sequences need to be checked?"). Nevertheless, while for testing it is clear that coverage corresponds to activation during the execution on the given input sequence, it is less clear what coverage should correspond to in model checking. Indeed, all reachable parts of the system may be visited during the model-checking process, regardless of the role they play in the satisfaction of the specification. Early work on coverage metrics in model checking [15] adopted the idea of *mutation testing*, developed in the context of software testing [5]. The metric in [15], later followed by [8–10], is based on *mutations* applied to the model of the system. Essentially, a state in the model of the system is covered by the specification if modifying the value of a variable in the state renders the specification untrue. In [11], the authors took the adoption of coverage from testing to model-checking one step forward and adjusted various coverage metrics that are used in testing to the model-

checking setting. For example, branch coverage in testing checks that the test suites has led to an execution of all branches. Accordingly, branch coverage in model checking, studies the effect of disabling branches on the satisfaction of the specification.

In this paper we adopt the research that has been done on vacuity in the model-checking community to the setting of testing. Essentially, as in model checking, a test suite T passes a specification vacuously if we can modify the specification to one that would be harder to satisfy, and still T passes. We define and study vacuity for three settings of testing. The first setting considers the expected extension of LTL vacuity in the context of model checking. Thus, we check whether a given test suite could have passed even a stronger specification.

The second setting is that of *run-time verification*: we assume the specification is a monitor that is executed in parallel to the system, and an input sequence passes the test if the monitor does not get stuck on it [14]. Thus, the properties we are checking are safety properties, and the monitor gets stuck when a violation of the safety property has been detected. Our definition of vacuity in this setting refers to the transitions of the monitor: a test suite passes vacuously if it passes also with a monitor with fewer transitions.

The third setting is that of *software checking*: we assume the system is a procedure that terminates, and the specification is Boolean function, to which we feed both the input to the systems and its output. For example, the system may be a procedure that sorts a list of numbers, and the specification is a function that, given two lists, checks that a second list is the first list, sorted. We model both the system and the specification in a simple programming language. Our definition of vacuity refers to branches in the specification: a test suite passes vacuously if it does not branch-cover the specification, i.e., not all branches of the specification procedure are executed on the process of checking the test suite.

Our vacuity check is complementary to coverage checking. The rational behind vacuity checking in testing is that the structure of the specification is often different from the structure of the system: they may be designed by different designers, and the specification may refer to properties that are irrelevant in the system (for example, the output list being a permutation of the input list, in the case of sorting). Thus, there are cases in which all elements of the system have been covered, and still some elements of the specification are not covered, causing the test suite to pass vacuously. As we demonstrate in our examples, an attempt to cover all elements of the specification can then reveal bugs. In addition, we show that in most cases, the complexity of detecting a vacuous pass does not exceed that of testing. In particular, in the case of a deterministic specification, vacuity checking can be easily combined with the testing process.

2 Vacuity in Model Checking

In this section we describe the basic definitions of vacuity. We model a system by a sequential circuit (*circuit*, for short) $\mathcal{C} = \langle I, O, S, \theta, \delta, \rho \rangle$, where I is a set of input signals, O is a set of output signals, S is a set of states, $\theta : 2^I \rightarrow S$ is an initialization function that maps every input assignment (that is, assignment to the input signals) to a state, $\delta : S \times 2^I \rightarrow S$ is a transition function that

maps every state and input assignment to a successor state, and $\rho : S \to 2^O$ is an output function that maps every state to an output assignment (that is, an assignment to the output signals).[3]

Note that the interaction between the circuit and its environment is initiated by the environment. Once the environment generates an input assignment $i \in 2^I$, the circuit starts reacting with it from the state $\theta(i)$. Note also that the circuit is deterministic and receptive. That is, $\theta(s)$ and $\delta(s, i)$ are defined for all $s \in S$ and $i \in 2^I$, and they suggest a single state.

Given an input sequence $\xi = i_0, i_1, \ldots \in (2^I)^\omega$, the *execution* of $\mathcal{C}$ on ξ is the path π_ξ that $\mathcal{C}$ traverses while reading ξ. Formally, $\pi_\xi = s_0, s_1, \ldots \in S^\omega$, where $s_0 = \theta(i_0)$ and for all $j \geq 0$, we have $s_{j+1} = \rho(s_j, i_j)$. The *computation* of $\mathcal{C}$ on ξ is then the word $w_\xi = w_0, w_1, \ldots \in (2^{I \cup O})^\omega$ such that for all $j \geq 0$, we have $w_j = i_j \cup \rho(s_j)$. The language of $\mathcal{C}$, denoted $L(\mathcal{C})$ is union of all its computations. We sometimes refer also to executions and computations of $\mathcal{C}$ on finite words.

A specification to a system can be given either in terms of an LTL formula over atomic propositions in $I \cup O$ (we will consider also specifications given in terms of a *monitor* over the alphabet $2^{I \cup O}$; this setting, however, was not yet studied in the context of vacuity). We assume the reader is familiar with the syntax and the semantics of LTL. We recall the definition of vacuity in model checking.

Consider a circuit $\mathcal{C}$ and an LTL formula φ that is satisfied in $\mathcal{C}$. In the *single-occurrence* approach to LTL vacuity [17], we check that each of the occurrences of a subformula of φ has played a role in the satisfaction of φ in $\mathcal{C}$. Each occurrence σ of a subformula ψ of φ has a *polarity*. The polarity is positive if σ appears under an even number of negations, and is negative is σ appears under an odd number of negations. When the polarity of σ is positive, replacing σ with $false$ results in a formula that is harder to satisfy. Dually, when the polarity is negative, replacing σ with $true$ results in a formula that is harder to satisfy. Let $\bot_\sigma$ stand for $false$ if the polarity of σ is positive, and stand for $true$ if the polarity is negative.

We say that an occurrence σ of a subformula of φ *does not affect the satisfaction of* φ *in* $\mathcal{C}$ if $\mathcal{C}$ also satisfies the formula $\varphi[\sigma \leftarrow \bot_\sigma]$, in which σ is replaced by the most challenging replacement. For example, if $\varphi = G(\neg req \vee Fack)$, then $\varphi[req \leftarrow \bot_{req}] = GFack$ and $\varphi[ack \leftarrow \bot_{ack}] = G\neg req$. A specification φ is *vacuously satisfied in* $\mathcal{C}$ (in the single-occurrence approach) if φ has an occurrence of a subformula that does not affect its satisfaction in $\mathcal{C}$.

The *multiple-occurrence* approach to LTL vacuity in model checking considers LTL formulas augmented with universal quantification over atomic propositions. Recall that an LTL formula over a set AP of atomic propositions (typically $AP = I \cup O$) is interpreted over computations of the form $w = w_0, w_1, w_2, \ldots$, with $w_j \in 2^{AP}$. The path then satisfies a formula of the form $\forall x.\varphi$, where φ is an LTL formula and x is an atomic proposition, if φ is satisfied in all the computations that agree with w on all the atomic propositions except (maybe) x.

[3] Typically, $S = 2^C$ for a set C of control signals. Here, we are not going to refer to the control signals, and hide them in the notation.

Thus, $w \models \forall x.\varphi$ iff $w' \models \varphi$ for all $w' = w'_0, w'_1, w'_2, \ldots$ such that $w'_j \cap (AP \setminus \{x\}) = w_j \cap (AP \setminus \{x\})$ for all $j \geq 0$. As with LTL, a circuit $\mathcal{C}$ satisfies $\forall x.\varphi$ if all computations of $\mathcal{C}$ satisfy $\forall x.\varphi$.

We say that a subformula ψ of φ *does not affect the satisfaction of* φ *in* $\mathcal{C}$ if $\mathcal{C}$ also satisfies the formula $\forall x.\varphi[\psi \leftarrow x]$, in which ψ is replaced by a universally quantified proposition [1]. Intuitively, this means that $\mathcal{C}$ satisfies φ even with the most challenging assignments to ψ. Finally, a specification φ is *vacuously satisfied in* $\mathcal{C}$ (in the multiple-occurrence approach) if φ has a subformula that does not affect its satisfaction in $\mathcal{C}$.

3 Vacuity Checking in Testing

In the context of testing, we have two types of vacuous satisfaction. Consider a system $\mathcal{C}$, a specification $\mathcal{S}$, and a test suite T such that all the input sequences in T result in computations of $\mathcal{C}$ that satisfy $\mathcal{S}$.

- **Strong vacuity** is independent of T, and it coincides with vacuity in model checking. That is, some element of $\mathcal{S}$ does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{C}$. As in model checking, strong vacuity suggests that $\mathcal{C}$ and $\mathcal{S}$ should be re-examined: some behavior that the specifier expect cannot happen.
- **Weak vacuity** depends on T and it refers to the role that the elements of $\mathcal{S}$ play in the fact that T passes. Weak vacuity suggest that either there is strong vacuity or that more tests are needed.

We define and study vacuity for three settings of testing. The first setting considers the expected extension of LTL vacuity in the context of model checking. Thus, we check whether a given test suite could have passed even a stronger specification.

The second setting is that of *run-time verification*: we assume the specification is a monitor that is executed in parallel to the system, and an input sequence passes the test if the monitor does not get stuck on it [14]. Thus, the properties we are checking are safety properties, and the monitor gets stuck when a violation of the safety property has been detected. Our definition of vacuity in this setting refers to the transitions of the monitor: a test suite passes vacuously if it passes also with a monitor with fewer transitions.

The third setting is that of *software checking*: we assume the system is a procedure that terminates, and the specification is Boolean function, to which we feed both the input to the systems and its output. For example, the system may be a procedure that sorts a list of numbers, and the specification is a function that, given two lists, checks that a second list is the first list, sorted. We model both the system and the specification in a simple programming language. Our definition of vacuity refers to branches in the specification: a test suite passes vacuously if it does not branch-cover the specification, i.e., not all branches of the specification procedure are executed on the process of checking the test suite.

3.1 Vacuity in LTL specifications

We first consider specifications in LTL. Verification of a circuit $\mathcal{C}$ with respect to an LTL formula φ amounts to checking that for all infinite sequences $\xi \in (2^I)^\omega$,

the execution of $\mathcal{S}$ on the computation of ξ satisfies φ. Model checking of $\mathcal{C}$ with respect to φ is done by checking that the language of $\mathcal{C}$ is contained in that of an automaton accepting exactly all the models of φ. Technically, this is reduced to checking the emptiness of the product of $\mathcal{C}$ with an automaton accepting exactly all the models of $\neg\varphi$ [21]. One computational challenge is to cope with the exponential size of the automaton, which, in the worst case, is exponential in the length of the formula. The computational bottleneck, however, relies in the size of $\mathcal{C}$, which is typically much larger than φ. The alternative that testing suggests is to examine a vector $T \subseteq (2^I)^\omega$ of *lasso-shaped* input sequences. That is, each input sequence in T is of the form $u.v^\omega$, for $u, v \in (2^I)^*$. We say that T *passes* φ in $\mathcal{C}$ if for all $\xi \in T$, the computation w_ξ of $\mathcal{C}$ satisfies φ. We define the length of T, denoted $\|T\|$, as $\sum_{\xi \in T} |\xi|$.

As with vacuity in model checking, we say that an occurrence σ of a subformula of φ *does not affect* φ in $\mathcal{C}$ and T if $w_\xi \models \varphi[\sigma \leftarrow \bot_\sigma]$ for all input sequences ξ in T. Similarly, we say that a subformula ψ of a specification φ *does not affect* φ in $\mathcal{C}$ and T if w_ξ satisfies $\forall x.\varphi[\psi \leftarrow x]$ for all input sequences ξ in T. We then say that T passes φ vacuously in $\mathcal{C}$ in the single-occurrence approach if some occurrence of a subformula of φ does not affect φ in $\mathcal{C}$ and T, and say that T passes φ vacuously in $\mathcal{C}$ in the multiple-occurrence approach if some subformula of φ does not affect φ in $\mathcal{C}$ and T.

Theorem 1. *The problem of deciding whether T passes φ vacuously in $\mathcal{C}$ is in PTIME in the single-occurrence approach and is in PSPACE in the multiple-occurrence approach.*

Proof: In the single-occurrence approach, we go over all occurrences σ of subformulas of φ and model-check $\varphi[\sigma \leftarrow \bot_\sigma]$ in all computations w_ξ, for all $\xi \in T$. Since LTL model checking for a single path can be done in PTIME [18], the whole check is in PTIME.

In the multiple-occurrence approach, we go over all subformulas ψ of φ and model-check $\forall x.\varphi[\psi \leftarrow x]$ in all computations w_ξ, for all $\xi \in T$. Now each check requires PSPACE, and so does the whole check. □

We note that the lower bounds in Theorem 1 are open. In the single-occurrence approach, the challenge has to do with the open problem of the complexity of LTL model-checking with respect to a single path (known PTIME upper bound, only an NLOGSPACE lower bound [18]). In the multiple-occurrence approach, the challenge has to do with the open problem of the complexity of LTL vacuity detection (known PSPACE upper bound, only an NPTIME lower bound [1]).

3.2 Vacuity in run-time verification

We now turn to study specifications given by monitors. A *monitor* for a circuit $\mathcal{C} = \langle I, O, S, \theta, \delta, \rho \rangle$ is an automaton $\mathcal{S} = \langle 2^{I \cup O}, Q, Q_0, M \rangle$, where Q is a set of states $Q_0 \subseteq Q$ is a set of initial states, and $M : Q \times 2^{I \cup O} \to 2^Q$ is a transition function that maps a state and a letter to a set of possible successor states. We refer to the transitions of $\mathcal{S}$ as relations and write $M(s, \sigma, s')$ to indicate that $s' \in M(s, \sigma)$. Note that while $\mathcal{S}$ has no acceptance condition, it may be that

$M(s,\sigma) = \emptyset$, in which case $\mathcal{S}$ gets stuck.[4] A word $w \in (2^{I \cup O})^*$ is accepted by $\mathcal{S}$ if $\mathcal{S}$ has a run on w that never gets stuck. The language of $\mathcal{S}$, denoted $L(\mathcal{S})$, is the set of words that $\mathcal{S}$ accepts. All safety properties can be translated to monitors [16, 19].

The complexity considerations in the setting of model checking safety properties are similar to these in LTL model checking. Verification of $\mathcal{C}$ with respect to $\mathcal{S}$ amounts to checking that for all infinite sequences $\xi \in (2^I)^\omega$, the execution of $\mathcal{S}$ on the computation of ξ never gets stuck. Model checking of $\mathcal{C}$ with respect to $\mathcal{S}$ is done by checking that the language of $\mathcal{C}$ is contained in that of $\mathcal{S}$. Technically, this is reduced to checking the emptiness of the product of $\mathcal{C}$ with an automaton that complements $\mathcal{S}$. One computational challenges is to cope with the complementation of $\mathcal{S}$, which involves an exponential blow up. The computational bottleneck, however, relies in the size of $\mathcal{C}$, which is typically much larger than $\mathcal{S}$. The alternative that testing suggests is to examine a vector $T \subseteq (2^I)^*$ of finite prefixes of input sequences. We say that T *passes* $\mathcal{S}$ in $\mathcal{C}$ if for all $\xi \in T$, the computation w_ξ of $\mathcal{C}$ is accepted by $\mathcal{S}$.

We first describe the algorithm for checking whether T passes $\mathcal{S}$ in $\mathcal{C}$. The algorithm is a simple membership checking algorithm for monitors, applied to all input sequences in T. Given an input sequence $\xi = i_0, i_1, \ldots, i_{n-1} \in (2^I)^n$, we define the product of its computation in $\mathcal{C}$ with $\mathcal{S}$ as a sequence $\langle s_0, R_0\rangle, \langle s_1, R_1\rangle, \ldots, \langle s_{n-1}, R_{n-1}\rangle \in (S \times 2^Q)^n$ as follows. Intuitively, $s_0, \ldots, s_{n-1}$ is the execution of $\mathcal{C}$ on ξ, and R_j, for $0 \leq j \leq n-1$, is the set of states that $\mathcal{C}$ can be in after reading the prefix of the computation on ξ up to the letter i_j. Formally, $s_0 = \theta(i_0)$ and $R_0 = Q_0$, and for all $0 \leq j \leq n+1$, we have $s_{j+1} = \rho(s_j, i_{j+1})$ and $R_{j+1} = M(R_j, i_j \cup \rho(s_j))$. The pairs $\langle s_j, R_j\rangle$ can be constructed on-the-fly, and the computation w_ξ is accepted by $\mathcal{S}$ if $R_j \neq \emptyset$ for all $0 \leq j \leq n-1$.

Assume that T passes $\mathcal{S}$. The traditional approach to coverage checks that all the transitions of $\mathcal{C}$ have been taken during the execution of $\mathcal{C}$ on the input sequences in T. Here, we consider vacuity, which corresponds to coverage in the specification. Consider a transition $\langle s, \sigma, s'\rangle \in M$. Let $\mathcal{S}_{\langle s,i,s'\rangle}$ denote the monitor obtained from $\mathcal{S}$ by removing the transition $\langle s, i, s'\rangle$ from M. We say that $\langle s, i, s'\rangle$ *does not affect* $\mathcal{S}$ in $\mathcal{C}$ and T if T also passes $\mathcal{S}_{\langle s,i,s'\rangle}$ in $\mathcal{C}$. We then say that T *passes* $\mathcal{S}$ *vacuously* in $\mathcal{C}$ if some transition of $\mathcal{S}$ does not affect its pass.

Theorem 2. *The problem of checking whether T passes $\mathcal{S}$ vacuously in $\mathcal{C}$ is NLOGSPACE-complete.*

Proof: We start with the upper bound. Consider a circuit $\mathcal{C} = \langle I, O, S, \theta, \delta, \rho\rangle$, a monitor $\mathcal{S} = \langle 2^{I \cup O}, Q, Q_0, M\rangle$, and a test suite $T \subseteq (2^I)^*$. Consider an input sequence $\xi \in T$ and a transition $\langle s, i, s'\rangle \in M$. It is easy to see that the problem of deciding whether ξ is accepted by $\mathcal{S}_{\langle s,i,s'\rangle}$ is in NLOGSPACE. Indeed, an

[4] Readers familiar with Büchi automata would notice that a monitor is a *looping* Büchi automaton – a Büchi automaton in which all states are accepting. Here, we execute $\mathcal{C}$ on finite words.

algorithm that guesses an accepting run has to store the location in ξ, as well as the states in S and Q are currently visited. Since NLOGSPACE is closed under complementation, we conclude that the problem of deciding whether ξ is rejected by $\mathcal{S}_{\langle s,i,s' \rangle}$ is also in NLOGSPACE.

Now, given a transition $\langle s, i, s' \rangle$, the problem of deciding whether $\langle s, i, s' \rangle$ affects $\mathcal{S}$ in $\mathcal{C}$ and T can be solved in NLOGSPACE. Indeed, by definition, $\langle s, i, s' \rangle$ affects $\mathcal{S}$ in $\mathcal{C}$ and T if there is $\xi \in T$ such that ξ is rejected by $\mathcal{S}_{\langle s,i,s' \rangle}$, and the algorithm can guess such an input sequence $\xi \in T$ and check in NLOGSPACE that it is rejected by $\mathcal{S}_{\langle s,i,s' \rangle}$. Again we apply the closure of NLOGSPACE under complementation, and conclude that the problem of deciding whether a given transition does not affect $\mathcal{S}$ in $\mathcal{C}$ and T can be solved in NLOGSPACE. Thus, an algorithm for checking whether T passes $\mathcal{S}$ vacuously in $\mathcal{C}$ guesses a transition $\langle s, i, s' \rangle$ in M, and checks in NLOGSPACE that it does not affect $\mathcal{S}$ in $\mathcal{C}$ and T.

Hardness in NLOGSPACE can be easily proven by a reduction from reachability. □

We note that the straightforward algorithm for the problem requires polynomial time: it goes over all transitions $\langle s, i, s' \rangle$ and input sequences $\xi \in T$, and checks whether w_ξ is accepted by $\mathcal{S}_{\langle s,i,s' \rangle}$. The algorithm concludes that T passes $\mathcal{S}$ vacuously in $\mathcal{C}$ iff there is a transition $\langle s, i, s' \rangle$ for which the answer is positive for all input sequences.

Remark 3 When $\mathcal{S}$ is deterministic, things are much simpler, and it is easy to extend the algorithm for checking whether T passes $\mathcal{S}$ in $\mathcal{C}$ so that it also checks whether the pass is vacuous. Indeed, checking whether T passes is done by executing the input sequences in T. When we execute an input sequence $\xi \in T$, we mark the transitions in M that have been traversed during the monitoring of w_ξ. The test suite T then passes vacuously if not all transitions have been marked. □

The idea behind weak vacuity is that the structure of the specification is often different from the structure of the system. Hence, a test suite may cover all the transitions of the system, yet may not cover all the transitions of the specification. Adding to the test suite input sequences that cover the specification may reveal errors. We demonstrate this in Example 1 below.

Example 1. Consider the circuit $\mathcal{C}$ and its specification $\mathcal{S}$ appearing in Figure 1. We assume that $I = \{req\}$ and $O = \{ack\}$. The specification $\mathcal{S}$ corresponds to the LTL formula $\psi = G(req \rightarrow (ack \vee X(ack \vee Xack)))$. For convenience, we describe the input and output assignments by Boolean assertions. Note that a single assertion may correspond to several assignment, and thus a single edge in the figure may correspond to several transitions. For example, the self loop in state q_0 of $\mathcal{S}$, which is labeled $\neg req \vee ack$, stands for the three transitions $\langle q_0, \{\ \}, q_0 \rangle$, $\langle q_0, \{ack\}, q_0 \rangle$, and $\langle q_0, \{req, ack\}, q_0 \rangle$. It is not hard to see that $\mathcal{S}$ does not satisfy ψ (and $\mathcal{S}$). For example, an input sequence of the form $req, \neg req, \neg req, \neg req, \neg req, \ldots$ would loop forever in s_0 and s_2 and the first request is never acknowledged.

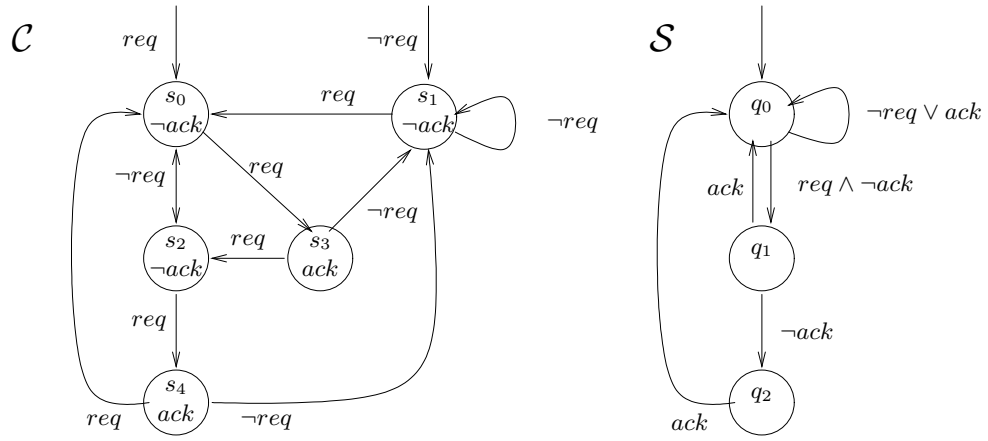

Fig. 1. A system and its monitor.

Below we describe the behavior of $\mathcal{C}$ and $\mathcal{S}$ on a test suite $T = \{\xi_1, \xi_2.\xi_3\}$, as follows.

- $\xi_1 = req, req, req, \neg req, req, \neg req$. The execution of $\mathcal{C}$ on ξ_1 is $\pi_1 = s_0, s_3, s_2, s_0, s_3, s_1$, and the run of $\mathcal{S}$ on the induced computation traverses the following sequence of transitions: $\langle q_0, \{req\}, q_1\rangle$, $\langle q_1, \{req, ack\}, q_0\rangle$, $\langle q_0, \{req\}, q_1\rangle$, $\langle q_1, \{\ \}, q_2\rangle$, $\langle q_2, \{req, ack\}, q_0\rangle$, $\langle q_0, \{\ \}, q_0\rangle$.
- $\xi_2 = \neg req, \neg req, req, \neg req, req, req$. The execution of $\mathcal{C}$ on ξ_2 is $\pi_2 = s_1, s_1, s_0, s_2, s_4, s_0$, and the corresponding run of $\mathcal{S}$ is $\langle q_0, \{\ \}, q_0\rangle$, $\langle q_0, \{\ \}, q_0\rangle$, $\langle q_0, \{req\}, q_1\rangle$, $\langle q_1, \{\ \}, q_2\rangle$, $\langle q_2, \{req, ack\}, q_0\rangle$, $\langle q_0, \{req\}, q_1\rangle$.
- $\xi_3 = req, \neg req, req, \neg req$. The execution of $\mathcal{C}$ on ξ_3 is $\pi_3 = s_0, s_2, s_4, s_1$, and the corresponding run of $\mathcal{S}$ is $\langle q_0, \{req\}, q_1\rangle$, $\langle q_1, \{\ \}, q_2\rangle$, $\langle q_2, \{req, ack\}, q_0\rangle$, $\langle q_0, \{\ \}, q_0\rangle$.

Note that all runs are accepting, thus T passes $\mathcal{S}$ in $\mathcal{C}$. Moreover, all the transitions of $\mathcal{C}$ have been taken during the execution of T. Thus, T covers $\mathcal{C}$, which indicates that T satisfies some quality criteria.

Consider the transition $\langle q_2, \{ack\}, q_0\rangle$ of $\mathcal{S}$. The transition does not affect $\mathcal{S}$ in $\mathcal{C}$ and T. Indeed, it was not traversed in the three runs. We claim that there is strong vacuity with respect to the transition $\langle q_2, \{ack\}, q_0\rangle$, and that detecting this strong vacuity is likely to reveal the bug. To see why, note that no computation of $\mathcal{C}$ generates the letter $\{ack\}$. Thus, acknowledgments are issued only with requests. Thus, if a request is not acknowledged immediately, another request is needed in order to issue an acknowledgement. This information should urge the designer to test $\mathcal{C}$ with respect to an input vector with a single request, which would reveal the bug. Note that the transitions $\langle q_0, \{req, ack\}, q_0\rangle$ and $\langle q_1, \{req\}, q_2\rangle$ also do not affect $\mathcal{S}$ in $\mathcal{C}$ and T, and in fact there is strong vacuity also with respect to them. □

3.3 Vacuity in software checking

We now turn to consider the third setting, of terminating software procedures. We assume the system is a procedure P that terminates, and the specification

is a Boolean function $\mathcal{S}$ to which we feed both the input and output of P. Given an input v to P, the Boolean value $\mathcal{S}(v, P(v))$ indicates whether P satisfies the specification modeled by $\mathcal{S}$. For the definition of vacuity, we adopt the standard *branch coverage* metric: all branches of $\mathcal{S}$ should be executed.

We consider procedures in a simple programming language, with branches induced by the *if-then-else*, *case*, and *while* statements. For a Boolean function $\mathcal{S}$ and a branch b, let $\mathcal{S}_b$ denote the function obtained from $\mathcal{S}$ by replacing the statement guarded by b by "return(*false*)". Given a procedure P, we say that a branch b of $\mathcal{S}$ *does not affect* $\mathcal{S}$ *in* P if $\mathcal{S}(v, P(v)) = \mathcal{S}_b(v, P(v))$ for all input vectors v. Then, given a procedure P, a specification $\mathcal{S}$ for it, and a test suite T, we say that a branch b *does not affect* $\mathcal{S}$ *in* P *and* T if $\mathcal{S}$ agrees with $\mathcal{S}_b$ on all the inputs in T. That is, for all $v \in T$, we have that $\mathcal{S}(v, P(v)) = \mathcal{S}_b(v, P(v))$. Finally, T passes $\mathcal{S}$ vacuously in P if some branch of $\mathcal{S}$ does not affect its pass.

Deciding whether a branch b does not affect $\mathcal{S}$ in P and T can be done by testing T with respect to $\mathcal{S}_b$. Since, however, $\mathcal{S}$ is deterministic, it is easy to combine the testing of T with a vacuity check: whenever a branch of $\mathcal{S}$ is taken, we mark it, and T passes $\mathcal{S}$ vacuously in P if we are done testing T and some branch is still not marked. Clearly, such a branch does not affect $\mathcal{S}$ in P and T.

Note that finding a bug in P amounts to covering a branch in $\mathcal{S}$ in which false is returned. Thus, an attempt to cover all branches is at least as challenging as finding a bug. We therefore seek to cover all "hopeful branches" – these that can be taken in an execution of $\mathcal{S}$ that returns *true*. In the rest of this section we describe two examples that demonstrate the effectiveness of vacuity checking in this setting.

Example 2. Consider the sorting program appearing in Figure 2. We use the operator :: to denote concatenation between elements or lists. For a list of the form $x :: y :: tail$, where x and y are numbers and $tail$ is a list of numbers, the procedure sorts the list by sorting x and y and then recursively sorting $y :: tail$ (in case $x \leq y$) or $x :: tail$ (in case $y < x$).

```
sort(list):
  case list is
    nil          → nil
    x :: nil     → x :: nil
    x :: y :: tail → if x ≤ y then return(x :: sort(y :: tail))
                          else  return(y :: sort(x :: tail))
```

Fig. 2. A buggy sorting procedure.

The Boolean function **sorted** in Figure 3 is a specification for the sorting procedure. It gets as input two lists of numbers and returns true iff the second list is a permutation of the first list, and the second list is sorted. For that, it calls two Boolean functions, each checking one condition.

In our example, it not hard to see that the procedure **sort** is correct with respect to input vectors v of length n for which, for all $2 \leq i \leq n$, the prefix of length i contains all the smallest $i-1$ numbers in v. For example, the procedure

```
sorted(list, list′):
  if permutation(list, list′) and sort_check(list′) then return(true)
                                                    else return(false)
```

Fig. 3. A specification for the sorting procedure.

correctly sorts $4 :: 1 :: 2 :: 3$ or $3 :: 1 :: 2 :: 5 :: 4$. Note that these two vectors branch cover **sort**, which indicates that a test suite consisting of them satisfies some quality criteria. Still the procedure sort is buggy. For example, it fails on $3 :: 1 :: 2$.

Assume now that the Boolean function **permutation**$(list, list')$ distinguishes between cases where the first and last elements of $list$ are switched in $list'$ and cases it does not. Then, at attempt to cover a branch that considers the first case would reveal the bug. Indeed, no input (of length at least three) in which the biggest number is first and the smallest number is last, would be correctly sorted by **sort**. □

Example 2 shows how vacuity checking is useful in cases the specification checks properties that are not referred to by the system. In Example 3 below we show how vacuity checking is useful also in cases the specification need not check additional properties, yet the structure of the system and the specification is different.

Example 3. The procedure **convert** in Figure 4 gets as input a string over 0 and 1. Its goal is to replace all substrings of the form 01 by 21. It is, however, buggy: by leaving a 00 head as is, **convert** ignores cases in which the second 0 is followed by 1. For example, the output of convert on 011001 is 211001, where the correct output is 211021.

```
convert(list):
  case list is
    nil          → return(nil)
    1 :: tail    → return(1 :: convert(tail))
    0 :: 1 :: tail → return(2 :: 1 :: convert(tail))
    0 :: 0 :: tail → return(0 :: 0 :: convert(tail))
```

Fig. 4. The procedure **convert** replaces a substring 01 by 21.

The specification **check**, appearing in Figure 5, gets as input the original string $list$ and the output $list'$ of **convert** and it outputs true iff $list'$ is indeed obtained from $list$ by replacing all substrings of the form 01 by 21. Note that while **check** looks complicated, one cannot simplify it.

The input 01101000 branch covers **convert**, and **convert** is correct with respect to it. Indeed, **convert**$(01101000) = 21121000$, and **check**$(01101000, 21121000)$ is true. On the other hand, $(01101000, 21121000)$ does not branch cover **check**. Indeed, the branch

$$b = (0 :: 0 :: 1 :: tail, 0 :: 2 :: 1 :: tail') \rightarrow \textbf{check}(tail, tail')$$

```
check(list, list′):
  case (list, list′) is
    (nil, nil)                          → return(true)
    (0 :: nil, 0 :: nil)                → return(true)
    (1 :: tail, 1 :: tail′)             → check(tail, tail′)
    (0 :: 0 :: tail, 0 :: 0 :: tail′)   → check(0 :: tail, 0 :: tail′)
    (0 :: 1 :: tail, 2 :: 1 :: tail′)   → check(tail, tail′)
    (0 :: 0 :: 1 :: tail, 0 :: 2 :: 1 :: tail′)  → check(tail, tail′)
    else                                → return(false)
```

Fig. 5. A specification for **convert**.

is not covered by $(01101000, 21121000)$.

We argue that an attempt to cover the branch b would reveal the bug in **convert**. To see this, note that the simplest way to cover b is by feeding **check** with the output of **convert** on 001. However, **convert**$(001) = 001$, and **check**$(001, 001)$ returns false and reveals the bug.

References

1. R. Armon, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M.Y. Vardi. Enhanced vacuity detection for linear temporal logic. In *Proc 15th Int. Conf. on Computer Aided Verification*. Springer, 2003.
2. D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. 31st Design Automation Conf.*, pages 596–602. IEEE Computer Society, 1994.
3. 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.
4. L. Bening and H. Foster. *Principles of verifiable RTL design – a functional coding style supporting verification processes*. Kluwer Academic Publishers, 2000.
5. T. Budd. *Mutation Analysis*. PhD thesis, Yale University, 1979.
6. D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Conf. on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2005.
7. M. Chechik and A. Gurfinkel. Extending extended vacuity. In *Proc. 5th Int. Conf. on Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 2004.
8. H. Chockler and O. Kupferman. Coverage of implementations by simulating specifications. In *Proceedings of 2nd IFIP Int. Conf. on Theoretical Computer Science*, volume 223 of *IFIP Conf. Proceedings*, pages 409–421. Kluwer Academic Publishers, 2002.
9. H. Chockler, O. Kupferman, R.P. Kurshan, and M.Y. Vardi. A practical approach to coverage in model checking. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 66–78. Springer, 2001.
10. H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for temporal logic model checking. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 528 – 542. Springer, 2001.

11. H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for formal verification. In *Proc. 12th Conf. on Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2003.
12. E.M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
13. D.L. Dill. What's between simulation and formal verification? In *Proc. 35st Design Automation Conf.*, pages 328–329. IEEE Computer Society, 1998.
14. K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):18–173, 2004.
15. Y. Hoskote, T. Kam, P.-H Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. 36st Design Automation Conf.*, pages 300–305, 1999.
16. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
17. O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *Software Tools for Technology Transfer*, 4(2):224–233, 2003.
18. N. Markey and Ph. Schnoebelen. Model checking a path. In *14th Int. Conf. on Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2003.
19. A.P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994.
20. S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.
21. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
22. Verisity. Surecove's code coverage technology. http://www.verisity.com/products/surecov.html, 2003.
23. H. Zhu, P.V. Hall, and J.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.