Sanity Checks in Formal Verification[*]

Orna Kupferman [**]

Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
Email:orna@cs.huji.ac.il, URL: http://www.cs.huji.ac.il/~orna

Abstract. One of the advantages of temporal-logic 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. On the other hand, when the answer to the correctness query is positive, most model-checking tools provide no additional information. In the last few years there has been growing awareness to the importance of suspecting the system or the specification of containing an error also in the case model checking succeeds. The main justification of such suspects are possible errors in the modeling of the system or of the specification. The goal of sanity checks is to detect such errors by further automatic reasoning. Two leading sanity checks are *vacuity* and *coverage*. In vacuity, the goal is to detect cases where the system satisfies the specification in some unintended trivial way. In coverage, the goal is to increase the exhaustiveness of the specification by detecting components of the system that do not play a role in verification process. For both checks, the challenge is to define vacuity and coverage formally, develop algorithms for detecting vacuous satisfaction and low coverage, and suggest methods for returning to the user helpful information. We survey existing work on vacuity and coverage and argue that, in many aspects, the two checks are essentially the same: both are based on repeating the verification process on some mutant input. In vacuity, mutations are in the specifications, whereas in coverage, mutations are in the system. This observation enables us to adopt work done in the context of vacuity to coverage, and vise versa.

1 Introduction

In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior [CGL93]). 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 are very important and they can be essential in detecting subtle errors in complex designs [CGMZ95]. On the other hand, when the answer to the correctness query is positive, most model-checking tools provide no additional information. Since a positive answer means that the system is correct with respect to the

[*] The paper is based on joint work with Hana Chockler, Moshe Y. Vardi, and Robert Kurshan, appearing in [CKV01,CKKV01,CKV03,Cho03].

[**] Supported in part by BSF grant 9800096, and by a grant from Minerva.

specification, this at first seems like a reasonable policy. 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 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 [BB94]. 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 [BBER01], 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" [BBER01]. The definition of vacuity according to [BBER01] 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. 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 [KV03], and the problem was further studied in [AFF$^+$03,CG04a,BFG$^+$05]. It is shown in these papers that for temporal logics such as LTL and $\mathrm{CTL}^\star$, where an occurrence of the subformula ψ can be replaced by a universally quantified proposition, vacuity detection can be reduced to model checking specifications in the logic obtained by adding universally quantified atomic propositions. This leaves vacuity detection for LTL in PSPACE [AFF$^+$03], but makes vacuity detection for CTL and $\mathrm{CTL}^\star$ EXPTIME and 2EXPTIME complete — as hard as their satisfiability [CG04a]. Moreover, adding to the specification formalism a regular layer, such as the ability to use regular expressions in the formula as in Sugar [BBE$^+$01] and ForSpec [AFF$^+$02], also adds a need to replace some subformulas ψ by a university quantified interval, which makes vacuity detection more complex 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 design 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.

The challenge of making the verification process as exhaustive as possible is even more crucial in *simulation-based* verification. Each input vector for the system induces a different execution of it, and a system is correct if it behaves as required for all possible input vectors. Checking all the executions of a system is an infeasible task. Simulation-based verification is traditionally used in order to check the system with respect to some input vectors [BF00]. The vectors are chosen so that the verification would be as exhaustive as possible, and it is crucial to measure the exhaustiveness of the input sequences that are checked. Indeed, there has been an extensive research in the simulation-based verification community on *coverage metrics*, which provide such a measure [TK01]. 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 [Ver03]. For a survey on the variety of metrics that are used in simulation-based verification, see [ZHM97,Dil98,Pel01,TK01]

Measuring the exhaustiveness of a specification in formal verification ("are more properties need to be checked?") has a similar flavor as measuring the exhaustiveness of the input sequences in simulation-based verification ("are more sequences need to be checked?"). Nevertheless, while for simulation-based verification it is clear that coverage corresponds to activation during the execution on the input sequence, it is less clear what coverage should correspond to in formal verification, as in model checking all reachable parts of the system are visited. Early work on coverage metrics in formal verification [HKHZ99,KGG99] suggested two directions. Both directions reason about a state-transition graph that models the system. The metric in [HKHZ99], later followed by [CKV01,CKKV01,CK02], is based on *mutations* applied to the graph. Essentially, a state s in the graph is covered by the specification if modifying the value of a variable in the state renders the specification untrue. The metric in [KGG99] is based on a comparison between the graph and a reduced tableau for the specification.

In [CKV03], we adapted the work done on coverage in simulation-based verification to the formal-verification setting in order to obtain new coverage metrics. Interestingly, the adoption of metrics from simulation-based verification has brought vacuity to the front of the stage again, and this time, in the context of coverage. To see why, consider for example code-based coverage, where we check, for example, whether both branches of an *if* statement have been executed during the simulation. A straightforward adoption would check the satisfaction of the specification in a mutant system, one for each branch, in which the branch is disabled. Such a mutant system, however, has less behaviors than the original system, and would clearly satisfy all universal specifications (i.e., specifications that apply to *all* behaviors, as in linear temporal logic) that are satisfied by the original system. In general, the problem we are facing is the need to assess the role a behavior has played in the satisfaction of a universal specification – one that is clearly satisfied in the system obtained by removing this behavior. The way we suggested to do so is to check whether the specification is vacuously satisfied in a mutant system in which this behavior is disabled: a vacuous satisfaction of the specification in such a system (we assume that the specification is not vacuously satisfied in the original system) indicates that the specification does refer to this behavior; on the other hand, a non-vacuous satisfaction of the specification in the mutant system indicates that the specification does not refer to the missing behavior. Accordingly, coverage metrics adopted from the simulation-based word check both the satisfaction and the vacuous satisfaction of the specification in mutant systems.

The definition of vacuity coverage in [CKV03] has related vacuity and coverage. In this paper we strengthen the link between the two sanity checks further and argue that, from the algorithmic point of view, the problems are essentially identical. In both problems, we check whether all the components of the input to the model-checking problem have played a role in the model-checking process. In the case of vacuity, the components we check are subformulas of the specification. In the case of coverage, the components are elements of the system. This suggests that the solutions to the vacuity and coverage problems may be based on the same algorithm. We show that, indeed, ideas developed for coverage can be adopted for vacuity, and vice versa.

2 Vacuity and Coverage

In this section we describe the basic definitions of vacuity and coverage. We consider specifications in either linear or branching temporal logics. For a formula φ, a subformula ψ of φ, and a formula ξ, we use $\varphi[\psi \leftarrow \xi]$ to denote the formula obtained from φ by replacing all the occurrences of ψ in φ by ξ.

We define the semantics of temporal-logic formulas with respect to a *Kripke structure* $K = \langle AP, W, R, w_{in}, L\rangle$, where AP is a set of atomic propositions, W is a set of states, $R \subseteq W \times W$ is a total transition relation, $w_{in} \in W$ is an initial state, and $L : W \to 2^{AP}$ maps each state to the set of atomic propositions that hold in this state. A Kripke structure K can be unwound into an infinite computation tree in a straightforward way. Formally, the tree that is obtained by unwinding K is denoted by $\mathcal{K}$ and is the 2^{AP}-labeled W-tree $\langle T^K, V^K\rangle$, in which a node $x \cdot w$, for $x \in W^*$ and $w \in W$, is associated with state w. Formally, $\varepsilon \in T^K$ is associated with w_{in} and $V^K(\varepsilon) = L(w_{in})$. Now, for all w with $R(w_{in}, w)$, we have that $w \in T^K$, and for all $x \cdot w \in T^K$ and $v \in W$ with $R(w, v)$, we have $x \cdot w \cdot v \in T^K$ and $V^K(x \cdot w) = L(w)$. That is, V^K maps a node that was reached by taking the direction w to $L(w)$.

The definition of vacuity involves formulas with an atomic proposition that is universally quantified. Consider an atomic proposition x. A Kripke structure K satisfies a temporal logic formula $\forall x \varphi(x)$ iff φ is satisfied in all computation trees $\langle T, V\rangle$ that differ from $\langle T^K, V^K\rangle$ only in the label of the atomic proposition x. Note that different occurrences of the same state in K may have different x labels.

Let us start with the basic definition of vacuous satisfaction. Intuitively, a Kripke structure K satisfies a formula φ vacuously if K satisfies φ yet it does so in a non-interesting way, which is likely to point on some trouble with either K or φ. For example, a system in which requests are never sent satisfies $AG(req \to AF grant)$ vacuously. In order to formalize this intuition, it is suggested in [BBER01] to formalize first the notion of a subformula of φ affecting its truth value in K. We use the following definition for the latter:

Definition 1. [AFF$^+$03] *A subformula ψ of φ* does not affect the truth value of φ in K *(ψ does not affect φ in K, for short) if K satisfies $\forall x \varphi[\psi \leftarrow x]$ iff K satisfies φ.*

The definition in [AFF$^+$03] is semantic. Earlier definitions, and in particular the one in [BBER01], were syntactic, in the sense they consider a replacement of ψ by other subformulas. Thus, according to [BBER01], ψ does not affect φ in K if K satisfies $\varphi[\psi \leftarrow \xi]$ for all formulas ξ. A good reason to switch to the semantic-based definition is the fact that [BBER01]'s definition is not effective, as it requires evaluation of $\varphi[\psi \leftarrow \xi]$ for all formulas ξ. To deal with this difficulty, [BBER01] considers only a small class, called w-ACTL, of branching temporal logic formulas. Once we have defined when a subformula of φ affects its truth value in K, the definition of vacuity is as expected[1]:

Definition 2. *A system K satisfies a formula φ* vacuously *iff $K \models \varphi$ and there is some subformula ψ of φ such that ψ does not affect φ in K.*

In [KV03], we showed that when all the occurrences of a subformula ψ in φ are of a *pure polarity* (that is, they are either all under an even number of negations (positive polarity), or all are under an odd number of negations (negative polarity)), the syntactic and semantic definitions coincide, and checking whether ψ affects φ in K is easy. Formally, for a formula φ and a subformula ψ of φ, let $\varphi[\psi \leftarrow \perp]$ denote the formula obtained from φ by replacing ψ by $\mathbf{false}$, in case ψ is positive in φ, and replacing ψ by $\mathtt{true}$, in case ψ is negative in φ. Now, by [KV03], ψ does not affect φ iff K satisfies the formula obtained from φ by the single extreme modification of ψ. Formally, we have the following.

Theorem 1. [KV03] *For every formula φ, a subformula ψ of a pure polarity of φ, and a system K that satisfies φ, we have that ψ does not affect φ in K iff K satisfies $\varphi[\psi \leftarrow \perp]$.*

By Definition 2, vacuity detection can be reduced to checking whether K satisfied $\forall x \varphi[\psi \leftarrow x]$ for all subformulas ψ of φ. Also, by Theorem 1, when ψ is of a pure polarity [2], the latter can be done by checking whether K satisfies $\varphi[\psi \leftarrow \perp]$. In particular, when φ is *polar* (that is, all its subformulas are of a pure polarity), vacuity detection can be reduced to a sequence of model-checking executions, each for a single subformula (this is a naive algorithm for this task, and we later mention some heuristics). When, however, some subformula ψ is of a mixed polarity, the check is harder and requires model checking of formulas with quantified atomic propositions. For the case of CTL and CTL*, the problem of vacuity detection is then as hard as the satisfiability problem, namely it is EXPTIME and 2EXPTIME complete, respectively [CG04a]. For LTL, it can still be reduced to LTL model checking and stay PSPACE-complete, but is more complicated than simple model checking [AFF$^+$03].

Remark 2 The semantic approach turned out to be appropriate also when the specification formalism has a regular layer [BFG$^+$05]. There, the subformula ψ may be a regular

[1] In [CG04b], the authors study an alternative definition of vacuity in which the *mutual vacuity* of some subformulas is taken into a consideration.

[2] Note that one can talk about a subformula ψ affecting φ in K or about an *occurrence* of ψ affecting φ in K. Since a single occurrence is of a pure polarity, Theorem 1 always applies in this setting.

event, and the universal quantification that is needed is over intervals. Consider for example the formula $\varphi = G\,((req \cdot (\neg ack)^* \cdot ack)\ triggers\ X grant)$, which says that a grant is given exactly one cycle after the cycle in which a request is acknowledged. Note that if ack does not affect the satisfaction of φ in K, we can learn that acknowledgments are actually ignored: grants are given, and stay on forever, immediately after a request. Such a behavior is not referred to in the specification, but is detected by regular vacuity. Thus, while LTL vacuity involved only *monadic* quantification (over the set of points in which x may hold), regular vacuity also involves *dyadic* quantification (over intervals – sets of pairs of points, in which *int* may hold). This transition, from monadic to dyadic quantification, is technically very challenging, yet, as was shown in [BFG$^+$05], the automata-theoretic approach to LTL [VW94] can be extended to handle regular vacuity, but the problem is much harder than LTL vacuity (it is in EXPSPACE and is EXPTIME-hard).

As with usual vacuity, when a subformula ψ has a pure polarity, checking whether it affects the truth value of φ can be reduced to checking whether K satisfies $\varphi[\psi \leftarrow \perp]$, with $\perp$ being $\mathbf{true}^*$ in case ψ is of a negative polarity and is $\mathbf{false}$ in case ψ is of a positive polarity. Thus, in the context of regular vacuity, pure polarity is even more crucial. $\square$

We now turn to the basic definition of coverage in model checking. The idea, due to [HKHZ99], is to define coverage by examining the effect of modifications in the system on the satisfaction of the specification. Given a system modeled by a Kripke structure K, a formula φ that is satisfied in K, and a signal (atomic proposition) q, let us denote by $\tilde{K}_{w,q}$ the Kripke structure obtained from K by flipping the value of q in w. Thus, $\tilde{K}_{w,q} = \langle AP, W, R, W_{in}, \tilde{L}_{w,q}\rangle$, where $\tilde{L}_{w,q}(v) = L(v)$ for all $v \neq w$, and $\tilde{L}_{w,q}(w) = L(w) \setminus \{q\}$ if $q \in L(w)$ and $\tilde{L}_{w,q}(w) = L(w) \cup \{q\}$ if $q \notin L(w)$.

Definition 3. [HKHZ99] *A state w of a Kripke structure K is q-*covered by *φ, for a formula φ and an atomic proposition q, if K satisfies φ but $\tilde{K}_{w,q}$ does not satisfy φ.*

Thus, w is q-covered by φ if the Kripke structure obtained from K by flipping the value of q in w no longer satisfies φ. Indeed, this indicates that the value of q in w is crucial for the satisfaction of φ in K. Definition 3 is very basic not only since it considers only mutations of a very limited nature, but also, as pointed out in [CKV01], it ignores the fact that often, replacing the value of an atomic proposition also causes a change in the transitions of K, which are typically defined by means of the values of the atomic propositions in the target and source of each transition. As shown, however, in [CKV01], the latter weakness is technical and it is possible to extend coverage algorithms that consider mutations that do not change the transitions to mutations that do change them.

3 Adopting Ideas from Vacuity to Coverage

In this section we show how ideas that have been suggested in the context of coverage are actually an adoption of ideas in vacuity. We also point to ideas in vacuity that have not yet been adopted in coverage.

3.1 Single vs. multiple occurrences

We start with the definition of coverage. Recall that the basic definition of coverage considered a very simple mutation: flip the value of one atomic proposition in one state. Recall that the Kripke structure models a system, and that the execution of the system corresponds to unwinding the Kripke structure into an infinite computation tree. A state w of K may correspond to several nodes in the computation tree. The basic definition of coverage flips the value of q in w in all these occurrences. In a similar way, in the definition of vacuity, we have distinguished between a single occurrence of a subformula ψ of φ and all its occurrence. In the first case, we have replaced only this occurrence by a universally quantified proposition (in fact, in this case it is sufficient to replace the single occurrence by $\perp$), and in the second, we have replaced all the occurrences. Each approach may return a different answer to the vacuity query.

This suggest that the definition of coverage should also be refined to reflect the fact that the flipping of q in w can be performed in different ways. Such a refinement was suggested in [CKKV01], which made a distinction between "flipping always", "flipping once", and "flipping sometimes", which are formalized in the definitions of *structure coverage*, *node coverage*, and *tree coverage* below. We first need some notations.

For a domain Y, a function $V : Y \to 2^{AP}$, an observable signal $q \in AP$, and a set $X \subseteq Y$, the *dual function* $\tilde{V}_{X,q} : Y \to 2^{AP}$ is such that $\tilde{V}_{X,q}(x) = V(x)$ for all $x \notin X$, $\tilde{V}_{X,q}(x) = V(x) \setminus \{q\}$ if $x \in X$ and $q \in V(x)$, and $\tilde{V}_{X,q}(x) = V(x) \cup \{q\}$ if $x \in X$ and $q \notin V(x)$. When $X = \{x\}$ is a singleton, we write $\tilde{V}_{x,q}$. Recall that $\tilde{K}_{w,q} = \langle AP, W, R, W_{in}, \tilde{L}_{w,q} \rangle$. For $X \subseteq T^K$ we denote by $\tilde{\mathcal{K}}_{X,q}$ the tree that is obtained by flipping the value of q in all the nodes in X. Thus, $\tilde{\mathcal{K}}_{X,q} = \langle T^K, \tilde{V}^K_{X,q} \rangle$. When $X = \{x\}$ is a singleton, we write $\tilde{\mathcal{K}}_{x,q}$.

Definition 4. *Consider a Kripke structure K, a formula φ satisfied in K, and an observable signal $q \in AP$.*

- *A state w of K is* structure q-covered *by φ iff the structure $\tilde{K}_{w,q}$ does not satisfy φ.*
- *A state w of K is* node q-covered *by φ iff there is a w-node x in T^K such that $\tilde{\mathcal{K}}_{x,q}$ does not satisfy φ.*
- *A state w of K is* tree q-covered *by φ iff there is a set X of w-nodes in T^K such that $\tilde{\mathcal{K}}_{X,q}$ does not satisfy φ.*

Note that, structure coverage coincides with Definition 3. Also note that a state is structure q-covered iff $\tilde{\mathcal{K}}_{X,q}$ does not satisfy φ for the set X of all w-nodes in $\mathcal{K}$. In other words, a state w is structure q-covered if flipping the value of q in all the instances of w in $\mathcal{K}$ falsifies φ, it is node q-covered if a single flip of the value of q falsifies φ, and it is tree q-covered if some flips of the value of q falsifies φ.

3.2 A semantic approach to coverage

Recall that earlier definitions of vacuity were syntactic and considered replacements of a subformula ψ by another formula [BBER01]. Later, in [AFF$^+$03], researchers have moved to the semantic approach, where ψ is replaced by a universally quantified atomic proposition. We would like to use the idea of a universally quantified atomic proposition also in the context of coverage. Thus, we seek a definition according to which w is not covered by φ if $\forall x K[w \leftarrow x] \models \varphi$. In general, it is not clear what x is and what does $K[w \leftarrow x]$ stands for. There are, however, settings in which an appropriate definition for x exists. In particular, in symbolic methods, the state space is encoded by propositional variables [BCM$^+$92,BCC$^+$99], and universal quantification is naturally defined. The induced definition of coverage captures exactly our intuition of w not playing a role in the verification process. Indeed, if, for example, we have reduced bounded model checking to the non-satisfiability of a propositional formula θ and a vector $\boldsymbol{x}$ of variables encodes the value of the system's variables in state w in time t, then satisfiability of $\forall \boldsymbol{x} \theta$ indicates that the values of the variables in $\boldsymbol{x}$ did play a role in the model-checking procedure. Note that, as with usual coverage, there is a need to distinguish between structure, tree, and node coverage.

3.3 Returning an interesting witness to the user

A witness for the satisfaction of a specification in a system is a sub-system, usually a computation, that satisfies the specification. A witness is interesting if it satisfies the specification non-vacuously [BBER01,KV03]. For example, a computation in which both req and $\neg grant$ hold is an interesting witness for the satisfaction of $AG(req \rightarrow AF grant)$. An interesting witness gives the user a confirmation that his specification models correctly the desired behavior, and enables the user to study some nontrivial executions of the system.

An interesting witness in the context of coverage is a subformula that causes a component to be covered. It is easy to extend existing coverage algorithms to return, for each component of the system, the parts of the specification with respect to which it is covered. More informative, however, and closer to the way interesting witnesses are used in vacuity, is to return to the user information on how the component is covered. Thus, for every component c of the system, the user should be able to get witnesses to the coverage of c by means of an erroneous computations in which c is mutated. From an algorithmic point of view, this involves solving exactly the same problem as the problem of generating interesting witnesses in vacuity, namely the problem of generating counter examples [CGMZ95,KV03]. In the context of coverage, however, we return to the user a family of counterexamples – one for each sub-specification that is no longer satisfied in the system with c mutated.

4 Adopting Ideas from Coverage to Vacuity

The adoption we suggested in Section 4 considers the challenges of defining vacuity and coverage and of returning helpful information to the user. In the context of coverage, much effort has been put in order to develop efficient algorithms for computing coverage. As we shall detail below, this has to do with the fact that a naive algorithm for coverage increases the complexity of model checking by a factor that depends on the size of the system, whereas one for vacuity detection increases the complexity only by a factor that depends on the specification. While the specification is typically much smaller than the system, it is still desirable to get rid of this factor.

A naive algorithm for the detection of components of the system that are not covered by the specification proceeds by model checking mutations of the system. For example, in order to find the set of states not q-covered by φ in a Kripke structure K with n states, the naive algorithm executes the model-checking procedure n times, where in each execution $\tilde{K}_{w,q}$ is checked for a different state w. Likewise, a naive algorithm for vacuity detection proceeds by checking mutations of the specification, each obtained by replacing a single subformula by a universally quantified proposition (in case the subformula is of a mixed polarity) or by $\mathbf{true}$ or $\mathbf{false}$ (in case it is of a pure polarity).

In [CKV01,CKKV01], we presented two alternatives to the naive algorithm for coverage. The first is symbolic, and the second makes use of overlaps among different mutations of the same Kripke structures. In this section we briefly describe the two algorithms and show how exactly the same ideas can be used in order to detect vacuous satisfaction of polar formulas.

4.1 A symbolic approach

We start with the symbolic coverage detection algorithm for LTL specifications. The algorithm is described in [CKKV01]. For simplicity, we start with node coverage, and then explain how tree and structure coverage can be checked with the same idea. The algorithm extends the LTL automata-based model-checking algorithm. There, we translate an LTL specification φ to a nondeterministic Büchi automaton $\mathcal{A}_{\neg\varphi}$ that accepts all words that do not satisfy φ [VW94]. Model checking of K with respect to φ can then be reduced to checking the emptiness of the product $K \times \mathcal{A}_{\neg\varphi}$. Let $K = \langle AP, W, R, w_{in}, L\rangle$ be a Kripke structure that satisfies φ, and let $\mathcal{A}_{\neg\varphi} = \langle 2^{AP}, S, \delta, S_0, \alpha\rangle$ be the nondeterministic Büchi automaton for $\neg\varphi$. The product of K with $\mathcal{A}_{\neg\varphi}$ is the fair Kripke structure $K \times \mathcal{A}_{\neg\varphi} = \langle AP, W \times S, M, \{w_{in}\} \times S_0, L', W \times \alpha\rangle$, where $M(\langle w, s\rangle, \langle w', s'\rangle)$ iff $R(w, w')$ and $s' \in \delta(s, L(w))$, and $L'(\langle w, s\rangle) = L(w)$. Note that an infinite path π in $K \times \mathcal{A}_{\neg\varphi}$ is fair iff the projection of π on S satisfies the acceptance condition of $\mathcal{A}_{\neg\varphi}$. Since K satisfies φ, we know that no initialized path of K is accepted by $\mathcal{A}_{\neg\varphi}$. Hence, $L(K \times \mathcal{A}_{\neg\varphi})$ is empty.

Let $P \subseteq W \times S$ be the set of pairs $\langle w, s\rangle$ such that $\mathcal{A}_{\neg\varphi}$ can reach the state s as it reads the state w. That is, there exists a sequence $\langle w_0, s_0\rangle, \ldots, \langle w_k, s_k\rangle$ such that $w_0 = w_{in}$, $s_0 \in S_0$, $w_k = w$, $s_k = s$, and for all $i \geq 0$ we have $R(w_i, w_{i+1})$ and $s_{i+1} \in \delta(s_i, L(w_i))$. Note that $\langle w, s\rangle \in P$ iff $\langle w, s\rangle$ is reachable in $K \times \mathcal{A}_{\neg\varphi}$. For an observable signal $q \in AP$ and $w \in W$, we define the set $P_{w,q} \subseteq W \times S$ as the set of pairs $\langle w', s'\rangle$ such that w' is a successor of w and $\mathcal{A}_{\neg\varphi}$ can reach the state s' as it reads the state w' in a run in which the last occurrence of w has q flipped. Formally, if we denote by $\tilde{L}_q : W \to 2^{AP}$ the labeling function with q flipped (that is, $\tilde{L}_q(w) = L(w) \cup \{q\}$ if $q \notin L(w)$, and $\tilde{L}_q(w) = L(w) \setminus \{q\}$ if $q \in L(w)$), then

$$P_{w,q} = \{\langle w', s'\rangle : \text{ there is } s \in S \text{ such that } \langle w, s\rangle \in P, R(w, w'), \text{ and } s' \in \delta(s, \tilde{L}_q(w))\}.$$

Recall that a state w is node q-covered in K iff there exists a a w-node x in T^K such that $\tilde{\mathcal{K}}_{x,q}$ does not satisfy φ. We can characterize node q-covered states also as follows

Theorem 3. *Consider a Kripke structure K, an LTL formula φ, and an observable signal q. A state w is node q-covered in K by φ iff there is a successor w' of w and a state s' such that $\langle w', s' \rangle \in P_{w,q}$ and there is a fair $\langle w', s' \rangle$-path in $K \times \mathcal{A}_{\neg\varphi}$.*

Theorem 3 reduces the problem of checking whether a state w is node q-covered to computing the relation $P_{w,q}$ and checking for the existence of a fair path from a state in the product $K \times \mathcal{A}_{\neg\varphi}$. Model-checking tools compute the relation P and compute the set of states from which we have fair paths. Therefore, Theorem 3 suggests an easy implementation for the problem of computing the set of node-covered states. We describe a possible implementation in the tool COSPAN, which is the engine of FormalCheck [HHK96,Kur98]. We also show that the implementation can be easily modified to handle structure and tree coverage.

In COSPAN, the system is modeled by a set of modules, and the desired behavior is specified by an additional module $\mathcal{A}$. The language $\mathcal{L}(\mathcal{A})$ is exactly the set of wrong behaviors, thus the module $\mathcal{A}$ stands for the automaton $\mathcal{A}_{\neg\varphi}$ in cases the specification is given an LTL formula φ. In order to compute the set of node q-covered states, the system has to nondeterministically choose a step in the synchronous composition of the modules, in which the value of q is flipped in all modules that refer to q. Note that this is the same as to choose a step in which the module $\mathcal{A}$ behaves as if it reads the dual value of q. This can be done by introducing two new Boolean variables *flip* and *flag*, local to $\mathcal{A}$. The variable *flip* is nondeterministically assigned $\mathbf{true}$ or $\mathbf{false}$ in each step. The variable *flag* is initialized to $\mathbf{true}$ and is set to $\mathbf{false}$ one step after *flip* becomes $\mathbf{true}$. Instead of reading q, the module $\mathcal{A}$ reads $q \oplus (\mathit{flip} \wedge \mathit{flag})$. Thus, when both *flip* and *flag* hold, which happens exactly once, the value of q is flipped ($\oplus$ stands for exclusive or). So, the synchronous composition of the modules is not empty iff the state that was visited when *flip* becomes $\mathbf{true}$ for the first time is node q-covered.

With a small change in the implementation we can also check tree coverage. Since in tree coverage we can flip the value of q several times, the variable *flag* is no longer needed. Instead, we need $\log |W|$ variables $x_1, \ldots, x_{\log |W|}$ for encoding the state w that is now being checked for tree q-coverage. The state w is not known in advance and the variables $x_1, \ldots, x_{\log |W|}$ are initialized to some special value $\perp$. The variable *flip* is nondeterministically assigned $\mathbf{true}$ or $\mathbf{false}$ in each step. When *flip* is changed to $\mathbf{true}$ for the first time, the variables $x_1, \ldots, x_{\log |W|}$ are set to encode the current state w. Instead of reading q, the module $\mathcal{A}$ reads $q \oplus (\mathit{flip} \wedge \mathit{at\_w})$, where $\mathit{at\_w}$ holds iff the encoding of the current state coincides with $x_1, \ldots, x_{\log |W|}$. Thus, when both *flip* and $\mathit{at\_w}$ hold, which may happen several times, yet only when the current state is w, the value of q is flipped. So, the synchronous composition of the modules is not empty iff the state that was visited when *flip* becomes $\mathbf{true}$ for the first time is tree q-covered. Finally, by nondeterministically choosing the values of $x_1, \ldots, x_{\log |W|}$ at the first step of the run and fixing *flip* to $\mathbf{true}$, we can also check structure coverage.

Note that our algorithm is independent of the fairness condition being Büchi, and it can handle any fairness condition for which the model-checking procedure supports the check for fair paths. Also, it is easy to see that the same algorithm can handle systems with multiple initial states. Finally, it is also easy to adjust the algorithm to definitions of coverage in which several mutations are checked mutually.

Remark 4 The above algorithm handles specifications in LTL. A different symbolic algorithm for coverage computation is described for CTL in [CKV01]. The algorithm addresses the fact that even if model checking of each of the mutant Kripke structures is checked symbolically, there may be many mutations to check. If we have, for example, a mutant structure for each state, then there are $|W|$ mutant structures to check, and we would like to refer also to these structures symbolically. Consider a Kripke structure $K = \langle AP, W, R, w_0, L\rangle$ and an atomic proposition $q \in AP$. For a CTL formula φ, we define

$$P(\varphi) = \{\langle w, v\rangle : \tilde{K}_{v,q}, w \models \varphi\}.$$

Thus, $P(\varphi) \subseteq W \times W$ contains exactly all pairs $\langle w, v\rangle$ such that w satisfies φ in the structure where we dualize the value of q in v. The q-covered set in K for φ can be derived easily from $P(\varphi)$ as it is the set $\{w : \langle w_0, w\rangle \notin P(\varphi)\}$.

The symbolic algorithm in [CKV01] computes the OBDDs $P(\psi)$ for all subformulas ψ. The algorithm works bottom-up, and is based on the symbolic CTL model-checking algorithm. The symbolic algorithm for CTL model-checking uses a linear number of OBDD variables. The algorithm in [CKV01] above doubles the number of OBDD variables, as it works with sets of pairs of states instead of sets of states. By the nature of the algorithm, it performs model-checking for all $\tilde{K}_{w,q}$ globally, and thus the OBDDs it computes contain information about the satisfaction of the specification in all the states of all the dual Kripke structures, and not only in their initial states. $\square$

We now turn to describe how the same idea can be used in order to symbolically detect vacuity of polar formulas. The algorithm we describe can be viewed as a special case of the symbolic algorithm in [CKV03] for the detection of vacuity coverage. Recall that checking whether a system satisfies a specification vacuously involves model checking of a mutant specification. We can use the idea in [CKKV01] in order to check symbolically for vacuous satisfaction by adding a new variable x that encodes the subformula ψ that is being replaced with $\perp$. The subformula ψ belongs to the set $cl(\varphi)$ of subformulas of φ. The variable x is an integer in the range $0, \ldots, |cl(\varphi)|$, thus it can be encoded with $O(\log|\varphi|)$ Boolean variables. The value 0 of x stands for "no replacement", thus it checks the satisfaction of φ in the system. The value of (the variables that encode) x is chosen nondeterministically at initialization and is kept unchanged. For example, if $\varphi = y_1 \vee y_2$, and 1 encodes y_1 and 2 encodes y_2, then the value 1 of x corresponds to the replacement of y_1 with **false** (which is the $\perp$ value for y_1 in φ) resulting in the formula $(y_1 \vee y_2)[y_1 \leftarrow \textbf{false}] = y_2$. In the automaton $\mathcal{A}_{\neg\varphi}$, each state variable corresponds to a subformula (cf. [BCM$^+$92]), thus the nondeterministic choice of the subformula leads to a mutant automaton $\mathcal{A}_{\neg\varphi[\psi\leftarrow\perp]}$. The state space of the augmented product now consists of triples $\langle x, u, s\rangle$, where x encodes the subformula replaced with $\perp$, and u and s are the components of the product automaton. The successors of $\langle x, u, s\rangle$ are the triples $\langle x, u', s'\rangle$ such that $\langle u', s'\rangle$ is a possible successor of $\langle u, s\rangle$ in a product between the system with the automaton $\mathcal{A}_{\neg\varphi[\psi\leftarrow\perp]}$, where ψ is the subformula encoded by x. The subformulas that affect the value of φ in the systems are these encoded by a value x for which there are initial states u_0 and s_0 of the system and the automaton, respectively, such that there is a fair path from $\langle x, u_0, s_0\rangle$. Let P be the set of triples from which a fair path exists in the augmented product (as above, P can be found symbolically), and let P' be the intersection of P with the initial states of the system and the automaton, projected on the first element. Note that $x \in P'$ iff the subformula associated with x affects the value of φ in the system. Thus, ψ is satisfied vacuously in the system if $\neg P'(0)$ and $P' \neq \{1, \ldots, cl(\psi)\}$.

4.2 Improving average complexity

Consider a Kripke structure $K = \langle AP, W, R, w_0, L\rangle$, a formula φ, and an atomic proposition q. Recall that the naive CTL coverage algorithm, which performs model checking for all dual Kripke structures, has running time of $O(|K| \cdot |\varphi| \cdot |W|)$. While for some dual Kripke structures model-checking may require less than $O(|K| \cdot |\varphi|)$, the naive algorithm always performs $|W|$ iterations of model checking; thus, its average complexity cannot be substantially better than its worst-case complexity. This unfortunate situation arises even when model checking of two dual Kripke structures is practically the same, and even when some of the states of K obviously do not affect the satisfaction of φ in K. In [CKV01] we presented an algorithm that makes use of such overlaps and redundancies. The expectant running time of our algorithm is $O(|K| \cdot |\varphi| \cdot \log |W|)$. Formally, we have the following:

Theorem 5. *The set q-cover(K, φ) can be computed in average*[3] *running time of* $O(|K| \cdot |\varphi| \cdot \log |W|)$.

Our algorithm is based on the fact that for each w, the dual Kripke structure $\tilde{K}_{w,q}$ differs from K only slightly. Therefore, there should be a large amount of work that we can share when we model check all the dual structures. In order to explain the algorithm, we introduce the notion of *incomplete model checking*. Informally, incomplete model checking of K is model checking of K with its labeling function L partially defined. The solution to the incomplete model checking problem can rely only on the truth values of the atomic propositions in states for which the corresponding L is defined. Obviously, in the general case we are not guaranteed to solve the model-checking problem without knowing the values of all atoms in all states. We can, however, perform some work in this direction, which is not needed to be performed again when missing parts of L are revealed.

Consider a partition of W into two equal sets, W_1 and W_2. Our algorithm essentially works as follows. For all the dual Kripke structures $\tilde{K}_{w,q}$ such that $w \in W_1$, the states in W_2 maintain their original labeling. Therefore, we start by performing incomplete model checking of φ in K with L that does not rely on the values of q in states in W_1. We end up in one of the following two situations. It may be that the values of q in states in W_2 (and the values of all the other atomic propositions in all the states) are sufficient to imply the satisfaction of φ in K. Then, we can infer that all the states in W_1 are not q-covered. It may also be that the values of q in states in W_2 are not sufficient to imply the satisfaction of φ in K. Then, we continue and partition the set W_1 into two equal sets, W_{11} and W_{12}, and perform incomplete model checking that does not rely on the values of q in states in W_{11}. The important observation is that incomplete model checking is now performed in a Kripke structure to which we have already applied incomplete

[3] Average is taken with respect to all possible inputs to the algorithm as well as all random choices made by the algorithm.

model checking in the previous iteration. Thus, we only have to propagate information that involves the values of q in W_{12}. Thus, as we go deeper in the recursion described above, we perform less work. The depth of the recursion is bounded by $\log |W|$. As analyzed in [CKV01], the work in depth i amounts in average to model checking of φ in a Kripke structure of size $\frac{|K|}{2^i}$. Hence the $O(|K| \cdot |\varphi| \cdot \log |W|)$ complexity.

In case of vacuity for polar formulas, the naive algorithm performs model checking for each subformula, and thus has running time $O(|K| \cdot |\varphi|^2)$. The quadratic dependency in φ is less crucial than the quadratic dependency in $|W|$ in the case of coverage, but it is still a problem, and efforts to come up with better algorithms are described in [PS02,Nam04]. In order to improve the average complexity, we can encode all the mutations to the formula (each mutation corresponds to a subformula that is replaced by $\perp$) with a vector $\boldsymbol{x}$ of variables. We then proceed with incomplete model checking where in each iteration more variables get values. As with coverage, in each iteration we handle smaller structures, and the overall complexity is, in average, $O(|K| \cdot |\varphi| \log |\varphi|)$[Cho03].

5 Discussion

Sanity checks are applied to the system after model checking has successfully terminated. In addition to vacuity and coverage, other checks that have recently been advocated are *query checking* [Cha00] and *certification* [Nam01]. In query checking, some subformulas in the specification are replaced by the symbol "?" and the query-checking algorithm returns strongest possible replacements to "?" with which the specification is satisfied. In certification, the positive answer of the model-checking procedure is accompanied by a proof that the specification indeed holds. The idea is that it is much easier to check a given certificate than to find one.

The different checks have a lot in common, both conceptually and from the algorithmic point of view. Still, each approach has its own algorithms and tools. We believe that an effort should be made in order to accommodate sanity checks in one algorithmic framework. A good candidate is the theory of multi-valued logic (in fact, this has already been done for query checking in [BG01]). The idea is that typical sanity checks repeat the model-checking procedure with respect to "mutant inputs" — inputs that are slightly different from the original model-checking input. By associating different sets of mutations with different values, we can hopefully reduce the question of finding the set of mutants for which model checking no longer succeeds to the problem of multi-valued model checking [BG04]. In addition, as suggested in [Nam04] for the case of vacuity, it may be possible to carry the sanity checks with respect to the model-checking certificate, rather than with respect to the system and the specification.

References

[AFF$^+$02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211, Grenoble, France, April 2002. Springer-Verlag.

[AFF$^+$03] R. Armon, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M.Y. Vardi. Enhanced vacuity detection for linear temporal logic. In *Computer Aided Verification, Proc. 15th International Conference*. Springer-Verlag, 2003.

[BB94] D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. 31st Design Automation Conference*, pages 596–602. IEEE Computer Society, 1994.

[BBE$^+$01] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367, Paris, France, July 2001. Springer-Verlag.

[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.

[BCC$^+$99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Design Automation Conference*, pages 317–320. IEEE Computer Society, 1999.

[BCM$^+$92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BF00] L. Bening and H. Foster. *Principles of verifiable RTL design – a functional coding style supporting verification processes*. Kluwer Academic Publishers, 2000.

[BFG$^+$05] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 191–206. Springer-Verlag, 2005.

[BG01] Glenn Bruns and Patrice Godefroid. Temporal logic query checking. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS-01)*, pages 409–420, Los Alamitos, CA, June 16–19 2001. IEEE Computer Society.

[BG04] Bruns and Godefroid. Model checking with multi-valued logics. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2004.

[CG04a] M. Chechik and A. Gurfinkel. Extending extended vacuity. In *5th International Conference on Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 306–321. Springer-Verlag, 2004.

[CG04b] M. Chechik and A. Gurfinkel. How vacuous is vacuous? In *10th International Conference on Tools and algorithms for the construction and analysis of systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 451–466. Springer-Verlag, 2004.

[CGL93] E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School)*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer-Verlag, 1993.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automation Conference*, pages 427–432. IEEE Computer Society, 1995.

[Cha00] W. Chan. Temporal-logic queries. In *Computer Aided Verification, Proc. 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463. Springer-Verlag, 2000.

[Cho03] H. Chockler. *Coverage metrics for model checking*. PhD thesis, Hebrew University, Jerusalem, Israel, 2003.

[CK02] H. Chockler and O. Kupferman. Coverage of implementations by simulating specifications. In R.A. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Proceedings of 2nd IFIP International Conference on Theoretical Computer Science*, volume 223 of *IFIP Conference Proceedings*, pages 409–421, Montreal, Canada, August 2002. Kluwer Academic Publishers.

[CKKV01] H. Chockler, O. Kupferman, R.P. Kurshan, and M.Y. Vardi. A practical approach to coverage in model checking. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 66–78. Springer-Verlag, 2001.

[CKV01] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for temporal logic model checking. In *7th International Conference on Tools and algorithms for the construction and analysis of systems*, number 2031 in Lecture Notes in Computer Science, pages 528 – 542. Springer-Verlag, 2001.

[CKV03] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for formal verification. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 2003.

[Dil98] D.L. Dill. What's between simulation and formal verification? In *Proc. 35st Design Automation Conference*, pages 328–329. IEEE Computer Society, 1998.

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *Computer Aided Verification, Proc. 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, 1996.

[HKHZ99] Y. Hoskote, T. Kam, P.-H Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. 36th Design automation conference*, pages 300–305, 1999.

[KGG99] S. Katz, D. Geist, and O. Grumberg. "Have I written enough properties ?" a method of comparison between specification and implementation. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 280–297. Springer-Verlag, 1999.

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

[KV03] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *Journal on Software Tools For Technology Transfer*, 4(2):224–233, February 2003.

[Nam01] K.S. Namjoshi. Certifying model checkers. In *13th Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 2–13. Springer-Verlag, 2001.

[Nam04] K.S. Namjoshi. An efficiently checkable, proof-based formulation of vacuity in model checking. In *16th Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 57–69. Springer-Verlag, 2004.

[Pel01] D. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.

[PS02] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Proc. 14th Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages 485–499. Springer-Verlag, July 2002.

[TK01] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.

[Ver03] Verisity. Surecove's code coverage technology. http://www.verisity.com/products/surecov.html, 2003.

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

[ZHM97] H. Zhu, P.V. Hall, and J.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.