Coverage Metrics for Formal Verification

Hana Chockler, Hebrew University[*]
Orna Kupferman, Hebrew University[†]
Moshe Y. Vardi, Rice University[‡]

November 14, 2003

Abstract

In formal verification, we verify that a system is correct with respect to a specification. Even when the system is proven to be correct, there is still a question of how complete the specification is, and whether it really covers all the behaviors of the system. The challenge of making the verification process as exhaustive as possible is even more crucial in *simulation-based* verification, where the infeasible task of checking all input sequences is replaced by checking a test suite consisting of a finite subset of them. It is very important to measure the exhaustiveness of the test suite, and indeed, there has been an extensive research in the simulation-based verification community on *coverage metrics*, which provide such a measure. It turns out that no single measure can be absolute, leading to the development of numerous coverage metrics whose usage is determined by industrial verification methodologies. On the other hand, prior research of coverage in formal verification has focused solely on state-based coverage. In this paper we adapt the work done on coverage in simulation-based verification to the formal-verification setting in order to obtain new coverage metrics. Thus, for each of the metrics used in simulation-based verification, we present a corresponding metric that is suitable for the setting of formal verification, and describe an algorithmic way to check it.

1 Introduction

Today's rapid development of complex hardware designs requires reliable verification methods. In *formal verification*, we verify the correctness of a design with respect to a desired behavior by checking whether a labeled state-transition graph that models the design satisfies a specification of this behavior, expressed in terms of a temporal logic formula or a finite automaton [CGP99]. Beyond being fully-automatic and reliable, an additional attraction of formal-verification tools is their ability to accompany a negative answer to the correctness query by a counterexample to the satisfaction of the specification in the design [CGMZ95]. On the other hand, when the answer to the correctness query is positive, most formal-verification tools terminate with no further information to the user. Since a positive answer means that the design is correct with respect to the specification, this seems like a reasonable policy. In the last

[*] Address: School of Engineering and Computer Science, Jerusalem 91904, Israel. Email: hanac@cs.huji.ac.il
[†] Address: School of Engineering and Computer Science, Jerusalem 91904, Israel. Email: orna@cs.huji.ac.il. Supported in part by by NSF grant CCR-9988172.
[‡] Address: Department of Computer Science, Houston, TX 77251-1892, U.S.A. Email: vardi@cs.rice.edu. Supported in part by NSF grants CCR-9988322, CCR-0124077, IIS-9908435, IIS-9978135, and EIA-0086264, by BSF grant 9800096, and by a grant from the Intel Corporation.

few years, however, there has been growing awareness of the importance of suspecting the design of containing an error also in the case verification succeeds. The main justification for such suspicion are possible errors in the modeling of the design or of the behavior, and possible incompleteness in the specification.

Several *sanity checks* have been suggested for further assessment of the modeling of the design and the specification [Kur97]. One direction is to detect *vacuous satisfaction* of the specification [BBER01, KV03, PS02], where cases like antecedent failure [BB94] make parts of the specification irrelevant to its satisfaction. For example, the specification "every request is eventually granted" is vacuously satisfied in a design in which no requests are sent. A similar direction is to check the *validity* of the specification (a specification is valid if it holds for all designs). Clearly, vacuity or validity of the specification suggests some problem. It is less 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 design induces a different execution of it, and a design is correct if it behaves as required for all possible input vectors. Checking all the executions of a design is an infeasible task. Simulation-based verification is traditionally used in order to check the design with respect to some input vectors [BF00]. The vectors are chosen so that the verification would be as exhaustive as possible, but still, design errors may escape the verification process. Since simulation-based verification is a heuristic that replaces the infeasible task of checking all input vectors, it is very important 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 design. Essentially, the metrics measure the part of the design that has been activated by the input sequences. For example, in *code-based* coverage metrics, the design is given as a program in some *hardware description language* (HDL), and one measures the number of code lines executed during the simulation. In Section 3, we survey the variety of metrics that are used in simulation-based verification (see also [ZHM97, Dil98, Pel01, TK01]). Coverage metrics today play an important role in the design validation effort [Ver03].

Measuring the exhaustiveness of a specification in formal verification ("do 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 design are visited. Early work on coverage metrics in formal verification [HKHZ99, KGG99] suggested two directions. Both directions reason about a finite-state machine (FSM) that models the design. The metric in [HKHZ99], later followed by [CKV01, CKKV01, CK02], is based on *mutations* applied to the FSM. Essentially, a state s in the FSM 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 FSM and a reduced tableau for the specification. See [CKV01] for a discussion of pros and cons of this metric.

Coming up with an exhaustive specification is of great importance and challenge in formal verification. Sanity checks in formal verification have been helpful in detecting design errors that escape the verification process [BBER01, HKHZ99, PS02]. The main lesson to be learned from several years of research in coverage in simulation-based verification [Pel01, TK01] is that coverage is a heuristic that measures the exhaustiveness of the verification effort, but no single measure can be absolute. Consequently, research in simulation-based coverage has identified numerous coverage metrics; their usage is determined by practical verification methodologies. Prior research of coverage in formal verification [HKHZ99, KGG99, CKV01, CKKV01, CK02] has focused solely on state-based coverage. In contrast, in simulation-based coverage one finds many other coverage metrics, including several metrics of *code coverage*, which measure that all syntactic aspects of the design have been covered [Pel01, TK01]. Our goal in this paper is to adapt the work done on coverage in simulation-based verification to the formal-verification setting in order to obtain new coverage metrics. Thus, for each of the metrics used in simulation-based verification, we present a corresponding metric that is suitable for the setting of formal verification. In addition, we describe symbolic algorithms for computing each of the new metrics.

The adoption of metrics from simulation-based verification is not straightforward. To see this, consider for example code-based coverage and a check 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 design, one for each branch, in which the branch is disabled. Such a mutant design, however, has less behaviors than the original design, 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 design. 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 design obtained by removing this behavior. The way we suggest to do so is to check whether the specification is vacuously satisfied in a mutant design in which this behavior is disabled: a vacuous satisfaction of the specification in such a design (we assume that the specification is not vacuously satisfied in the original design) indicates that the specification does refer to this behavior; on the other hand, a non-vacuous satisfaction of the specification in the mutant design indicates that the specification does not refer to the missing behavior. Accordingly, some of the new metrics we suggest reduce coverage to queries about vacuous satisfaction. On the other hand, a code-based metric that checks whether a particular assignment in the code has been executed may also be reduced to a metric that checks the satisfaction of the specification in a mutant design in which the assignment is changed. Accordingly, some of the metrics we suggest follow the approach in [HKHZ99] and reduce coverage to queries about satisfaction of the specification in mutant designs. Unlike previous work, however, the mutant designs we consider are not arbitrary, and capture the different metrics of coverage used in simulation-based verification.

2 Preliminaries

2.1 Simulation-Based Verification

In *simulation-based verification*, the implementation of a hardware design is executed in parallel with a reference model described at a different level of abstraction or with monitors and assertions that check for certain behavior of the implementation [KN96]. The execution is done with respect to a selected set of finite input sequences, referred to as *tests*. Thus, assuming the implementation has a set I of input signals, a test is a finite sequence $t = i_0, i_1, \ldots, i_n \in (2^I)^*$ of input assignments. Implementations of hardware designs can be described by different formalisms. We consider two formalisms with respect to which coverage metrics are naturally defined.

The first formalism is that of *hardware description languages* (HDL). A typical HDL program specifies the input and output variables of the various modules of the design, and, using control and assignment statements, the interaction of the modules among themselves and with an environment that provides the input signals. Reasoning about rich HDL such as Verilog involves difficult technical details.[1] We consider here the simplified model of control flow graph (CFG). Each HDL statement corresponds to a control state and induces a node in the CFG. We refer to CFG nodes as *locations*. Assignment statements have a single successor, and control statements, such as *if* or *while*, have several successors, corresponding to the possible locations to which the control can jump. Transitions from a control statement to its successors are labeled by an expression that guards the transition. Recall that the design interacts with an environment that supplies its input signals. When the design is described as a CFG, the interaction induces a traversal of the CFG. Formally, given a CFG G with a set V of locations, and a test $t = i_0, i_1, \ldots, i_n \in (2^I)^{n+1}$ of input assignments, the *execution* of G on t is a sequence $\langle i_0, l_0^0, \ldots, l_{m_0}^0, o_0\rangle, \langle i_1, l_0^1, \ldots, l_{m_1}^1, o_1\rangle, \ldots, \langle i_n, l_0^n, \ldots, l_{m_n}^n, o_n\rangle \in (2^I \times V^+ \times 2^O)^{n+1}$ such that l_0^0 is the initial location of G, for all $0 \leq i \leq n$, the location $l_{m_i}^i$ corresponds to a read and write assertion, o_i is the new assignment to the output variables, and l_0^{i+1} matches the control flow of the CFG from location $l_{m_i}^i$ upon reading i_{i+1}. The locations $l_1^{i+1}, \ldots, l_{m_{i+1}}^{i+1}$ then correspond to the control flow of the CFG from l_0^{i+1} until the next input assignment is read. We often ignore the input and output variables and refer to the interaction as a word in V^* obtained by projecting the execution above on V.

The second formalism is that of *sequential circuits*. We refer to a circuit as a tuple $\mathcal{S} = \langle I, O, \mathcal{L}, \mathcal{F}, \mathcal{G}, c_0\rangle$, where I and O are the sets of input and output signals, respectively, $\mathcal{L}$ is a set of latches, $c_0 \in 2^{\mathcal{L}}$ describes the initial values of the latches, and $\mathcal{F}$ and $\mathcal{G}$ are families of the next-state and output functions. Thus, each latch $l \in \mathcal{L}$ has a function $f_l : 2^I \times 2^{\mathcal{L}} \to \{0,1\}$ in $\mathcal{F}$, and each output signal $o \in O$ has a function $g_o : 2^I \times 2^{\mathcal{L}} \to \{0,1\}$ in $\mathcal{G}$. A *configuration* $c \in 2^{\mathcal{L}}$ of the circuit describes the value of each latch. The circuit starts its interaction with the environment in configuration c_0. When the circuit is in configuration c and it reads a set $i \in 2^I$ of input signals, it moves to configuration c' in which the value of each latch l is $f_l(i,c)$, and in which it sends to the environment the set of output signals o with $g_o(i,c) = 1$. Accordingly, the *execution* of a circuit $\mathcal{S}$ on a test $t = i_0, i_1, \ldots, i_n \in (2^I)^{n+1}$, is a sequence $\langle i_0, c_0, o_0\rangle, \langle i_1, c_1, o_1\rangle, \ldots, \langle i_n, c_n, o_n\rangle \in (2^I \times 2^{\mathcal{L}} \times 2^O)^{n+1}$ that satisfies the conditions above.

Both HDL and circuits enable a description of the design at different levels of abstraction [Hos95], yet abstraction is most naturally supported when the design is modeled as a *symbolic finite state machine* (FSM). We assume that the design is defined with respect to a set X of state variables, and it is specified by predicates on X and X' – a primed version of the variables in X. Formally, an FSM is a tuple $F = \langle I, O, X, \theta_{in}, \theta_{next}, \mathcal{G}\rangle$, where I and O are the input and output variables, X is the set of state variables, inducing the state space 2^X, θ_{in} is a predicate on X describing the set of initial states, θ_{next} is a predicate on $X \cup X'$ describing the transition relation (there is a transition from state u to state v iff $\theta_{next}(u, v')$), and $\mathcal{G}$ is a family of predicates that associates with each input or output variable s a predicate τ_s on X describing the set of states in which s holds. Likewise, predicates on X are used to describe other sets of interest, for example, the set of fair states when the design comes with an unconditional fairness constraint. Formally, a *fair* FSM F is a tuple $F = \langle I, O, X, \theta_{in}, \theta_{next}, \mathcal{G}, \alpha\rangle$, where α is a predicate on X describing the accepting condition. A behavior π is accepted by F if it

[1] For a description of a formal model of a real-life HDL see, for example, [FLLO95].

satisfies α. The simplest accepting condition is Büchi condition [Büc62] (called *impartiality* in [MP92]), where α is a predicate on X and a behavior π satisfies α if it visit a state satisfying α infinitely often.

A non-symbolic FSM for the design is usually created manually by the designer in early stages of the design. Symbolic FSMs are automatically extracted from HDL programs, which is were abstraction plays an important role [Hos95]. The HDL program typically contains too many variables. The designer then selects a subset $X' \subset X$ of the state variables, and the predicates that define the FSM are viewed as predicates over X' to obtain an abstract FSM. Note that an FSM may be nondeterministic, thus its interaction with the environment may produce several outputs.

2.2 Model checking, vacuity, and coverage

In *linear-time model checking*, we check whether a design has a desired behavior by checking whether a Büchi automaton for the negation of the specification has accepting runs on an FSM describing the design [VW86]. The specification can be expressed as an LTL formula [Hol97], as a ForSpec formula [AFG+02], or as a Büchi automaton [HHK96, Kur98]. A specification φ in linear temporal logic can be translated to a nondeterministic Büchi automaton $\mathcal{A}_{\neg\varphi}$ that accepts all words that do not satisfy φ [VW94]. Given an automaton $\mathcal{A}_{\neg\varphi}$, we check that the product of F with $\mathcal{A}_{\neg\varphi}$, which is a fair FSM $F \times \mathcal{A}_{\neg\varphi}$, does not contain accepting paths.

Sanity checks for model checking address the problem of errors in the modeling of the design and the desired behavior, which are not discovered by model checking, for example, incompleteness of the specification. These problems may cause "false positive" results of model checking and conceal errors in the design. Two such checks are *vacuity* and *coverage*, which we briefly review below (for the full details, see [BBER01, KV03, HKHZ99, CKV01]).

Intuitively, an FSM F satisfies a formula φ vacuously if F satisfies φ yet it does so in a non-interesting way, which is likely to point on some trouble with either F or φ. In order to formalize this intuition, we first say that a subformula ψ of φ *does not affect* φ *in* F if for every formula ξ, the FSM F satisfies $\varphi[\psi \leftarrow \xi]$ iff F satisfies φ, where $\varphi[\psi \leftarrow \xi]$ denote the formula obtained from φ by replacing ψ with ξ [BBER01]. As shown in [KV03], when ψ has a single occurrence in φ, then instead of checking the replacement of ψ by all formulas ξ, one can check only the replacement of ψ by the formulas $\mathbf{true}$ and $\mathbf{false}$. Thus, ψ does not affect φ in F whenever F satisfies $\varphi[\psi \leftarrow \mathbf{true}]$ iff F satisfies $\varphi[\psi \leftarrow \mathbf{false}]$. Now, an FSM F satisfies a formula φ *vacuously* iff $F \models \varphi$ and there is some subformula ψ of φ such that ψ does not affect φ in F. Equivalently, F satisfies φ vacuously if $F \models \varphi$ and there is some subformula ψ of φ such that F also satisfies $\varphi[\psi \leftarrow \bot]$, where $\bot$ is either $\mathbf{false}$ or $\mathbf{true}$, depending on the polarity of ψ in φ. It is easy to see that vacuous satisfaction can be detected by a naive algorithm that model checks F with respect to formulas obtained from φ. More sophisticated algorithms are suggested in [PS02, KV03, Cho03, AFF+03].

Coverage in model checking was introduced in [HKHZ99, KGG99]. The metric in [HKHZ99] is based on mutations applied to an FSM. For an FSM $F = \langle I, O, X, \theta_{in}, \theta_{next}, \mathcal{G}\rangle$, a state $w \in 2^X$ and an output variable $q \in O$, a *mutant* FSM $\tilde{F}_{w,q}$ is obtained from F by dualizing the value of q in the state w. Thus, if τ_q is the predicate describing the set of states satisfying q in F, then the predicate $\tilde{\tau}_{w,q}$, which describes the set of states satisfying q in $\tilde{F}_{w,q}$, is satisfied by w iff τ_q is not satisfied by w. For all states $v \neq w$, the predicate $\tilde{\tau}_{w,q}$ is satisfied by v iff τ_q is satisfied by v. For an FSM F, a specification φ that is satisfied in F, and an output variable q, we say that φ *q-covers* w iff $\tilde{F}_{w,q}$ no longer satisfies φ. By [HKHZ99], a state is covered if it is q-covered for some output variable q. It is easy to see that the set of states q-covered by φ can be computed by a naive algorithm that performs model checking of φ in $\tilde{F}_{w,q}$ for each state w of F. More sophisticated algorithms are suggested in [HKHZ99, CKV01, CKKV01].

Chockler et al. also suggest the following refinement of coverage metrics [CKKV01]. Instead of performing local mutations in F, we can perform local mutations in the infinite tree T_F obtained by unwinding F. A state w of F can appear many (possibly an infinite number of) times in T_F. Flipping the value of q in one occurrence of w in T_F can have a different effect from flipping the value of q in all or some of the occurrences of w in T_F. These differences are captured by the notions of *node, structure*, and *tree* coverage. Node coverage of a state w corresponds to flipping the value of q in one occurrence of w in the infinite tree. Structure coverage corresponds to flipping the value of q in all the occurrences of w in the tree. It may seem that node coverage defines more fine-grained mutations than structure coverage. As shown in [CKKV01], the two metrics are incomparable: a state may be node-covered and not structure-covered, and vice versa. Tree coverage generalizes node and structure coverage and corresponds to flipping the value of q in some occurrences of w in the tree. Thus, w is q-tree-covered by φ if there exists a subset of occurrences of w in the infinite tree obtained by unwinding F such that flipping q in this subset falsifies φ in F. Chockler et al. describe a framework in which node, structure, and tree coverage can be computed by a symbolic algorithm; minor changes are required to capture the different types of coverage [CKKV01]. We describe their algorithm in more detail in Section 5.

In this paper we introduce new types of mutations and new types of coverage metrics in model checking in order to capture better the different notions of coverage used in simulation-based verification. Coverage in model checking is performed by applying mutations to a given FSM and then examining the resulting mutant FSMs with respect to a given specification. Each mutation is generated in order to check whether a specific element of the design is essential for the satisfaction of the specification. As we explain in more detail in Section 4, mutations correspond to omissions and replacements of small elements of the design, which can be given as an HDL program, an FSM, or a sequential circuit. Once we have a mutant FSM, there are two coverage checks we can perform on it.

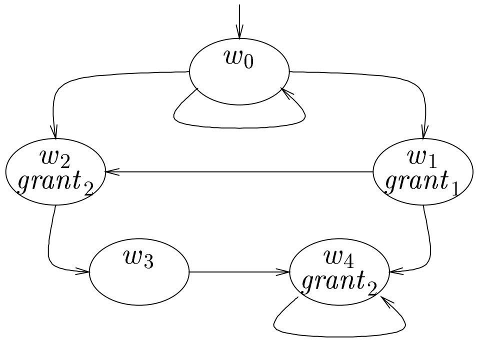
1. *Falsity coverage:* does the mutant FSM still satisfy the specification?

2. *Vacuity coverage:* if the mutant FSM still satisfies the specification, does it satisfy it vacuously?

Falsity coverage is the metric introduced in [HKHZ99], and we extend it here to handle mutations richer than these studied in the literature so far. Vacuity coverage is new. As we demonstrate in Example 2.1, it often provides information that falsity coverage fails to detect. In particular, in mutations that are based on omission of elements from the original design (as we are going to see in Section 4, such mutations are popular in metrics adopted from simulation-based verification), falsity coverage is useless for universal specifications. Indeed, having less behaviors, the mutant design is guaranteed to satisfy all the specifications satisfied by the original design.

Example 2.1 Consider the FSM F described below, which abstracts a design with respect to the output signals $grant_1$ and $grant_2$. Let $\varphi = G(grant_1 \rightarrow F grant_2)$. Thus, φ requires that (in all execution paths) each grant to the first user is followed by a grant to the second user. It is easy to see that φ is satisfied in F. Recall that the goal of coverage metrics is to check whether all the elements of the design play some role in the satisfaction of φ. Let us see which parts of F are covered by φ. We refer only to structure coverage in this example.

- The positive value of $grant_2$ in w_4 is essential to the satisfaction of φ: One way to see this, is to observe that changing it to **false** creates the behavior $\{\} \cdot \{grant_1\} \cdot \{\}^\omega$ in which $grant_1$ is not



followed by $grant_2$. the state w_4 is falsity covered by φ with respect to mutations that flip the value of $grant_2$.

- The value of $grant_1$ in w_1 is not essential to the satisfaction of φ. On the other hand, the designer had a reason to set it to **true** in w_1, as it is essential to the non-vacuous satisfaction of φ: Indeed, omitting w_1 or flipping the value of $grant_1$ there results in a mutant in which $grant_2$ does not affect the satisfaction of φ. the state w_1 is vacuity covered by φ with respect to mutations in which w_1 is omitted and with respect to mutations that flip the value of $grant_1$.

- One may also question negative values of variables. For example, while the negative value of $grant_2$ in w_0 is not essential to the satisfaction of φ, it is essential to its non-vacuous satisfaction: Indeed, flipping the value of $grant_2$ in w_0 to **true** results in a mutant in which $grant_1$ does not affect the satisfaction of φ. the state w_0 is vacuity covered by φ with respect to mutations that flip the value of $grant_2$.

- Consider now the value of $grant_2$ in the state w_2. All the paths of F that pass through w_2 describe a behavior in which two grants – in both w_2 and in w_4, are given to the second user, after at most one grant was given to the first user. The specification does not require such a behavior, nor does it require a correspondence between the number of grants that each user gets. The labeling of w_2 indeed does not play a role in the satisfaction of φ: flipping the value of $grant_2$ in w_2 or omission of w_2 do not falsify φ in F, nor it causes φ to be satisfied vacuously. This information may hint on a possible impreciseness or incompleteness in the definition of φ. For example, if the intention of the designer had been that each $grant_1$ should be followed by a $grant_2$ in the next step, the correct specification should have been $G(grant_1 \rightarrow X\, grant_2)$, in which case w_2 would have been falsity covered with respect to mutations that flips the value of $grant_2$. Moreover, if the intention of the design had been to require a correspondence between the number of grants that each user gets, the coverage information would lead to a refinement of the specification, after which an error would be discovered. Another possibility is that F contains redundancies, in which case the state w_2 can be omitted from the FSM without violating φ.

Recall that the positive value of $grant_1$ in w_4 is essential for the satisfaction of φ. An execution that reaches w_4 stays there forever, and the second user is kept being granted. One may wonder whether all these grants are essential. This is where the difference between structure-based and node-based coverage appears: while w_4 is structure-covered by φ with respect to the mutation that flips the value of $grant_2$ in w_4, it is not node-covered by φ with respect to the same mutation, for both the falsity and vacuous coverage. Indeed, node-coverage checks whether the positive value of $grant_2$ plays a role in all the visits to w_4 [2].

3 Coverage Metrics in Simulation-based Verification

In this section we briefly survey coverage metrics in simulation-based verification – metrics we are going to adopt for the setting of formal verification in the next section. Each of the metrics is "tailored" for a specific representation of the design or a specific verification goal. The reader is referred to [TK01] for a detailed survey. All metrics refer to a set of input sequences (or tests) $t \in (2^I)^*$ with respect to which the design is simulated.

3.1 Syntactic coverage metrics

Syntactic coverage metrics assume a specific formalism for the description of the design and measure the syntactic part of the design visited in the process of execution of a given input sequence. Commonly [Mar99, TK01], high coverage according to syntactic-based metrics is considered a precondition to moving to other more sophisticated (and time consuming) coverage metrics.

Code coverage Code-based coverage metrics refer to the HDL program that describes the design or to its CFG. Measuring code coverage requires little overhead and it is easy to interpret the coverage information. This makes code coverage the most popular metric [UZ98, TK01]. The most widely used code-coverage metrics are *statement* and *branch* coverage. Essentially, an object is covered if it is visited during the execution of the input sequence. Again, the fully-formal definition depends on the particular HDL used, but a semi-formal definition is given in terms of the computation of the CFG as follows. Let G be a CFG. For an input sequence $t \in (2^I)^*$ such that the execution of G on t, projected on the sequence of locations, is $l_0, \ldots, l_m$, we say that a statement τ is covered by t if there is $0 \leq j \leq m$ such that the control location l_j corresponds to τ. We say that a branch $\langle l, l' \rangle$ between two control locations is covered by t if there is $0 \leq j \leq m-1$ such that $l_j = l$ and $l_{j+1} = l'$. More sophisticated metrics measure the way expressions in the guards labeling the CFG's transitions are satisfied. For example, *expression coverage* checks whether a Boolean expression has been satisfied by all its satisfying assignments (e.g., whether $a_1 == a_2$ has been satisfied by both an $a_1 = a_2 = 0$ and an $a_1 = a_2 = 1$ assignment).

Circuit coverage Circuit-structure based coverage metrics refer to the circuit that describes the design. Thus they identify the physical parts of the circuit that are covered. Measuring circuit coverage is usually easy and it is easy to interpret the coverage information. Unlike code coverage, however, it is not easy to use the coverage information in order to generate new tests that direct simulation towards the unexplored areas of the design. The most widely used circuit-coverage metrics are *latch* and *toggle* coverage [HH96, KN96]. Essentially, a latch is covered if it changes its value at least once during the execution of the input sequence. Similarly, an output variable is covered if its value has been toggled. Formally, for a circuit $\mathcal{S}$ and an input sequence $t \in (2^I)^{n+1}$ such that the execution of $\mathcal{S}$ on t is $\langle i_0, c_0, o_0 \rangle, \langle i_1, c_1, o_1 \rangle, \ldots, \langle i_n, c_n, o_n \rangle$, we say that a latch $l \in \mathcal{L}$ is covered by t if there is $j \geq 0$ such

[2]It may seem that node coverage defines more fine-grained mutations than structure coverage. As shown in [CKKV01], the two metrics are incomparable: a state may be node-covered and not structure-covered, and vice versa.

that $l \in c_0$ iff $l \notin c_j$. Similarly, an output variable $o \in O$ is covered by t if there are $0 < j_1 < j_2$ such that $o \in o_0$ iff $o \notin o_{j_1}$ iff $o \in o_{j_2}$. Note that toggle coverage requires that the value of an output variable should be changed at least twice during the execution of t.

Hit count The notion of *hit count*, introduced in [Ver03], replaces binary coverage queries with quantitative measurements (how many times an object has been visited). Intuitively, the more times we visit an element, the bigger is our confidence that its functionality is tested.

3.2 Semantic coverage metrics

Semantic coverage metrics measure the part of the functionality of the design exercised by the set of input sequences. Semantic coverage metrics require user help and are more sophisticated than syntactic coverage metrics. We consider the following metrics.

FSM coverage Due to the large size of FSMs for complete systems, FSM-based coverage metrics refer to more abstract FSMs constructed manually by the designer, or automatically extracted from the design by projecting its symbolic description on a subset of the state variables as explained in Section 2.1 [TK01]. Similarly to code coverage, a state or a transition of the abstract FSM is covered if it is visited during the execution of the input sequence. The fact that coverage is checked with respect to an abstract FSM makes the interpretation of the coverage information harder (linking the uncovered parts of the FSM to uncovered parts of the HDL program is not trivial) and have led to the use of more sophisticated metrics. In particular, limited-path coverage metrics check that important sequences of behavior are exercised [SA99]. Transition coverage can be viewed as a special case of path coverage, for paths of length 1.

Assertion coverage In assertion coverage, also called "functional coverage" in [TK01, Cad03]), the user provides a list of assertions referring to the variables of the design. The assertions describe some conditions that may be satisfied during the execution or a state of the design during the execution. They may be propositional ("snapshot tasks") or temporal (describing a behavior along several clock cycles). A test t covers an assertion a if the execution of the design on t satisfies a. The assertion-coverage metric measures what assertions are covered by a given set of input sequences.

Mutation coverage In mutation coverage, the user introduces a small change (aka "mutation") to the design, and checks whether the change leads to an erroneous behavior [DLS78, Bud81, ZHM97]. The coverage of a test t is measured as the percentage of the mutant designs that fail on t, that is, the percentage of the mutations that t "catches". The list of interesting mutations can be written manually or automatically following some mutation criteria. For example, a local mutation can be flipping a value of one output variable in a circuit. Budd and Angluin [BA82] define the *neighborhood* $\Phi(P)$ of a program P as a set of all possible mutants obtained from P by a (predefined) local mutation. Similarly, we can define a neighborhood of an FSM F (for example, all FSMs obtained from F by merging two states) or a circuit $\mathcal{S}$. In mutation coverage the goal is to cover the whole neighborhood of the design, that is, to find a set of input sequences such that for each mutant design there exists at least one test that fails on it. (this notion is similar to the notion of *local correctness* in [BA82]). As discussed in Section 2.2, mutation coverage is the metric that inspired most of the work on coverage in model checking.

4 Coverage Metrics in Model Checking

In this section we discuss how the coverage metrics from simulation-based verification can be adopted in model checking. Thus, for each of the metrics described in Section 3, we define a metric that can be used in the context of model checking.

4.1 Syntactic coverage

In syntactic coverage, we assume that we are given the syntactic representation of the design (an HDL code or a CFG) with respect to which we measure the coverage. Since in the process of model checking we visit the whole reachable part of the design, metrics that measure the part of the design exercised during the simulation cannot be applied directly to model checking. Essentially, we adopt these metrics by replacing the question whether a part of the design has been visited during the simulation by the question whether the part plays a role in the success of the verification process, where playing a role means that the part is essential for the satisfaction or the non-vacuous satisfaction of the specification. The latter is checked by reasoning about the behavior of a mutant design in which the part is modified or omitted.

Code coverage Let G be a CFG and φ a specification that is satisfied in G. We say that a statement τ of G is covered by φ if omitting τ from G causes vacuous satisfaction of φ in the mutant CFG. Similarly, a branch $\langle l, l' \rangle$ of G is covered if omitting it causes vacuous satisfaction of φ. Note that falsity coverage would be meaningless here, since omitting a statement or a branch of CFG results in a design with fewer behaviors, which is guaranteed to satisfy the universal specification. In expression coverage, we check whether omitting the behaviors in which the variables have a particular satisfying assignment for a particular expression leads to vacuous satisfaction of φ. For example, if the expression is $a_1 == a_2$ and it guards a block B of statements, we check whether the mutant design obtained by skipping B in case $a_1 = 0$ and $a_2 = 0$ satisfies the specification vacuously, and similarly for $a_1 = 1$ and $a_2 = 1$.

Circuit coverage Recall that latch and toggle coverage metrics check whether the value of a specific latch or variable in the circuit changes during the execution of an input sequence. We replace this question by the question whether disabling the change causes the specification to be satisfied vacuously. Thus, a latch $l \in \mathcal{L}$ is covered if the specification is vacuously satisfied in the circuit obtained by fixing the value of l to its initial value. Similarly, an output variable $o \in \mathcal{O}$ is covered if the specification is vacuously satisfied in the circuit obtained by allowing o to change its value only once. Thus, if the initial value of o is 0, the circuit is obtained by fixing o to 1 as soon as it changes its value to 1, and if the initial value of o is 1, the circuit is obtained by fixing o to 0 as soon as it changes its value to 0.

Hit count Recall that the distinction between structure, node, and tree coverage enable us to choose whether to perform a mutation on every visit of the model checker to the mutated element, on exactly one visit, or on a subset of the visits. We define hit count in model-checking coverage as the minimal number of visits in which we have to perform the mutation (or omit the element) in order to falsify the specification in the design or to make it vacuously satisfied.

4.2 Semantic coverage

Among the semantic coverage metrics, mutation coverage has already been adopted to the setting of model checking. As discussed in Section 2.2, we suggest a strengthening of the adopted metrics by checking the effect of the mutation not only on the satisfaction of the specification, but also on its vacuous satisfaction. Below we describe the adoption of the other semantic coverage metrics.

FSM coverage In FSM coverage we are given an abstract FSM F and we check the influence of mutations and omissions in this FSM on the result of model checking of the specification φ in the design. In state coverage, for a state w of F we check the influence of omission of w or changing the values of output variables in w on the (non-vacuous) satisfaction of the specification in the design. Clearly, a mutant FSM $\tilde{F}_w$ obtained from F by omitting w has fewer behaviors than F, thus for omissions of a state we only check vacuity coverage. On the other hand, a mutant FSM $\tilde{F}_{w,o}$ obtained from F by flipping the value of the output variable $o \in O$ in w can also falsify the specification, thus we check falsity and vacuity coverage.

In path coverage, we check the influence of omitting or mutating a finite path on the (non-vacuous) satisfaction of the specification in the design. A path π of length c in F is a sequence of states $w_1, \ldots, w_c$ of F such that for all $1 \leq i \leq c - 1$ we have $\theta_{next}(w_i, w_{i+1})$. Let us first define coverage for omissions of a path. A path π is covered by φ if the mutant FSM $\tilde{F}_\pi$ obtained from F by omitting all behaviors that contain π satisfies φ vacuously. On the other hand, we can also introduce mutations that replace π with a mutant path $\tilde{\pi}$ in the FSM. Then, the mutant FSM $\tilde{F}_{\pi,\tilde{\pi}}$ is obtained from F by replacing π with $\tilde{\pi}$. The mutant FSM $\tilde{F}_{\pi,\tilde{\pi}}$ can falsify φ or can satisfy φ vacuously, thus for mutations that replace a path with another, mutant, path we check both falsity and vacuity coverage. We note that all possible mutations in the FSM can be introduced consistently on each occurrence of the mutated element, on exactly one occurrence, or on a subset of occurrences, thus resulting in structure, node, or tree coverage, respectively.

Assertion coverage An input to assertion-coverage check is an FSM F, a specification φ that is satisfied non-vacuously in F, and a list of LTL assertions $a_1, \ldots, a_k$. An assertion a_i is covered by φ in F if the mutant FSM $\tilde{F}_{a_i}$ obtained from F by omitting all behaviors that satisfy a satisfies φ vacuously. We note that this definition is similar to the definition of FSM path coverage. The only difference is in the description of the mutation: in FSM path coverage we omit behaviors that contain a given finite path π, whereas in assertion coverage we omit behaviors that satisfy a given assertion.

5 Coverage Computation

In Section 4 we described new coverage metrics for model checking. In this section we discuss how to compute these metrics. We first show that both vacuity and falsity coverage can be reduced to model checking (possibly of mutant specifications and/or mutant designs). Let F be an FSM, φ a specification that is satisfied in F non-vacuously, and $\tilde{F}$ a mutant FSM. If $\tilde{F}$ does not satisfy φ, we say that $\tilde{F}$ is falsity covered by φ. If $\tilde{F}$ satisfies φ, it still may be vacuity covered by φ if it satisfies φ vacuously. Formally, $\tilde{F}$ satisfies φ vacuously if $\tilde{F} \models \varphi$ and there exists $\psi \in cl(\varphi)$ such that $\tilde{F}$ satisfies $\varphi[\psi \leftarrow \bot]$. Thus, like falsity coverage, we check whether a mutant design $\tilde{F}$ satisfies a specification, only that here the specification is also mutated.

Mutation coverage The algorithm we present for falsity-coverage computation is based on the coverage algorithm described in [CKKV01]. That algorithm computes symbolically falsity coverage for mutations that flip the value of a variable $q \in O$ in one state w of the FSM. The idea is to look for a fair path in the product of the mutant FSM $\tilde{F}$ and an automaton $\mathcal{A}_{\neg\varphi}$ for the negation of φ. The state space of the product is $2^X \times S$, where X is the set of state variables of F, S is the state space of $\mathcal{A}_{\neg\psi}$, and the transitions of the product are induced by the transition relations of F and $\mathcal{A}_{\neg\psi}$. In order to compute the set of covered states, it is suggested in [CKKV01] to add $|X|$ new variables that encode the state w in which the value of q is flipped. It is now possible to define symbolically an augmented product, with state space $2^X \times 2^X \times S$, where the first component of a state $\langle w, u, s\rangle$ is the state w that is being considered, and the two other components are as in the usual product automaton. The value of the first component is chosen nondeterministically at initialization and is kept unchanged. The copy of the augmented product with first component w checks whether the mutation of F in which q is flipped in w contains a fair path (in which case flipping q in w violates the specification). Thus, when the augmented product is in a state $\langle w, w, s\rangle$, the set of successor states contains all triples $\langle w, u, t\rangle$ such that u is a successor of w and $t \in \delta(s, \tilde{\sigma})$, where $\tilde{\sigma}$ is the label of w in $\tilde{F}_{w,q}$. The above describes structure coverage, where the value of w is flipped in all visits. Likewise, we can define an augmented product in which the value of q in w is flipped only one time (node coverage) or some of the times (tree coverage). We can now use a symbolic algorithm in order to find the set P of all triples $\langle w, u, s\rangle$ from which there exists a fair path in the augmented product automaton. The covered states are those w such that $\langle w, u_0, s_0\rangle \in P$, for some initial states u_0 of F and s_0 of $\mathcal{A}_{\neg\psi}$ In particular, triples of the form $\langle w, u_0, s_0\rangle$ in P, for an initial states u_0 of F and initial state s_0 of $\mathcal{A}_{\neg\psi}$, correspond to a path of $\tilde{F}_{w,q}$ that violates the specification. Hence, the set of states that are covered by the specification can be calculated symbolically by intersecting P with the initial state set of the product automaton and projecting the result on the first component.

Vacuity coverage Recall that checking whether a system satisfies a specification vacuously involves model checking of a mutant specification. We adjust the symbolic algorithm in [CKKV01] to this setting by adding a new variable x that encodes the subformula $\psi \in cl(\varphi)$ that is being replaced with $\perp$. The variable x is an integer in the range $0, \dots, |cl(\varphi)|$, 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. As with mutations, the values of these variables are chosen nondeterministically at initialization and are 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 $P' \neq \{1, \dots, cl(\psi)\}$.

In order to get a symbolic algorithm for vacuity coverage, we combine the above algorithm with the one of [CKKV01]. For example, if we want to find the set of states w such that flipping the value of q in w causes the specification to be satisfied vacuously, we augment the state space of the product of F and $\mathcal{A}_{\neg\varphi}$ by variables that encode both the state in which we do the mutation and the subformula that is being replaced with $\perp$. Thus, the state space of the augmented product now consists of 4-tuples $\langle w, x, u, s\rangle$, where w is the state in which we flip q, x encodes the subformula replaced with $\perp$, and u and s are the components of the product automaton. The successors of $\langle w, x, u, s\rangle$ are the 4-tuples $\langle w, x, u', s'\rangle$ such that $\langle u', s'\rangle$ is a possible successor of $\langle u, s\rangle$ in a product between the system q flipped in w and the automaton $\mathcal{A}_{\neg\varphi[\psi\leftarrow\perp]}$, where ψ is the subformula encoded by x. As above, the initial values of w and x are chosen nondeterministically at initialization and are kept unchanged. The states that are not vacuity covered are states w such that for all the assignments x, there are initial states u_0 and s_0 of the system and the automaton, respectively, such that there is a fair path from $\langle w, x, u_0, s_0\rangle$. Thus, if P is the set of 4-tuples from which a fair path exists in the augmented product (as above, P can be found symbolically), then the set of mutations that are not covered can be obtained from P as follows. Let P' be the intersection of P with the initial states of the system and the automaton, projected on the first two elements. Note that a pair $\langle w, x\rangle$ is in P' if flipping q in w violates the specification obtained by replacing the subformula associated with x by $\perp$. Consider now a the set of mutations defined by $(\forall x)P'(w, x)$. It contains states that lead to failure of ϕ regardless of what subformula ψ of ϕ is replaced by $\perp$, which means that these mutations do not lead to vacuous satisfaction and therefore they are not covered. As we specify below, if we want to check vacuity coverage for other types of mutations, we replace the variables that encode w by variables that encode other types of mutations.

Code coverage Recall that in code coverage we need to check whether the omission of parts of the code causes the specification to be satisfied vacuously. Accordingly, for code coverage, it is simpler to define the mutations with respect to the HDL code. Let k be the number of elements in the code we want to check (e.g., the number of lines). We introduce a new variable mut, which is an integer in the interval $[1, \ldots, k]$. The value i of mut indicates that the mutation is in element l_i, which we want to omit, and we need $O(\log k)$ Boolean variables to encode it. The HDL code is instrumented using source-to-source translation in (see [BKM02] as an example of such instrumentation) so that l_i in the code is replaced by the statement "*if* $(mut \neq i)$ *then* l_i *else* **skip**". The instrumented code represents all the mutant designs[3]. The product of the FSM induced by the instrumented code and $\mathcal{A}_{\neg\varphi}$ subsumes all the mutations of the code. It is now possible to apply the symbolic algorithm described above (instead of the variables that encode w, we now have the variables that encode mut) for detecting the mutations that lead to vacuous satisfaction.

In expression coverage, we do something similar. Let $e_1, \ldots, e_m$ be the expressions we want to check, and let $V_i = \{v_1^i, \ldots, v_n^i\}$ be the Boolean variables over which e_i is defined. Assume that n bounds the number of variables in every expression. Let "*if* e_i *then* B_i" be the statement that contains e_i as a guard (handling of "*while*" or "*until*" statements is similar). Recall that we want to check, for each e_i and for each satisfying assignment $f \in 2^{V_i}$, whether skipping B_i when the variables have value f causes the specification to be satisfied vacuously. Accordingly, we add a variable mut (encoded by $O(\log m)$ Boolean variables) that indicates the expression to be checked, and n variables $u_1, \ldots, u_n$ that encode assignments to n variables. As usual, the variables get their value nondeterministically at initialization.

[3]The user may wish to include 0 (no mutation) in the range of mut, in which case the instrumented code represents also the original design.

The HDL code is now instrumented so that "*if* e_i *then* B_i" in the code is replaced by "*if* $(mut \neq i)$ *or* $(e_i \wedge \bigvee_{1 \leq j \leq n} v_j^i \neq u_j)$ *then* B_i *else* **skip**". It is now possible to apply the symbolic algorithm described above for detecting the expressions and assignments that lead to vacuous satisfaction.

Circuit coverage In latch coverage, we want to restrict the product of F and $\mathcal{A}_{\neg\varphi}$ to paths in which the value of a latch is not allowed to change, and check whether this causes vacuous satisfaction. Thus, we augment the product with variables that encode the examined latch and (for the vacuity check) the subformula of φ that we replace with $\bot$. When the augmented product is in state $\langle l, x, u, s \rangle$, it proceeds to states $\langle l, x, u', s' \rangle$ such that $\langle u', s' \rangle$ is a possible successor of $\langle u, s \rangle$ in a product between the system with l that is not changed and the automaton $\mathcal{A}_{\neg\varphi[\psi \leftarrow \bot]}$, where ψ is the subformula encoded by x. The values of l and x are chosen nondeterministically at initialization. Let P be the set of 4-tuples from which a fair path exists in the augmented product, and let P' be the intersection of P with the initial states of the system and the automaton, projected on the first two elements. Note that a pair $\langle l, x \rangle$ is in P' if disallowing l to change violates the specification obtained by replacing the subformula associated with x with $\bot$. Thus, the set of latches defined by $(\forall x) P'(l, x)$ contains latches that are not covered. Now, toggle coverage is similar, only that we allow the value of an output variable to be changed only once. For that, we need an additional new Boolean variable $flag$ that indicates whether the variable has already changed its value once. That is, the state space of the augmented product consists of 5-tuples $\langle o, flag, x, u, s \rangle \in O \times \{0, 1\} \times cl(\varphi) \times 2^X \times S$. When $flag = 0$ and the value of o changes, then $flag$ is assigned 1. Then, the augmented product continues as in the case of latch coverage, with the value of o unchanging.

FSM coverage State and transition coverage can be computed using the techniques of mutation-based metrics. We now describe the computation of path coverage. We start with mutations that omit all behaviors that contain a given finite path $\pi = w_1, \ldots, w_c$. Let M_π be a *monitor* that filters away paths that contain π as a sub-path. That is, M_π is a fair FSM that accepts paths ρ such that π is not a sub-path of ρ. Since M_π only cares for the values of control variables that encode the states (and not, for example, for the values of output variables in these states), the set of input variables of M_π is the set of control variables X of F, and M_π does not have output variables. For a given path π, the mutant FSM $\tilde{F}_\pi$ is the product FSM $F \times M_\pi$, which contains only the computations of F that do not have π as a sub-path. Then, π is vacuity covered by φ if $\tilde{F}_\pi$ satisfies φ vacuously. For a set of paths $\{\pi_1, \ldots, \pi_k\}$, we can compute the set of covered paths symbolically using the techniques as described above for vacuity coverage. The state space of the augmented product consists of 5-tuples $\langle mut, x, u, r, s \rangle$, where mut encodes the path $\pi_i \in \{\pi_1, \ldots, \pi_k\}$, x encodes the subformula of φ that we replace with $\bot$, and u, r, and s are the components of the product $F \times M_{\pi_{mut}}$ with $\mathcal{A}_{\neg\varphi[\psi \leftarrow \bot]}$. The successors of $\langle mut, x, u, r, s \rangle$ are the 5-tuples $\langle mut, x, u', r', s' \rangle$ such that $\langle u', r', s' \rangle$ is a possible successor of $\langle u, r, s \rangle$ in a product between the fair FSM $\tilde{F}_{\pi_{mut}}$, which is $F \times M_{\pi_{mut}}$ and the automaton $\mathcal{A}_{\neg\varphi[\psi \leftarrow \bot]}$, where ψ is the subformula encoded by x. The vacuity uncovered paths are these encoded by values of mut such that for all the assignments x, there are initial states u_0, r_0, and s_0 of the FSM and the automaton, respectively, such that there is a fair path from $\langle x, mut, u_0, r_0, s_0 \rangle$. As above, the set of 5-tuples P from which a fair path exists in the augmented product can be found symbolically, and the set of vacuity covered paths can be computed from it.

In a similar way we can define mutations of paths that replace a finite path π with a path $\tilde{\pi}$ of the same length, redirecting the system to another execution. If a mutated path is of length 1, the mutation redirects one transition. For a path π replaced with a mutant path $\tilde{\pi}$, we use a monitor $M_{\pi,\tilde{\pi}}$. In the product $\tilde{F}_\pi$ of F with $M_{\pi,\tilde{\pi}}$, all the occurrences of π are replaced by $\tilde{\pi}$. Note that for mutated (rather than omitted) paths we can compute both falsity and vacuity coverage. The computation of falsity coverage is similar to falsity coverage of CFG described above, where the state space of the augmented product consists of 4-tuples $\langle mut, u, r, s\rangle$, with mut encoding the mutated path, and u, r, and s are the components of the product automaton of $\tilde{F}_\pi$ with $\mathcal{A}_{\neg\varphi}$. Vacuity coverage is computed similarly to the case of omission of paths.

Assertion coverage For an LTL assertion a, a *monitor* for a is the automaton $\mathcal{A}_{\neg a}$. Given assertions $a_1, \ldots, a_k$, the mutant FSM is the product $F \times \mathcal{A}_{\neg a_1} \times \ldots \times \mathcal{A}_{\neg a_k}$. Falsity and vacuity coverage of a set of assertions is computed similarly to FSM path coverage, where the variable mut encodes the assertion a_{mut} for $1 \leq mut \leq k$.

Hit count For a state $w \in 2^X$ of F, an output variable $q \in O$, and a specification φ that is satisfied in F, the q-hit count of w in F for φ is the minimal number of occurrences of w in the infinite tree obtained by unwinding F in which q has to be flipped in order to falsify φ in F. The computation of hit count may be hard in the general case. We restrict the computation of hit count to the following easier question: "given a *threshold* t, is the q-hit count of w in F for φ at least t?" The tree coverage computation algorithm of [CKKV01] can be modified in order to decide whether the hit count is below or above the threshold. We introduce a new variable *count* that counts the number of occurrences of w in which q is flipped. We now proceed as in tree coverage, except that whenever we visit a state w and we choose to flip q we also increment *count* until we hit the threshold t. When we compute the set of covered states, we return only these whose projection on *count* is greater than t. This can also be applied to other types of coverage.

6 Concluding Remarks

We showed how work done on coverage in simulation-based verification can be adapted to formal-verification. We also described symbolic algorithms to compute coverage in the formal-verification setting. Clearly, these algorithm incur costs beyond the costs already incurred by formal-verification algorithms. As in simulation-based verification, the use of our algorithms will be determined by practical verification methodologies developed in industry.

References

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

[AFG+02] R. Armoni, L. Fix, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, A. Tiemeyer, E. Singerman, M.Y. Vardi, and Y. Zbar. The ForSpec temporal language: A new temporal property-specification language. In *Proc. 8th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, Lecture Notes in Computer Science 2280, pages 296–311. Springer-Verlag, 2002.

[BA82] T.A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18:31–45, 1982.

[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.

[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.

[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.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysys (ISSTA)*, pages 123–133, Rome,Italy, 2002.

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

[Bud81] T.A. Budd. Mutation analysis: Ideas, examples, problems, and prospects. *Computer Program Testing*, pages 129–148, 1981.

[Cad03] Cadence. Assertion-based verification. http://www.cadence.com, 2003.

[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.

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

[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 *Computer Aided Verification, Proc. 13th International Conference*, 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 *Tools and algorithms for the construction and analysis of systems*, number 2031 in Lecture Notes in Computer Science, pages 528 – 542. Springer-Verlag, 2001.

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

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, 1978.

[FLLO95] R.S. French, M.S. Lam, J.R Levitt, and K. Olukotun. A general method for compiling event-driven simulations. In *Design Automation Conference*, pages 151–156, 1995.

[HH96] R.C. Ho and M.A. Horowitz. Validation coverage analysis for complex digital designs. In *International Conference on Computer Aided Design*, pages 146–151, November 1996.

[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.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[Hos95] Y.V. Hoskote. *Formal techniques for verification of synchronous sequential circuits*. PhD thesis, The University of Texas at Austin, 1995.

[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.

[KN96] M. Kantrowitz and L. Noack. I'm done simulating: Now what? verification coverage analysis and correctness checking of the dec chip 21164 alpha microprocessor. In *Proceedings of Design Automation Conference*, pages 325–330, June 1996.

[Kur97] R.P. Kurshan. Formal verification in a commercial setting. In *Proc. Conf. on Design Automation (DAC'97)*, volume 34, pages 258–262, 1997.

[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.

[Mar99] B. Marick. How to misuse code coverage. In *Proc. 16th International Conference on Testing Computer Software (TCS)*, June 1999.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.

[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.

[SA99] J. Shen and J.A. Abraham. An rtl abstraction technique for processor microarchitecture validation and test generation. *Journal of Electronic Testing*, 16(1-2):67–81, February 1999.

[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.

[UZ98] S. Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *Proceedings of International Conference on Software Testing, Analysis, and Review*, 1998.

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

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[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.