Proof-Based Coverage Metrics for Formal Verification

Elaheh Ghassabani, Michael Whalen, Mats Heimdahl
Department of Computer Science & Engineering
University of Minnesota
MN, USA
ghass013, mwwhalen, heimdahl@umn.edu

Andrew Gacek, Lucas Wagner
Rockwell Collins
Advanced Technology Center
IA, USA
andrew.gacek, lucas.wagner@rockwellcollins.com

***Abstract*—When using formal verification on critical software, an important question involves whether we have we specified enough properties for a given implementation model. To address this question, coverage metrics for property-based formal verification have been proposed. Existing metrics are usually based on *mutation*, where the implementation model is repeatedly modified and re-analyzed to determine whether mutant models are "killed" by the property set. These metrics tend to be very expensive to compute, as they involve many additional verification problems.**

This paper proposes an alternate family of metrics that can be computed using the recently introduced idea of Inductive Validity Cores (IVCs). IVCs determine a minimal set of model elements necessary to establish a proof. One of the proposed metrics is both rigorous and substantially cheaper to compute than mutation-based metrics. In addition, unlike the mutation-based techniques, the design elements marked as necessary by the metric are guaranteed to preserve provability. We demonstrate the metrics on a large corpus of examples.

***Keywords*-coverage; requirements completeness; formal verification; inductive proofs; inductive validity cores;**

I. Introduction

In safety critical systems development, an important question involves whether the requirements and testing process are adequate for the implementation. For example, in DO178B/C [30], tests are derived from requirements and the adequacy of the tests is measured by examining *coverage* of the implementation. As the criticality of the software increases, more rigorous adequacy measures (statement, decision, MC/DC) are required, with the justification that code must be more rigorously executed by tests to demonstrate its compliance. If complete coverage is not achieved, analysis is performed to determine whether additional tests *or additional requirements* are necessary to achieve coverage.

For critical systems, it has been argued that formal methods should be applied to gain higher assurance than is possible with testing [19], [27], [31]. Unfortunately, proof-based approaches tend not to answer the question as to whether implementations have functionality that is not covered by requirements. Unlike testing, for proof the whole system is 'executed', but many parts of the system may be irrelevant to the property that is proved.

This work was carried out within the HACMS and SOSITE Phase II grants (DARPA FA8750-12-9-0179 and FA8650-16-C-7656).

Relatively recently, techniques have been devised for analyzing adequacy of requirements against formal implementation models specified as transition systems or Kripke structures [5], [7], [10], [15]. The mechanism used is based on *mutation* and *proof*: is it possible to prove that the requirements still hold of the system after mutating the model in some way? If so, then the requirements are incomplete with respect to the mutated part of the model. Unfortunately, previous approaches to computing coverage metrics can *underapproximate* which portions of a program are necessary to prove the requirements. In addition, the mutation-based analyses tend to be computationally very expensive because there are many possible mutant models to verify. For example, for model checkers, state of the art techniques have runtimes of (in the best case) several times more than is required for proof [4].

We wish to have a graduated set of metrics, suitable for use in certification, for checking the adequacy of requirements against an implementation model that: (1) can be applied early and throughout a development cycle on different implementation artifacts, (2) are *proof preserving*: the portion of the implementation that is identified as covered demonstrates the fulfillment of the requirement but does not contain irrelevant information, and (3) are efficient to compute. Towards this end, we propose measures of requirements adequacy based on *minimal proofs of requirements*. We measure the adequacy of a set of requirements by examining an (approximately) minimal set of model elements necessary to construct a proof of all the requirements.

Our approach is applicable to *reactive software* that does a bounded amount of computation in response to an input for an unbounded sequence of inputs. This set contains most embedded and especially safety-critical software, and can be described as transition systems (equivalently, infinite-state "circuits" that compute next states given an input and current state). It is implemented using *Inductive Validity Cores* (IVCs) [13] for transition systems, and integrated into a branch of the `JKind` model checker [1]. In our benchmarks, we demonstrate that one of the proof-based metrics is considerably more computationally tractable than previous approaches based on mutation, averaging 24% overhead over model-checking alone, rather than the 2369% overhead required for the state of the art of the mutation-based metrics. Thus, the contributions of this

```
node asw(alt1, alt2: int; inhibit: bool)
        returns (doi_on: bool);
var
   a1_below, a2_below, a1_above, a2_above,
   below, above_hyst, d1, d2: bool;
let
   a1_below = (alt1 < THRESHOLD);          // (1)
   a2_below = (alt2 < THRESHOLD);          // (2)
   a1_above = (alt1 >= T_HYST);            // (3)
   a2_above = (alt2 >= T_HYST);            // (4)
   below = a1_below or a2_below;           // (5)
   above_hyst = a1_above and a2_above;     // (6)
   doi_on = if (below and not inhibit)     // (7)
        then true else d1;
   d1 = if (inhibit or above_hyst)         // (8)
        then false else d2;
   d2 = (false -> pre(doi_on));            // (9)
tel;
```

Fig. 1. Altitude Switch Model

work are:

1) A family of coverage metrics for formal verification based on *minimal* Inductive Validity Cores (MIVCs). Most of the proposed metrics are *proof preserving*,
2) A discussion of the relationship between proof-based metrics and mutation-based metrics, including a proof of equivalence between non-deterministic mutation coverage and one of the proof-based metrics (MUST-COV).
3) An implementation that efficiently computes property coverage over symbolic transition systems,
4) An initial experiment that compares our technique against a state of the art mutation-based notion of completeness.

II. RUNNING EXAMPLE

We use a very simple system from the avionics domain to illustrate our approach. An Altitude Switch (ASW) is a hypothetical device that turns power on to another subsystem, the Device of Interest (DOI), when the aircraft descends below a threshold altitude and turns the power off again after we ascend over the threshold plus some hysteresis factor. An implementation of an ASW containing two altimeters written in the Lustre [17] language (simplified and adapted from [20]) is shown in Figure 1. If the system is not "inhibited" by the user and either altimeter is below the constant THRESHOLD, then it turns on the DOI; else, if the system is inhibited or both altimeters are above the threshold plus the hysteresis factor (THRESHOLD + HYST `T_Hyst`), then the DOI is turned off, and if neither condition holds, then in the initial computation it is false and thereafter retains its previous value. The notation `(false -> pre(doi_on))` in equation (9) describes an initialized register in Lustre: in the initial state, the expression is `false`, and thereafter it is the previous value of `doi_on`. In the remainder of the paper, we will use this model to illustrate aspects of requirements completeness.

III. PRELIMINARIES

This section presents formalizations of transition systems, inductive validity cores, and background information on mutation-based coverage metrics. Although we focus the formalism below on safety properties, the approach is able to handle liveness properties through reduction to safety properties, as is performed by, e.g., K-liveness [8].

A. Models, Requirements, and Provability

Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \to bool$ and a transition step predicate $T : U \times U \to bool$. We define the notion of reachability for (I, T) as the smallest predicate $R : U \to bool$ which satisfies the following formulas:

$$\forall u.\ I(u) \Rightarrow R(u)$$
$$\forall u, u'.\ R(u) \wedge T(u, u') \Rightarrow R(u')$$

A safety property $P : U \to bool$ is a state predicate that holds on a transition system (I, T) if it holds on all reachable states, i.e., $\forall u.\ R(u) \Rightarrow P(u)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$. We assume the transition relation has the structure of a top-level conjunction. This assumption gives us a structure that we can easily manipulate. Given $T(u, u') = T_1(u, u') \wedge \cdots \wedge T_n(u, u')$ we will write $T = T_1 \wedge \cdots \wedge T_n$ for short. By further abuse of notation, T is identified with the set of its top-level conjuncts. Thus, $x \in T$ means that x is a top-level conjunct of T, and $S \subseteq T$ means all top-level conjuncts of S are top-level conjuncts of T. When a top-level conjunct x is removed from T, it is written as $T \setminus \{x\}$. Such a transition system can easily encode our example model in Section II. We assume each equation defines a conjunct within the transition system which we will denote by the variable assigned, so $T = \{$ `a1_below, a2_below, a1_above, a2_above, below, above_hyst, d1, d2` $\}$.

Definition 1. Inductive Validity Core (*IVC*) [13]: *$S \subseteq T$ for $(I, T) \vdash P$ is an Inductive Validity Core, denoted by IVC(P, S), iff $(I, S) \vdash P$.*

In examining provability, we are interested in minimal sets that satisfy a property P; tracing a property to the entire model is not particularly enlightening. Fortunately, *IVC*s have the following monotonicity property [13]: given $(I, T) \vdash P$, $\forall S_1 \subseteq S_2 \subseteq T.\ IVC(P, S_1) \Rightarrow IVC(P, S_2)$. We next introduce the notion of *minimal* inductive validity cores.

Definition 2. Minimal Inductive Validity Core (*MIVC*) [13]: *$S \subseteq T$ is a minimal Inductive Validity Core, denoted by MIVC(P, S), iff IVC$(P, S) \wedge \forall T_i \in S.\ (I, S \setminus \{T_i\}) \nvdash P$.*

Note that given $(I, T) \vdash P$, P always has at least one *MIVC*, which implies *MIVC*s are not necessarily unique. For example, take $I = a \wedge b$, $T = a' \wedge b'$, and $P = a \vee b$. Then both $\{a'\}$ and $\{b'\}$ are *MIVC*s for $(I, T) \vdash P$. To capture this fact, the *all MIVCs (AIVC)* relation has been introduced [29].

$$AIVC(P) \equiv \{\ S \mid S \subseteq T \wedge \textit{MIVC}(P, S)\}$$

Establishing the $AIVC$ for a single property, one gets a clear picture of the all the model elements constrained by the property. The set of $AIVC$s for all properties demonstrates a complete mapping from the requirements to the design elements, which is called complete traceability [29].

B. Coverage and Mutations

In general, specification completeness can be defined with regard to the notion of coverage. In fact, the way that coverage is formalized plays a key part in the strength/effectiveness of a method for the assessment of completeness. The goal of a coverage metric is usually to assign a numeric score that describes how well properties cover the design. The majority of the work on coverage metrics has focused on *mutations*, which are "atomic" changes to the design, where the set of possible mutations depends on the notation that is used. A mutant is "killed" if one of the properties that is satisfied by the original design is violated by the mutated design [4]–[6], [24], [25]. There are many different kinds of mutations that have been proposed, primarily focused on checking sequential bit-level hardware designs. Similarly, for software, it is possible to apply any of the "standard" source code mutation operators used for software testing [3] towards requirements coverage analysis.

We assume each element $T_i \in T$ has a set of possible mutations associated with it. Depending on the modeling formalism used, this may be the value of a gate or signal or an expression within a statement in a program. We will further assume the existence of a mutation function f_m that, given a model element, will return a finite set of mutations for that element. We can then define the set of mutant models M as follows:

$$M = \{(T \setminus \{T_i\}) \cup \{m\} \mid T_i \in T, m \in f_m(T_i)\}$$

and then define the mutation score for property P in the standard way:

Definition 3. Generalized mutation coverage.

$$\textsc{Mutant-Cov} = \frac{|\{m \mid m \in M \ \wedge \ (I, m) \nvdash P\}|}{|M|}$$

Note that without loss of generality, we consider a single property P, which can be viewed as the conjunction of all the properties of the model.

In our example in Figure 1, applying the software mutations from [3] would involve manipulating the constants used in the definitions of `a1_below`, `a2_below`, `a1_above`, `a2_above`, swapping `or` and `and` in the definition of `below`, `above_hyst`, or negating the conditions in the if/then/else statements. Even for this small model, there are many possible mutations (57). The number of single-mutation programs is roughly the product of the leaf elements of the program abstract syntax tree (AST) and the size of the chosen set of mutations, which can lead to an impractical number of verification problems.

Mutations for hardware are discussed in [4], [24], [25]. The state of the art of mutation-based coverage can be found in [4], where a design is considered as a net-list with nodes of types {`AND, INVERTER, REGISTER, INPUT`}. Each mutant design changes the type of a single node to `INPUT`. When property ϕ satisfied by the original net-list fails on the mutant design, it is said that a mutant is discovered for ϕ. Then, the coverage metric for ϕ is defined as the fraction of the discovered mutants, based on which the coverage of a set of properties is measured as the fraction of mutants discovered by at least one property. To decrease the cost of computation, coverage analysis is performed at several stages; first, all the nodes that do not appear in the resolution proof of a given property are marked as *not-covered*, and the rest of the nodes are marked as *unknown*. Then, for the unknown nodes, the basic mutation check is performed: if a corresponding mutant design violates the property, it will be considered as *covered*.

IV. Proof-Based Metrics

We propose a new approach for measuring property completeness based on proof rather than mutation. We first define notation, then describe different possible metrics given a set of *minimal proofs*.

Definition 4. IVC coverage (IVC-Cov):
Given $S \in AIVC(P)$, T_i is covered by P via S iff $T_i \in S$.

We call Definition 4 a *proof-preserving* metric because, with a set of the model elements marked as covered by IVC-Cov, P is provable. Other notions, as will be discussed in Section IV-A, may yield subsets of the model that are insufficient to reconstruct the proof of the property. The coverage score for IVC-Cov can be calculated with:

$$\frac{|S|}{|T|}$$

Because P may have multiple *MIVC*s, IVC-Cov metric can lead to various scores that belong to the following set:

$$\{ \ \frac{|S|}{|T|} \mid S \in AIVC(P) \ \}$$

Note that if an *MIVC* contains all model elements (i.e., the model is *completely covered*), then there is only one possible *MIVC*, so in this case there is no diversity of scores.

Given *all* proofs of a particular property, it is possible to define additional, complementary coverage notions. To do so, we use the following categorization of the model elements based on *MIVC* and $AIVC$ relations for P:

- $MUST(P) = \bigcap AIVC(P)$
- $MAY(P) = (\bigcup AIVC(P)) \setminus MUST(P)$
- $IRR(P) = T \setminus (\bigcup AIVC(P))$

This categorization helps to identify the role and relevance of each design element in satisfying a property. Function $MUST$ specifies the parts of the model absolutely necessary for the property satisfaction. Any change to these parts will affect provability of the property. On the other hand, any single element in $MAY(P)$, may be modified without affecting satisfaction of P(though modifying multiple elements may require re-proof). The IRR denotes model elements that are irrelevant to the validity of P [29]. Using the notions of MAY and $MUST$, we can introduce additional coverage metrics.

Definition 5. (May-Cov): $T_i \in T$ *is covered by P* iff $T_i \in$ May-Cov(P), *where* May-Cov$(P) = \{T_i \mid \exists S \in AIVC(P) \cdot T_i \in S\}$.

Definition 6. (MUST-COV): *$T_i \in T$ is covered by P* iff *$T_i \in$ MUST-COV(P), where* MUST-COV$(P) = \{T_i \mid \forall S \in AIVC(P) \cdot T_i \in S\}$.

The MAY-COV notion aims to deal with the fact that a property P may have several distinct *MIVC*s. In such cases, IVC-COV only looks at an arbitrary *MIVC* that may contain a subset of $MAY(P)$, which means, depending on which *MIVC* it considers, every time it may report a different part of $MAY(P)$ as uncovered. However, MAY-COV resolves this issue reporting the entire set of $MAY(P)$ as covered, which also leads to higher coverage scores. MUST-COV takes the opposite view, considering a model element as covered only if it affects all the proofs of P.

It is still possible to build more relaxed coverage metrics in which coverage is captured by looking at individual properties, rather than their conjunction. We can, for example, describe a metric in which any element used by an *MIVC* for any property is considered covered. The next definition, MODEL-COV, formalizes this notion.

Definition 7. (MODEL-COV): *Given a set of properties Δ over T, $T_i \in T$ is covered* iff *$T_i \in$ MODEL-COV(T), where* MODEL-COV$(T) = \{T_i \mid \exists P \in \Delta,\ S \in AIVC(P).\ T_i \in S\}$.

A. Discussion

Based on the categorization of elements, we will state some relationships about *MIVC*s so to compare different proof-based metrics proposed earlier.

Lemma 1. *If $MAY(P) \neq \varnothing$, then P is not provable by $MUST(P)$.*

Now we focus on the relationship between non-deterministic mutation-based coverage and proof-based metrics. In Chockler et. al. [4], each mutant design changes the type of a single node to `INPUT`. Given a suitable encoding of the netlist, assigning a "fresh" input is an isomorphic operation to simply removing a T_i from T. The mapping is as follows: the netlist becomes a conjunction of equations, where each vertex becomes a variable $v_i \in U$, and where each non-input vertex becomes an assignment equation $T_i \in T$. For example, given an `AND`-vertex v_i with three input edges from other vertexes $\{v_a, v_b, v_c\}$, we would define an equation $T_i \in T$ of the form $(v_i = (v_a \wedge v_b \wedge v_c))$.

Given this encoding, we can reframe the non-deterministic coverage proposed in [4] as follows:

Definition 8. Nondeterministic coverage (alternate specification) (NONDET-COV*) [4]. *$T_i \in T$ is covered by property P* iff *$T_i \in$ NONDET-COV$^*(P)$, where* NONDET-COV$^*(P) = \{T_i \mid (I, T) \vdash P \wedge (I, T \setminus \{T_i\}) \nvdash P\}$.

Given this definition, it becomes straightforward to define some additional properties.

Lemma 2. $T_i \in$ NONDET-COV$^*(P) \Leftrightarrow T_i \in$ MUST-COV(P).

In light of Lemma 2, the NONDET-COV* coverage score of specification P can be also calculated by

$$\frac{|MUST(P)|}{|T|}$$

Immediate from Lemmas 1 and 2, NONDET-COV* is not proof-preserving, while IVC-COV is proof-preserving by definition. Note that one can define many more proof-based coverage metrics based on the *MIVC*$/AIVC$ idea. Metrics that make use of the $AIVC$ relation are computationally more expensive to compute than IVC-COV although they might be easier to satisfy (i.e., result in higher coverage scores). In general, when coverage scores of a given property are obtained from the proposed metrics, the following relationship holds:

NONDET-COV* $\leq$ IVC-COV $\leq$ MAY-COV $\leq$ MODEL-COV

IVC-COV and NONDET-COV* are equivalent when all elements within the model are covered: if all model elements are MUST elements, then there can only be one *MIVC*, and this *MIVC* uses all of the model elements. In the implementation and experiments, we will focus on the IVC-COV and NONDET-COV* metrics. Both metrics are fairly rigorous and can be computed reasonably efficiently. The equivalence of MUST-COV and NONDET-COV* allows us to compare our algorithms against state-of-the-art mutation based coverage.

V. ILLUSTRATION

We illustrate the idea of completeness metrics and the "score" from different metrics on our altitude switch (ASW) example from Section II. The ASW is responsible for turning on and off a device of interest, so we formulate two requirements that describe when the ASW should be *on* and when it should be *off*. The first attempt at formalization (property set 1) is as follows:

```
on_p = (a1_below and a2_below) and not inhibit =>
    doi_on = true;
off_p = (a1_above and a2_above) and inhibit =>
    doi_on = false;
all_p = on_p and off_p;
```

For each of the IVC-COV, MAY-COV, and MUST-COV metrics, `all_p` only requires {`below, d1, doi_on`}. This small set of elements is due to a classic specification problem: using computed variables as the antecedents of implications. We therefore modify our properties to use inputs and constants as antecedents and derive:

```
on_p = ((alt1 < THRESHOLD) and (alt2 < THRESHOLD))
   and not inhibit => doi_on = true;
off_p = ((alt1 >= T_HYST) and (alt2 >= T_HYST))
   and inhibit => doi_on = false;
```

In this version, distinctions emerge between the metrics. `all_p` has two *MIVC*s: {{`a1_below, below, doi_on, d1`}, {`a2_below, below, doi_on, d1`}}, because of the `on_p` property: in the implementation, the DOI is turned on when either of the altimeters is below the threshold, while our property states that they both must be below. The MUST elements remain the same: {`below, doi_on, d1`}, because neither `a1_below` or `a2_below` is required for all proofs, and the MAY elements contain both `a1_below` and `a2_below`.

The `above_hyst`, `a1_above`, `a2_above`, and `d2` equations are still missing, meaning that the "above" thresholds are irrelevant to our properties. We realize we are missing a hysteresis case, and end up with the following set of properties:

```
on_p = ((alt1 < THRESHOLD) and (alt2 < THRESHOLD))
   and not inhibit => doi_on = true;
off_p = ((alt1 >= T_HYST) and (alt2 >= T_HYST))
   or inhibit => doi_on = false;
hyst_p = not inhibit and
         (alt1 > THRESHOLD and alt2 > THRESHOLD) and
         (alt1 < T_HYST or alt2 < T_HYST) =>
   (doi_on = false -> doi_on = pre(doi_on))
all_p = on_p and off_p and hyst_p;
```

The hysteresis property states that if the antecedent conditions hold, then in the initial state, the `doi_on` variable is assigned false, and in subsequent steps, it retains the same value as it previously had. This property set, which includes all variables for all measures, can reasonably be seen as describing the interesting behavior of the model.

VI. Implementation and Initial Results

For implementation we have made use of the technique proposed by Ghassabani et al. [13] using the Algorithm IVC_UC that computes an *approximately minimal* IVC in an efficient way.[1] To compute the *must* set of a given property as efficiently as possible, we use the algorithm from [4], adapted to use the more scalable PDR [11] algorithm rather than interpolation for model checking. We call this algorithm Must-Cov. These algorithms have been implemented in the `JKind` model checker [1], [9], [12]. `JKind` proves safety properties using multiple cooperative engines in parallel including k-induction [32], property directed reachability (PDR) [11], and template-based lemma generation [22].

To examine the efficiency and effectiveness of our approach, we ran the Must-Cov and the IVC-Cov metric against a benchmark suite consisting of 475 Lustre models (395 from [16] and 80 industrial models derived from [28] and other sources). Most of the benchmark models from [16] are small (10kB or less, with 6-40 equations) and the industrial models each are $\geq$80kB with over 600 equations. [2] The overhead of the IVC_UC algorithm is, on average, 24% over the baseline proof, as opposed to the 2369% runtime overhead for the Must-Cov computation. In many cases, the IVC-Cov metric is efficient enough to run as part of the verification process.

To examine effectiveness, we examined the portion of the model required by each of the coverage metrics. Any proposed metric should be justified and yield meaningful information about the adequacy of the property set. As a baseline, we use backwards static slicing, which is used in the hardware verification community for measuring adequacy. Our claim is that slicing is *too loose* a metric to be useful. This claim was justified in our benchmarks, where coverage reported by static slicing is, on average, 20.5x higher than the IVC-Cov metric.

[1]For complete information, see Ghassabani et. al [13].

[2]Tools and complete presentation of results are accessible from [2].

Our metrics have a justification from proof: for each of the metrics, any model element returned must be necessary to *some* proof. However, there are differences between the reported coverage of the metrics. On average, the IVC-Cov metric yields 1.15x higher coverage than the Must-Cov metric. This might lead one to prefer the Must-Cov metric, but it is more expensive to compute and the resulting set of elements does not always lead to a proof. Additional study is necessary to determine the best circumstances for use of each metric.

VII. Related work

Coverage in verification was introduced in [21], [23]. Hoskote et al. [21] suggested a state-based metric in model checking based on FSM mutations, which are small atomic changes to the design. Later in [6], Chockler et al. provided corresponding notions of metrics used in simulation-based verification for formal verification. However, the proposed algorithm in [6] is both computationally expensive (approximately linear in the number of mutations). Most of the mutation-based metrics, including [5], [25], are focused on finite state systems and hardware systems. A more recent work in [4] performs coverage analysis through interpolation [26]. This work is also based on design-dependent mutations [6], as explained in III-B. The proposed algorithm is similar to the IVC_MUST algorithm as discussed in IV-A except that our implementation uses PDR and k-induction rather than interpolation.

A similar notion to ours was outlined in a patent [18], which sketches a family of *proof core*-based metrics for hardware verification. While the approach described by the patent is general, no formal description of the models, metrics, algorithms, or experimental results are provided.

VIII. Conclusions & Future Work

In this paper, we have examined the use of proof-based coverage notions for formal verification. These provide an alternate way of measuring property completeness, which, in some cases, are much more efficient than existing work in the literature while still providing accurate information about the covered parts of a given design. Most of our proposed metrics preserve provability, which means that the set of elements considered covered by our algorithm is sufficient to establish the validity proof for every requirement in the set of specifications. The notion of proof preservation is appealing because it allows a concrete demonstration to the user of the irrelevance of portions of the implementation. The utility of the proposed metrics are being evaluated by Rockwell Collins on a pilot project. The proposed metrics appear to be useful for both traceability and adequacy checking [33].

Since some of the proposed metrics need to compute all IVCs, we have investigated efficient algorithms for computing all IVCs [14]. In cases where there are multiple minimal satisfying sets, the all IVCs give insight on multiple ways by which the model meets a requirement.

REFERENCES

[1] JKind. http://loonwerks.com/tools/jkind.html.

[2] Tools and Experimental Results. https://github.com/elaghs/Working/tree/master/ASE17.

[3] J. Andrews et al. Using mutation analysis for assessing and comparing testing coverage criteria. *TSE*, 2006.

[4] H. Chockler et al. Coverage in interpolation-based model checking. In *DAC*.

[5] H. Chockler et al. A practical approach to coverage in model checking. In *CAV01*, 2001.

[6] H. Chockler et al. Coverage metrics for formal verification. *CHARME*, 2003.

[7] K. Claessen. A coverage analysis for safety property lists. In *FMCAD'07*, 2007.

[8] K. Claessen and N. Srensson. A liveness checking algorithm that counts. In *FMCAD*, pages 52–59. IEEE, 2012.

[9] D. D. Cofer et al. Compositional verification of architectural models. In *NFM'12*, 2012.

[10] S. Das et al. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In *VLSID*.

[11] N. Een et al. Efficient implementation of property directed reachability. In *FMCAD '11*, 2011.

[12] A. Fifarek et al. Spear v2.0: Formalized past LTL specification and analysis of requirements. In *NFM'17*, 2017.

[13] E. Ghassabani et al. Efficient generation of inductive validity cores for safety properties. In *FSE2016*, 2016.

[14] E. Ghassabani et al. Efficient generation of all minimal inductive validity cores. In *FMCAD'17*, 2017.

[15] D. Grosse et al. Estimating functional coverage in bounded model checking. In *DATE'07*, 2007.

[16] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with smt-based techniques. In *FMCAD '08*, 2008.

[17] N. Halbwachs et al. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 1991.

[18] Z. Hanna et al. Formal verification coverage metrics for circuit design properties, May 14 2015. US Patent App. 14/474,280.

[19] D. Hardin et al. Development of security software: A high-assurance methodology. In *ICFEM'09*, 2009.

[20] M. P. Heimdahl et al. Deviation analysis via model checking. In *ASE'02*, 2002.

[21] Y. Hoskote et al. Coverage estimation for symbolic model checking. In *DAC*.

[22] T. Kahsai et al. Instantiation-based invariant discovery. In *NFM'11*, 2011.

[23] S. Katz et al. Have i written enough properties?-a method of comparison between specification and implementation. In *CHARME*, 1999.

[24] O. Kupferman. Sanity checks in formal verification. In *CONCUR'06*, 2006.

[25] O. Kupferman et al. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *FMCAD'08*, 2008.

[26] K. L. McMillan. Interpolation and sat-based model checking. In *CAV'03*.

[27] S. P. Miller et al. Software model checking takes off. 2010.

[28] A. Murugesan et al. Compositional verification of a medical device system. In *HILT'13*, 2013.

[29] A. Murugesan et al. Complete traceability for requirements in satisfaction arguments. In *RE@Next*, 2016.

[30] RTCA/DO-178C. Software considerations in airborne systems and equipment certification.

[31] J. Rushby. Software verification and system assurance. In *SEFM'09*.

[32] M. Sheeran et al. Checking safety properties using induction and a SAT-solver. In *FMCAD*, 2000.

[33] L. Wagner. Automating DO-178C objectives with SpeAR: A case study. http://www.mys5.org/Proceedings/2017/Day_1/2017-S5-Day1_1335_Wagner.pdf, Safe and Secure Systems and Software Symposium (S5), AFRL, 2017.