Computing Mutation Coverage in Interpolation-based Model Checking

Hana Chockler, Daniel Kroening, and Mitra Purandare

***Abstract*—*Coverage* is a means to quantify the quality of a system specification, and is frequently applied to assess progress in system validation. Coverage is a standard measure in testing, but is very difficult to compute in the context of formal verification. We present efficient algorithms for identifying those parts of the system that are covered by a given property. Our algorithm is integrated into state-of-the-art SAT-based Model Checking using Craig interpolation. The key insight of our algorithm is the re-use of previously computed inductive invariants and counterexamples. This re-use permits a a rapid completion of the vast majority of tests, and enables the computation of a coverage measure with 96% accuracy with only 5x the runtime of the Model Checker.**

***Index Terms*—Model Checking, Coverage, Interpolation**

I. Introduction

Due to the ever-growing complexity of modern hardware and ever-decreasing time-to-market, functional verification has become a major bottleneck in producing correct hardware. *Model checking* is a functional verification technique that performs an exhaustive search of the state space of systems. Formal properties describing certain functionality of the system are written and a model checker proves whether a given system satisfies these properties.

If the model checker proves that all of the properties are satisfied by the system, it is guaranteed that the system behaves as specified in those properties. However, the behaviors of the system not specified in the properties are left unverified. It is rarely the case that properties are a full specification of the system and hence, an erroneous behavior of the system may escape the verification effort. Thus, there is a need to expose those features of the system not verified by a model checking run. This necessitates new properties to be written that specify these features.

In order to quantify the progress of the design validation phase, a metric for the exhaustiveness or *coverage* of a set of properties is highly desirable. The use of coverage metrics is commonplace in testing. As testing all conceivable executions of a design is usually infeasible, there is a need to assess the exhaustiveness of a given suite of test vectors [1]. There has been extensive research in the simulation-based verification community on coverage metrics, which provide a heuristic measure of exhaustiveness of the test suite [2]. In this context, coverage metrics answer the question "*Have I written enough tests?*".

The basic approach to coverage in testing is to record which parts of the design are exercised/activated during the execution of the test suite. Such a metric cannot be used in model checking, as model checking performs an exhaustive exploration of the state space of the system during which all parts of the design are exercised. In model checking, coverage metrics therefore serve a different purpose: they are used as an indicator for the completeness of the *specification*. A low-coverage specification could result in erroneous behavior that escapes the verification effort. Intuitively, coverage metrics in formal methods answer the question "*Have I written enough properties?*".

The earliest research on coverage in model checking suggested metrics based on *mutations*, which are small "atomic" changes in the design. A part of the design is said to be *covered* by a property if the original design satisfies the property, whereas mutating that part of the model renders the property invalid [3]–[9]. The straightforward way to measure mutation coverage is to run a model checker on each of the mutant designs, checking if the mutant design satisfies the property. Due to the sheer number of conceivable mutations, this approach is prohibitively expensive even on medium-size designs.

This paper focuses on alleviating the complexity of computing mutation coverage in interpolant-based model checking by applying the *incremental verification* approach: since the mutant designs differ from each other only very slightly, most of the work of the model checker can be reused. We present a novel algorithm that computes coverage of a given property by means of Craig interpolation, which is a state-of-the-art technique used in model checking [10]. The algorithm performs an analysis of the proof of unsatisfiability and the Craig interpolant generated during model checking. The proof and the interpolant provide hints about the parts of the system that play a role in satisfying the property. Then, the algorithm attempts to reuse the same proof and interpolant for the mutant designs. Mutations that are not covered by the property usually do not affect the proof and interpolant at all, making it possible to use the same proof and interpolant in the mutant design. For mutations that affect the proof of satisfaction of the property

Supported by the Semiconductor Research Corporation (SRC) under contract no. 2006-TJ-1539, the EU FP7 STREP MOGENTES (project ID 216679) and the EU FP7 STREP PINCETTE (project ID 257647).

H. Chockler is with IBM Research, Haifa, 31905, Israel.

D. Kroening is with the Computer Science Department, Oxford University, UK.

M. Purandare is with IBM Research - Zurich, Saeumerstrasse 4, 8803 Rueschlikon, Switzerland.

This paper is an extended version of our DAC 2010 publication "Coverage in Interpolation-based Model Checking". The journal version contains a detailed background section, further illustrative examples as well as additional experimental results on industrial designs.

for the design and yet are not covered by the property, in most cases, slight modifications to the proof suffice to make it valid for mutant designs, thus eliminating the need to model check mutants. We describe how these modifications are computed and introduced into the proof. We remark that in the worst case, when the proof and interpolant need to be recomputed, the complexity of our algorithm for a single mutant design is the same as that of model checking.

The algorithm is implemented and tested on a broad range of circuits. The experimental results demonstrate that the parts of the design covered by the property can be computed with reasonable cost in relation to the time taken by the model checking run. To the best of our knowledge, this is the first interpolant-based algorithm for computing coverage.

The rest of the paper is organized as follows. The necessary background on model checking appears in Section II. In Section III, we formally define coverage of circuits given as net-lists. In Section IV, we present our algorithm for computing coverage in an interpolant-based model checker. Details of the implementation of our algorithm and its experimental evaluation are presented in Section V. Related work appears in Section VI and Section VII presents our conclusions.

II. Background

Model checking [11], [12] is a technique for verifying a model of a system against a given property. Given a system M and its property ϕ, model checking systematically explores the state space of the system to decide if the system satisfies the property, i.e., if $M \models \phi$. In this paper, we consider hardware designs which are finite-state systems. A sample model is shown in Fig. 1. A property is typically expressed as a temporal logic formula, e.g., as Linear Temporal Logic (LTL) [13].

A. LTL Properties

Consider the Verilog model in Fig. 1. Some of the properties that it satisfies are:

- $count[0]$ *eventually* becomes 1;
- $count[1]$, $count[0]$ and $count[2]$ are *always* not 1 at the same time.

An LTL formulation of the first property is $(\mathbf{F}\ count[0] = 1)$, where $\mathbf{F}$ is the "eventuality" operator. An LTL formulation of the second property is $(\mathbf{G}\neg(count[0] \wedge count[1] \wedge count[2]))$, in which $\mathbf{G}$ is the "always" operator. Formal details on the syntax and semantics of LTL can be found in [13].

B. Net-lists

Hardware designs are frequently represented as net-lists. A net-list is a collection of primitive combinational elements. We use and-inverter graphs (AIGs) to store the netlist, i.e., the netlist consists of "and" gates, inverters and memory elements referred to as registers.

Definition 1: A net-list N is a directed graph (V_N, E_N, τ_N) where V_N is a finite set of vertices, $E_N \subseteq V_N \times V_N$ is the set of directed edges and $\tau_N : V_N \rightarrow \{\text{AND}, \text{INV}, \text{REG}, \text{INPUT}\}$ maps a node to its type, where AND is an "and" gate, INV is an inverter, REG is a register, and INPUT is a primary input. The in-degree of a vertex of type AND is at least two, of type INV and REG is exactly one and of type INPUT is zero. Any cycle in N must contain at least one REG node.

```
module counter(clk, count);
  input clk;
  output [2:0] count;
  reg [2:0] count;

  wire cin = ~count[0] & ~count[1] & ~count[2];

  initial count = 3'b0;

  always @ (posedge clk) begin
       count[0] <= cin;
       count[1] <= count[0];
       count[2] <= count[1];
  end
endmodule
```

Fig. 1. Verilog module of a counter

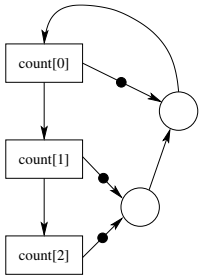

Fig. 2. Net-list of the counter in Fig. 1

As an example, consider the 3-bit counter whose Verilog module is shown in Fig. 1. The corresponding net-list is shown in Fig. 2. A rectangular node represents a REG. A circle-shaped node is an AND gate. An incoming edge of a node marked with a circle indicates negation.

A *state* of a net-list is a mapping of its registers to the Boolean values $\mathbb{B} = \{0,1\}$. A net-list N with r registers gives rise to a transition system $M = (S_N, T_N)$ where $S_N = \mathbb{B}^r$ is the set of states and T_N is the transition relation specifying what pairs of states are connected by transitions. The set S_0 of *initial states* is determined by the values of the registers immediately after reset. In the example above, $S_0 = \neg count[0] \wedge \neg count[1] \wedge \neg count[2]$. The state-transition diagram for the circuit is shown in Fig. 3. Note that S_N for the 3-bit counter consists of $2^3 = 8$ states. Unreachable states are not shown in Fig. 3. An algorithm for obtaining a transition relation for a net-list is given in [14].

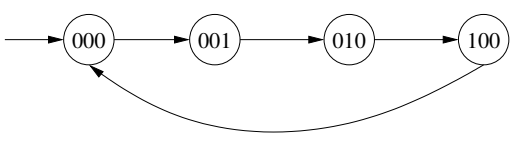

Fig. 3. State-transition diagram of the counter in Fig. 1

C. Finite-State Model Checking

Consider a transition system $M = (S,T)$ with a finite set of states S and transition relation $T \subseteq S \times S$. Let $S_0 \subseteq S$ and $F \subseteq S$ be the sets of initial and failure states, respectively. A system is correct if no state in F is reachable from any state in S_0. The image operator $img : \wp(S) \rightarrow \wp(S)$ maps a set of states to its successors:

$$img(Q) = \{s' \in S | s \in Q \text{ and } (s,s') \in T\}.$$

Let $img^0(Q) = Q$ and $img^{i+1}(Q) = img(img^i(Q))$. A set of states P is *inductive* if $img(P) \subseteq P$. The set P is an *inductive invariant* if P is inductive and $S_0 \subseteq P$. Given S_0 and F, the *strongest inductive invariant* R_{S_0} is the set of states reachable from S_0. If $R_{S_0} \cap F = \emptyset$, then F is not reachable from S_0.

Observe that $img(Q) = \exists s.Q(s) \wedge T(s,s')$. The computation of the precise image is expensive as it involves quantifier elimination.

Image Approximation: The cost of the image computation necessitates an approximate image operator. An over-approximate image operator $\hat{post} : \wp(S) \rightarrow \wp(S)$ satisfies $img(Q) \subseteq \hat{post}(Q)$ for all $Q \in \wp(S)$. An over-approximation of the set of reachable states is the set $\hat{R}_{S_0} = \bigcup_{i \geq 0} \hat{post}^i(S_0)$. Observe that if $\hat{R}_{S_0} \cap F = \emptyset$, then F is not reachable from S_0. Thus, it suffices to compute an over-approximation $\hat{R}_{S_0}$ to conclude correctness. However, if $\hat{R}_{S_0} \cap F \neq \emptyset$, it is not known if a state reachable from S_0 has a successor in F.

Finite sets and their relations can be encoded in propositional logic by using their *characteristic functions*. Instead of representing them as Binary Decision Diagrams (BDDs) [15] which may grow exponentially, propositional decision procedures (SAT solvers) can be used. Next, we briefly explain the essentials of propositional satisfiability.

D. Propositional Satisfiability and Resolution

The Boolean satisfiability problem (SAT) is a decision problem to determine if there exists a satisfying assignment for a given Boolean formula. Propositional SAT solvers [16] operate on Boolean expressions but do not use canonical forms. They do not suffer from the potential state space explosion of BDDs and can handle problems with thousands of variables. A number of efficient implementations are available [17]–[20].

Modern DPLL-style [21] solvers operate as follows: they perform decisions on values of variables followed by Boolean Constraint Propagation (BCP). The implications discovered by BCP are recorded in the form of an *implication graph*. The SAT solver either successfully assigns values to all variables or discovers a *conflict*. In the latter case, a conflict clause is generated by means of resolution. This conflict clause is added to the set of clauses to avoid repetition of the conflicting assignments. The solver then performs backtracking to undo some of the assignments. Eventually, the SAT solver may rule out all possible assignments and terminate with an unsatisfiable answer.

When the SAT-solver terminates with an unsatisfiable answer, the resolution steps can be used to produce a *proof of unsatisfiability*. These proofs can be formalized as follows. Let X be a set of propositional variables. Let $\neg x$ denote the negation of a variable x. A literal is either a variable or its negation. Hence, the set of literals over variables in X is $\{x, \neg x | x \in X\}$. For example, in the propositional formula

$$(\neg x_1 \wedge x_2) \vee \neg x_3 \vee (x_3 \wedge x_4) \quad (1)$$

the set of variables is $\{x_1, x_2, x_3, x_4\}$ and the set of literals is $\{\neg x_1, x_2, x_3, \neg x_3, x_4\}$.

A propositional formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of disjunctions of literals. For example, the formula

$$(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2) \wedge \neg x_3 \wedge \neg x_1 \quad (2)$$

is in CNF. A clause $\mathscr{C}$ is a disjunction of literals. In Formula 2, $x_1 \vee \neg x_2$, $x_3 \vee x_2$, $\neg x_3$, and $\neg x_1$ are clauses. The empty clause $\square$ contains no literals. There exist polynomial-time algorithms [22], [23] to transform an arbitrary propositional formula into an equisatisfiable propositional formula in CNF.

We now state the *resolution principle* which forms the basis of resolution proofs. The disjunction of two clauses $\mathscr{C}$ and $\mathscr{D}$ is their union, denoted as $\mathscr{C} \vee \mathscr{D}$, which is further simplified to $\mathscr{C} \vee x$ if $\mathscr{D}$ is the singleton $\{x\}$. The resolution principle states that an assignment satisfying the clauses $\mathscr{C} \vee x$ and $\mathscr{D} \vee \neg x$ also satisfies $\mathscr{C} \vee \mathscr{D}$. Let $\text{Res}(\mathscr{C}, \mathscr{D}, x)$ denote the *resolvent* of the clauses $\mathscr{C}$ and $\mathscr{D}$ with the pivot x. For example, the resolvent of $\mathscr{C} = (x_1 \vee \neg x_2)$ and $\mathscr{D} = (x_3 \vee x_2)$, denoted as $\text{Res}(\mathscr{C}, \mathscr{D}, x_2)$, is $(x_1 \vee x_3)$.

Definition 2: A *resolution proof* $\mathscr{P}$ is a directed acyclic graph $(V_\mathscr{P}, E_\mathscr{P}, piv_\mathscr{P}, \ell_\mathscr{P}, \mathtt{s}_\mathscr{P})$, where $V_\mathscr{P}$ is a set of vertices, $E_\mathscr{P}$ is a set of edges, $piv_\mathscr{P}$ is a pivot function, $\ell_\mathscr{P}$ is the clause function, and $\mathtt{s}_\mathscr{P} \in V_\mathscr{P}$ is the sink vertex. An *initial vertex* has in-degree 0. All other vertices are *internal* and have in-degree 2. The sink has out-degree 0. The pivot function maps internal vertices to variables. For an internal vertex v and $(v_1, v), (v_2, v) \in E_\mathscr{P}$, $\ell_\mathscr{P}(v) = \text{Res}(\ell_\mathscr{P}(v_1), \ell_\mathscr{P}(v_2), piv_\mathscr{P}(v))$.

A vertex v_1 in $\mathscr{P}$ is a *parent* of v_2 if $(v_1, v_2) \in E_\mathscr{P}$. Note that the value of $\ell_\mathscr{P}$ at internal vertices is determined by that of $\ell_\mathscr{P}$ at initial vertices and the pivot function. A proof $\mathscr{P}$ is a *resolution refutation* if $\ell_\mathscr{P}(\mathtt{s}_\mathscr{P}) = \square$. Henceforth, the words "proof" and "refutation" connote resolution proofs and resolution refutations.

The set of initial vertices of $\mathscr{P}$ is denoted as $Roots(\mathscr{P})$. An (A,B)-*refutation* $\mathscr{P}$ of an inconsistent CNF pair (A,B) is one in which $\ell_\mathscr{P}(v)$ is an element of A or B for each $v \in Roots(\mathscr{P})$. A subset of the clauses from A and B that appear in an (A,B)-refutation is said to form an *UNSAT core*.

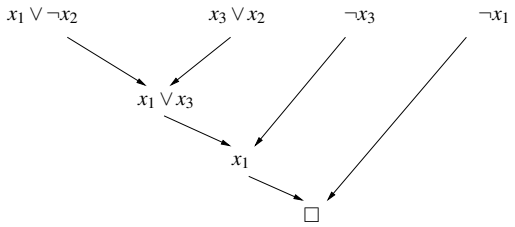

Fig. 4. Resolution proof showing unsatisfiability of Formula 2

Fig. 4 depicts a resolution refutation of Formula 2. The

labels of initial vertices are the clauses of Formula 2. The two intermediate vertices are labeled as

$$\text{Res}((x_1 \vee \neg x_2),(x_2 \vee x_3)) = x_1 \vee x_3 \text{ and}$$
$$\text{Res}((x_1,x_3),\neg x_3) = x_1.$$

The label of the sink vertex is $\ell_P(\mathtt{s}_{\mathscr{P}}) = \text{Res}(x_1,\neg x_1) = \square$. Hence, the proof in Fig. 4 is a resolution refutation.

Next, we discuss two state-of-the-art SAT-based model checking techniques.

E. Bounded Model Checking (BMC)

Bounded Model Checking (BMC) [24] leverages the success of fast propositional SAT solvers to model checking. The basic concept behind verifying a system M using BMC is to check if there exists a trace in M of a bounded length k that reaches a faulty state. Since LTL formulas are path formulas, finding a counterexample corresponds to checking whether there exists a trace in M that falsifies the formula. Hence, BMC is well-suited to finding counterexamples to LTL properties.

Consider a set of states Q, a transition relation T, a set of failure states F, and a constant $k \geq 1$. A BMC_k instance from Q with bound k checks if Q reaches F in k steps. The corresponding formula is $BMC_k \stackrel{\text{def}}{=} A(s_0,s_1) \wedge B(s_1,\ldots,s_k)$, where A and B are as follows.

$$\begin{array}{lcl} A(s_0,s_1) & \stackrel{\text{def}}{=} & Q(s_0) \wedge T(s_0,s_1) \\ B(s_1,\ldots,s_k) & \stackrel{\text{def}}{=} & T(s_1,s_2) \wedge \ldots \wedge T(s_{k-1},s_k) \wedge \\ & & (F(s_1) \vee \ldots \vee F(s_k)) \end{array} \quad (3)$$

If BMC_k is satisfiable, F is reachable from a state in Q in k steps. If BMC_k is unsatisfiable, F is not reachable from a state in Q in $\leq k$ steps. An instance of the BMC problem, denoted as $BMC_k(M,\phi)$, checks if $M \models_k \phi$ where $\models_k$ is the satisfaction relation from the initial states of M up to bounded depth k. Details on encoding the failure states for various LTL properties can be found in [24], [25].

F. Interpolant-based Model Checking

As stated earlier, the computation of the precise image $img(Q)$ is expensive, as it involves quantifier elimination. This necessitates an approximate image operator. The image $I(s_1)$ computed using an over-approximate image operator is such that $\exists s_0.A(s_0,s_1) \rightarrow I(s_1)$ is valid. An efficient procedure for computing the formula $I(s_1)$ provides an implementation of $\hat{post}$ applicable to compute $\hat{R}_{S_0}$. An interpolation system ITP is such a procedure.

Craig Interpolation: Interpolant-based Model Checking for circuits uses a special case of Craig's interpolation theorem for the case of propositional logic. Given a propositional formula β, let $\text{Var}(\beta)$ denote the set of propositional variables occurring in β.

Definition 3: An *interpolant* for a pair of inconsistent propositional formulas (A,B) is a propositional formula I such that

1) $A \rightarrow I$,
2) I and B are inconsistent, and
3) $\text{Var}(I) \subseteq \text{Var}(A) \cap \text{Var}(B)$.

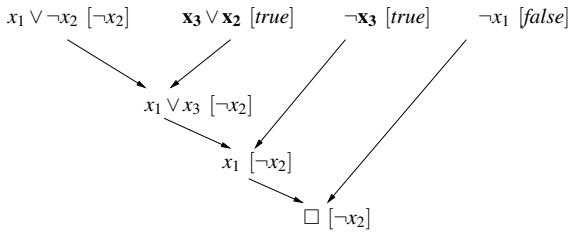

Fig. 5. Resolution proof showing unsatisfiability of Formula 2, annotated with $\text{ITP}_M(\mathscr{P})(v)$. The clauses in bold letters belong to B. The partial interpolants are shown in square brackets next to each vertex.

Example 1: Recall Formula 2, which is unsatisfiable. Let $A = (x_1 \vee \neg x_2) \wedge \neg x_1$ and $B = (x_3 \vee x_2) \wedge \neg x_3$. The interpolant for the unsatisfiable pair (A,B) is $\neg x_2$. Simplifying A and B, we have $A = \neg x_1 \wedge \neg x_2$ and $B = x_2 \wedge \neg x_3$. It is easy to see that $A \rightarrow \neg x_2$ is valid and $B \wedge \neg x_2$ is false.

Interpolants can be computed efficiently from resolution refutations. Different methods for computing interpolants from proofs exist [10], [26]–[28]. We very briefly discuss McMillan's interpolation system.

McMillan's Interpolation System: Let $\mathscr{P}$ be an (A,B)-refutation of an inconsistent CNF pair (A,B). Let $\text{ITP}_M(\mathscr{P},A,B)$ be a function that maps a vertex in $\mathscr{P}$ to a Boolean formula over the variables in $\text{Var}(A) \cap \text{Var}(B)$ such that $\text{ITP}_M(\mathscr{P},A,B)(\mathtt{s}_{\mathscr{P}})$ is an interpolant of (A,B). The formula $\text{ITP}_M(\mathscr{P},A,B)(v)$ is computed in the following manner: Let $g(v)$ denote the disjunction of literals in $\ell_{\mathscr{P}}(v)$ that appear in B.

For an initial vertex v,

$$\text{ITP}_M(\mathscr{P},A,B)(v) := \begin{cases} g(v) & : \text{ if } \ell_{\mathscr{P}}(v) \in A \\ \mathit{true} & : \text{ otherwise.} \end{cases}$$

For an internal vertex v with parents v_1 and v_2, $\text{ITP}_M(\mathscr{P},A,B)(v) := \text{ITP}_M(\mathscr{P},A,B)(v_1) \vee \text{ITP}_M(\mathscr{P},A,B)(v_2)$ if $piv_{\mathscr{P}}(v) \notin B$. Otherwise, $\text{ITP}_M(\mathscr{P},A,B)(v_1) \wedge \text{ITP}_M(\mathscr{P},A,B)(v_2)$. When A and B are clear from the context, we write $\text{ITP}_M(\mathscr{P})$ for $\text{ITP}_M(\mathscr{P},A,B)$.

Example 2: Consider Formula 2 with A and B given in Example 1. Fig. 5 shows the refutation proof $\mathscr{P}$ for unsatisfiable Formula 2 with $\text{ITP}_M(\mathscr{P})(v)$ for each vertex shown in square brackets next to $\ell_{\mathscr{P}}(v)$. The clauses from B are shown in bold. The interpolant for the unsatisfiable pair (A,B) is $\text{ITP}_M(\mathscr{P})(\mathtt{s}_{\mathscr{P}}) = \neg x_2$.

Interpolants as Approximation Operators: Interpolant-based model checking [10] is a method for computing an over-approximation $\hat{R}_{S_0}$ as discussed in Section II-F. An approximate operator $\hat{post}$ is implemented using a SAT solver that is able to generate a refutation and an interpolation system. An interpolation-based model checker is shown in Algorithm 1. If the pair $(A(s_0,s_1),B(s_1,\ldots,s_k))$ in Formula 3 is inconsistent (line 7), an interpolant $\text{ITP}(\mathscr{P},A,B)$ is an approximate image. Successive images (line 9) are computed by replacing Q in A by $Q(s_0) \vee \text{ITP}(\mathscr{P},A,B)(s_0)$ (line 13). When $\text{ITP}(\mathscr{P},A,B)(s_0) \subseteq Q(s_0)$, a fixed point is reached and $\hat{R}_{S_0} = Q(s_0)$ (line 11). If Formula 3 is satisfiable at any point

Algorithm 1 INTERPOLANT-MC(M,ϕ)

Input: Model M with initial states S_0, property ϕ
Output: A Yes/No answer to whether the faulty states F according to ϕ are reachable in M

1: Initialize k
2: **while** true **do**
3: $Q = S_0$
4: **if** $BMC_k(M,\phi)$ is SAT **then**
5: **return** Yes
6: **end if**
7: **while** $BMC_k(M,\phi)$ is UNSAT **do**
8: Let $\mathscr{P}$ be a proof of unsatisfiability of $BMC_k(M,\phi)$
9: Compute ITP$(\mathscr{P},A,B)$ where A and B are as shown in Formula 3.
10: **if** ITP$(\mathscr{P},A,B)(s_0) \rightarrow Q(s_0)$ **then**
11: **return** No
12: **end if**
13: $Q(s_0) = Q(s_0) \vee$ ITP$(\mathscr{P},A,B)(s_0)$
14: **end while**
15: Increase k
16: **end while**

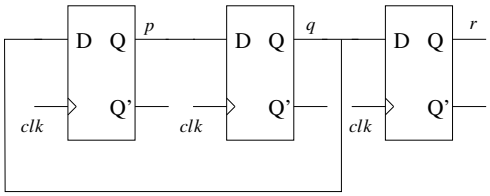

Fig. 6. A sequential circuit N satisfying $\phi_N = \mathsf{G}(p \vee q \vee r)$

in the successive image computation, the complete procedure is repeated with a higher k (line 15).

III. Formalizing Coverage

We restrict the presentation to models that are circuits given as net-lists. The mutations we consider depend on the representation of the design [7]. A mutation is any modification of the design. Mutations considered in mutation-based testing and coverage are chosen such that the semantic impact of the mutation is as small as possible. Consequently, mutations are typically restricted to a single change to the net-list: this prevents one mutation masking another. We begin with an illustrative example which explains the intuition behind the coverage notions adopted in this paper.

A. Illustrative Example

Consider the sequential circuit shown in Fig. 6. It has three outputs, namely, p, q, and r. The design has three registers; one corresponding to each output. The registers corresponding to the outputs p, q, and r are initialized with zero, one, and zero, respectively. Observe that the design satisfies the property $\mathsf{G}(p \vee q \vee r)$.

Since q is initialized to 1, the initial state of the circuit ($p = 0, q = 1$, and $r = 0$) satisfies the property. At the first positive clock edge, the values of the outputs change to $p = 1, q = 0$, and $r = 1$. At the second positive clock edge, the circuit returns to the initial state.

A careful analysis of the circuit reveals that the register r does not play any role in the property satisfaction. If the output of the register r is forced to zero, the property continues to be satisfied as either p or q is one. Forcing p to zero forces the outputs q and r to zero in the subsequent cycles, falsifying the property. Forcing q to zero has the same effect. Hence, the registers p and q are covered by the property and the register r is not.

We use this sequential circuit as a running example to explain the algorithm for computing coverage of a circuit.

B. Definition of Mutant Net-lists

When a design is modeled as a net-list, the smallest possible modification is changing the type of a single node. We begin by changing the type of a node to INPUT. This new input can be kept open or fixed to 0 or 1. Formally, the semantics of a mutant net-list is defined by means of a new labeling function τ_N^v, which replaces a node v by a new primary input:

$$\tau_N^v(u) := \begin{cases} \tau(u) & : \text{ if } u \neq v \\ \text{INPUT} & : \text{ otherwise.} \end{cases}$$

We say that τ_N^v *cuts* v from N. If a property satisfied by the original net-list fails on the mutant net-list, we say that the node is NONDET-COVERED. The new input v can also be held to zero or one.[1] These mutations are known as *stuck-at-0* and *stuck-at-1* mutation, respectively. Note that the stuck-at mutations are also the most commonly used fault models in fault simulation and automatic test pattern generation (ATPG). The corresponding coverage notions are called ZERO-COVERAGE and ONE-COVERAGE.

Lemma 1: Consider a net-list N and a property ϕ. If a node v in N is ZERO-COVERED or ONE-COVERED by ϕ, v is also NONDET-COVERED by ϕ.

Proof: Observe that the set of values a node is forced to at different time instances in a non-deterministic mutation is a superset of $\{0,0,0,\ldots\}$ (stuck-at-0) and $\{1,1,1,\ldots\}$ (stuck-at-1). If a stuck-at-0 (or stuck-at-1) mutation at a node results in a counterexample, the non-deterministic mutation also permits this counterexample. ■

Note that the reverse of Lemma 1 does not hold: a node NONDET-COVERED by ϕ need not be ZERO-COVERED or ONE-COVERED by ϕ.

We do not consider mutations that affect the initial state. Furthermore, we focus on mutations that change the value of a single node at a time, as there is no obvious definition of coverage for multiple mutations – the effect of changing the value of one node could be masked by the effect of changing the value of a second node.

The coverage of the design by a property ϕ is defined as the *percentage of mutant designs* out of all mutant designs that are covered by ϕ, that is, mutant designs on which ϕ fails

[1] The use of the proposed algorithms is not limited to the mutations above. Further mutations can be applied by tying input v to a more elaborate fault/mutation generator.

(assuming that ϕ passes on the original design). Coverage of a set of properties is the percentage of mutant designs covered by at least one property from the set.

A straightforward way to compute coverage is to mutate each node and run a model checker on each mutated system. This approach is prohibitively expensive even for small designs. In the next section, we propose a more efficient algorithm.

IV. Computing Coverage

This section explains the coverage computation algorithm (Algorithm 2). We use the sequential circuit from Section III-A as an illustrative running example. Let N denote the circuit and ϕ_{N} denote the property.

A. Overview

Algorithm 2 takes a net-list $N = (V_N, E_N, \tau_N)$ and a set of failure states F as inputs. It computes three maps nc, zc, and oc. These variables track the status of the nodes and map a node v to COVERED, NOTCOVERED, or UNKNOWN. If $nc[v] =$ COVERED, the node v is NONDET-COVERED. If $nc[v] =$ NOTCOVERED, the node v is not NONDET-COVERED. Otherwise, it is not yet known if v is NONDET-COVERED. Similarly, the maps zc and oc indicate whether a node is ZERO-COVERED and ONE-COVERED, respectively. Initially, these variables map all nodes to UNKNOWN.

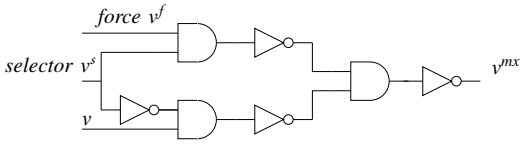

Fig. 7. Mutating v using a multiplexer, represented using AND gates and inverters

In order to force a vertex v in N to zero, one, or to make it non-deterministic, a multiplexer is introduced in N. The resulting AND-Inverter netlist is shown in Fig. 7. We further modify the edges of the net-list such that the tails of all the edges directed from v are changed to v^{mx}. Forcing the selector vertex v^s to one cuts a vertex v. In addition, fixing v^f to zero or one causes v^{mx} to be stuck-at-0 or stuck-at-1, respectively. A vertex v can be forced to a non-deterministic value by leaving v^f unconstrained. The vertices v^s and v^f are called the *selector* and *force* of the multiplexer, respectively.

We write $T_{N_{mx}}$ for the transition relation of this new net-list N_{mx}. We denote a formula that constrains the selectors to zero or one by *SEL*. The transition relation and the constraints together are denoted as $T_{N_{mx}^{SEL}} = T_{N_{mx}} \wedge SEL$. The resulting net-list is denoted as N_{mx}^{SEL} and the corresponding transition system is $(S_N, T_{N_{mx}^{SEL}})$. The following example illustrates how we insert multiplexers in N.

Example 3: Consider the sequential circuit described in Section III-A. We mutate this circuit by introducing the multiplexer shown in Fig. 7 at the inputs of all the three registers as shown in Fig. 8. The selectors of the multiplexers are named r^s, p^s, and q^s. The corresponding force vertices are r^f, p^f, and q^f, respectively. Observe that if r^s, q^s, and p^s are tied to zero, then the mutated circuit functions as it was designed originally. If p^s is tied to one, the output p^{mx} is equal to p^f, which can be held to zero, one, or a non-deterministic value. The same holds for the two remaining multiplexers.

The first step of the algorithm is to run an interpolant-based model checker on the model in which all selectors are set to zero, i.e., the model is equivalent to the original circuit. This selector constraint is denoted as SEL_0. Let v_t^s denote the selector variable of the multiplexer of vertex v in timeframe t. Thus, for a bound k, $SEL_0 = \bigwedge_{t=0}^{k} \bigwedge_{v \in V_N} \neg v_t^s$. The algorithm saves the final proof $\mathscr{P}$ and the inductive invariant $\hat{R}_{S_0}$ produced by the model checker. The algorithm proceeds

Algorithm 2 COVERAGECHECKS

Input: Net-list $N = (V_N, E_N, \tau_N)$ and failure states F
Output: nc, zc, and oc (coverage results)

1: **for all** $v \in V_N$ **do** $nc[v] := zc[v] := oc[v] :=$ UNKNOWN;
2: **end for**
3: Check $(S_N, T_{N_{mx}^{SEL_0}}) \models \neg F$ using Algorithm 1.
4: Let $\mathscr{P}$ be the final resolution proof, $\hat{R}_{S_0}$ the inductive invariant, k the final bound, m #(approx. image steps at bound k)
5: Let SEL_v represent the selector settings corresponding to mutating vertex v of N.
6: **for all** $v \in V_N$ **do**
7: **if** $\forall t. v_t^s$ are not present in $\mathscr{P}$ **then** ▷ CORE
8: Mark $nc[v]$, $zc[v]$, and $oc[v]$ as NOTCOVERED.
9: **end if**
10: **end for**
11: **for all** $v \in V_N$ **do** ▷ COUNTEREXAMPLE
12: **for** each $nc[v]$, $zc[v]$, and $oc[v] =$ UNKNOWN **do**
13: Mutate according to the selected mutation
14: **if** a CE of length $k+m$ exists **then**
15: Mark $nc[v]$, $oc[v]$ or $zc[v]$ as COVERED.
16: **end if**
17: **end for**
18: **end for**
19: **for all** $v \in V_N$ **do** ▷ INDUCTION
20: **for** each $nc[v]$, $zc[v]$, and $oc[v]$ = UNKNOWN **do**
21: Mutate according to the selected mutation
22: **if** $\hat{R}_{S_0}$ is an inductive invariant of the mutated system **then**
23: Mark $nc[v]$, $oc[v]$ or $zc[v]$ as NOTCOVERED.
24: **end if**
25: **end for**
26: **end for**
27: **for all** $v \in V_N$ **do** ▷ INTERPOLATION
28: **for** each $nc[v]$, $zc[v]$, and $oc[v] =$ UNKNOWN **do**
29: Run interpolant-based model checking on the new design
30: Mark $nc[v]$, $oc[v]$ or $zc[v]$ according to the outcome
31: **end for**
32: **end for**

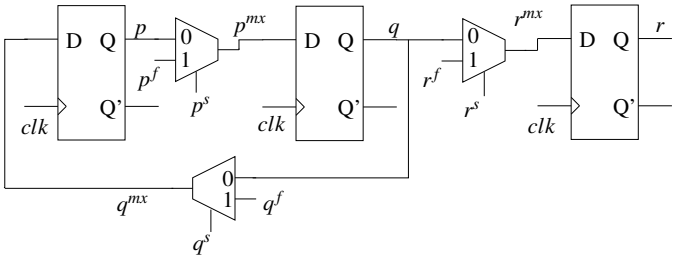

Fig. 8. The mutation N_{mx} of N in Fig. 6. Implementation of the multiplexer is shown in Fig 7.

Symbol	Meaning
N	net-list
S_N	state space of N
T_N	transition relation of N
N_{mx}	N with multiplexers introduced
N_{mx}^{SEL}	N_{mx} with selectors constrained by SEL
SEL_0	constraint forcing all selectors to zero
SEL_v	constraint for mutating a node v
v^{mx}, v^f, v^s	output, force, and selector of multiplexer at node v
M_o	$(S_N, T_{N_{mx}^{SEL_0}})$

TABLE I
IMPORTANT SYMBOLS RELATED TO NET-LISTS AND MUTATIONS

with four tests:

1) CORE: In the first test, nodes not mentioned in the proof are identified, and the corresponding nodes in the net-list are declared not covered.
2) COUNTEREXAMPLE: For each of the three types of mutations, a BMC run provides a counterexample which is used to identify some covered nodes in the net-list.
3) INDUCTION: We check if an inductive argument that a node in the net-list is not covered can be made, using the inductive invariant supplied by the model checker. There are two variants of this test. Initially, the test uses only those nodes mentioned in the proof, and not the full net-list. The full net-list is used only if the coverage of the node is undecided after the first variant.
4) INTERPOLATION: Full interpolant-based model checking is applied to all nodes that are still undecided.

We provide a summary of the most important symbols in Table I. These symbols are used extensively in the formalizations in the following subsections.

We now elaborate on the various tests used in Algorithm 2; Sec. IV-B describes the test that uses full interpolation, Sec. IV-C the unsatisfiable core test, Sec. IV-D the counterexample test, and Sec. IV-E two tests based on induction.

B. Coverage Analysis with Interpolants

The first step is to apply INTERPOLANT-MC (Algorithm 1) to $M_o = (S_N, T_{N_{mx}^{SEL_0}})$. The BMC-style unwinding for M_o with a bound k is shown below:

$$\begin{aligned} BMC_I &\stackrel{\text{def}}{=} Q(s_0) \wedge T_{N_{mx}}(s_0, s_1) \\ BMC_T &\stackrel{\text{def}}{=} \textstyle\bigwedge_{i=1}^{i=k-1} T_{N_{mx}}(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{i=k} F(s_i) \\ BMC_S &\stackrel{\text{def}}{=} SEL_0 \end{aligned} \tag{4}$$

Seq. Cct. Element	Formula
Initial states	$p_0 = 0, q_0 = 1, r_0 = 0$
q^{mx}	$q_0^{mx} \leftrightarrow ((q_0^s \vee q_0) \wedge (\neg q_0^s \vee q_0^f))$
p^{mx}	$p_0^{mx} \leftrightarrow ((p_0^s \vee p_0) \wedge (\neg p_0^s \vee p_0^f))$
r^{mx}	$r_0^{mx} \leftrightarrow ((r_0^s \vee q_0) \wedge (\neg r_0^s \vee r_0^f))$
q_1	$q_1 \leftrightarrow p_0^{mx}$
p_1	$p_1 \leftrightarrow q_0^{mx}$
r_1	$r_1 \leftrightarrow r_0^{mx}$
Faulty States	$\neg(p_1 \vee q_1 \vee r_1)$
Selectors	$\neg p_0^s \wedge \neg q_0^s \wedge \neg r_0^s$

TABLE II
IMPORTANT FORMULAS IN $BMC_1(M_o, \phi_{\mathsf{N}})$

Note that the vector variables s_i include the variables corresponding to the vertices v^f and v^s of the multiplexers. For over-approximate image computation using Craig interpolants, let

$$\begin{aligned} A &\stackrel{\text{def}}{=} BMC_I \wedge \textstyle\bigwedge_{v \in V_N} \neg v_0^s \text{ and} \\ B &\stackrel{\text{def}}{=} BMC_T \wedge \textstyle\bigwedge_{t=1}^{k} \bigwedge_{v \in V_N} \neg v_t^s. \end{aligned} \tag{5}$$

The intuition behind the partitioning of the BMC instance above is that only s_1 variables are shared between A and B. Initially, $Q = S_0$. The interpolant-based model checking procedure successively computes approximate images by interpolating the pair (A,B) as described in Algorithm 1. Assume that it reaches a fixed point after m approximate image steps at the bound k. Let $\hat{R}_{S_0}$ denote the final inductive invariant and $\mathscr{P}$ denote the final proof of unsatisfiability.

Example 4: Continue Example 3, and tie the selectors r^s, p^s, and q^s to zero. We now show how interpolant-based model checking proves that the property is satisfied by the circuit. Table II lists the relevant clauses of $BMC_1(M_o, \phi_{\mathsf{N}})$.

A resolution proof of $BMC_1(M_o, \phi_{\mathsf{N}})$ for the first iteration with $k = 1$ and $Q = S_0 = \neg p_0 \wedge q_0 \wedge \neg r_0$ is shown in Fig. 9. The only clause in the proof that belongs to B is $\neg p_1$. Hence, the interpolant for the pair (A,B) is p_1. It is straightforward to see that p_1 over-approximates the next state $(p_1 \wedge \neg q_1 \wedge r_1)$ of the initial state.

In the next iteration, $Q = p \vee (\neg p \wedge q \wedge \neg r) = (q \vee p) \wedge (\neg r \vee p)$. The resolution refutation of $BMC_1(M_o, \phi_{\mathsf{N}})$ is shown in Fig. 10. The interpolant for the new pair (A,B) is $p_1 \vee q_1$.

Continuing, $Q = (p \vee q) \vee p \vee (\neg p \wedge q \wedge \neg r) = (p \vee q) \vee (\neg p \wedge q \wedge \neg r) = p \vee q$. Since Q is contained in the previous image $p \vee q$, a fixed point is reached. An invariant of the circuit, i.e., union of all the images computed so far is $p \vee q$.
At this point, we compute another interpolant with a different partitioning:

$$(BMC_S, BMC_I \wedge BMC_T)$$

Let $\mathscr{I} \stackrel{\text{def}}{=} \text{ITP}(\mathscr{P}, BMC_S, BMC_I \wedge BMC_T)(s_{\mathscr{P}})$ denote the interpolant when a fixed point is reached. The intuition behind this partitioning is that only the selector variables are shared between the two partitions. The following holds by definition of Craig interpolants:

$$\begin{aligned} BMC_S &\rightarrow \mathscr{I} \\ BMC_I \wedge BMC_T &\rightarrow \neg\mathscr{I} \end{aligned} \tag{6}$$

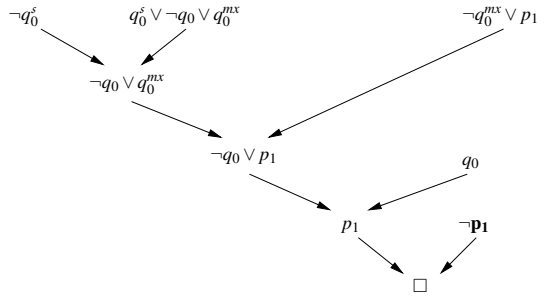

Fig. 9. Resolution refutation $\mathscr{P}_1$ of BMC_1 for the mutated sequential circuit in Fig. 8 with p_0^s, q_0^s, and r_0^s forced to zero with $F = \neg p_1 \wedge \neg q_1 \wedge \neg r_1$ and $Q = \neg p_0 \wedge q_0 \wedge \neg r_0$. The clauses shown in bold belong to B.

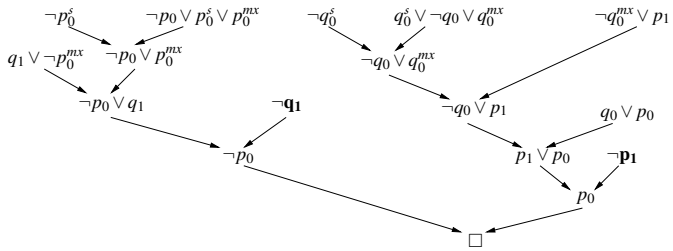

Fig. 10. Resolution refutation $\mathscr{P}_2$ for the final iteration during interpolant-based model checking of N_{mx} in Fig. 8 with p_0^s, q_0^s, and r_0^s forced to zero, $F = \neg p_1 \wedge \neg q_1 \wedge \neg r_1$, and $Q = (q_0 \vee p_0) \wedge (\neg r_0 \vee p_0)$. The clauses shown in bold belong to B.

We now make observations about the form of $\mathscr{I}$, which give rise to a simple test for detecting not-covered mutations. We prove that $\mathscr{I}$ contains only conjunctions of negative selector literals from BMC_S.

Theorem 1: Let $A_i \wedge B_i$ be an unsatisfiable formula such that $A_i = a_0 \wedge a_1 \wedge \ldots$ and let $\mathscr{P}_i$ be a proof of its unsatisfiability. An interpolant $\text{ITP}(\mathscr{P}_i, A_i, B_i)$ consists of conjunctions of a_i. It holds that $a_i \in \text{ITP}(\mathscr{P}_i, A_i, B_i) \leftrightarrow (\ell_{P_i}(v_i) = a_i$ and $a_i \in A_i)$ where v_i is an initial vertex of $\mathscr{P}_i$.

Proof: For each a_i appearing in $\mathscr{P}_i$, B_i contains $\neg a_i$. Hence, the variable a_i appearing in $\mathscr{P}_i$ is common to A_i and B_i, owing to the common vocabulary constraint of Craig interpolants.

Consider a formula I_i that is a conjunction of $\ell_{P_i}(v_i)$ of all initial vertices v_i of $\mathscr{P}_i$ such that $\ell_{P_i}(v_i) \in A$. It is easy to see that $A_i \rightarrow I_i$.

The formula I_i contains conjunctions of only those a_is that appear in the proof $\mathscr{P}_i$. Hence, we know that $I_i \wedge B_i$ is unsatisfiable from the existence of $\mathscr{P}_i$. Thus, I_i is an interpolant of the unsatisfiable pair (A_i, B_i). ∎

The following corollary of Theorem 1 states that the interpolant $\mathscr{I}$ consists of conjunctions of negative literals corresponding to selector variables present in the proof of unsatisfiability.

Corollary 1: Let the final proof of unsatisfiability of $BMC_I \wedge BMC_T \wedge BMC_S$ during model checking of M_o be $\mathscr{P}$. An interpolant $\mathscr{I} \stackrel{\text{def}}{=} \text{ITP}(\mathscr{P}, BMC_S, BMC_I \wedge BMC_T)(s_{\mathscr{P}})$ consists of conjunctions of $\ell_{\mathscr{P}}(v_i)$ for all initial vertices v_i of $\mathscr{P}$ with $\ell_{\mathscr{P}}(v_i) \in BMC_S$.

Example 5: Consider the final proof $\mathscr{P}_2$ computed in Example 4. The proof $\mathscr{P}_2$ is shown in Fig. 10. We have $BMC_S = \neg p_0^s \wedge \neg q_0^s \wedge \neg r_0^s$. Only $\neg p_0^s$ and $\neg q_0^s$ from BMC_S appear in the proof $\mathscr{P}_2$. It is easy to see that $\mathscr{I} = \neg p_0^s \wedge \neg q_0^s$.

The following theorem states that the selector settings according to $\mathscr{I}$ preserve the inductive invariant.

Theorem 2: The inductive invariant $\hat{R}_{S_0}$ of $M_o = (S_N, T_{N_{mx}^{SEL_0}})$ is also an inductive invariant of $M_{\mathscr{I}} = (S_N, T_{N_{mx}^{\mathscr{I}}})$.

Proof: The following formula is unsatisfiable for M_o.

$$\overbrace{\hat{R}_{S_0}(s_0) \wedge T_{N_{mx}}(s_0, s_1)}^{BMC_I} \wedge BMC_T \wedge BMC_S \tag{7}$$

Let $\mathscr{P}_o$ be the proof of unsatisfiability of Formula 7. Recall that for approximate image computation, we create a pair of unsatisfiable formulas (A, B) as shown in Formula 5. Let $\text{ITP}(\mathscr{P}_o, A, B)(s_{\mathscr{P}})$ be denoted as $\mathscr{I}_o$, which is the approximate image of $\hat{R}_{S_0}$ contained in $\hat{R}_{S_0}$ (the fixed point). Hence, $\mathscr{I}_o \subseteq \hat{R}_{S_0}$.

For $M_{\mathscr{I}}$, we begin with $Q = \hat{R}_{S_0}$. From Equation 6,

$$\hat{R}_{S_0}(s_0) \wedge T_{N_{mx}}(s_0, s_1) \wedge BMC_T \wedge \mathscr{I} \tag{8}$$

is unsatisfiable. From Corollary 1, we know that $\mathscr{I}$ conjoins only those selector literals from BMC_S that are used in $\mathscr{P}_o$. The proof $\mathscr{P}_o$ is also a proof of unsatisfiability of Formula 8. The interpolant $\text{ITP}(\mathscr{P}_o, A, B)(s_{\mathscr{P}})$ where A and B are partitions of Formula 8 is also $\mathscr{I}_o$. Hence, the approximate image of $\hat{R}_{S_0}$ in $M_{\mathscr{I}}$, i.e., $\mathscr{I}_o$ is contained in $\hat{R}_{S_0}$. This indicates that a fixed point is reached. ∎

Example 6: From Example 5, we have $\mathscr{I} = \neg p^s \wedge \neg q^s$. The inductive invariant $p \vee q$ of the circuit is also an inductive invariant of the circuit in which only p^s and q^s are set to zero (no mutation) and r^s is set to one (mutated) or zero (not mutated). Observe that $\neg p^s \wedge \neg q^s \wedge \neg r^s \rightarrow \mathscr{I}$ is valid and so is $\neg p^s \wedge \neg q^s \wedge r^s \rightarrow \mathscr{I}$.

Lemma 2: Let SEL_v represent the selector settings corresponding to mutating a vertex v of N according to some coverage criterion. If $SEL_v \rightarrow \mathscr{I}$, then v is not covered by that mutation.

Proof: According to Corollary 1, $\mathscr{I}$ is a conjunction of negative selector literals. Using Theorem 2, we conclude that $\hat{R}_{S_0}$ is an inductive invariant for a mutated system in which the selectors are constrained by $\mathscr{I}$. Note that SEL_v and $\mathscr{I}$ consist of conjunctions. Thus, if mutating v forces selectors in $\mathscr{I}$ to zero, i.e., $SEL_v \rightarrow \mathscr{I}$, $\hat{R}_{S_0}$ is still an inductive invariant of the mutated system, and thus, the node v is not covered by the mutation. ∎

Example 7: Continuing Example 6, we set r_0^s to one and the rest to zero, i.e., $SEL_r = r^s \wedge \neg p^s \wedge \neg q^s$. Hence, $SEL_r \rightarrow (\neg p^s \wedge \neg q^s)$. Using Lemma 2, register r is not covered by the property $\phi_{\mathsf{N}} = \mathsf{G}(p \vee q \vee r)$.

In summary, we run interpolant-based model checking on $M_o = (S_N, T_{N_{mx}^{SEL_0}})$ and compute the interpolant $\mathscr{I}$ from the final proof of unsatisfiability. The selectors of the multiplexers occurring in $\mathscr{I}$ when held to zero guarantee that the resulting system does not violate the property. The following section presents an improvement of this idea and demonstrates that a simple analysis of the proof of unsatisfiability is sufficient for

coverage computation, and that it is actually unnecessary to compute an interpolant.

C. The Unsatisfiable Core Test

The following lemma states that the absence of some selector variables in the final proof $\mathscr{P}$ during interpolant-based model checking of M_o can be used to identify which vertices are not covered.

Lemma 3: Consider a vertex v in the net-list N. For all t such that $0 \leq t \leq k$ if the selector variables v_t^s do not appear in $\mathscr{P}$, then v is not NONDET-COVERED, ZERO-COVERED, and ONE-COVERED.

Proof: We show that the selector formula corresponding to mutating v into a new primary input implies $\mathscr{I}$. The selector formula

$$SEL_v = \bigwedge_{t=0}^{t=k} v_t^s \wedge \bigwedge_{t=0}^{t=k} \bigwedge_{w \in V_N} \{\neg w_t^s | w \neq v\}$$

corresponds to this mutation. Using Corollary 1, we conclude

$$\mathscr{I} = \bigwedge_{v_p \in V_{\mathscr{P}}} \{\ell_{\mathscr{P}}(v_p) \mid \ell_{\mathscr{P}}(v_p) \in SEL_0\}.$$

Each conjunct in SEL_v is a conjunct in $\mathscr{I}$, except v_t^s, as $\mathscr{P}$ does not contain v_t^s in any timeframe. Hence, $SEL_v \rightarrow \mathscr{I}$. From Lemma 1, it follows that if v is not NONDET-COVERED, v is also not ZERO-COVERED and ONE-COVERED. ∎

Thus, a proof-logging SAT solver is not required; a solver able to identify the clauses that result in a proof is sufficient. This may result in significant savings in memory usage, as the size of the proof is worst-case exponential. Note that even if v is not NONDET-COVERED, it is possible that v_t^s appears in the proof $\mathscr{P}$ under consideration.

Example 8: Consider the final proof $\mathscr{P}_2$ shown in Fig. 10, computed in Example 4. The selector variable r^s does not appear in the proof in any timeframe, indicating that the register r is not needed to prove the property. Thus, the register r is neither NONDET-COVERED, ONE-COVERED nor ZERO-COVERED by the property. On the other hand, p^s and q^s appear in the proof, but we do not know whether the registers p and q are NONDET-COVERED.

As illustrated by the example above, the core test may fail to conclude coverage of some nodes in the net-list. In the next section, we discuss a BMC-based low-cost technique to compute coverage of the remaining nodes.

D. The Counterexample Test

A counterexample, i.e., a path from a state in S_0 to a state in F in the mutant circuit indicates at least one covered mutation. It is essential to identify such mutations early. Counterexamples of a given length can be obtained at moderate cost using BMC.

Since $\hat{R}_{S_0} \supseteq \bigcup_{i=0}^{m} img^i(S_0)$, there is a path of length $\leq m$ from a state in S_0 to a state in R_{S_0} in the original system. As R_{S_0} does not reach F in k steps, all paths of length $\leq (k+m)$ from S_0 in M do not reach F. We check if a counterexample of length $\leq (k+m)$ exists in $(S_N, T_{N_{mx}^{SEL_v}})$. Note that we only mutate one vertex at a time. For each trace obtained from the propositional SAT solver, we record the mutations that are found covered. If a mutation is ZERO-COVERED or ONE-COVERED, the counterexample test for NONDET-COVERED can be skipped according to Lemma 1.

Example 9: Consider the interpolant-based model checking steps worked out in Example 4. Note that $m = 2$ and $k = 1$.

In the counterexample test, the selector p^s is forced to one and the force p^f is either tied to zero, one, or kept non-deterministic. We check if a path of length ≤ 3 exists to faulty states, i.e., $\neg p \wedge \neg q \wedge \neg r$.

a) NONDET-COVERAGE *of p:* Consider the mutation that forces p^s to one and makes p^f non-deterministic. Note that a non-deterministic p^f can assume any value in any timeframe. Starting with the initial state $\neg p_0 \wedge q_0 \wedge \neg r_0$, $p_0^s = 1$, and $p_0^f = 0$, the next state is $p_1 \wedge \neg q_1 \wedge r_1$, the next state of which is in turn $\neg p_2 \wedge \neg q_2 \wedge \neg r_2$, which violates the property ϕ_{N}. Hence, the register p is NONDET-COVERED by the property.

b) ONE-COVERAGE *of p:* Consider a stuck-at-1 mutation that forces p^s to one and p^f to one. For $0 \leq i \leq 3$, the value of p_i^{mx} is one, which is shifted to the register q on each positive clock edge. Hence, there is no counterexample of length ≤ 3 that violates ϕ. The counterexample test is inconclusive about whether p is ONE-COVERED by ϕ.

c) ZERO-COVERAGE *of p:* Consider a stuck-at-0 mutation that forces p^s to one and p^f to zero. For $0 \leq i \leq 3$, $p_i^{mx} = 0$. The image of the initial state is $p_1 \wedge \neg q_1 \wedge r_1$, whose next state is $\neg p_1 \wedge \neg q_1 \wedge \neg r_1$. As none of p, q, or r are one in this state, this state violates ϕ_{N}. Hence, the register p is ZERO-COVERED by the property.

Similarly, q is NONDET-COVERED and ZERO-COVERED by the property. The counterexample test is inconclusive about whether q is ONE-COVERED by the property.

As illustrated by the example above, the core test and the counterexample test may be inconclusive about coverage of some nodes in the net-list. In the next section, we discuss a further low-cost technique to compute coverage of the remaining nodes.

E. Two Induction-based Tests

We exploit the inductive invariant $\hat{R}_{S_0}$ of M_o to analyze the remaining mutations at low cost. As $\hat{R}_{S_0}$ is an inductive invariant of M_o, the following formula is unsatisfiable:

$$\overbrace{\hat{R}_{S_0}(s_0) \wedge T_{N_{mx}}(s_0, s_1) \wedge \neg \hat{R}_{S_0}(s_1)}^{AI} \wedge \overbrace{\textstyle\bigwedge_{v \in V_N} (\neg v_0^s \wedge \neg v_1^s)}^{BI} \qquad (9)$$

Let $\mathscr{P}_I$ be the proof of unsatisfiability of Formula 9.

Lemma 4: Let SEL_v be the selector settings for some mutation of v. Let $AI' = \bigwedge_{v \in V_{\mathscr{P}_I}} \{\ell_{\mathscr{P}_I}(v) \mid \ell_{\mathscr{P}_I}(v) \notin BI\}$. If $SEL_v \wedge AI'$ is unsatisfiable, v is not covered by the mutation.

Proof: Note that $\ell_{\mathscr{P}_I}(v) \notin BI$ iff $\ell_{\mathscr{P}_I}(v) \in AI$. It holds that $AI \rightarrow AI'$. Thus, if AI' and SEL_v are inconsistent, AI and SEL_v are also inconsistent. Also, $S_0 \subseteq \hat{R}_{S_0}$, as our mutation does not affect the initial states. Hence, $\hat{R}_{S_0}$ is an inductive invariant of $(S_N, T_{N_{mx}^{SEL_v}})$. ∎

Example 10: The proof $\mathscr{P}_2$ computed in Example 4 is also the resolution proof of unsatisfiability of Formula 9 for M_o. We

Register	NONDET-COVERED	ZERO-COVERED	ONE-COVERED
p	✓	✓	×
q	✓	✓	×
r	×	×	×

TABLE III
COVERAGE OF THE THREE REGISTERS IN N (FIG. 6). A ✓ DENOTES "COVERED" AND A × DENOTES "NOT COVERED".

now check if the clauses in $\mathscr{P}_2$ belonging to AI' can be reused to conclude whether q is ONE-COVERED by the property ϕ_{N}.

The AI-clauses from $\mathscr{P}_2$ form AI' and $SEL_q = \neg p_0^s \wedge q_0^s \wedge q_0^f \wedge \neg r_0^s$. Observe that $AI' \wedge SEL_q$ is satisfiable. Hence, the test is inconclusive about whether q is ONE-COVERED by the property. Similarly, we do not know yet whether p is ONE-COVERED by the property.

If Formula 9 is satisfiable, it may be that the remaining clauses from AI are needed to prove inductiveness. This can be achieved by checking $AI \wedge SEL_v$ using the following SAT instance:

$$\hat{R}_{S_0}(s_0) \wedge T_{N_{mx}}(s_0, s_1) \wedge SEL_v \wedge \neg \hat{R}_{S_0}(s_1) \tag{10}$$

Lemma 5: Let SEL_v be the selector setting for some mutation of v. If Formula 10 is unsatisfiable, v is not covered by the mutation.

Proof: Recall $S_0 \subseteq \hat{R}_{S_0}$. Since Formula 10 is unsatisfiable, the image of $\hat{R}_{S_0}$ for the mutated transition relation is contained in $\hat{R}_{S_0}$, which is strong enough to prove the property. ■

Example 11: Consider the mutated sequential circuit N_{mx} in Fig. 8 and the property ϕ_{N}. We check if q is ONE-COVERED by ϕ_{N}. The inductive invariant of M_o is $p \vee q$. The formula $(p_0 \vee q_0) \wedge \neg p_1 \wedge \neg q_1 \wedge \neg p_0^s \wedge q_0^s \wedge q_0^f \wedge \neg r_0^s$ conjoined with the clauses related to N_{mx} in Table II (rows 2–7) is unsatisfiable. Fig. 11 shows the proof of unsatisfiability. Hence, the register q is not ONE-COVERED by ϕ_{N}. We check that the register p is not ONE-COVERED by ϕ_{N} in a similar way.

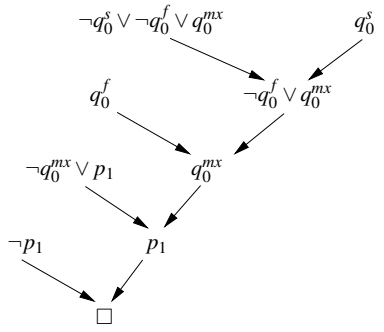

Fig. 11. Proof of unsatisfiability of Formula 10 for N_{mx} and $SEL_q = \neg p_0^s \wedge q_0^s \wedge q_0^f \wedge \neg r_0^s$

If Formula 10 is satisfiable, there may still exist a different inductive invariant for the mutated system. In this case, we fall back on interpolant-based model checking of the mutated circuit.

Table III lists the coverage results for the sequential circuit N and the property ϕ_{N}. In this example, the coverage tests described above successfully identify coverage of each register and the algorithm does not resort to interpolant-based model checking for coverage analysis of any of the registers.

F. Approximate coverage computation

We are often not interested in the precise percentage of covered states, but rather whether this measure is above or below a certain threshold. For example, it is reasonable to expect that even the most exhaustive properties will not achieve 100% coverage. However, the difference between, for example, 30% and 90% coverage is very significant. Hence, in most cases, it suffices to compute an approximation of the coverage metric, and precision can be traded for efficiency to some extent.

Algorithm 2 lends itself naturally to such an approximate computation. Indeed, all steps except the final one—interpolation-based model checking of the whole mutant design—are relatively lightweight. As our experiments show (see Section V), all these steps incur very little computational overhead on top of model checking. This gives rise to *approximate coverage computation*. We can start with performing steps (1)–(4) of Algorithm 2, incurring only modest computational overhead. Then, for the nodes whose coverage remains unknown, we perform full model checking only until we reach the desired accuracy of the coverage metric. We demonstrate approximate coverage computation on industrial designs in Section V.

V. EXPERIMENTS

We have implemented the proposed algorithm in the tool EBMC. The tool accepts Verilog as well as SMV files as input. In addition, EBMC supports SystemVerilog assertions for specifying safety properties.

A. Coverage Analysis on the HWMCC 2008 Circuits

We first reports results on the circuits from the Hardware Model Checking Competition 2008. Each of the benchmarks is shipped with one property. We ran our tool on 161 benchmarks (from the set) which our interpolant-based model checker was able to complete within a timeout of 1800 seconds. The average number of registers/latches in these benchmarks is about 110 with a median 75, and the maximum number of latches is 567. About 25% of the benchmarks have a high-coverage property (an average of 40% of the latches are NONDET-COVERED). The remainder of the benchmarks have low-coverage properties (an average of 5% of the latches are NONDET-COVERED).

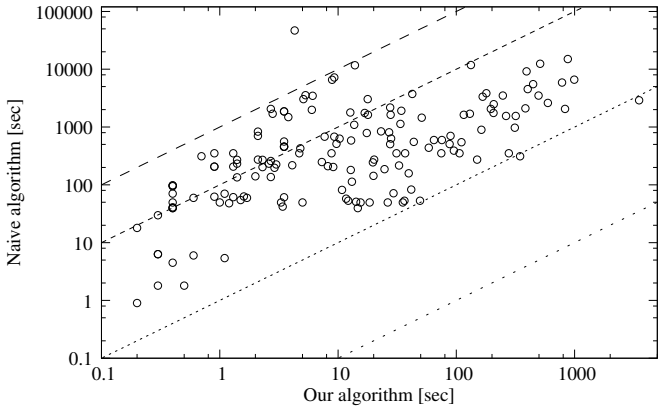

Fig. 12. Comparison of naive vs. COVERAGECHECKS

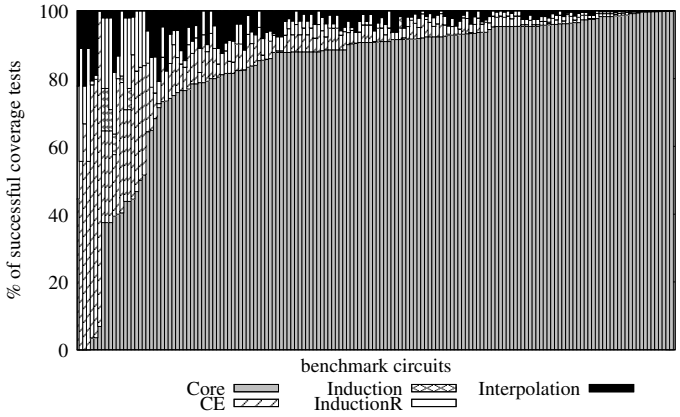

Fig. 13. Histogram of the success rates of the proposed tests in COVERAGECHECKS

We restrict the coverage analysis to the registers. For each register in the design, we check whether it is ZERO-COVERED, ONE-COVERED, or NONDET-COVERED. Fig. 12 is a log-scale scatter plot comparing the time taken by our algorithm with the time it would take to run model checking on each of the mutated designs.[2] The speedup obtained using our method is several orders of magnitude. The CORE test, COUNTEREXAMPLE test and INDUCTION test run extremely quickly. About 85% of the time taken by our algorithm is spent in interpolation-based tests. The rest of the time is spent in interpolation-based model checking of the original design.

We quantify the success rates of the proposed five methods presented in Section IV. We call a test "successful" if it is able to classify a given mutation as "covered" or "not covered". The histogram in Fig. 13 depicts the relative success rates of each test for computing coverage (recall that the CORE test identifies nodes not mentioned in the proof, the COUNTEREXAMPLE test looks for a counterexample, INDUCTION attempts to construct an inductive argument that a node is not covered using the inductive invariant and only the nodes that are mentioned in the proof; the full version of INDUCTION, the test INDUCTIONR, uses the full net-list; finally, the INTERPOLATION test applies full-blown interpolation-based model checking to all undecided nodes). The coverage of a vast majority of mutations is decided by the CORE and the COUNTEREXAMPLE (CE) tests. Our method is indeed able to avoid most of the expensive calls to the interpolant-based model checker: in 50% of the benchmarks, we call the model checker in only 10% of the cases.

For benchmarks with a low-coverage property, most of the selector constraints are not in the core. Thus, the CORE test is frequently successful on this category of benchmarks. On benchmarks with a high-coverage property, a counterexample for one mutant is often applicable to many others, which means that the COUNTEREXAMPLE test is successful. The remaining tests contribute roughly equal parts.

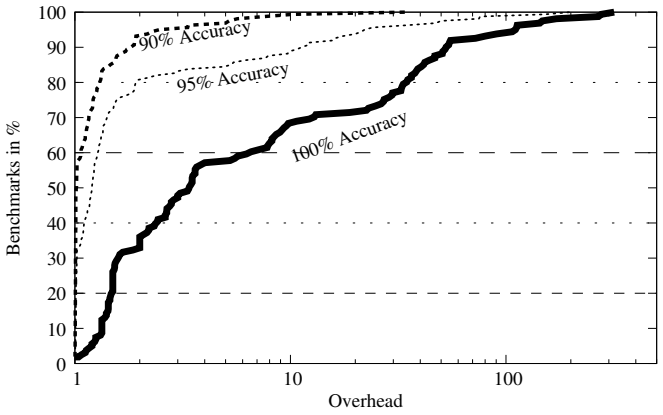

Fig. 14. Overhead for a given accuracy

In most cases, an approximate coverage metric is sufficient. We observe that our algorithm is very well-suited for obtaining a quick approximation. Fig. 14 reports the time overhead in relation to the original model checking run for a given accuracy, averaged over the benchmark set. We note that 90% accuracy can be obtained for 90% of the benchmarks analyzed with an overhead of only 2.

B. Coverage Analysis on IBM Circuits

We have analyzed 10 industrial designs from IBM with COVERAGECHECKS and compared the results to the naive algorithm. Table IV reports the success of the proposed tests on these examples. The timeout is set to 7200 seconds. The column #LAT reports the number of latches in the design. For each latch, we check if it is ZERO-COVERED, ONE-COVERED, or NONDET-COVERED. Hence, there are #LAT $\times$ 3 coverage tests for each example.

The column #CORE reports the number of successful CORE tests. Note that if a CORE test is successful for a latch, the latch is NOT-NONDET-COVERED, and hence, it is NOT-ONE-COVERED and NOT-ZERO-COVERED as well. For example, out of the 8 latches in Example 3, 18/3=6 latches do not appear in the proof. Hence, these 6 latches are NOT-ZERO-COVERED, NOT-ONE-COVERED, and NOT-NONDET-COVERED.

The column #IND reports the number of successful induction tests. In Example 3, out of the remaining $24-18=6$ tests, the coverage results of two tests are identified by the induction

[2]This time is estimated based on the time spent on model checking some mutated systems, as the full check is prohibitively expensive for many examples.

		CoverageChecks							Naive		
Ex	#LAT	#CORE	#IND	#CE	T1 (sec)	#INT	T2 (sec)	%COV$_C$	#NAIVE	T$_N$(sec)	%COV$_N$
1	125	372	1	0	2	2	2.5	100	375	65	100
2	125	372	1	0	2	2	2.5	100	375	65	100
3	8	18	2	2	0.03	2	0.08	100	24	0.1	100
4	52	156	0	0	0.5	0	0.5	100	156	7	100
5	125	372	1	0	2	2	2.5	100	375	65	100
6	125	372	1	0	2	2	2.5	100	375	65	100
7	42	87	4	0	4.4	1	7200*	73	21	7200*	16.6
8	42	87	4	0	6.1	1	7200*	73	20	7200*	15.8
9	42	87	4	0	75	1	7200*	73	21	7200*	16.6
10	42	87	4	0	6.1	1	7200*	73	21	7200*	15.8

TABLE IV
COVERAGE DATA ON IBM BENCHMARKS. A TIMEOUT IS INDICATED BY *.

test, classifying the corresponding latch as "not covered" by that mutation.

The column #CE reports the number of successful counterexample tests. The counterexample test (CE test) was only successful on Example 3. In Example 3, the coverage results of two tests are identified by the counterexample test classifying the corresponding latch as covered by that mutation. It is likely that the low success rate of the counterexample test is due to the low coverage of the properties on these examples: only 1–2% of the latches are covered.

We need to run the interpolation based-model checking in order to compute coverage of the remaining latches. The column #INT reports the number of successful (complete) interpolation runs. In Example 3, coverage results of two tests could be obtained only by means of the interpolation-based model checker.

The column T2 reports the total running time of CoverageChecks. The column T1 reports the running time of CoverageChecks without the interpolation-based model checker. The column #NAIVE reports the number of coverage tests executed by the naive algorithm in two hours. The column T$_N$ reports the running time of the naive algorithm.

The columns %COV$_C$ and %COV$_N$ report the percentage of successful coverage tests using CoverageChecks and the naive algorithm, respectively. For the examples in which all the coverage tests are completed in two hours, CoverageChecks is 14–26 times faster than the naive coverage computation algorithm. For the examples in which both the algorithms timed out, CoverageChecks computed approximately 70% of the coverage results (approximate coverage) in 4–75 seconds. For the same examples, the naive algorithm computed less than 20% of the coverage results in two hours.

VI. Related Work

Two approaches for defining and developing algorithms for coverage metrics in temporal logic model checking have been studied in the literature. The first approach, of Katz et al. [29], is based on a comparison of the system with a tableau of the specification. Essentially, a tableau of a universal specification φ is a system that satisfies φ and subsumes all the behaviors allowed by φ. By comparing a system with the tableau of φ, Katz et al. are able to detect parts of the systems that are irrelevant to the satisfaction of the specification, to detect different behaviors of the system that are indistinguishable by the specification, and to detect behaviors that are allowed by the specification but not generated by the system. Such cases imply that the specification is incomplete or not sufficiently restrictive. The tableau used in [29] is *reduced*: a state of the tableau is associated with subformulae that have to be true in it, and it induces no obligations on the other, possibly propositional, subformulae. This leads to smaller and less restrictive tableau. Roughly speaking, a system passes the criteria in [29] if and only if it is bisimilar to the tableau of the specification. This is also the main drawback of this approach, as we want specifications to be much more abstract than their implementations[3].

The second approach, of Hoskote et al. [30], is to define coverage by examining the effect of modifications in the system on the satisfaction of the specification. Given a design M, a formula φ satisfied in M, and an observable signal q, a state w of M is *q-covered by* φ if the design obtained from M by flipping the value of q in w no longer satisfies φ. This indicates that the value of q in w is crucial for the satisfaction of φ in M. Hoskote et al. describe an algorithm for computing the set of states that are q-covered by a formula φ in the logic *acceptable ACTL*, where acceptable ACTL is a restriction of the universal fragment ACTL of CTL in which no disjunctions are allowed and all the implications $\alpha \rightarrow \beta$ are such that α is propositional. The algorithm in [30] is applied to φ after an *observability transformation* and has a linear time complexity, like CTL model-checking. Essentially, the linear running time is allowed by the restricted syntax of acceptable ACTL and the observability transformation that force the mutant designs to satisfy φ in exactly the same way the original design M does. The algorithm has been improved to support a larger subset of CTL and increase its accuracy [31].

The majority of recent research on coverage in formal verification relies on the notion of mutation-based coverage based on the definition in [30], extending it to different types of mutations and different representations of designs and properties [32]–[34].

Most attempts in the literature to alleviate the complexity of

[3]The approach in [29] also has some technical and computational disadvantages: the specification considered is the (big) conjunction of all the properties the system should satisfy, the complexity of the algorithm is exponential in the specification (for φ in ACTL), and it is restricted to universal safety specifications whose tableau have no fairness constraints.

computing coverage involve mutating the symbolic representation of the design (by adding nondeterministic BDD variables for mutations) or exploiting the similarity between mutant designs, thus performing most of the model-checking effort just once [35]. The idea of exploiting the similarity between mutant designs in order to speed up coverage computation also inspires our work.

It is also common to combine model checking and testing with the goal of increasing coverage. Variants include symbolic searches from deep states obtained by simulation, and the use of coverage goals from testing as targets for the formal engine. These algorithms have found their way into industrial practice [36]. Test case generation based on model checking is proposed in [37]. Use of model checking and various analyses such as property coverage of test cases, model-coverage of the property, and specification vacuity for test case generation is proposed in [38], [39]. A game-based approach between the model and its test bench to create intelligent test cases to cover corner case behaviors is proposed in [40].

An altogether different direction of coverage computation is *coverage without design*, where the set of properties is said to cover the set of output signals if the value of each output signal is fully determined by the property given a combination of input values [41], [42]. This approach is inspired by the earlier work by Das et al. which introduces the notion of *fault-based coverage*: a stuck-at-fault signal is covered by a property if and only if an implementation that satisfies the property and has the fault is unrealizable [43]. These approaches do not suffer from complexity-related issues, because properties are usually very small compared to the design, and any computation that is carried out only on the property is regarded as having negligible complexity with respect to model checking.

In [44]–[47], a different notion of coverage called *design intent coverage* is proposed. The idea is to compare two formal properties at different levels of abstraction and find a set of properties that close the coverage gap between the two specifications. One of the applications is to check if Register Transfer Level (RTL) properties cover architecture-level properties.

VII. Conclusion

Low coverage indicates possible incompleteness in the specification, which may lead to missed bugs in the non-covered parts of the design. We present an algorithm for computing mutation coverage of designs represented as net-lists. Our algorithm is integrated and implemented in a Craig interpolant-based model checker. The main advantage of our algorithm is that it exploits the information in the final resolution proof, and thus, the model checker needs to be run only on a very small percentage of mutations. Our tool also allows the trading of precision for speed. We show that a very accurate coverage measure can be computed with little overhead compared to the time taken by the model checking run on the original design. Our technique provides a speedup of several orders of magnitude compared to the naive method, and it is well-suited to computing very inexpensive approximations.

Options for future work include an extension of our algorithm to word-level verification techniques [48], and an investigation into whether proof-restructuring techniques [49] can result in further, inexpensive coverage tests. Our technique can also be applied to other problems in which many similar model checking instances have to be solved, such as sequential equivalence checking.

Acknowledgements

The authors would like to thank Ofer Strichman for discussions on the paper.

References

[1] L. Bening and H. Foster, *Principles of verifiable RTL design – a functional coding style supporting verification processes*. Kluwer, 2000.
[2] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001.
[3] Y. V. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *DAC*. ACM, 1999, pp. 300–305.
[4] H. Chockler, O. Kupferman, and M. Vardi, "Coverage metrics for temporal logic model checking," in *TACAS*, ser. LNCS 2031. Springer, 2001, pp. 528 – 542.
[5] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi, "A practical approach to coverage in model checking," in *CAV*, ser. LNCS 2102. Springer, 2001, pp. 66–78.
[6] H. Chockler and O. Kupferman, "Coverage of implementations by simulating specifications," in *Conference on Theoretical Computer Science*, ser. IFIP Conference Proceedings 223. Kluwer, 2002, pp. 409–421.
[7] H. Chockler, O. Kupferman, and M. Vardi, "Coverage metrics for formal verification," in *CHARME*, ser. LNCS 2860. Springer, 2003, pp. 111–125.
[8] O. Kupferman, "Sanity checks in formal verification," in *Concurrency Theory (CONCUR)*, ser. LNCS 4137. Springer, 2006, pp. 37–51.
[9] O. Kupferman, W. Li, and S. A. Seshia, "A theory of mutations with applications to vacuity, coverage, and fault tolerance," in *FMCAD*. IEEE, 2008, pp. 1–9.
[10] K. L. McMillan, "Interpolation and SAT-based model checking," in *CAV*, ser. LNCS 2725. Springer, 2003, pp. 1–13.
[11] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs*. Springer, 1981, pp. 52–71.
[12] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Symposium on Programming*, ser. LNCS, vol. 137. Springer, 1982, pp. 337–351.
[13] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE, 1977, pp. 46–57.
[14] E. Clark, O. Grumberg, and D. Peled, "Model checking," in *MIT Press*, 1999.
[15] R. Bryant, "Graph-based algorithms for boolean-function manipulation," *IEEE Trans. on Computers*, vol. C-35, no. 8, 1986.
[16] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.
[17] J. P. M. Silva and K. A. Sakallah, "GRASP – a new search algorithm for satisfiability." in *ICCAD*. ACM, 1996, pp. 220–227.
[18] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver." in *DAC*, 2001, pp. 530–535.
[19] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver." in *DATE*. IEEE, 2002, pp. 142–149.
[20] N. Eén and N. Sörensson, "An extensible SAT-solver." in *SAT*, ser. LNCS. Springer, 2003, pp. 502–518.
[21] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.
[22] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *J. Symb. Comput.*, vol. 2, no. 3, pp. 293–304, 1986.
[23] G. S. Tseitin, "On the complexity of derivation in the propositional calculus," *Zapiski nauchnykh seminarov LOMI*, vol. 8, pp. 234–259, 1968.
[24] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[25] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, ser. LNCS. Springer, 1999, pp. 193–207.
[26] G. Huang, "Constructing Craig interpolation formulas," in *Computing Combinatorics (COCOON)*, ser. LNCS, vol. 959. Springer, 1995, pp. 181–190.
[27] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *The Journal of Symbolic Logic*, vol. 62, no. 3, pp. 981–998, 1997.
[28] J. Krajíček, "Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic," *The Journal of Symbolic Logic*, vol. 62, no. 2, pp. 457–486, 1997.
[29] S. Katz, D. Geist, and O. Grumberg, ""Have I written enough properties?" a method of comparison between specification and implementation," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, ser. LNCS, vol. 1703. Springer, 1999, pp. 280–297.
[30] Y. V. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *DAC*. ACM, 1999, pp. 300–305.
[31] N. Jayakumar, M. Purandare, and F. Somenzi, "Dos and don'ts of CTL state coverage estimation," in *DAC*. ACM, 2003, pp. 292–295.
[32] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi, "A practical approach to coverage in model checking," in *CAV*, ser. LNCS 2102. Springer, 2001, pp. 66–78.
[33] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for temporal logic model checking," *Formal Methods in System Design*, vol. 28, no. 3, pp. 189–212, 2006.
[34] ——, "Coverage metrics for formal verification," *STTT*, vol. 8, no. 4-5, pp. 373–386, 2006.
[35] N. He, P. Rümmer, and D. Kroening, "Test-case generation for embedded Simulink via formal concept analysis," in *DAC*. ACM, 2011, pp. 224–229.
[36] Jasper Design Automation Inc., "Reaching 100% coverage using formal," 2008, panel talk at HVC.
[37] P. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *ICFEM*. IEEE, 1998, pp. 46–54.
[38] S. Rayadurgam and M. P. Heimdahl, "Coverage based test-case generation using model checkers," in *Engineering of Computer Based Systems (ECBS)*. IEEE, April 2001, pp. 83–91.
[39] G. Fraser and F. Wotawa, "Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis," in *ICSEA*. IEEE, 2006, p. 16.
[40] A. Banerjee, B. Pal, S. Das, A. Kumar, and P. Dasgupta, "Test generation games from formal specifications," in *DAC*. ACM, 2006, pp. 827–832.
[41] S. Das, A. Banerjee, P. Basu, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, and L. Fix, "Formal methods for analyzing the completeness of an assertion suite against a high-level fault model," in *Proceedings of 18th International Conference on VLSI Design*. IEEE, 2005, pp. 201–206.
[42] K. Claessen, "A coverage analysis for safety property lists," in *FMCAD*. IEEE, 2007, pp. 139–145.
[43] S. Das, A. Banerjee, P. Basu, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, and L. Fix, "Formal methods for analyzing the completeness of an assertion suite against a high-level fault model," in *VLSI Design*. IEEE, 2005, pp. 201–206.
[44] S. Das, P. Basu, A. Banerjee, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, L. Fix, and R. Armoni, "Formal verification coverage: computing the coverage gap between temporal specifications," in *ICCAD*. IEEE, 2004, pp. 198–203.
[45] P. Basu, S. Das, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, and L. Fix, "Formal verification coverage: Are the RTL-properties covering the design's architectural intent?" in *DATE*. IEEE, 2004, pp. 668–669.
[46] P. Basu, S. Das, A. Banerjee, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, L. Fix, and R. Armoni, "Design-intent coverage – a new paradigm for formal property verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1922–1934, 2006.
[47] S. Das, P. Basu, P. Dasgupta, and P. P. Chakrabarti, "What lies between design intent coverage and model checking?" in *DATE*. European Design and Automation Association, 2006, pp. 1217–1222.
[48] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word level predicate abstraction and refinement for verifying RTL Verilog," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, pp. 366–379, February 2008.
[49] V. D'Silva, M. Purandare, G. Weissenbacher, and D. Kroening, "Interpolant strength," in *Proceedings of VMCAI*, ser. LNCS, vol. 5944. Springer, 2010, pp. 129–145.

Hana Chockler received the B.Sc. degree in computer science and mathematics from the Hebrew University of Jerusalem, the M.Sc. degree in computer science from Tel-Aviv University, and the Ph.D. degree in computer science from the Hebrew University of Jerusalem, Israel, in 1994, 1997, and 2003, respectively. She was a postdoctoral researcher at Northeastern University and a visiting scientist at MIT between 2003 and 2005.

She joined the Formal Verification group at IBM Research Laboratory in Israel in 2005. Her research interests include automated formal verification of hardware and software systems, coverage metrics for formal verification, hybrid verification methods, and combinatorial optimization.

Daniel Kroening received the M.E. and doctoral degrees in computer science from the University of Saarland, Saarbrücken, Germany, in 1999 and 2001, respectively. He joined the Model Checking group in the Computer Science Department at Carnegie Mellon University, Pittsburgh PA, USA, in 2001 as a Post-Doc.

He was an assistant professor at the Swiss Technical Institute (ETH) in Zürich, Switzerland, from 2004 to 2007. He is now Professor of Computer Science at the Computer Science Department at Oxford University. His research interests include automated formal verification of hardware and software systems, decision procedures, embedded systems, and hardware/software co-design.

Mitra Purandare received the B.E. degree from Government College of Engineering Pune, Pune, India in 2000. She received the M.Sc. degree in Computer Engineering from University of Colorado Boulder, Boulder, USA in 2003. She joined ETH Zurich, Switzerland in 2006 as a doctoral student and received the doctoral degree in computer science in 2010.

She is now a postdoctoral researcher in the Accelerator Technologies group at IBM Research Laboratory in Zurich, Switzerland. Her research interests include automated formal verification of hardware and software systems, hardware/software co-design, and use of hardware-based accelerators in various applications.