Attention-based Coverage Metrics *

Shoham Ben-David[1]**, Hana Chockler[2], and Orna Kupferman[3]

[1] David Cheriton School of Computer Science, University of Waterloo, Canada
[2] Department of Informatics, King's College, London, UK.
[3] School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel

Abstract. Over the last decade, extensive research has been conducted on coverage metrics for model checking. The most common coverage metrics are based on mutations, where one examines the effect of small modifications of the system on the satisfaction of the specification. While it is commonly accepted that mutation-based coverage provides adequate means for assessing the exhaustiveness of the model-checking procedure, the incorporation of coverage checks in industrial model checking tools is still very partial. One reason for this is the typically overwhelming number of non-covered mutations, which requires the user to somehow filter those that are most likely to point to real errors or overlooked behaviors.

We address this problem and propose to filter mutations according to the *attention* the designer has paid to the mutated components in the model. We formalize the attention intuition using a multi-valued setting, where the truth values of the signals in the model describe their level of importance. Non-covered mutations of signals of high importance are then more alarming than non-covered mutations of signals with low intention. Given that such "importance information" is usually not available in practice, we suggest two new coverage metrics that automatically approximate it. The idea behind both metrics is the observation that designers tend to modify the value of signals only when there is a reason to do so. We demonstrate the advantages of both metrics and describe algorithms for calculating them.

1 Introduction

Today's rapid development of complex hardware designs requires reliable verification methods. A major challenge in these methods is to make the verification process as exhaustive as possible. Exhaustiveness is crucial in simulation-based verification [5]. There, coverage metrics have been traditionally 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 [12, 22, 24]. During the last decade, there has been an extensive research

* This work is partially supported by the EC FP7 programme, PINCETTE 257647, and by the ERC (FP7/2007-2013) grant agreement QUALITY 278410.
** Shoham Ben-David is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

on coverage metrics for model checking. Such metrics are used for assessing the exhaustiveness of the specification, and information obtained from them is used in order to reveal behaviors of the system that are not referred to in the specification [19, 18, 10, 16, 21, 7].

The most common coverage metrics for model checking are based on mutations, where one examines the effect of small modifications of the system on the satisfaction of the specification. For example, *state-based* mutations flip the value of some (control or output) signal, and *logic-based* mutations fix the value of a signal to 0 or 1 [10, 21]. While there is an agreement that mutation-based coverage provides adequate means for assessing the exhaustiveness of the model-checking procedure, the incorporation of coverage checks in industrial-strength model-checking tools is still very partial. One possible reason for this is the fact that coverage checking requires model checking many mutations. As it turns out though, the fact the mutations are only slightly different from the original system enables a reuse of much of the information gathered during model checking and leads to coverage algorithms that do not incur a significant computational overhead on top of the model-checking procedure [9, 6, 7]. Another reason for the slow integration of coverage checks in practice, is the overwhelming number of non-covered mutations that current metrics involve [3], a problem reported also in the context of test-case generation using model checking [15]. Typically, a user gets a long list of mutations that are non-covered, and is expected to analyze them and filter out the non-interesting ones. When a significant portion of the non-covered mutations are false alarms, it may cause the user to disregard coverage information altogether, potentially causing real problems to be ignored.

We address this problem and propose to filter mutations according to the *attention* the designer has paid to the mutated components in the original model. We first formalize the intuition of attention using a multi-valued setting. In this setting, the truth values of the signals in the model are real numbers taken from the range $[-1, 1]$. The higher the absolute value of a signal is, the "more intentional" this value is. In particular, 1 stand for "very intentional true", -1 for "very intentional false", and 0 corresponds to "don't care". We consider specifications described by means of formulas in linear temporal logic (LTL). The semantics of LTL can be adjusted to the multi-valued setting, lifting the intention interpretation from the output signals to the whole specification [1]. Recall that in the traditional approach to coverage, we check coverage by flipping the value of a signal in a state, and checking whether the specification is satisfied in the new model. In the multi-valued setting, mutations reduce the absolute value of the truth value of a signal, and we check the effect of this on the truth value of the specification. Non-covered mutations of signals with high intention are then more alarming than non-covered mutations of signals with low intention.

While the multi-valued setting offers a very precise ranking of mutations, it requires the user to manually provide the intention information, which is a serious drawback. Accordingly, we suggest two new coverage metrics that automatically approximate the intention information. The idea behind both metrics is the observation that designers tend to modify the value of signals only when

there is a reason to do so. Thus, the value of a signal that has just been assigned is "more intentional" than the value of a signal that maintains its value. Before we turn to describe the new coverage metrics, let us point out that the above "lazy assignment" assumption, which is the key to our two metrics, is supported by power gating and clock gating considerations. Power consumption is an important consideration in modern chip design, from portable servers to large server farms. As the chips become more complex, the cost of powering a server farm can easily outweigh the cost of the servers themselves, thus design teams go to great lengths in order to reduce power consumption in their designs. Existing power saving techniques can be divided into electrical, such as using more efficient transistors, and logical, which attempt to introduce power-saving changes into designs without changing their logic. Logical power saving techniques attempt to reduce the number of changes in the values of signals, the main source of power consumption in chips. The most widely researched logical power saving techniques are *clock gating*, in which a clock is prevented from making a "tick" if it is redundant (c.f., [2]), and *power gating*, in which whole sections of the chip are powered off when not needed and then powered on again [20, 13]. The goal of these techniques is to make sure that a change in the value of a signal happens only when there is a good reason for it, that is, leaving the value of a signal unchanged would result in a different logic than intended by the designer.

Our first coverage metric is *stuttering coverage*, where mutations flip the value of a signal along a sequence of states in which the signal is fixed (rather than flipping the value in a single state). Consider, for example, the property "every request is eventually granted" and a chip design where "grant" signals, once raised, only fall when the current transaction terminates. In this design, "grant" will stay up for several consecutive cycles. Applying the traditional mutation-based coverage metrics results in all these states being identified as non-covered with respect to the property and the "grant" signal. On the other hand, considering mutations that flip the value of a signal in the whole block at once will filter out these blocks, resulting in fewer and more meaningful coverage results.

Our second metric is applied to *netlist mutations*. Such mutations set a signal in the netlist to a constant value or make it "free" to change nondeterministically in every cycle. Here, the goal is to define as interesting mutations of signals whose values have received a lot of attention of the designer. We associate attention with the frequency in which signals flip their value. Thus, our metric filters mutations of signals that are not often flipped. We formalize "often" by means of windows of a fixed length along the computation.

We discuss the advantages of both metrics and describe algorithms for calculating them. Our algorithms output a "pass" result if all mutations are covered. If a non-covered block or signal (in the first and the second metrics, respectively) is found, it is presented as a counterexample. Note that, in contrast to existing algorithms for computing coverage, our algorithms do not output all non-covered mutations at once. There are two advantages to this strategy. First, it allows us to construct algorithms with the same complexity as model checking

(essentially, we reduce coverage computation to model checking a property of almost the same size as the original one). Second, it mimics the real verification process, where bugs are found and corrected one by one. Since fixing a coverage hole results in a modification of either a property or a design, the rest of the coverage holes might become irrelevant, and the verification process should be re-executed.

2 Preliminaries

2.1 Linear temporal logic

We specify on-going behaviors of reactive systems using the linear temporal logic *LTL* [23]. Formulas of LTL are constructed from a set AP of atomic proposition using the usual Boolean operators and the temporal operators **X** ("next time"), **U** ("until"), **G** ("always"), and **F** ("eventually"). We define the semantics of LTL with respect to a *computation* $\pi = \sigma_0, \sigma_1, \sigma_2, \ldots$, where for every $j \geq 0$, we have that σ_j is a subset of AP, denoting the set of atomic propositions that hold in the j's position of π. We use $\pi \models \psi$ to indicate that an LTL formula ψ holds in the path π.

2.2 Circuits

We model reactive systems by *sequential circuits*. A sequential circuit (a *circuit*, for short) is a tuple $\mathcal{S} = \langle I, O, C, \theta, \rho, \delta \rangle$, where I is a set of input signals, O is a set of output signals, and C is a set of control signals that induce the state space 2^C. The sets I and C and the sets I and O are disjoint. Accordingly, $\theta \in 2^C$ is an initial state, $\rho : 2^C \times 2^I \to 2^C$ is a deterministic transition function, and $\delta : 2^C \to 2^O$ is an output function. Possibly $O \cap C \neq \emptyset$, in which case for all $x \in O \cap C$ and $t \in 2^C$, we have $x \in t$ iff $x \in \delta(t)$. Thus, $\delta(t)$ agrees with t on signals in C. We partition the signals in $O \cup C$ into three classes as follows. A signal $x \in O \setminus C$ is a *pure-output* signal. A signal $x \in C \setminus O$ is a *pure-control* signal. A signal $x \in C \cap O$ is a *visible-control* signal. While pure output signals have no control on the transitions of the system, a specification of the system can refer only to the values of the pure-output or the visible-control signals.

An input sequence $i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$ induces a run $s_0, s_1, s_2, \ldots$ of states of $\mathcal{S}$, where $s_0 = \theta$ and $s_{j+1} = \rho(s_j, i_j)$ for all $j \geq 0$. Recall that only signals in $I \cup O$ are visible, thus LTL formulas that specify $\mathcal{S}$ are over the set $AP = I \cup O$ of atomic propositions, and a *computation* of $\mathcal{S}$ is a sequence $\sigma_0, \sigma_1, \sigma_2, \ldots \in (2^{I \cup O})^\omega$, such that there is an input sequence $i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$, inducing the run $s_0, s_1, s_2, \ldots$, and $\sigma_j = i_j \cup \delta(s_j)$ for all $j \geq 0$.

2.3 Mutations in circuits

Let $\mathcal{S}$ be a circuit that satisfies a specification φ. We consider two types of mutations – *state-based* and *logic-based* – reflecting the possible ways in which a small change (mutation) can be introduced into $\mathcal{S}$ (see also [10, 21]).

State-based mutations. For a circuit $\mathcal{S} = \langle I, O, C, \theta, \rho, \delta \rangle$, a state $t \in 2^C$, and a signal $x \in C$, we define the *x-twin of t*, denoted $\mathit{twin}_x(t)$, as the state t' obtained from t by dualizing the value of x. Thus, $x \in t'$ iff $x \notin t$. A state-based mutation of x in t replaces t by $\mathit{twin}_x(t)$. The resulting mutant circuit is denoted by $\tilde{\mathcal{S}}_{t,x}$. The effect of this mutation for a pure-output signal x is changing the value of x in t. Mutations that dualize control signals introduce more aggresive changes. Indeed, dualizing a control signal x in a state s in $\mathcal{S}$ causes all transitions leading to t to be directed to its t-twin. In particular, the state t is no longer reachable in $\tilde{\mathcal{S}}_{t,x}$. Formally, given $\mathcal{S}$, s, and a signal $x \in O \cup C$, we define the *dual circuit* $\tilde{\mathcal{S}}_{s,x} = \langle I, O, C, \tilde{\theta}, \tilde{\rho}, \tilde{\delta} \rangle$ as follows.

- If x is a pure-output signal, then $\tilde{\theta} = \theta$, $\tilde{\rho} = \rho$, and $\tilde{\delta}$ is obtained from δ by dualizing the value of x in s, thus $x \in \tilde{\delta}(s)$ iff $x \notin \delta(s)$.
- If x is a pure-control signal, then $\tilde{\delta} = \delta$, and $\tilde{\theta}$ and $\tilde{\rho}$ are obtained by replacing all the occurrences of s in θ and in the range of ρ by $\mathit{twin}_x(s)$. Thus, if $\theta = s$, then $\tilde{\theta} = \mathit{twin}_x(s)$; otherwise, $\tilde{\theta} = \theta$. Also, for all $s' \in 2^C$ and $i \in 2^I$, if $\rho(s', i) = s$, then $\tilde{\rho}(s', i) = \mathit{twin}_x(s)$; otherwise, $\tilde{\rho}(s', i) = \rho(s', i)$.
- If x is a visible-control signal, then we do both changes. Thus, $\tilde{\delta}$ is obtained from δ by dualizing the value of x in s, and $\tilde{\theta}$ and $\tilde{\rho}$ are obtained by replacing all the occurrences of s in θ and in the range of ρ by $\mathit{twin}_x(s)$.

For a specification φ such that $\mathcal{S} \models \varphi$, a state t is *x-covered* by φ if $\tilde{\mathcal{S}}_{t,x}$ does not satisfy φ.

Note that it makes no sense to define coverage with respect to observable input signals. This is because an open system has no control on the values of the input signals, which just resolve the external nondeterminism of the system.

Logic-based mutations. These mutations describe changes resulting from freeing a control signal of $\mathcal{S}$ or fixing it to 0 or 1. Freeing a control signal is, from the design perspective, equivalent to turning this signal into an input signal. Fixing a signal to 0 or to 1 is known as "stuck-at-0" and "stuck-at-1" mutations, respectively, and these are the most commonly-used fault models in fault simulation and automatic test pattern generation (ATPG). For a circuit $\mathcal{S} = \langle I, O, C, \theta, \rho, \delta \rangle$ and a control signal $x \in C$, the mutant circuits are defined according to the type of the logic-based mutation applied to $\mathcal{S}$ as follows:

- The *x-freed circuit* $\mathcal{S}_x = \langle I', O, C', \theta', \rho', \delta' \rangle$ is obtained from $\mathcal{S}$ by moving x to the set of input signals (that is, $I' = I \cup \{x\}$, and $C' = C \setminus \{x\}$), and removing it from the definition of θ, from the range of ρ, and from the domain of δ.
- The *x-fixed-to-1 circuit* $\mathcal{S}_{x,1} = \langle I, O, C, \theta', \rho', \delta \rangle$ is obtained from $\mathcal{S}$ by replacing all the occurrences of x in θ and in the range of ρ by 1; i.e., $\theta' = \theta \cup \{x\}$, and for all $s \in 2^C$ and $i \in 2^I$, we have $\rho'(s, i) = \rho(s, i) \cup \{x\}$.
- The *x-fixed-to-0 circuit* $\mathcal{S}_{x,0}$ is defined by replacing all the occurrences of x in θ and in the range of ρ by 0.

A control signal x is *nondet-covered* if $\mathcal{S}_x$ does not satisfy φ, *1-covered* if $\mathcal{S}_{x,1}$ does not satisfy φ, and *0-covered* if $\mathcal{S}_{x,0}$ does not satisfy φ.

2.4 Mutations in netlists

Hardware designs are frequently represented as *netlists*. A netlist is a collection of primitive combinational elements. *And-Inverter* graphs (AIGs) are often used to store the netlist; i.e., the netlist consists of input gates, AND-gates, inverters, and memory elements (registers). Formally, a netlist N is a directed graph $\langle V_N, E_N, \tau_N \rangle$, where V_N is a finite set of vertices, $E_N \subseteq V_N \times V_N$ is a set of directed edges, and $\tau_N : V_N \to \{AND, INV, REG, INPUT\}$ maps a node to its type. The in and out degree of the vertices respects the expected requirements from the corresponding type.

When a design is modeled as a netlist, the smallest possible mutation is changing the type of a single node. We use the definition from [7], where the mutation changes the type of a single node in V_N to an input. This new input can be kept open, in which case its value is non-deterministically set at each cycle, or it can be fixed to 0 or 1.

Netlists can be naturally represented as a special case of sequential circuits, where the registers are viewed as control signals, that is, a state of the netlist is defined by a combination of values of the registers. Changing the type of a single node in a netlist can, therefore, be viewed as a logic-based mutation in the corresponding circuit.

3 Attention-Based Coverage

Ranking of coverage results according to the level of alarm they should cause could have been a much easier task if the designer of the verified system had provided information regarding his understanding of the importance of the different components of the system. In this section we develop a multi-valued approach for ranking of coverage results in case such an information is provided. We are aware of the fact that current modeling formalisms do not require the user to provide such an information. We still find it interesting and useful, both as further motivation to future modeling standards (especially given the tendency today to move to multi-valued approaches), and as a starting point for methods that approximate the multi-valued settings without information from the user, like those we suggest in Sections 4 and 5.

3.1 Multi-valued circuits

For our model of intention based coverage, we assume that the assignments to the pure-output variables are not Boolean. Rather, each output signal is assigned a real value in the range $[-1, 1]$, reflecting the level of importance that the designer gives to this assignment. Formally, in a multi-valued circuit $\mathcal{S} = \langle I, O, C, \theta, \rho, \delta \rangle$, the output function is $\delta : 2^C \to [-1, 1]^O$, which is not Boolean. Note that $C \cap O$ need not be empty, in which case we require, for all states $s \in 2^C$ and visible-control signals $x \in C \cap O$, that either $x \in s$ and $\delta(s)(x) \geq 0$ or $x \notin s$ and $\delta(s)(x) \leq 0$. Note that the values of signals in I are still Boolean. For uniformity,

we map them to $\{-1, 1\}$, in the expected way: a *computation* of the multi-valued circuit $\mathcal{S}$ is a sequence $\sigma_0, \sigma_1, \sigma_2, \ldots \in ([-1, 1]^{I \cup O})^\omega$, such that there is an input sequence $i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$, inducing the run $s_0, s_1, s_2, \ldots$, and for all $j \geq 0$, the assignment σ_j describes the values of the input and output signals in the j-th position in the run. Thus, for an input signal $x \in I$ we have that $\sigma_j(x)$ is 1 if $x \in i_j$ and is -1 if $x \notin i_j$, and for an output signal $x \in O$, we have $\sigma_j(x) = \delta(s_j)$.

Intuitively, the higher the absolute value of a signal is, the "more intentional" this value is. In particular, 1 stand for "very intentional true", -1 for "very intentional false", and 0 corresponds to "don't care". The semantics of LTL can be adjusted to the multi valued setting, lifting the intention interpretation from the output signals to the whole specification. We use $val(\pi, \varphi)$ to denote the value (in $[-1, 1]$) of an LTL formula φ in a computation $\pi = \sigma_0, \sigma_1, \ldots ([-1, 1]^{I \cup O})^\omega$. The value $val(\pi, \varphi)$ is defined by induction on the structure of φ as follows (c.f., [1]).

- $val(\pi, p) = \sigma_0(p)$,
- $val(\pi, \neg\varphi_1) = -val(\pi, \varphi_1)$,
- $val(\pi, \varphi_1 \wedge \varphi_2) = \min\{val(\pi, \varphi_1), val(\pi, \varphi_2)\}$,
- $val(\pi, \mathbf{X}\varphi_1) = val(\pi^1, \varphi_1)$,
- $val(\pi, (\varphi_1 \,\mathbf{U}\, \varphi_2)) = \max\{val(\pi, \varphi_2), \min\{val(\pi, \varphi_1), val(\pi^1, (\varphi_1 \,\mathbf{U}\, \varphi_2))\}\}$.

Note that, by De-Morgan rules, we have that $val(\pi, \varphi_1 \vee \varphi_2) = \max\{val(\pi, \varphi_1, val(\pi, \varphi_2)\}$, which matches our intuition. When φ is propositional, we sometimes use $val(s, \varphi)$ (rather than $val(\pi, \varphi)$), for the first state s of π.

Example 1. Consider the computation π in Figure 1 and the property $\varphi = G(p \rightarrow Fq)$. The signal q is "strongly false" in states s_0, s_1, and s_2, and is "strongly true" in the state s_3. Afterwards, the value of q reduces to $\frac{1}{2}$, indicating a "weaker true", possibly because the value is kept on only in order to reduce power consumption or as a back-up for the case that the q that is true in state s_3 would fail.

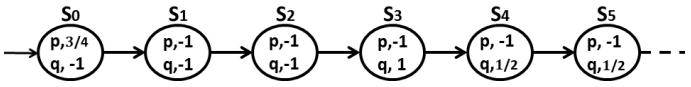

Fig. 1. A multi-valued computation.

We compute the value of φ on π. By the multi-valued semantics, we have

$$val(\pi, \varphi) = \min_{0 \leq i \leq 5} \{val(\pi^i, \neg p \vee Fq)\}.$$

Opening $val(\pi^i, \neg p \vee Fq)$, we get

$$val(\pi, \varphi) = \min_{0 \leq i \leq 5} \{\max_{i \leq j \leq 5} \{val(s_j, \neg p), val(s_j, q), \ldots, val(s_5, q)\}\}.$$

In all indices i that correspond to states in which p is strongly false we get that the maximum is 1 (since p is negated in s_j, its -1 value contributes 1). When $i = 0$, we have that $val(s_0, p) = \frac{3}{4}$. But since $val(s_3, q) = 1$, the maximal value in $\max\{val(s_0, \neg p), val(s_1, q),\ val(s_2, q),\ val(s_3, q),\ val(s_4, q), val(s_5, q)\}$ is still 1. Hence, the value of φ on π is 1.

Theorem 1. *The model-checking problem for multi-valued LTL has the same complexity as the model-checking problem for regular (Boolean) LTL.*

Proof Sketch. Essentially, the states of the automata constructed from the LTL specifications are now extended to be associated with functions from formulas in the closure of the specification to values in $[-1, 1]$, rather than with subsets of the formulas in the closure [1]. □

3.2 Multi-valued coverage

We now turn to the question of coverage in multi-valued circuits. Recall that in a regular (Boolean) model we check coverage by flipping the value of a signal in a state, and checking whether the specification is satisfied in the new model. In the multi-valued setting, mutations reduce the importance of a truth value of a signal, and check the effect of this on the truth value of the specification. Consider the example in Figure 1 again. If we reduce the truth value of q is state s_3 we expect to get a different coverage result from the case where the truth value of q is reduced in state s_4, since the value in s_3 is more important than that in s_4 to begin with.

We parameterize the coverage query by two values $v_1, v_2 \in (0, 1]$. The first value, v_1, describes the change in the truth value of the mutated signal. The second value describes the threshold for reporting non-coverage. That is, if after changing the truth value of the signal by v_1, the change in the truth value of the specification is less than v_2, then the signal is non-covered.

We now turn to formalize this intuition. For a state $s \in 2^C$ and an output signal $x \in O$ with $|\delta(s)(x)| \geq v_1$, we define the v_1-mutated value of x in s as the value obtained from $\delta(s)(x)$ by "bringing it closer to 0" by changing it by at most v_1. Note that since $|\delta(s)(x)| \geq v_1$, we do not have to worry about a signal switching its positivity in the definition below. Also note that the assumption about $|\delta(s)(x)|$ being at least v_1 matches the intuition behind multi-valued coverage, as it makes little sense to compute the effect of mutating signals that the designer does not care much about.

Formally,

$$val_{v_1}(s, x) = \begin{cases} \delta(s)(x) - v_1 & \text{if } \delta(s)(x) \geq 0. \\ \delta(s)(x) + v_1 & \text{if } \delta(s)(x) < 0. \end{cases}$$

For a circuit $\mathcal{S}$ we define $\tilde{\mathcal{S}}_{s,x,v_1}$ to be the circuit obtained from $\mathcal{S}$ by replacing $\delta(s)(x)$ with $val_{v_1}(s, x)$. It is not hard to prove that reducing the absolute value of a signal by v_1 can reduce the absolute value of the whole specification by at most v_1. Formally, for all circuit $\mathcal{S}$, states s, signals x, values v_1,

and specifications φ, we have that $0 \leq val(\mathcal{S}, \varphi)$ iff $0 \leq val(\tilde{\mathcal{S}}_{s,x,v_1}, \varphi)$ and $|val(\mathcal{S}, \varphi) - val(\tilde{\mathcal{S}}_{s,x,v_1}, \varphi)| \leq v_1$. Hence, checking (v_1, v_2)-coverage, we take $v_2 \leq v_1$.

We say that x is (v_1, v_2)-covered in a state s by a formula φ in the model $\mathcal{S}$ if $|val(\mathcal{S}, \varphi) - val(\tilde{\mathcal{S}}_{s,x,v_1}, \varphi)| \geq v_2$.

Example 2. Consider again the computation π appearing in Figure 1. Suppose we want to evaluate the coverage of the signal q in state s_3 with respect to the specification $\varphi = G(p \rightarrow Fq)$ and the parameters $(\frac{1}{2}, \frac{1}{4})$. The value of q in s_3 is reduced to $\frac{1}{2}$ and we get that $\max\{val(s_0, \neg p),\ val(s_1, q),\ val(s_2, q), val(s_3, q), val(s_4, q), val(s_5, q)\}$ is now $\frac{1}{2}$, while the other values in the external minimum remain 1, so the value of φ in the mutated computation is $\min\{\frac{1}{2}, 1, 1, 1, 1\} = \frac{1}{2}$. Recall from Example 1 that $val(\pi, \varphi) = 1$. Thus $|val(\pi, \varphi) - val(\tilde{\pi}_{s_4,q,\frac{1}{2}}, \varphi)| = 1 - \frac{1}{2} \geq \frac{1}{4}$. Accordingly, we get that q is $(\frac{1}{2}, \frac{1}{4})$-covered in s_3 with respect to φ, we do not report a hole here, which matches our intuition.
Let us now check the $(\frac{1}{2}, \frac{1}{4})$-coverage of q in s_4. It is easy to see that $val(\tilde{\pi}_{s_4,q,\frac{1}{2}}, \varphi)$ is still 1. Thus, $|val(\pi, \varphi) - val(\tilde{\pi}_{s_4,q,\frac{1}{2}}, \varphi)| = 0 < \frac{1}{4}$. Accordingly, we report that q is $(\frac{1}{2}, \frac{1}{4})$-non-covered in s_4, which matches our intuition.

4 Stuttering Coverage

In a circuit, a mutation $\widetilde{\mathcal{S}}_{s,q}$ switches the value of a visible control signal q in the state s of the circuit $\mathcal{S}$ (see Section 2.3). In many cases, such a mutation is too subtle, resulting in a spurious 'non-covered' result. To see this, consider the formula $\varphi = G(p \rightarrow Fq)$, and the execution path π shown in Fig. 2. Clearly, $\pi \models \varphi$. If we apply the standard mutation based coverage check, we shall flip the

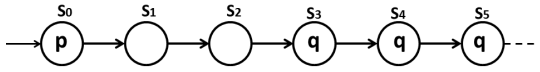

Fig. 2. An execution path in which existing metrics declare that q is not covered by $G(p \rightarrow Fq)$.

value of q in each of the states s_3, s_4 or s_5, and find that none of them is covered. It could be the case however, that q was left active simply because there was no reason to deactivate it, and therefore the 'non-covered' result that we report is a false alarm.

In this section we introduce *stuttering coverage*, which regards a block of states that agree on the value of a propositional formula as one unit, and flips their value together. In the case of Figure 2, the value of q will be flipped in all the states s_3, s_4, and s_5 together, causing φ to fail on the mutated path, and thus we get that q is covered in all states. Consider now a slightly modified example, where q holds in states s_3 and s_5, but not in s_4. In this case, the stuttering

coverage metric will indicate that both s_3 and s_5 are not covered, which seems a reasonable strategy, since the designer deliberately switched q off and on again.

Stuttering coverage is related to *statement coverage* metrics used in the context of code coverage [4, 10]. There, a mutation modifies or skips an assignment statement in the code. The effect of this in the corresponding circuit is a change in the block of states that starts with the execution of the statement and ends when the next assignment takes place. Unlike stuttering coverage, all blocks as above are affected. In contrast, stuttering coverage flips the value of q in a single block. Also, the boundaries of the block are determined by a propositional formula that may depend not only in q. Below we present the formal definition of stuttering coverage and suggest an algorithm to easily detect it.

4.1 Finding stuttering coverage holes

We examine mutations that flip the value of q in a sequence of states – a *block*. In the example, we define blocks as maximal sequences of states along which the mutated signal does not change its value. Here we generalize the setting to consider blocks defined by a predicate on the state space. For a Boolean assertion β over C, let $||\beta||$ denote the set of states that satisfy β. We are going to include in a block a sequence of states that are all satisfying β. That is, in stuttering coverage, we switch the value of q in β-blocks instead of in a single state. Note that typically there may be many β-blocks in a circuit, each suggesting a different mutation, and our metric considers them all.

Let $\mathcal{S} = \langle I, O, C, \theta, \rho, \delta \rangle$ be the circuit, q the signal to be flipped and β the Boolean expression. We construct a mutant circuit $\mathcal{S}' = \langle I', O', C', \theta', \rho', \delta \rangle$ that embodies all the mutations corresponding to a flip of q in a β-block. Essentially, as suggested in [8], we do so by nondeterministically guessing when a β-block starts. We now describe the details of the construction. We define $I' = I \cup \{x\}$ and $O' \cap C' = O \cap C \cup \{\mathit{start},\ \mathit{hold}\}$. The new input signal x is used to nondeterministically select a starting point for a β-block. The visible control signals *start* and *hold* are used to find the borders of the β-block: The signal *start* is initiated to *false* and is set to *true* by the transition function if x is *true* and β is *false*. Once *start* is set to *true* it stays *true* forever. Thus *start* uses the input x to "guess" that β is going to become active in the next state of the computation. Since *start* may be wrong in its guess, we use the signal *hold* to verify the guess and to indicate when β is no longer valid. The signal *hold* is initiated to *true*, and stays active as long as *start* is *false*, or, if *start* is *true*, it stays active until β is *false*.

Formally, $\theta' = \theta \cup \{\mathit{hold}\}$ and $\rho'(s, i) = \rho(s \cap C, i \cap I) \cup \gamma$, where γ is defined as follows:

$$\gamma = \begin{bmatrix} \{\mathit{start},\mathit{hold}\} & \text{If } (\mathit{start} \notin s, x \in i, \text{ and } s \notin ||\beta||) \text{ or} \\ & (\mathit{start} \in s, \mathit{hold} \in s, \text{ and } s \in ||\beta||). \\ \{\mathit{start}\} & \text{If } \mathit{hold} \notin s \text{ or } (\mathit{hold} \in s, \mathit{start} \in s \text{ and } s \notin ||\beta||). \\ \{\mathit{hold}\} & \text{If } \mathit{start} \notin s \text{ and } (x \notin i \text{ or } (x \in i \text{ and } s \in ||\beta||)). \\ \emptyset & \text{Otherwise.} \end{bmatrix}$$

Note that a state s is in the selected β-block iff *start*, *hold* and β are all active together in s. We denote such states by the predicate $\mathit{InBlock} = \mathit{start} \wedge \mathit{hold} \wedge \beta$. This construction however, cannot select a β-block if it begins in the initial state. This is because *start* is initialized to *false* and thus can be active only starting in the second state. This can be easily fixed by introducing a new initial state $\tilde{\theta} = \{\mathit{hold}\}$, with a single outgoing transition leading to the original initial state, and adding a leading X (next) before the formula. This way the formula will be checked starting from the original initial state of the model, and a β-block can be selected from the original initial state as well.

Note that in $\mathcal{S}'$ there are execution paths on which *InBlock* is never active: this happens when the input signal x never holds, or when *start* becomes active in wrong place and "misses" the beginning of a β-block.

In order to flip q in the selected β-block, we introduce a new observable signal $q' = q \oplus \mathit{InBlock}$. Note that q' holds the flipped value of q exactly on the selected β-block, and is equal to q in all other states. We define $\varphi' = \varphi[q \leftarrow q']$, replacing every occurrence of q in φ with q'. Thus, φ' "reads" the flipped value of q exactly on the selected β-block.

In order to search for a non-covered case, we look for a computation path on which φ' holds, but also *InBlock* is active at some point. Thus we search for a computation path on which $\varphi' \wedge F(\mathit{InBlock})$ holds. Such a computation path demonstrates a non-covered case of the original circuit $\mathcal{S}$.

The size of the $\mathcal{S}'$ is linear in the size of $\mathcal{S}$ and the Boolean expression β, size of the new property φ' is the same as the size of φ, and the algorithm performs model checking once, hence proving the following claim.

Claim. Finding a stuttering coverage hole is not harder than detecting a non-covered mutation as defined in Section 2.3, and is the same as model checking.

Remark 1. The logic LTL-X excludes the "next time" ($\mathbf{X}$) operator from LTL and is used for the specification of stutter-invariant properties [14]. Formulas in LTL-X are particularly suitable for stuttering coverage. Indeed, while the next-time operator can impose requirements on particular states in a computation (say, some valuation of signals should occur immediately after some event happens), stutter-invariants properties impose requirements on blocks. Even in the presence of the next-time operator, stuttering coverage has the advantage of reducing the number of mutations that needs to be checked.

Remark 2. Since stuttering coverage introduces larger changes in the circuit than the standard mutation-based coverage metrics described in Section 2.3, it may seem that stuttering coverage is strictly stronger than the standard coverage (in other words, if a state is stutter-covered, then it is covered according to the standard mutation-based metric). This is true for most properties, and, in particular, for properties used in the verification of real hardware designs, making this metric especially attractive in practice. However, this implication does not hold in general. One example is the properties using the $\mathbf{X}$ operator, as Remark 1 points out. Another example are properties that require that a particular signal holds its value for a large block of cycles (or for the duration of the whole design),

as in $p \rightarrow \mathbf{G}p$, which states that if a signal p holds in the initial state, then it should hold in the whole design.

5 Frequency-Based Coverage

We now consider logic-based mutations, typically modeling netlists [9]. Such a mutation takes a signal x and frees it or fixes it either to 0 or to 1. Here as well, coverage is reported when the specification holds on the mutated model.

In this metric we define as *important* signals that change a lot, assuming that a change in the signal's value is a result of an intentional action by the designer, whereas keeping the value constant whenever possible is the default behavior. We thus want to detect a signal that changes its value frequently, and yet, when mutated, does not influence the satisfaction of the specification.

We first have to formalize "frequently". There are different definitions that come to mind. We find the definition of *k-window*, specified below, to be most appropriate. It is possible to extend the idea here to other definitions. Let $\mathcal{S} = \langle I, O, C, \theta, \rho, \delta \rangle$ be the circuit modeling the netlist. For a control signal x, a computation π, and an integer $k \geq 1$, we say that x is *k-frequently flipped in* π if in each window of length k in π (that is, each subsequence of length k of assignments), the value of x is flipped at least once.

5.1 Finding frequency-based coverage holes

We are going to filter coverage results by frequency by defining a mutant circuit $\mathcal{S}'$ that keeps a log of flips of x in the last k transitions. , and enables the coverage check to restrict attention to computations in which x is flipped frequently. The circuit $\mathcal{S}'$ also applies the required mutation on x. The frequency check is, of course, with respect to the values of x before the mutation. Accordingly, $\mathcal{S}'$ keeps record of the original value of x in a new signal x'. In order to detect a change in the value of x', we also add a signal prev-x', recording the original value of x in the previous state.

We define the mutant circuit $\mathcal{S}' = \langle I, O', C', \theta', \rho', \delta \rangle$ as follows. First, we apply to x the desired mutation as specified in Section 2.3. We then add a set of control signals $V = \{x', \text{prev-}x', q_0, q_1, ..., q_k\}$. Thus, $C' = C \cup V$. The signals x' and prev-x' are described above. The signals $q_0, q_1, ..., q_k$ are used to count to k (note that as such, one could easily replace them by only $\lceil \log k \rceil$ signals. For simplicity, we describe the construction here with linearly many signals). Only q_k needs to be visible, thus $O' = O \cup \{q_k\}$.

The signal x' records the behavior of x in $\mathcal{S}$, namely, before the mutation was applied to it. The signal prev-x' records the value of x' in the previous state. Thus, we define $x' \in \theta'$ iff $x \in \theta$ and prev-$x' \in \theta'$ iff $x \notin \theta$. For all $s \in 2^{C'}$ and $i \in 2^I$, we set $\rho'(s, i) = \rho(s \cap C, i) \cup \{x'\}$ if $x \in \rho(s \cap C, i)$, and $\rho(s \cap C, i)$ otherwise. We set $\rho'(s, i) = \rho'(s, i) \cup \text{prev-}x'$ iff $x \in s$. A change in the value of x occurs in a state $s \in 2^{C'}$ if $s \models x' \oplus \text{prev-}x'$. That is, if $x' \in s$ and prev-$x' \notin s$ or $x' \notin s$ and prev-$x' \in s$.

In order to detect whether x is k-frequently flipped in the computations of $\mathcal{S}$, we record the behavior of x along k-windows. We do it using $q_0, ..., q_k$. For each state $s \in 2^{C'}$ we add exactly one of $q_0, ..., q_k$ as follows. We define $\theta' = \theta' \cup \{q_0\}$, and for all $s \in 2^{C'}$, $i \in 2^I$ and $0 \leq j < k$, we update ρ' as follows.

$$\rho'(s,i) = \begin{cases} \rho'(s,i) \cup \{q_k\} & \text{if } q_k \in s, \\ \rho'(s,i) \cup \{q_0\} & \text{if } s \models x' \oplus \text{prev-}x' \text{ and } q_k \notin s, \\ \rho'(s,i) \cup \{q_{j+1}\} & \text{if } s \not\models x' \oplus \text{prev-}x' \text{ and } q_j \in s. \end{cases}$$

It is easy to see that if x is k-frequently flipped in a computation, then q_0 would appear infinitely often on π. Otherwise, eventually a state with q_k would be reached, and from that point onwards q_k will appear in all states on π. Let ψ be the formula to be verified, and let $\mathcal{S}'$ be the mutated model as defined above. In order to check for coverage, we check for a computation satisfying $\psi' = \psi \wedge G\neg q_k$, asserting that ψ holds with the mutated behavior of x even though q_k is never reached. A path satisfying ψ' exhibit an interesting non-covered mutation.

6 A Case Study

We experimented with our ideas on a model of a PCI bus, taken from the NuSMV [11] example list. The model describes four master-slave units, communicating using the PCI bus protocol [17]. We briefly describe the protocol below, omitting details that are not essential for understanding our examples.

When a PCI master unit needs to start a transaction over the bus, it first asserts its request signal *req*, and keeps it asserted until permission is granted by the bus arbiter, indicated by the signal *gnt* being asserted. When permission is granted, the master can start a transaction by asserting its *frame* signal. We omit the details of the actual transaction over the PCI bus. A transaction terminates when *frame* is de-asserted, at which stage the bus is free for new transaction requests.

We note that in the formal PCI bus protocol, all signal are *active low*, meaning that they are considered active when their value is 0 and inactive when it is 1. In the PCI model we used, signals are *active high*, thus our example looks different than a typical PCI waveform.

The PCI model specifies more than 100 properties, which can roughly be divided into three categories. We examine each of the categories in light of the stuttering coverage method. The first are properties of the form

$$G((\neg req \wedge issue\_next) \rightarrow Xreq)$$

asserting that one event should be immediately followed by another event. As discussed in Remark 1, the advantage of stuttering coverage in formulas that impose requirements in specific states (in our example, those immediately after states with $\neg req$) is computational, and it does not change the coverage analysis.

The second type of properties have the form

$$G(req \rightarrow (req\ U\ grant)),$$

stating that once a signal becomes active, it should remain active until some other event occurs, similarly to the second type of properties discussed in Remark 2. Recall that coverage information is checked for specifications that hold in the system. Thus, checking the coverage of the signal req, we know that $G(req \rightarrow (req\ U\ grant))$ holds. When $\beta = req$, we flip the value of a full block of req. We distinguish between two cases: (1) We flip a block in which req is active. Then, the left hand side of the implication becomes false, and the formula continues to hold, thus req is not covered, which meets our intuition – we want the design to activate req only when required, thus the fact req is active high should be further challenged by other components of the specification. (2) We flip a block in which req is inactive. Here, the fact we flip the entire block puts the responsibility on the coverage on the signal $grant$, enabling the user to detect redundant activation of $grant$.

The third type of properties are eventual ones. For example,

$$\varphi = G((gnt \wedge \neg frame) \rightarrow F frame).$$

This specification states that if $frame$ is inactive and gnt is given, then a transaction must start eventually. We checked stuttering coverage of the signal $frame$ for the above specification with $\beta = frame$. That is, we switch $frame$ in blocks of consecutive states where $frame$ has value 1. As described in Section 4.1, this involves the introduction of the signal $block$, which is asserted during the selected β block, and the signal $frame'$, which agrees with $frame$ outside the selected block, and is the negation of $frame$ inside the block. We replaced the specification by

$$\varphi' = \neg(G((gnt \wedge \neg frame') \rightarrow F(frame')) \wedge F block).$$

The specification φ' failed, and Figure 3 presents the counterexample, which is an example of a non-covered block. In this example, three transactions take

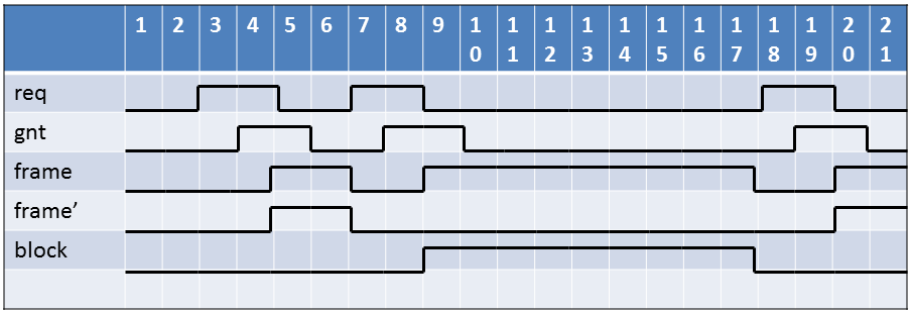

Fig. 3. A non-covered case for $G((gnt \wedge \neg frame) \rightarrow F(frame))$.

place. In cycles 5, $frame$ is asserted for a short transaction of 2 cycles. Then on cycle 9, $frame$ is asserted again for a longer transaction lasting until cycle 17.

Finally, a last transaction starts on cycle 20. The block of consecutive *frame*s selected for coverage check is the middle transaction, from cycle 9 to 17, as indicated by signal 'block' being asserted. Note that the signal *frame* is indeed not stutter-covered by φ. This is because many transactions take place on a typical execution path. Accordingly, a *gnt* is followed by many blocks of consecutive *frame*s, and eliminating one such block is not sufficient for causing φ to change its value. In order to cover the behavior of *frame*, a more detailed property should be introduced. Note further, however, that by using stuttering coverage we dramatically reduce the number of non-covered cases: in traditional "single state" coverage, each of the *frame*s in cycles 9 to 17 would be declared as non-covered.

7 Future Work

The algorithms we presented in this paper can be easily implemented on top of existing model checking tools – we need only to generate properties for detecting stuttering coverage and frequency-based coverage as described in Sections 4 and 5. As we already mentioned in the introduction, our algorithms generate one non-covered mutation at each run, hence mimicking the typical patterns of work of a verification engineer. Sometimes, however, we want to have a picture of how well our properties cover the design before we set up to fix coverage holes. In this context, a promising direction is to combine our definitions with the existing algorithms for efficient computation of coverage at once, for example those described in [7] and [6]. Based on our experience, the main obstacle in adoption of these algorithms as a part of the mainstream verification process is the sheer size of the output – the set of all non-covered mutations that need to be examined. We believe that using stuttering and frequency-based coverage will reduce the number of non-covered mutations by filtering the non-important mutations away, and we plan to perform these experiments as a future work. Finally, while the multi-value setting here comes mainly as a motivating framework to its approximation by stuttering and frequency-based coverage, we strongly believe that in the future we will see more and more quantitative specifications and systems, giving rise to quantitative verification methods, and making the multi-valued reasoning realistic in practice.

References

1. L. de Alfaro, M. Faella & M. Stoelinga (2004): *Linear and Branching Metrics for Quantitative Transition Systems.* In: *Proc. 31st ICALP*, pp. 97–109.
2. E. Arbel, O. Rokhlenko & K. Yorav (2009): *SAT-based synthesis of clock gating functions using 3-valued abstraction.* In: *Proc. 9th FMCAD*, pp. 198–204.
3. G. Auerbach, H. Chockler, S. Moran & V. Paruthi (2012): *Functional vs. Structural Verification – Case Study.* DAC User Track.
4. B. Beizer (1990): *Software Testing Techniques.* Van Nostrand Reinhold. 2nd edition.

5. L. Bening & H. Foster (2000): *Principles of verifiable RTL design – a functional coding style supporting verification processes.* Kluwer Academic Publishers.
6. H. Chockler, A. Ivrii, A. Matsliah, S. Moran & Z. Nevo (2011): *Incremental formal verification of hardware.* In: *Proc. 11th FMCAD*, pp. 135–143.
7. H. Chockler, D. Kroening & M. Purandare (2012): *Computing Mutation Coverage in Interpolation-Based Model Checking. IEEE Trans. on CAD of Integrated Circuits and Systems* 31(5), pp. 765–778.
8. H. Chockler, O. Kupferman, R.P. Kurshan & M.Y. Vardi (2001): *A Practical Approach to Coverage in Model Checking.* In: *Proc. 13th CAV*, *LNCS* 2102, pp. 66–78.
9. H. Chockler, O. Kupferman & M.Y. Vardi (2001): *Coverage Metrics for Temporal Logic Model Checking.* In: *Proc. 7th TACAS*, *LNCS* 2031, pp. 528–542.
10. H. Chockler, O. Kupferman & M.Y. Vardi (2006): *Coverage Metrics for Formal Verification. STTT* 8(4-5), pp. 373–386.
11. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani & A. Tacchella (2002): *NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking.* In: *Proc. 14th CAV*, *LNCS* 2404.
12. D.L. Dill (1998): *What's between simulation and formal verification?* In: *Proc. 35st DAC*, IEEE Computer Society, pp. 328–329.
13. C. Eisner, A. Nahir & K. Yorav (2009): *Functional verification of power gated designs by compositional reasoning. FMSD* 35(1), pp. 40–55.
14. K. Etessami (1999): *Stutter-Invariant Languages, ω-Automata, and Temporal Logic.* In: *Proc. 11th CAV*, *LNCS* 1633, pp. 236 – 248.
15. G. Fraser & F. Wotawa (2007): *Mutant Minimization for Model-Checker Based Test-Case Generation.* In: *TAIC PART – MUTATION*, pp. 161–168.
16. D. Große, U. Kühne & R. Drechsler (2008): *Analyzing Functional Coverage in Bounded Model Checking. IEEE Trans. on CAD of Integrated Circuits and Systems* 27(7), pp. 1305–1314.
17. PCI Special Interest Group (1998): *PCI Local Bus Specification*, 2.2 edition. Available at `http://www.ics.uci.edu/ harris/ics216/pci/PCI_22.pdf`.
18. Y. Hoskote, T. Kam, P.-H Ho & X. Zhao (1999): *Coverage estimation for symbolic model checking.* In: *Proc. 36st DAC*, pp. 300–305.
19. S. Katz, D. Geist & O. Grumberg (1999): *"Have I written enough properties ?" A method of comparison between specification and implementation.* In: *Proc. 10th CHARME*, *LNCS* 1703, pp. 280–297.
20. M. Keating, D. Flynn, R. Aitken, A. Gibbons & K. Shi (2007): *Low Power Methodology Manual.* Springer.
21. O. Kupferman, W. Li & S.A. Seshia (2008): *A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance.* In: *Proc. 8th FMCAD*, pp. 1–9.
22. D. Peled (2001): *Software Reliability Methods.* Springer.
23. A. Pnueli (1977): *The temporal logic of programs.* In: *Proc. 18th FOCS*, pp. 46–57.
24. S. Tasiran & K. Keutzer (2001): *Coverage Metrics for Functional Validation of Hardware Designs. IEEE Design and Test of Computers* 18(4), pp. 36–45.