Coverage of Implementations by Simulating Specifications

Hana Chockler and Orna Kupferman
School of Engineering and Computer Science
Hebrew University
Jerusalem 91904, Israel.
Email: {hanac,orna}@cs.huji.ac.il

December 21, 2001

Abstract

In formal verification, we verify that an implementation is correct with respect to a specification. When verification succeeds and the implementation 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 implementation. In this paper we study coverage for simulation-based formal verification, where both the implementation and the specification are modelled by labeled state-transition graphs, and an implementation $\mathcal{I}$ satisfies a specification $\mathcal{S}$ if $\mathcal{S}$ simulates $\mathcal{I}$. Our measure of coverage is based on small modifications we apply to $\mathcal{I}$. A part of $\mathcal{I}$ is covered by $\mathcal{S}$ if the mutant implementation in which this part is modified is no longer simulated by $\mathcal{S}$. Thus, "mutation coverage" tells us which parts of the implementation were actually essential for the success of the verification. We describe two algorithms for finding the parts of the implementation that are covered by $\mathcal{S}$. The first algorithm improves a naive algorithm that checks the mutant implementations one by one by exploiting the significant overlaps among the mutant implementations. The second algorithm is symbolic, and it improves a naive symbolic algorithm by reducing the number of variables in the OBDDs involved. In addition, we compare our coverage measure with other approaches for measuring coverage.

1 Introduction

In *formal verification* [CE81, QS81, LP85], we verify the correctness of a finite-state implementation with respect to a desired behavior by checking whether a labeled state-transition graph that models the implementation satisfies a specification of this behavior, expressed in terms of a temporal logic formula or a more abstract labeled state-transition graph. Beyond being fully-automatic, 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 implementation. Thus, together with a negative answer, the verifier returns some erroneous execution of the implementation. These counterexamples are very important and they can be essential in detecting subtle errors in complex designs [CGMZ95]. On the other hand, when the answer to the correctness query is positive, most verification tools terminate with no further information to the user. Since a positive answer means that the implementation is correct with respect to the specification, this seems like a reasonable policy. In the last few years, however, there has been growing awareness of the importance of suspecting the implementation of containing an error also in the case verification succeeds. The main justification of such suspects are possible errors in the modeling of the implementation or of the behavior, and possible incompleteness in the specification.

There are various ways to look for possible errors in the modeling of the implementation or the behavior. One direction is to detect *vacuous satisfaction* of the specification [BBER97, KV99], where cases like antecedent failure [BB94] make parts of the specification irrelevant to its satisfaction. For example, the specification $\varphi = AG(req \rightarrow AF grant)$ is vacuously satisfied in an implementation in which *req* is always **false**. A similar direction is to check the validity of the specification. Clearly, a specification that is valid or is vacuously satisfied 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 such a check is clear: an erroneous behavior of the implementation 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.

In simulation-based verification techniques, *coverage metrics* are used in order to reveal states that were not visited during the testing procedure (i.e., not "covered" by this procedure) [HMA95, HYHD95, DGK96, HH96, KN96, FDK98, MAH98, BH99, FAD99]. These metrics are a useful way of measuring progress of the verification process. However, the same intuition cannot be applied to formal verification because the process of formal verification may visit all states regardless of their essence to the success of the verification process. We can say that in testing, a state is "uncovered" if it is not essential to the success of the testing procedure. A similar idea can be applied to formal verification, where a state is defined as "uncovered" if its labeling is not essential to the success of the verification process. This approach was first suggested by Hoskote et al. [HKHZ99]. Low coverage can point to several problems. One possibility is that the specification is not complete enough to fully describe all the possible behaviors of the implementation. Then, the output of a coverage check is helpful in completing the specification. Another possibility is that the implementation contains redundancies. Then, the output of the coverage check is helpful in simplifying the implementation.

There are two different approaches to coverage in model checking, where the specification is

given as a temporal logic formula. One approach, introduced by Katz et al. [KGG99], states that a well-covered implementation should closely resemble the *reduced tableau* of its specification. Thus the coverage criteria of [KGG99] are based on the analysis of the differences between the implementation and the tableau of its specification[1]. A fully covered implementation, according to [KGG99], is *bisimilar* to the reduced tableau of its specification, that is, has the same set of behaviors as the specification. A deviation from bisimilarity indicates a problem in the implementation or in the specification. For example, a criterion *UnImplementedState* checks whether there exists a state in the reduced tableau that does not correspond to any state of the implementation. In this case, [KGG99] claims that the specification is not tight enough, or the implementation fails to implement an important behavior. We find the approach of [KGG99] too strict – we want specifications to be much more abstract than their implementations. In addition, it is computationally hard to compute the coverage criteria.

Another approach, introduced in [HKHZ99], is to check the influence of small changes in the implementation on the satisfaction of the specification. This approach is inspired by the definition of *mutation coverage* in simulation-based verification [Dil98]. For a given implementation, we can consider a set of *mutants*, each representing one small change in the original implementation. The specification *covers* a mutation in an implementation if it is not satisfied in the corresponding mutant. Formally, for an implementation $\mathcal{I}$, modeled as a labeled state-transition graph, a state w in $\mathcal{I}$, and an *observable signal* q, the *mutant implementation* $\tilde{\mathcal{I}}_{w,q}$ is obtained from $\mathcal{I}$ by flipping the value of q in w (the signal q corresponds to a Boolean variable that is **true** in w if w is labeled with q and is **false** otherwise; when we say that we flip the value of q, we mean that we switch the value of this variable). For a specification φ that is satisfied in $\mathcal{I}$ and an observable signal q, a state w of $\mathcal{I}$ is q-covered by φ if $\tilde{\mathcal{I}}_{w,q}$ does not satisfy φ. Indeed, this indicates that the value of q in w is crucial for the satisfaction of φ in $\mathcal{I}$. It is easy to see that for each observable signal, the set of q-covered states can be computed by a naive algorithm that performs model checking of φ in $\tilde{\mathcal{I}}_{w,q}$ for each state w of I. The naive algorithm, however, is very expensive, and is useless for practical applications [2].

The approach of [HKHZ99] is followed by [CKV01], where two alternatives to the naive algorithm are presented for specifications in the branching time temporal logic CTL. The first algorithm is symbolic and computes the set of pairs $\langle w, w' \rangle$ such that flipping the value of q in w' falsifies φ in w. The second algorithm improves the naive algorithm by exploiting overlaps in the many mutant implementations that we need to check. The "mutant approach" is also taken in [CKKV01], which studies coverage by specifications that are given as formulas in the linear temporal logic LTL or by automata on infinite words. [CKKV01] suggests alternative definitions of coverage, which suit better the linear case, and presents two algorithms for LTL and automata-based specifications. Both algorithms can be relatively easily implemented on top

[1] A tableau for a universal temporal logic formula is a labeled state-transition graph that embodies all the behaviors allowed by the formula. The reduced tableaux in [KGG99] are based on the *Particle tableaux* of [MP95]. In order to reduce the state space of the tableau further, [KGG99] requires that each state in the tableau refers only to the set of subformulas that are essential for the satisfaction of the specification. All other subformulas can have a "don't care" value, which allows to merge several different tableau states into one.

[2] Hoskote et al. describe an alternative algorithm that is symbolic and runs in linear time, but their algorithm handles specifications in a very restricted syntax (a fragment of the universal fragment $\forall$CTL of CTL) and it does not return the set of q-covered states, but a set that corresponds to a different definition of coverage, which is sometimes counter-intuitive. For example, the algorithm is syntax-dependent, thus, equivalent formulas may induce different coverage sets; in particular, the set of states q-covered by the tautology $q \rightarrow q$ is the set of states that satisfy q, rather than the empty set, which meets our intuition of coverage.

of existing model-checking tools.

In this paper we study coverage in design and verification methods in which the specification is given as a labeled state-transition graph. Consider an implementation and a specification. Both describe possible behaviors of the system, but the specification is more abstract than the implementation [AL91]. This approach, of representing both specifications and implementation as labeled state-transition graphs, suggests a top-down method for design development, called *hierarchical refinement* [LS84, Kur94]: starting with a highly abstract specification, we construct a sequence of behavior descriptions, each of which refers to its predecessor as a specification, and is thus less abstract than the predecessor. At each stage the current implementation is verified to satisfy its specification. Verifying that an intermediate implementation satisfies its specification leads to detection of errors in the design as soon as they are introduced. Likewise, measuring coverage of an intermediate implementation with respect to its specification would lead to early detection of low coverage.

There are several ways of defining what it means for an implementation $\mathcal{I}$ to satisfy a specification $\mathcal{S}$. The two main ones are *trace-based* and *tree-based*. The former requires each computation of $\mathcal{I}$ to correlate with some computation of $\mathcal{S}$, and the latter requires each computation tree embodied in $\mathcal{I}$ to correlate with some computation tree embodied in $\mathcal{S}$. The simplest definition of such correlation is equivalence with respect to the variables joint to $\mathcal{I}$ and $\mathcal{S}$, as the implementation is typically defined over a wider set of variables, reflecting the fact that it is more concrete than the specification. With this interpretation, trace-based verification corresponds to *trace containment* [Kur94], and tree-based verification corresponds to *simulation* [Mil71].

Simulation has several theoretically and practically appealing properties. First, since the definition of simulation is local, checking whether $\mathcal{S}$ simulates $\mathcal{I}$ can be done efficiently [Mil80, HHK95] and a witnessing relation for simulation can be computed symbolically [McM93, HHK95]. Second, simulation implies trace containment, whose checking for nondeterministic specifications is PSPACE-complete [MS72]. The computational advantage is so compelling as to make simulation useful also to researchers that favor the linear approach to specification: in automatic verification, simulation is widely used as a sufficient condition for trace containment [CPS93]; in manual verification, trace containment is most naturally proved by exhibiting local witnesses such as simulation relations or refinement mappings (a restricted form of simulation relations) [Lam83, LT87, Lyn96][3].

We apply mutation-based coverage to simulation and suggest efficient algorithms to measure coverage in simulation. As in [HKHZ99], for an implementation $\mathcal{I}$, a state w in $\mathcal{I}$, and an observable signal q, we say that w is *q-covered* by a specification $\mathcal{S}$ if $\tilde{\mathcal{I}}_{w,q}$ is not simulated by $\mathcal{S}$. Intuitively, w is q-covered by $\mathcal{S}$ if flipping the value of q in w creates a behavior that is not permitted by $\mathcal{S}$. As in the context of model checking, the naive algorithm computes coverage by executing a simulation computation algorithm $|W|$ times, once for each mutant implementation. We suggest two algorithms that improve the naive algorithm. Our algorithms are built on top of algorithms that compute the simulation relation. The first algorithm is built on top of the enumerative simulation algorithm of [HHK95]. The time complexity of the algorithm of [HHK95] is $O(m'n + mn')$, where m, n, and m', n' are the sizes of transition relations and state spaces of the implementation and specification. To the best of our knowledge, this is the

[3] In [AL91], it is shown that if auxiliary observable variables may be added to a implementation, then simulation is not only a sound proof technique but also complete for proving trace-containment.

best time complexity known for the problem[4]. Our algorithm exploits similarities between the mutant implementations, and has an average running time of $O((m'n + mn')\log n)$, while in the worst case its complexity does not exceed the complexity of the naive algorithm, which is $O((m'n+mn')n)$. The second algorithm is symbolic, and it computes, given an implementation $\mathcal{I}$ with state space W and a specification $\mathcal{S}$ with state space W', the following ternary relation.

$$\mathcal{C} = \{\langle w, v, w'\rangle : w, v \in W, w' \in W', \text{ and } w' \text{ simulates } w \text{ in } \tilde{\mathcal{I}}_{v,q}\}.$$

Thus, a triplet $\langle w, v, w'\rangle$ is in $\mathcal{C}$ iff w' simulates w in the mutant implementation $\tilde{\mathcal{I}}_{v,q}$, obtained from $\mathcal{I}$ by flipping the value of q in v. In particular, a state v is q-covered iff there exists an initial state w_0 of $\mathcal{I}$ such that for all initial states w'_0 of $\mathcal{S}$ we have $\langle w_0, v, w'_0\rangle$ not in $\mathcal{C}$, in which case $\mathcal{S}$ does not simulate $\tilde{\mathcal{I}}_{v,q}$. A naive implementation of Milner's fixed-point expression for simulation requires $2(n + n')$ OBDD variables [Mil80]. It has been recently shown in [KGG99, Kat01] how *early quantification* and *variable interleaving* in the OBDD can be used in order to reduce the number of required variables to $2n$. Similarly, a naive implementation of the fixed-point expression with which the relation $\mathcal{C}$ is computed requires $4n + 2n'$ OBDD variables. We show how early quantification and variable interleaving can be used also here, reducing the number of required variables to 3γ, where $\gamma = \max\{n, n'\}$.

Often, the designer is sufficiently familiar with the implementation and the specification to suspect that specific parts of the implementation are not covered by a specification. In such cases, it makes sense to replace the above described coverage algorithms by algorithms that get as input a set $MUT \subseteq W \times AP$ of mutations with respect to which coverage should be checked. A pair $\langle w, q\rangle \in MUT$ corresponds to the mutant implementation $\tilde{\mathcal{I}}_{w,q}$. Again, a naive algorithm checks the corresponding mutant implementations one by one, and is more complex than simulation in a factor of $|MUT|$. We show that our improved algorithms can be applied also in this case. The enumerative algorithm is more complex than simulation only in a factor of $\log|MUT|$, and the symbolic algorithm requires 3γ variables, with $\gamma = \max\{n, n', |MUT|\}$. In fact, the above described algorithms can be viewed as a special case where $MUT = W \times \{q\}$, for an observable signal q.

2 Preliminaries

We model systems by *labeled state transition graphs*. Formally, a system S is a tuple $S = \langle AP, W, R, W_0, L\rangle$, where AP is a set of atomic propositions, W is a set of states, $R \subseteq W \times W$ is a total transition relation, W_0 is a set of initial states, and $L : W \times AP \to \{\mathbf{true}, \mathbf{false}\}$ is a labeling function that maps a state w and an atomic proposition p to the value of p in w. We use $L(w)$ to denote the set $\{p : p \in AP \text{ and } L(w,p) = \mathbf{true}\}$. For a state w, we denote by $pre(w)$ the set of direct predecessors of w in the system, and by $post(w)$ the set of direct successors of w in the system. Formally, $pre(w) = \{v \in W : R(v, w)\}$, and $post(w) = \{v \in W : R(w, v)\}$.

Consider an implementation $\mathcal{I} = \langle AP, W, R, W_0, L\rangle$, and a specification $\mathcal{S} = \langle AP', W', R', W'_0, L'\rangle$. For technical convenience, we assume that $AP = AP'$; thus, the implementation and the specification are defined over the same set of atomic propositions[5]. A binary relation $B \subseteq W \times W'$

[4]The algorithm in [HHK95] is presented for the computation of the simulation relation in the same system, yet it can be easily adjusted to compute the simulation between an implementation and its specification.

[5]By replacing a set $L(w) \in 2^{AP}$ by the set $L(w) \cap AP'$, all our algorithms and results are valid also for the case $AP \supset AP'$.

is a *simulation* (of $\mathcal{I}$ by $\mathcal{S}$) if for all $\langle w, w' \rangle \in B$ the following conditions hold:

1. $L(w) = L'(w')$.
2. For each u such that $R(w, u)$ there exists u' such that $R'(w', u')$ and $B(u, u')$.

It is easy to see that the union of two simulations is a simulation. Consequently, the maximal simulation between $\mathcal{I}$ and $\mathcal{S}$, denoted $\mathcal{B}$, is the union of all simulations of $\mathcal{I}$ by $\mathcal{S}$. We say that $w' \in W'$ *simulates* $w \in W$ if $\langle w, w' \rangle \in \mathcal{B}$. We say that $\mathcal{S}$ *simulates* $\mathcal{I}$, or, equivalently, $\mathcal{I}$ *is simulated by* $\mathcal{S}$ (denoted $\mathcal{I} \leq \mathcal{S}$), if for every $w_0 \in W_0$ there exists $w'_0 \in W'_0$ that simulates w_0. Intuitively, it means that $\mathcal{S}$ has more behaviors than $\mathcal{I}$. In fact, every $\forall$CTL* formula that is satisfied in $\mathcal{S}$ is satisfied also in $\mathcal{I}$ [BCG88, GL94].

For the implementation $\mathcal{I}$, a state $w \in W$, and an atomic proposition $q \in AP$, the *mutant implementation* $\tilde{\mathcal{I}}_{w,q} = \langle AP, W, R, W_0, \tilde{L}_{w,q} \rangle$ is obtained from $\mathcal{I}$ by flipping the value of q in w. Formally, $\tilde{L}_{w,q}(w, q) = \neg L(w, q)$, and for all $\langle v, p \rangle \neq \langle w, q \rangle$, we have $\tilde{L}_{w,q}(v, p) = L(v, p)$. Consider a specification $\mathcal{S}$ such that $\mathcal{I} \leq \mathcal{S}$ and an atomic proposition $q \in AP$. We say that w is *q-covered* by $\mathcal{S}$ if $\tilde{\mathcal{I}}_{w,q} \not\leq \mathcal{S}$. Intuitively, w is q-covered by $\mathcal{S}$ if flipping the value of q in w creates a behavior that is not permitted by $\mathcal{S}$.

3 Enumerative Approach

In this section we describe an efficient algorithm for computing the set of q-covered states. Our algorithm is based on the enumerative algorithm of [HHK95] for simulation computation. Consider an implementation $\mathcal{I} = \langle AP, W, R, W_0, L \rangle$ and a specification $\mathcal{S} = \langle AP, W', R', W'_0, L' \rangle$ such that $\mathcal{I} \leq \mathcal{S}$. Given $q \in AP$, we compute the set of states of $\mathcal{I}$ that are q-covered by $\mathcal{S}$. When q is clear from the context, we omit it from our notations.

We first describe the algorithm of [HHK95]. The maximal simulation relation $\mathcal{B} \subseteq W \times W'$ is the set of all pairs $\langle w, w' \rangle \in W \times W'$ such that w' simulates w. That is, the labeling of w is equal to the labeling of w', and for each successor u of w there is a successor u' of w' such that u' simulates u. By the definition, $\mathcal{B}$ is the greatest fixed point of the equation

$$\mathcal{B} = \{\langle w, w' \rangle : \mathcal{B}_0(w, w') \wedge \forall u \, \exists u' \text{ such that } [R(w, u) \rightarrow R'(w', u') \wedge \mathcal{B}(u, u')]\}, \qquad (1)$$

where

$$\mathcal{B}_0 = \{\langle w.w' \rangle : L(w) = L'(w')\}.$$

An alternative way to compute the relation $\mathcal{B}$ is to compute, for each $w \in W$, the set $sim(w)$ of states of $\mathcal{S}$ that simulate w. The algorithm of [HHK95] starts with the maximal possible candidate for a simulation set for each w, namely, the set of all states in W' with the same labeling, and it it repeatedly reduces the sets until it reaches a fixed point: as long as there is a state w, a successor u of w, and a state $w' \in sim(w)$ such that there is no successor of w' in $sim(u)$, the set $sim(w)$ should be reduced by removing w'. A straightforward implementation of such a fixed-point calculation is presented in the procedure *Schematic_Similarity* in Figure 1 and has time complexity $O(mm'n^2n')$. Henzinger et al. improve this algorithm in the following way. For each $w \in W$, the algorithm maintains two sets of states: $oldsim(w)$ and $sim(w)$. The set $oldsim(w)$ is the candidate for the simulation set for w that was computed in the previous

iteration, and $sim(w)$ is the reduction of $oldsim(w)$ that is computed in this iteration. A fixed-point is reached when $sim(w) = oldsim(w)$ for all $w \in W$, thus no further reduction is possible. The reduction of $sim(w)$ is based on the same observation as in the straightforward algorithm: if a state w' is removed from the simulation set of w, then the predecessors of w' that have no other successors in the simulation set of w should be removed from the simulation sets of predecessors of w. An important observation that leads to the complexity of $O(m'n + mn')$ is that a state w' can be removed from the simulation set of w at most once during the algorithm. In addition, the algorithm uses a data structure that allows to compute the size of $post(v') \cap sim(u)$ for all $v' \in W'$ and $u \in W$ in constant time. The efficient version of the algorithm is described in the procedure *Efficient_Similarity* in Figure 1.

procedure *Schematic_Similarity*:
  for all $w \in W$ **do**
    $sim(w) = \{w' \in W' : L(w) = L'(w')\}$
  od;
  while there are $w, v \in W$, and $w' \in W'$ such that
      $v \in post(w)$, $w' \in sim(w)$, and $post(w') \cap sim(v) = \emptyset$ **do**
    $sim(w) = sim(w) \setminus \{w'\}$
  od;
  return *sim*.

procedure *Efficient_Similarity*:
  for all $w \in W$ **do**
    $oldsim(w) := W'$;
    $sim(w) := \{w' \in W' : L(w) = L'(w')\}$;
    $remove(w) := pre(oldsim(w)) \setminus pre(sim(w))$;
  od;
  while there exists $w \in W$ such that $remove(w) \neq \emptyset$ **do**
    for all $u \in pre(w)$ **do**
      for all $u' \in remove(w) \cap sim(u)$ **do**
        $sim(u) := sim(u) \setminus \{u'\}$;
        for all $v' \in pre(u')$ **do**
          if $post(v') \cap sim(u) = \emptyset$ **then** $remove(u) := remove(u) \cup \{v'\}$ **fi**
        od
      od
    od
    $oldsim(w) := sim(w)$;
    $remove(w) := \emptyset$;
  od;
  return *sim*.

Figure 1: The similarity algorithm of [HHK95].

The naive approach for coverage runs the algorithm of [HHK95] for $\tilde{\mathcal{I}}_{v,q}$, for all $v \in W$. The complexity of this is n times the complexity of the algorithm of [HHK95], which is $O((m'n + mn')n)$. A better approach is to use the fact that for all $w, v \in W$, the implementations $\tilde{\mathcal{I}}_{w,q}$

and $\tilde{\mathcal{I}}_{v,q}$ differ only slightly (that is, only in labeling of two states). Thus there is room for hope that the simulation computation for $\tilde{\mathcal{I}}_{w,q}$ and $\tilde{\mathcal{I}}_{v,q}$ is also almost the same. In order to explain our approach, we introduce the notion of *incomplete simulation*. Let X be the set of variables $\{x_w : w \in W\}$. For a subset of states $S \subseteq W$, the *incomplete labeling function* $L_S : W \times AP \to \{\mathbf{true}, \mathbf{false}\} \cup X$ maps a pair $\langle w, p \rangle$ to $L(w,p)$ if $w \notin S$ or $p \neq q$, and to x_w if $w \in S$ and $p = q$. As in the definition of L, we use $L_S(w)$ as a shortcut for the set $\{p : p \in AP$ and $L_S(w,p) = \mathbf{true}\}$. For two states $w \in W$ and $w' \in W'$, and a set $S \subseteq W$, we say that $L_S(w) = L'(w')$ if for every atomic proposition p, either $L_S(w,p) = L'(w',p)$, or $L_S(w,p)$ is a variable. For a set of states $S \subseteq W$ we define the implementation $\mathcal{I}_S = \langle AP, W, R, W_0, L_S \rangle$ as $\mathcal{I}$ with the incomplete labeling function L_S. Let $sim_S : W \to 2^{W'}$ denote the maximal simulation relation from $\mathcal{I}_S$ to $\mathcal{S}$. Also, for $w \in W$, let $\widetilde{sim}_w : W \to 2^{W'}$ denote the maximal simulation from $\tilde{\mathcal{I}}_{w,q}$ to $\mathcal{S}$.

Consider a state $w \in W$ and two sets $S_1 \subseteq S_2 \subseteq W$. It is easy to see that $sim_{S_1}(w) \subseteq sim_{S_2}(w)$. Indeed, the set $\{w' : w' \in W'$ and $L_{S_1}(w) = L'(w')\}$ is contained in the set $\{w' : w' \in W'$ and $L_{S_2}(w) = L'(w')\}$, and both simulation sets are computed from the above sets using the same monotonic fixed-point expression. In particular, when $S_1 = \emptyset$, we have that $sim(w) \subseteq sim_S(w)$ for all $w \in W$ and $S \subseteq W$.

Let $S_1 \subset S_2 \subseteq W$ be two sets of states of $\mathcal{I}$. Assume that we have computed sim_{S_2} and now we wish to compute sim_{S_1}. We claim that the computation of sim_{S_1} can be done using the algorithm of [HHK95] with the following modification: for each $w \in W$, we initialize the set $oldsim_{S_1}(w)$ to $sim_{S_2}(w)$ and the set $sim_{S_1}(w)$ to $sim_{S_2}(w) \cap \{w' \in W' : L'(w') = L_{S_1}(w)\}$. In other words, in the initialization of $oldsim_{S_1}(w)$ and $sim_{S_1}(w)$, we intersect the sets initialized in [HHK95] with the set $sim_{S_2}(w)$. Formally, consider the procedure *Efficient_Incomplete_Similarity* described in Figure 2. The procedure gets two parameters: $S \subseteq W$, and a simulation function $sim' : W \to 2^{W'}$. It differs from *Efficient_Similarity* only in the initialization stage: when S is not a singleton, the procedure computes the simulation relation by initializing sim and $oldsim$ with respect to L_S and sim'. When $S = \{w\}$, the procedure computes the simulation relation by initializing sim and $oldsim$ with respect to $\tilde{L}_{w,q}$ and sim'.

procedure *Efficient_Incomplete_Similarity*(S, sim'):
  for all $w \in W$ **do**
    $oldsim(w) := sim'(w)$;
    if $S \neq \{w\}$ **then** $sim(w) := \{w' \in W' : L_S(w) = L'(w')\} \cap sim'(w)$;
        else $sim(w) := \{w' \in W' : \tilde{L}_{w,q}(w) = L'(w')\} \cap sim'(w)$;
    $remove(w) := pre(oldsim(w)) \setminus pre(sim(w))$;
  od;
... % continues as in *Efficient_Similarity.*

Figure 2: Incomplete similarity algorithm

When $sim' = sim_{S'}$ for $S \subseteq S'$, the tighter initialization does not effect the correctness of the procedure. Formally, we have the following.

Lemma 3.1 *Let $S_1 \subseteq S_2$ be two subsets of W.*

- *If $|S_1| > 1$, then Efficient_Incomplete_Similarity(S_1, sim_{S_2}) returns sim_{S_1} (from $\mathcal{I}_{S_1}$ to $\mathcal{S}$).*

- *If $S_1 = \{v\}$, then Efficient_Incomplete_Similarity(S_1, sim_{S_2}) returns $\widetilde{sim}_v$ (from $\tilde{\mathcal{I}}_{v,q}$ to $\mathcal{S}$).*

Proof: Consider first the case where $|S_1| > 1$. Clearly, $sim_{S_1}(w) \subseteq sim_{S_2}(w)$ for all $w \in W$. Thus, *Efficient_Incomplete_Similarity* initializes $oldsim_{S_1}(w)$ to a set that contains $sim_{S_1}(w)$. In the same way, $sim_{S_1}(w) \subseteq \{w' \in W' : L'(w') = L_{S_1}(w)\} \cap sim_{S_2}(w)$, thus *Efficient_Incomplete_Similarity* initializes $sim_{S_1}(w)$ to a set that contains $sim_{S_1}(w)$. The algorithm then reduces the sets $oldsim_{S_1}(w)$ and $sim_{S_1}(w)$ until it reaches a fixed point. Since the reduction is independent of the order in which states are removed from $oldsim$, the same fixed point is reached.

Consider now the case where $S_1 = \{v\}$. Clearly, $\widetilde{sim}_v(w) \subseteq sim_{S_2}(w)$, for all $w \in W$. Thus, as in the previous case, *Efficient_Incomplete_Similarity* initializes $oldsim_{S_1}(w)$ and $sins_{S_1}(w)$ to sets that contain $\widetilde{sim}_v(w)$. Then, as in the previous case, this guarantees that a correct fixed-point is reached. □

We are now ready to describe our algorithm. The algorithm is based on a stepwise computation of the simulation relation sim from $\mathcal{I}$ to $\mathcal{S}$. In the first step, we compute incomplete simulation sim_W from $\mathcal{I}_W$ to $\mathcal{S}$. Note that sim_W refers to the labels of all the atomic propositions except q, and is very likely (a likelihood that increases for large sets of atomic propositions) to be much tighter than the initial candidate used in *Efficient_Similarity*. Consider a partition of W into two equal sets, W_1 and W_2. Our algorithm essentially works as follows. For all the mutant implementations $\tilde{\mathcal{I}}_{w,q}$ such that $w \in W_1$, the states in W_2 maintain their original labeling. Therefore, we start by computing incomplete simulation from $\mathcal{I}_{W_1}$ to $\mathcal{S}$; that is, we compute simulation with a labeling function that does not rely on the values of q in states in W_1. We end up with the function sim_{W_1}. Then, we continue and partition the set W_1 into two equal sets, W_{11} and W_{12}, and calculate incomplete simulation from $\mathcal{I}_{W_{11}}$ to $\mathcal{S}$. The important point is that we can start the computation of $sim_{W_{11}}$ from sim_{W_1}. Thus, we have to reduce the current candidate only with respect to information that involves the values of q in W_{12}. In a similar way, we compute incomplete simulation from $\mathcal{I}_{W_2}$ by $\mathcal{S}$, and then partition the set W_2 into two equal sets W_{21} and W_{22}, and compute incomplete simulation from $\mathcal{I}_{W_{21}}$ and $\mathcal{I}_{W_{22}}$ to $\mathcal{S}$. Here, we can start the computation of $sim_{W_{21}}$ and $sim_{W_{22}}$ from sim_{W_2}. Thus, as we go deeper in the recursion described above, we perform less work. The depth of the recursion is bounded by $\log|W|$. As we shall analyze exactly below, the work in depth i amounts in average to performing $1/2^i$ of the work required for computing the full simulation relation. Hence the $O((m'n + mn')\log n)$ complexity.

The coverage procedure is described in Figure 3. We call the procedure with the parameters $S = W$ and $sim' = sim_W$, where for all $w \in W$ we have $sim_W(w) = \{w' : L_W(w) = L'(w')\}$. The function bad that the procedure uses gets as input a simulation relation sim and returns **true** iff for all $w_0 \in W_0$, we have $sim(w_0) \cap W'_0 \neq \emptyset$. The procedure maintain a global variable COV in which the states of $\mathcal{I}$ that are q-covered are accumulated. Initially, $COV = \emptyset$. In step i of the recursion we have a partition of W to 2^i sets $W_1, W_2, \ldots, W_{2^i}$ of equal sizes $n/2^i$, and we perform incomplete simulation of $\mathcal{I}_{W_j}$ for $1 \leq j \leq 2^i$. Incomplete simulations in step i are computed using the values of incomplete simulations of the step $i-1$ as the initial values. Thus, the procedure *Efficient_Incomplete_Similarity* in step i receives the simulation sets computed with $n/2^{i-1}$ variables and computes the simulation sets with $n/2^i$ variables. In the last step, $i = \log n$, and we are left with one variable, which corresponds to the value of $L(w, q)$ for some state $w \in W$. Then, this value is flipped and the procedure *Efficient_Incomplete_Similarity*

computes the simulation of $\tilde{\mathcal{I}}_{w,q}$ by $\mathcal{S}$, using the simulation sets computed in the previous step. By Lemma 3.1, the functions $\widetilde{sim}_v$ computed by the algorithm are indeed such that for all $v, w \in W$, we have that $\widetilde{sim}_v(w) = \{w' : w' \text{ simulates } w \text{ in } \tilde{\mathcal{I}}_{v,q}\}$, thus our algorithm is correct.

procedure *Coverage* (S, sim')
  if $S = \{w\}$ **then**
    $\widetilde{sim}_w$ = *Efficient_Incomplete_Similarity* (S, sim');
    if *bad* (sim_w) **then** $COV := COV \cup \{w\}$ **fi**;
  else
    sim'' = *Efficient_Incomplete_Similarity* (S, sim');
    divide S **randomly to two equal sets** S_1 **and** S_2;
    Coverage(S_1, sim'');
    Coverage(S_2, sim'').

Figure 3: Coverage procedure

What is the complexity of our algorithm? The worst case for a state $w \in W$ is that the call to *Efficient_Incomplete_Similarity*$(\{w\}, sim')$ is executed with sim' for which $sim'(u) = W'$ for all u. Then, the calculation of $\widetilde{sim}_w$ coincides with its calculation by *Efficient_Incomplete_Similarity*. Thus, when the worst case applies to all $w \in W$, the overall complexity coincides with that of the naive algorithm (the work done before the call to *Efficient_Incomplete_Similarity*$(\{w\}, sim')$ is absorbed in the work done in the call itself) [6]. On the other hand, in the best case the simulation reaches its fixed point already in the first step, in which case the algorithm of [HHK95] is performed only once, which gives us a complexity of $O(m'n + mn')$.

Now that we have proved that the complexity of our algorithm is between $O(m'n + mn')$ and $O((m'n + mn')n)$, we analyze the average-case complexity. Let us look more closely at the complexity analysis of the algorithm of [HHK95]. In each iteration the algorithm reduces the set $sim(w)$, keeps the previous value in $oldsim(w)$, and removes all states $v' \in remove(w) = pre(oldsim(w)) \setminus pre(sim(w))$ from the sets $sim(u)$ for all $u \in pre(w)$. At the end of the iteration, the algorithm updates the set $oldsim(w)$ to $sim(w)$. Clearly, for all $w \in W$ and $j \geq 1$, the set $oldsim(w)$ in iteration j is a subset of the set $sim(w)$ in iteration $j-1$. Thus, a state v' can belong to $remove(w)$ in at most one iteration of the algorithm of [HHK95]. The complexity analysis of [HHK95] is based on this observation, and bounds, for each $w \in W$, the sum of sizes of the sets $remove(w)$ in all iterations by n'.

We claim that in our algorithm, a state v' also can belong to $remove(w)$ in at most one step. Indeed, for each $w \in W$ and for each $1 \leq i < \log n$, the initial value of the set $oldsim(w)$ in the step i is equal to the final value of $sim(w)$, computed in the step $i-1$, and the initial value of the set $sim(w)$ in each step is a subset of the initial value of the set $oldsim(w)$. Thus, if a test $v \in remove(w)$ is positive in step i, it is negative in all previous and in all subsequent steps. Thus, the sum of sizes of $remove(w)$ for all steps of the algorithm is bounded by n'. With each reduction in the size S, the set $sim_S(w)$ decreases. The reduction in $sim_S(w)$ is caused by

[6] Note that the worst case is very unlikely. Indeed, if the call to *Efficient_Incomplete_Similarity*$(\{w\}, sim')$ is executed with sim' for which $sim'(u) = W'$ for all u, then all states $u \neq w$ have the same label, which is also the label of all the states of the specification.

removing states u' that belong to $remove(w)$ from the simulation sets of the predecessors of w. In each step we handle a fraction of all states that belong to $remove(w)$ in one of the iterations of the algorithm. In the worst case, all these states are handled in the last step, while in the best case, they all are handled in the first step. In the average case, where average is taken over all random divisions of the set of variables S into two equal subsets, each replacement of a variable x_w by the value of $L(w, q)$ causes a removal of $1/n$ of the states that belong to $remove(w)$ for each $w \in W$. In step i we assign $1/2^i$ variables (replace the variables with the corresponding values of the labeling function), and thus in average in step i we handle the fraction $1/2^i$ of the states that belong to $remove(w)$. Recall that in step i we have 2^i copies of the sets $sim(w)$, so the complexity of the algorithm in this step has to be multiplied by 2^i. Thus, the overall complexity of the algorithm in the average case is

$$\sum_{i=1}^{\log n} 2^i (m'n + mn')1/2^i = O((m'n + mn') \log n).$$

Mutant vector As discussed in Section 1, often it is helpful to allow the designer to specify a set $MUT \subseteq W \times AP$ of mutations with respect to which coverage should be checked. Each pair $\langle w, q \rangle$ in MUT represents the mutant implementation $\tilde{\mathcal{I}}_{w,q}$. The algorithm above can be viewed as a special case where $MUT = W \times \{q\}$. It is easy to extend the algorithm to the more general case as follows. Given MUT, let $X_{MUT} = \{x_{w,q} : \langle w, q \rangle \in MUT\}$ be a set of variables that correspond to possible mutations. The incomplete labeling function L_{MUT} agrees with L for all $\langle w, q \rangle \notin MUT$ and is $x_{w,q}$ for $\langle w, q \rangle \in MUT$. In each step we randomly divide the set of variables into two equal subsets, and assign half of the variables their original values. Then we compute the incomplete simulation for this assignment. When the set of variables becomes a singleton $\{x_{w,q}\}$, we assign to $x_{w,q}$ the complementary value (that is, flip the value of q in w), and compute the simulation function from the mutant implementation $\tilde{\mathcal{I}}_{w,q}$ to $\mathcal{S}$. The number of steps in the algorithm is $O(\log |MUT|)$. By the same considerations detailed for the special case, this leads to an average time complexity of $O((m'n + mn') \log |MUT|)$.

4 Symbolic Approach

In this section we present an algorithm that symbolically computes the set of q-covered states. Note that the naive approach, which executes a symbolic algorithm $|W|$ times for all mutant implementations, is no longer symbolic, as it requires explicit enumeration of the state space. The algorithm we present in this section is symbolic, and it computes the relation $\mathcal{C}$ that is defined as follows.

$$\mathcal{C} = \{\langle w, v, w' \rangle : w, v \in W, w' \in W', \text{ and } w' \text{ simulates } w \text{ in } \tilde{\mathcal{I}}_{v,q}\}.$$

Then, v is q-covered by $\mathcal{S}$ if there is $w_0 \in W_0$ such that for all $w'_0 \in W'_0$, we have $\langle w_0, v, w'_0 \rangle \notin \mathcal{C}$. Several symbolic simulation algorithms are described in the literature [McM93, HHK95, KGG99]. We build our coverage algorithm on top of the straightforward symbolic implementation of Milner's fixed-point expression for simulation (see $\mathcal{B}$ of equation 1). The reason for this, as we elaborate below, is the small number of OBDD variables that are needed in this approach.

It is not hard to see that the relation $\mathcal{C}$ is the greatest fixed point of the following equation.

$$\mathcal{C} = \{\langle w, v, w' \rangle : \mathcal{C}_0(w, v, w') \wedge \forall u \; \exists u' \text{ such that } [R(w, u) \rightarrow R'(w', u') \wedge \mathcal{C}(u, v, u')]\},$$

where

$$\mathcal{C}_0 = \{\langle w, v, w' \rangle : w \neq v, \text{ and } L(w) = L'(w')\} \cup \{\langle w, w, w' \rangle : \tilde{L}_{w,q}(w) = L'(w')\}.$$

Thus, the calculation of $\mathcal{C}$ is very similar to that of $\mathcal{B}$, only that the state v affects the labels that are compared in $\mathcal{C}_0$. The straightforward symbolic implementation of the above fixed point involves OBDDs with $6n$ variables, where $n = \max(|W|, |W'|)$. We show how to reduce the number of OBDD variables to $3n$. In order to do so, we use *early quantification* and *variable interleaving* in the OBDDs. Both techniques are used in [KGG99] (see also [Kat01]) in order to reduce the number of OBDD variables required for computing the simulation relation $\mathcal{B}$ from $4n$ to $2n$.

We first explain the techniques in more detail. In early quantification, we try to push existential quantification inside in order to quantify out variables as soon as possible. Early quantification is traditionally used in *conjunctive partitioning* [CB98, Yan99] and is based on the property that sub-expressions can be moved out of the scope of an existential quantifier if they do not depend on any of the variables being quantified. In [KGG99], early quantification is used for computing simulation as follows. Recall that $\mathcal{B}$ is the greatest fixed point of the expression

$$\mathcal{B} = \{\langle w, w' \rangle : \mathcal{B}_0(w, w') \wedge \forall u \; \exists u' \text{ such that } [R(w, u) \rightarrow R'(w', u') \wedge \mathcal{B}(u, u')]\},$$

whose calculation involves OBDDs with $4n$ variables. By early quantification of u', we get

$$\mathcal{B} = \{\langle w, w' \rangle : \mathcal{B}_0(w, w') \wedge \forall u \; [R(w, u) \rightarrow \; \exists u' \text{ such that } R'(w', u') \wedge \mathcal{B}(u, u')]\}.$$

A naive implementation of the new fixed-point involves OBDDs with $3n$ variables. Indeed, the variables of u are introduced only after these of u' are quantified out. In order to reduce the number of variables further, Katz et al. order the variables in the OBDDs so that the variables of a binary relation $f(x, y)$ are interleaved: the variables of x are in the even levels of the OBDD for f and these of y are in the odd levels. Then, [KGG99] define two new operations on OBDD: *compose* and *compose_odd*. These operations compute "exist ... and" as one operation. Formally,

$$\textit{compose}(f(x, y), f(x, z)) = \exists x (f(x, y) \wedge g(x, z)),$$

and

$$\textit{compose_odd}(f(y, x), g(z, x)) = \exists x (f(y, x) \wedge g(z, x)).$$

When the variables of f and g interleaved as described above, the implementation of *compose* and *compose_odd* can proceed in levels, where the corresponding element of x is quantified simultaneously in the OBDDs of f and g. The resulting OBDD refers to the variables in y and z only, so we stay with an OBDD with $2n$ variables. For the detailed implementation of the operations see [Kat01].

Now $\mathcal{B}$ can be calculated using only $2n$ variables, by finding the greatest fixed point of the expression

$$\mathcal{B} = \{\langle w, w' \rangle : \mathcal{B}_0(w, w') \wedge \neg \textit{compose_odd}(R(w, u), \neg \textit{compose_odd}(R'(w', u'), \mathcal{B}(u, u')))\}.$$

Using the same ideas, we apply early quantification to the fixed-point expression for $\mathcal{C}$ and get

$$\mathcal{C} = \{\langle w, v, w'\rangle : \mathcal{C}_0(w, v, w') \wedge \forall u\ [R(w, u) \rightarrow \exists u' \text{ such that } R'(w', u') \wedge \mathcal{C}(u, v, u')]\}.$$

Then, we define two new operations on OBDDs, *(3:0)-compose* and *(3:2)-compose*, as follows.

$$\textit{(3:0)-compose}(f(x, y), g(x, z, u)) = \exists x(f(x, y) \wedge g(x, z, u)),$$

and

$$\textit{(3:2)-compose}(f(y, x), g(u, z, x)) = \exists x(f(y, x) \wedge g(u, z, x)).$$

The operations assume that the variables in the OBDDs of a ternary relation $g(x, z, u)$ are interleaved: the variables of x are in the 0 mod 3 levels, these of z are in the 1 mod 3 levels, and these of u are in the 2 mod 3 levels. Then, as in the case of binary relations, the existential quantification can be done in levels, with one pass on the OBDDs of f and g, and with only $3n$ variables.

Then, the relation $\mathcal{C}$ is the greatest fixed point of the expression

$$\mathcal{C} = \{\langle w, v, w'\rangle : \mathcal{C}_0(w, v, w') \wedge \neg\textit{(3:2)-compose}(R(w, u), \neg\textit{(3:2)-compose}(R'(w', u'), \mathcal{C}(u, v, u')))\},$$

which can be calculated with $3n$ variables.

Mutant vector Given a mutant vector $MUT \subseteq W \times AP$, the above algorithm can be adjusted to calculate symbolically which of the mutants are covered by the specification. Note that MUT can be given in some symbolic way (in particular, the above algorithm handles the case where $MUT = W \times \{q\}$ for some observable signal q), in which case it may be crucial to avoid an explicit enumeration of its members.

For a mutant $\lambda = \langle w, q\rangle \in MUT$, let $\tilde{L}_\lambda$ be the labeling function with q flipped in w, and let $\tilde{\mathcal{I}}_\lambda$ be the corresponding mutant implementation. We would like to calculate the relation

$$\mathcal{C} = \{\langle w, \lambda, w'\rangle : w \in W, \lambda \in MUT, w' \in W', \text{ and } w' \text{ simulates } w \text{ in } \tilde{\mathcal{I}}_\lambda\}.$$

The relation $\mathcal{C}$ is the greatest fixed point of the expression

$$\mathcal{C} = \{\langle w, \lambda, w'\rangle : \mathcal{C}_0(w, \lambda, w') \wedge \forall u\ \exists u' \text{ such that } [R(w, u) \rightarrow R'(w', u') \wedge \mathcal{C}(u, \lambda, u')]\},$$

where

$$\mathcal{C}_0 = \{\langle w, \lambda, w'\rangle : \lambda \in MUT \text{ and } \tilde{L}_\lambda(w) = L'(w')\}.$$

As in the algorithm above, we can rewrite $\mathcal{C}$ so that the quantification on u' is pushed inside. Unlike in the algorithm above, here the members of the triplets are not only states of $\mathcal{I}$ and $\mathcal{S}$ but also members of MUT. Accordingly, we define $n = \max\{|W|, |W'|, |MUT|\}$. Now, we can interleave the variables of w, λ, and w' as in the algorithm above, and calculate $\mathcal{C}$ as the greatest fixed point of the expression

$$\mathcal{C} = \{\langle w, \lambda, w'\rangle : \mathcal{C}_0(w, \lambda, w') \wedge \neg\textit{(3:2)-compose}(R(w, u), \neg\textit{(3:2)-compose}(R'(w', u'), \mathcal{C}(u, \lambda, u'))),$$

which can be calculated with $3n$ variables.

5 Discussion

We defined mutation-based coverage for implementations and specifications given by labeled state-transition graphs and described two algorithms for computing the set of states covered by the specification. The general idea of the algorithms is similar to the idea used in [CKV01] for mutation-based coverage in model-checking. The technical details, however, are different and nontrivial: in the enumerative algorithm, the overlaps between the mutant implementations lead to tighter candidates for simulation to start with[7]. In the symbolic approach, we addressed the problem of reducing the number of variables in the OBDDs involved, an issue that is not referred to in [CKV01], where a naive implementation requires only $2n$ variables. In addition, we show how the ideas in [CKV01] can be extended to handle a given vector of mutations.

Our work brings together the "mutant-based approach" of [HKHZ99] and the "simulation approach" of [KGG99]. As in [HKHZ99], coverage is measured with respect to mutant implementations. As in [KGG99], conformance to specification is checked by simulation. What is the relation between the two approaches? In order to answer this question, we first describe the approach of [KGG99] in more detail. In [KGG99], specifications are given in terms of $\forall$CTL safety formulas, and coverage is measured with respect to the reduced tableau of the specification. Katz et al. define four criteria according to which the implementation $\mathcal{I}$ is compared with the reduced tableau $\mathcal{S}$ of the specification. Each criterion corresponds to a set of states or transitions, essentially defined as follows[8].

1. *UnImplementedStartState*, which contains the set of states $w'_0 \in W'_0$ for which all $w_0 \in W_0$ have $w'_0 \notin sim(w_0)$. Thus, $\mathcal{S}$ simulates $\mathcal{I}$ even without having w'_0 as an initial state.

2. *UnImplementedState*, which contains the set of states $w' \in W'$ for which all $w \in W$ have $w' \notin sim(w)$. Thus, $\mathcal{S}$ simulates $\mathcal{I}$ even without the state w'.

3. *UnImplementedTransitions*, which contains the set of transitions $\langle w', u' \rangle \in R'$ for which $\mathcal{S}$ simulates $\mathcal{I}$ even without the transition $\langle w', u' \rangle$.

4. *ManyToOne*, which contains the set of states $w' \in W'$ for which $sim^{-1}(w')$ is not a singleton. Thus, multiple states of $\mathcal{I}$ are simulated by w'.

It is proved in [KGG99] that the four criteria are empty iff the implementation and the reduced tableau of the specification are bisimilar, in which case the implementation and the specification have exactly the same behaviors. How do the criteria of [KGG99] refer to our coverage measure? Do empty criteria in [KGG99] imply full coverage in our definition? Does full coverage in our definition imply empty criteria?

As we discuss below, the criteria of [KGG99] are orthogonal to our coverage measure. Consider the systems $\mathcal{I}_1$ and $\mathcal{S}_1$ described in Figure 4. The system $\mathcal{S}_1$ is the reduced tableau for the specification $AGq \vee AG\neg q$. Since $\mathcal{I}_1$ and $\mathcal{S}_1$ are identical, they are bisimilar, thus all the

[7] Alternatively, one could have followed the game-theoretic approach to simulation, show how incomplete simulation shrinks the game graph, and apply results from circuit complexity about shrinkage. This, less direct, approach would have been very similar to the approach taken in [CKV01].

[8] The criteria in [KGG99] refer to a restriction of the simulation relation to pairs $\langle w, w' \rangle$ such that w is reachable in $\mathcal{I}$ along a path that is simulated by a path along which w' is reachable in $\mathcal{S}$. The discussion and examples below apply also to this restriction.

criteria of [KGG99] are empty, and $\mathcal{I}_1$ is fully covered by $\mathcal{S}_1$. Is $\mathcal{I}_1$ indeed fully covered by $\mathcal{S}_1$? According to our definition of coverage, both states of $\mathcal{I}_1$ are not q-covered. Indeed, flipping the value of q in either states results in an implementation that still satisfies the specification.

Figure 4: Full coverage by [KGG99] does not imply full mutation coverage.

On the other hand, consider the systems $\mathcal{I}_2, \mathcal{S}_2, \mathcal{I}_3$, and $\mathcal{S}_3$ described in Figure 5. Atomic propositions whose value is not labeled (either positively or negatively) in the systems have a "don't care" value. The system $\mathcal{S}_2$ is the reduced tableau for the specification $(q \wedge (AXAGp \vee AXAGz)) \vee (r \wedge AXAGz) \vee AG\neg q$, and the system $\mathcal{S}_3$ is the reduced tableau for the specification AGq. According to our definition, $\mathcal{I}_2$ is fully covered by $\mathcal{S}_2$. Indeed, if we flip the value of each of the labeled atomic propositions, we no longer satisfy $\mathcal{S}_2$. On the other hand, criteria 1-3 of [KGG99] are not empty: the state s_4 is an unimplemented (start) state, and the transition $\langle s_0, s_3 \rangle$ is an unimplemented transition (and it connects states that are implemented). According to our definition, $\mathcal{I}_3$ is fully covered by $\mathcal{S}_3$. Indeed, if we flip the value of q in u_0 or u_1, we no longer satisfy $\mathcal{S}_3$. On the other hand, criteria 4 of [KGG99] is not empty: both states of $\mathcal{I}_3$ are simulated by the state t_0.

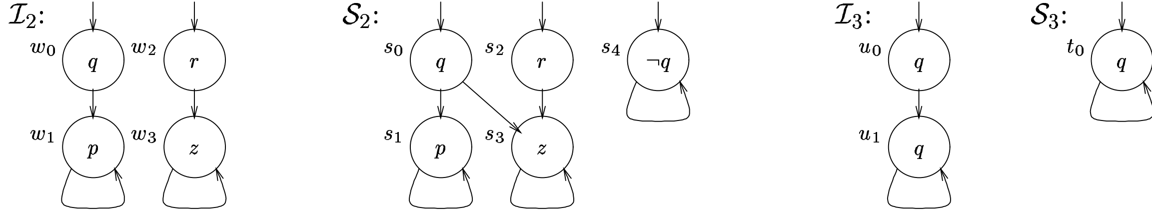

Figure 5: Full mutation coverage does not imply full coverage by [KGG99].

Intuitively, while the criteria of [KGG99] check that there is no redundancy in both the specification and the implementation, mutation coverage checks whether all the implementation details are essential. A point in favor of mutation coverage is the fact that it is *compositional*, in the sense that one could decompose the complete specification for the system to conjuncts and combine the coverage information measured for each conjunct. On the other hand, the approach in [KGG99] is not compositional. Finally, both approaches can be extended to handle fair labeled state-transitions graphs, but the extension would involve the need to check fair simulation [GL94, HKR97].

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

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

[BBER97] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc. 9th Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290, 1997.

[BCG88] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[BH99] J.P. Bergmann and M.A. Horowitz. Improving coverage analysis and test generation for large designs. In *IEEE International Conference for Computer-Aided Design*, pages 580–584, November 1999.

[CB98] Y. A. Chen and R. Bryant. Verification of floating point adders. In *Computer Aided Verification, Proc. 10th International Conference*, volume 1427 of *Lecture Notes in Computer Science*, pages 488–499. Springer-Verlag, 1998.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[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.

[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.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. on Programming Languages and Systems*, 15:36–72, 1993.

[DGK96] S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 418–425, November 1996.

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

[FAD99] F. Fallah, P. Ashar, and S. Devadas. Simulation vector generation from HDL descriptions for observability enhanced-statement coverage. In *Proceedings of the 36th Design Automation Conference*, pages 666–671, June 1999.

[FDK98] F. Fallah, S. Devadas, and K. Keutzer. OCCOM: efficient computation of observability-based code coverage metrics for functional simulation. In *Proceedings of the 35th Design Automation Conference*, pages 152–157, June 1998.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[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.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. 36th Symp. on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.

[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.

[HKR97] T.A. Henzinger, O. Kupferman, and S. Rajamani. Fair simulation. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 273–287, Warsaw, July 1997. Springer-Verlag.

[HMA95] Y. Hoskote, D. Moundanos, and J. Abraham. Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors. In *Proceedings of ICDD*, pages 532–537, October 1995.

[HYHD95] R. Ho, C. Yang, M. Horowitz, and D. Dill. Architecture validation for processors. In *Proceedings of the 22nd Annual Symp. on Computer Architecture*, pages 404–413, June 1995.

[Kat01] S. Katz. *Techniques for Increasing Coverage of Formal Verification.* PhD thesis, The Technion, 2001.

[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.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1994.

[KV99] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1999.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systenms*, 5:190–222, 1983.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LS84] S.S. Lam and A.U. Shankar. Protocol verification via projection. *IEEE Trans. on Software Engineering*, 10:325–342, 1984.

[LT87] N. A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[Lyn96] N.A. Lynch. *Distributed algorithms.* Morgan Kaufmann, 1996.

[MAH98] D. Moumdanos, J.A. Abraham, and Y.V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Trans. on Computers*, January 1998.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society, September 1971.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[MP95] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety.* Springer-Verlag, New York, 1995.

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Yan99] B. Yang. *Optimizing Model Checking Based on BDD Characterization.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.