Specification quality metrics based on mutation and inductive incremental model checking

Vassil Todorov[1,2], Safouan Taha[2] and Frédéric Boulanger[2]

[1] Groupe PSA, 78140 Vélizy-Villacoublay, France
[2] Université Paris-Saclay, CNRS, CentraleSupélec, LRI, 91405 Orsay, France.

Abstract. When using formal verification on Simulink or SCADE models, an important question about their certification is how well the specified properties cover the entire model. A method using unsatisfiable cores and inductive model checking called IVC (Inductive Validity Cores) has been recently proposed within modern SMT-based model checkers such as JKind. The IVC algorithm determines a minimal set of model elements necessary to establish a proof and gives back the traceability to the design elements (lines of code) necessary for the proof. These metrics are interesting but are rather coarse grain for certification purposes.
In this paper, we propose to use mutation combined with incremental inductive model checking to give more precision and quality to the traceability process and look inside the lines of code. Our algorithm, based on the result of IVC, mutates the source code to determine which parts inside a line of code have an impact on the properties (killed mutants) and which parts have no impact on the properties (survived mutants). Furthermore, using the incremental feature present in modern SMT-solvers, we observe that mutation can scale up to industrial models. We demonstrate the metrics first on a simple example, then on a complex industrial program and on the JKind benchmark.

Keywords: Formal verification · Model-based mutation · Incremental inductive model checking · Model coverage · Symbolic model checking

1 Introduction

Today, most of the embedded application software in the automotive industry is developed using model-based design tools such as Simulink or SCADE. This paradigm of using a model brings a higher level of abstraction compared to the code and has the possibility to automatically generate the final code. A system designer creates a model by dragging and dropping blocks from a library and simulates its behavior to check if it corresponds to what is expected.

For the development of critical systems, it has been argued that formal proof should be applied to gain higher confidence than with testing only [17], [20], [23]. Even if these tools can prove formal properties on the model, this feature is not well understood and used by the designers. Actually, specifying properties (based on the requirements specification) within the aforementioned tools is not more

complicated than designing the model itself because they are written with the same library blocks as the model. However, there are two main problems: the proof process does not always terminate, and when a property is proved to be valid, no further information is provided about its coverage. For certification of critical software, we should be able to measure quality and exhaustiveness of the proved properties.

For the first problem, we worked on the improvement of the invariant generation used in most of the modern symbolic model checkers and implemented it in JKind [11] to improve the provability of properties involving time. Actually, proving properties involving time is rather challenging when they involve long durations and timers. These properties are generally not inductive and even advanced techniques such as PDR/IC3 [5] are unable to handle them on production models in reasonable time. We proposed an algorithm [26] and a new methodology using physical types (speed, acceleration, etc.), which restricted the number of candidates to only those that made sense and thus outperformed the JKind and Kind2 model checkers.

The second problem is important in the sense that even if the model checker has proved all the properties to be valid, we cannot answer the question about whether our model contains features that are not covered by the properties. Unlike testing, where we can follow the execution trace, the proof process uses the whole model, but many parts of it may not be necessary to prove the properties. This problem has been studied using the following approaches: mutation proof [7], [9], [16], [24] and inductive validity cores [2], [3], [4], [12].

The mutation approach shown in Figure 1 consists in mutating a model for which safety properties were proved valid, and trying to prove the same properties on the mutated models (*mutants*) again. If they are proved to be valid (the mutant has *survived*), the mutant reveals a part of the model that is not covered by the properties. It can also be dead code that will never be accessed. The algorithms used to compute coverage in the aforementioned papers can under-approximate which parts of the model are necessary to prove the properties and tend to be computationally very expensive because there are many mutated models to be verified.

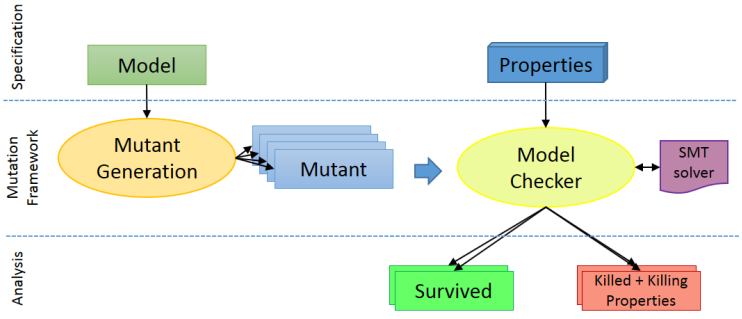


Fig. 1. Mutation proof framework

Inductive validity cores (IVCs) represent minimal sets of model elements necessary to construct inductive proofs for the specified properties. The algorithms proposed in the articles cited above are based on the *Unsatisfiable Core* support built into current SMT solvers. They can efficiently generate over-approximated inductive validity cores or exhaustively compute minimal ones. The authors show that calculating IVCs is more efficient than classical state of the art mutation. Calculating IVCs gives the coverage of properties in terms of lines of code of the model, which is more precise than a simple syntactic slicing, but does not look inside the lines of code and therefore does not consider the coverage of elementary operations inside an equation.

In this paper, we propose to go further in the precision of the coverage and zoom into the lines of code. Actually, a property can be covered by a line of code but inside the line there may still be some code that has no impact on the property. We argue that it is inside the lines of code that some subtle bugs can still subsist, and it is useful to uncover them. We use mutation to mutate some operators of the model, and symbolic model checking combined with induction-based techniques (k-induction [25], IC3/PDR [5], [10]), and take advantage of the incremental query capabilities of modern SMT solvers. We observed that mutation-based coverage for model checking is no longer out of reach, and this technique scales with our industrial use cases. We implemented this algorithm in the JKind open-source model checker [11], which is based on the Lustre [6] formal language. Lustre is used as base language for SCADE, so we could transform a SCADE model into Lustre. Simulink can also be transformed into Lustre using the CoCoSim framework developed at NASA Ames[3].

2 Preliminaries

In this section, we introduce the architecture of the industrial inductive model checker JKind [11] which is representative of other model checkers such as Kind2 and PKind.

2.1 The JKind Model Checker

JKind is an open-source industrial infinite-state model checker for safety properties. Models and properties are written in Lustre, a synchronous data-flow language, using theories of real and integer arithmetic. JKind uses SMT-solvers (SMTInterpol, Z3, Yices, CVC4, MathSAT) to prove or falsify the properties. It is structured as several parallel *engines* that cooperate to prove properties. Some engines are directly responsible for proving properties, some contribute to that effort by generating invariants, and others are for post-processing proofs or counterexample results. Each engine can be enabled or disabled separately. The architecture of JKind is shown in Figure 2. At the center of this architecture the **Director** allows any engine to broadcast information (invariants, valid and invalid properties) to the other engines.

[3] CoCoSim: `https://ti.arc.nasa.gov/tech/rse/research/cocosim`

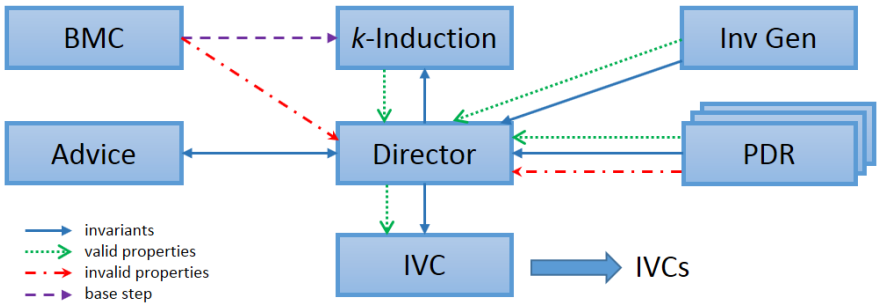


Fig. 2. The JKind model checker architecture

The **Bounded Model Checking (BMC)** engine performs a standard iterative unrolling of the transition relation to find counterexamples or to serve as the base case of k-induction. The BMC engine guarantees that any counterexample it finds is minimal in the number of steps from the initial state. The ***k*-Induction** engine performs the inductive step of k-induction, possibly using invariants generated by other engines. The **Invariant Generation** engine uses a template-based invariant generation technique [19] using its own k-induction loop. The **Property Directed Reachability (PDR)** engine performs property directed reachability [10] using the implicit abstraction technique [8]. Unlike BMC and k-induction, each property is handled separately by a different PDR sub-engine. The **Advice** engine saves invariants from previous runs of JKind and reuses them for new proofs to decrease the verification time.

A great effort was done in JKind on the post-processing of the results. We can cite the *Smoothing* counterexamples feature based on MaxSat which minimizes the number of changes to input variables. The other important post-processing feature is IVC.

Inductive Validity Cores (IVC). For a proven property, an inductive validity core is a subset of Lustre equations from the input model for which the property still holds [15], [13]. An IVC is *minimal* when no equation can be removed without breaking the provability. Depending on the model and property, there may exist several IVCs with different sizes. A *minimum* IVC has the smallest number of equations, and is not necessarily unique. Computing a minimum IVC is more difficult than computing any IVC, because it involves an exhaustive search. The IVC engine uses a heuristic algorithm to efficiently produce minimal IVCs but not minimum ones. As a side-effect, the IVC algorithm also minimizes the set of invariants used to prove a property, and shares this reduced set with other engines (notably the Advice engine).

2.2 IVC formalizations

In this section we re-use and adapt the formalization of IVC given by Ghassabani et al. in [14] to compare IVC to our mutation proof using similar definitions of coverage.

Models, Requirements and Provability. Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \to \mathit{bool}$ and a transition step predicate $T : U \times U \to \mathit{bool}$. A safety property $P : U \to \mathit{bool}$ is a state predicate that holds on a transition system (I, T) when it satisfies the following formulas:

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

When this is the case, we write $(I, T) \vdash P$.

Coming from the Lustre model that is a set of equations $\{eq_1 \dots eq_n\}$, the transition relation T has the structure of a top-level conjunction $T = t_1 \wedge \cdots \wedge t_n$ where each t_i is an equality corresponding to eq_i. By further abuse of notation, T is identified with the set of its top-level equalities. When an equation is removed from the Lustre model, an equality t_i is removed from T and the transition relation becomes $T \setminus \{t_i\}$.

Definition 1. Inductive Validity Core (IVC). *$S \subseteq T$ for $(I, T) \vdash P$ is an Inductive Validity Core, iff $(I, S) \vdash P \wedge \forall t_i \in S.\ (I, S \setminus \{t_i\}) \nvdash P$.*

As defined here, we are only interested in minimal sets that satisfy a property P. Note that given $(I, T) \vdash P$, P always has at least one *IVC*, which is not necessarily unique. For example, consider 2 boolean variables a and b initialized to true, *i.e.* $I = a \wedge b$, and assigned true at each step $T = (t_1 : a = \mathit{true}) \wedge (t_2 : b = \mathit{true})$. If $P = a \vee b$ then both $\{t_1\}$ and $\{t_2\}$ are *IVC*s. We note $AIVC(P)$ the set of all *IVC*s of P. Computing the $AIVC$ for each property, one gets a clear picture of all the model elements constrained by the property. The set $AIVC$ for all properties demonstrates a complete mapping from the requirements to the design elements, which is called *complete traceability* [21].

Property and Model coverage. The article by Ghassabani et al. [14] defines the two following metrics of coverage.

Definition 2. (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 3. (Must-Cov): *$t_i \in T$ is covered by P* iff *$t_i \in$ Must-Cov(P), where* Must-Cov$(P) = \{t_i \mid \forall S \in AIVC(P) \cdot t_i \in S\}$.

This categorization of coverage helps to identify the role and relevance of each design element in satisfying a property. Must-Cov specifies the parts of the model that are absolutely necessary for the property satisfaction. Any change to these parts will affect the provability of the property. On the other hand, May-Cov parts are relevant to the proof but may be modified without affecting the satisfaction of P. The May-Cov heuristic leads to higher coverage scores, because Must-Cov$(P) \subseteq$ May-Cov(P).

In JKind, the IVC engine computes one IVC and avoids exploring all possible ones. Therefore, it partially computes the May-Cov(P) and it does not handle Must-Cov(P).

Mutation. A mutator is a function that mutates any transition predicate T to a set of mutants $\{T^1_{mut}, \ldots, T^m_{mut}\}$, where each mutant T^i_{mut} is obtained by applying a small change to T.

A very simple mutator is the one that simply removes an equality t_i from T, which amounts to removing the corresponding line of code from the Lustre model. In our framework, we call this basic mutator `eq_remove` (see section 4). The authors of [15] only consider this simple mutator and define the corresponding coverage as follows:

Definition 4. Mutation coverage (Mut-Cov) $t_i \in T$ *is covered by property* P iff $t_i \in$ Mut-Cov(P), *where* Mut-Cov$(P) = \{t_i \mid (I,T) \vdash P \wedge (I, T \setminus \{t_i\}) \nvdash P\}$.

An immediate corollary proved in [15] states that if an equation is covered by such a mutation, it is also covered by all IVCs and conversely:

Corollary 1. Mut-Cov(P) = Must-Cov(P).

The Mut-Cov metrics can be generalized to more advanced mutators. In section 4, we will show how the Mut-Cov metrics can be improved to give a very precise coverage inside each t_i detected within Must-Cov or May-Cov.

3 Model coverage techniques

An important question for the certification of safety-critical systems is whether the requirements and tests are covering the implementation. For example, in ISO 26262 [18], which is the functional safety standard for road vehicles, tests are derived from requirements. An argumentation of why the performed tests give sufficient coverage shall be provided. As the critical level increases, a more rigorous method for test coverage (statement, branch, MC/DC) is required. If complete coverage is not achieved, an analysis is performed to decide whether additional tests or/and requirements are needed to increase coverage. DO-178C [22] with its supplement DO-333 (Formal Methods) go further in offering the possibility to use formal methods in replacement of all structural coverage objectives (including heavyweight MC/DC), but arguments showing that coverage is achieved by the formal proof should then be provided, see Table 1.

In this section, we present different techniques for model coverage, going progressively from coarse-grained coverage to fine-grained coverage. We consider the application of these techniques to the domain of inductive symbolic model checking. We propose to use mutation-based proof, taking advantage of the possibility to request SMT solvers in an incremental way, in order to look inside the operators in a way MC/DC does for testing. We show that the performance of this technique is equivalent to IVC and therefore quite faster than state of the art mutation-based methods. To the best of our knowledge, this technique has never been studied. The closest related work on mutation-based proof does not use inductive model checking for software verification nor incremental SMT solving. The work of Chockler et al. [7] presents an algorithm to re-use previously computed inductive invariants and counterexamples to identify the parts

DO-178C Table A-7 Objective	DO-333 Table FM.A-7 Objective
1. Test procedures are correct.	FM1. Formal analysis cases and procedures are correct.
2. Test results are correct and discrepancies explained.	FM2. Formal analysis results are correct and discrepancies explained.
3. Test coverage of High Level Requirements (HLRs) is achieved.	FM3. Coverage of HLRs is achieved.
4. Test coverage of Low Level Requirements (LLRs) is achieved.	FM4. Coverage of LLRs is achieved.
5. Test coverage of software structure (modified condition/decision coverage) is achieved.	FM5 – FM8. Verification coverage of software structure is achieved.
6. Test coverage of software structure (decision coverage) is achieved.	(A single objective that replaces the four structural coverage objectives in DO-178C)
7. Test coverage of software structure (statement coverage) is achieved.	
8. Test coverage of software structure (data coupling and control coupling) is achieved.	
9. A verification of additional code, that cannot be traced to source code, is achieved.	FM9. Verification of property preservation between source and object code.
N/A	FM10. Formal method is correctly defined, justified, and appropriate.

Table 1. DO-333 accepts replacing MC/DC coverage by formal proof coverage

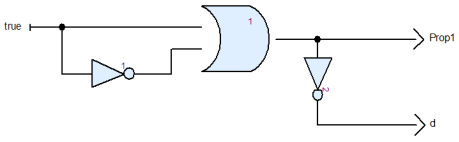
of a hardware system that are covered by a property. In [9], Claessen presents a coverage analysis based on LTL that gives the possibility to have underconstrained properties. In [16], the authors present an approach to estimate coverage in BMC (Bounded Model Checking). They generate coverage properties for each important signal for hardware verification purposes. Finally, in [24], Sayantan et al. present a method for determining the coverage of a formal LTL specification against a high-level fault model for hardware verification.

3.1 Simple running example

We use a simple running example to illustrate the difference between slicing, IVC and mutation proof. Consider the SCADE model shown both graphically and textually in Figure 3. The property `Prop1` we want to prove is the output of an `OR` block which takes a constant input equal to true and its negation. Obviously this property is always true. The Lustre code is obtained by using the SCADE option "Convert to textual" and we just add the comment on line 11 to tell JKind which output represents our safety property to be proved (invariant that shall always be true).

3.2 Slicing

The backward static slicing (or slicing for short) is a coarse-grained technique that allows to remove the parts of the code that do not affect the properties to be proved. It works by simply calculating the dependency graph for the variables used in the properties. Modern inductive model checkers use slicing to reduce the size of the queries sent to the SAT/SMT solver. It is interesting to see how much of the code is removed and to check if we really need this code or if our




```
node demo () returns (Prop1: bool; d: bool);
var
    L1, L2, L3, L4: bool;
let
    L1 = L2 or L3;
    L2 = true;
    L3 = not L2;
    L4 = not L1;
    Prop1 = L1;
    d = L4;
    --%PROPERTY Prop1;
tel
```

Fig. 3. A simple running example in Lustre

properties are simply not complete enough. After slicing, `d` and `L4` are removed and we obtain the lines:

```
    L1 = L2 or L3;
    L2 = true;
    L3 = not L2;
    Prop1 = L1;
```

3.3 Inductive Validity Cores (IVCs)

IVCs are much smaller and more precise than static slicing. For our short example, the IVC engine will either remove the equation of `L3` because `L1` does not depend on it since `L2` is true, or it will keep the equation of `L3` and remove the equation of `L2` since the equation of `L1` is a tautology when we consider the equation of `L3`. When running IVC on `Prop1`, it turns out that we obtain the first inductive validity core: `{L1, L2}`

```
    L1 = L2 or L3;
    L2 = true;
```

3.4 A simple mutator for Must-Cov: equation remover

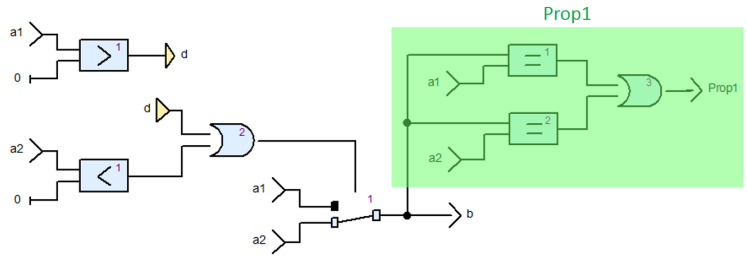
We want to go further than IVC, so we propose to use a simple mutator called "equation remover" which removes equations one by one and replays the proof process in an incremental way (using the SMT-LIB [1] *pop* and *push* commands). Our equation remover does not affect the properties because we want to mutate only the model and not the specification. If after removing an equation the

properties are still proved (surviving mutant), it means that the removed equation has no impact on the proof. If the properties do not hold anymore (killed mutant), this means that the removed equation is essential for the proof. This mutator computes the *minimum* core defined as MUST-COV in section 2, whereas IVC is working in MAY-COV mode. Using this technique, we obtain that only the equation of `L1` is essential for any proof of `Prop1` :

```
1     L1 = L2 or L3;
```

3.5 Using other mutators for deep coverage

We propose to add other mutation operators to zoom inside a line of code/equation and see what is covered by the properties. We explain these operators in detail in Section 4. For the moment, we give an example to see the difference between mutation and IVC. Our example is shown in Figure 4.




```
1  node demo2 (a1, a2: int)
2    returns (Prop1: bool; b: int);
3  var
4    d: bool;
5  let
6    d = (a1 > 0);
7    b = if d or (a2 < 0) then a1 else a2;
8    Prop1 = (b = a1 or b = a2);
9    --%PROPERTY Prop1;
10 tel;
```

Fig. 4. Example of inlined code and *if-then-else* operator mutations

This model takes two inputs `a1` and `a2`, and depending on whether their value is positive or negative, `a1` or `a2` is assigned to the output `b`. We have a property `Prop1` specifying that the output `b` should take the value of `a1` or `a2`. If slicing is applied to this model, it will remove nothing because `Prop1` depends statically on the entire model. However, applying IVC tells us that we should only keep `b` to cover our property `Prop1`. It is more precise than slicing because `d` is not necessary to prove that property (`b` is always equal to `a1` or `a2`).

Now, let us apply some mutations such as: replacing the condition of the *if* statement by *true* or *false*, replacing *or* by *xor*, replacing $>$ by $<$ etc. This leads to 22 possible mutations.

For `Prop1` we have 5 mutants killed out of 22. If we want to cover 100% of the code, we need to kill all mutants. To achieve this coverage, we need to strengthen our properties. We add a second property: `Prop2 = ((a1 > 0) => b = a1)`. At this stage IVC covers 100% of the model as `d` is now necessary to `Prop2`. However, only 14 mutants are killed out of 22, see Figure 5. For example, if the condition of the *if* statement at line 7 (Figure 4) is replaced by *true*, *Prop1* and *Prop2* are proved valid. This means that the condition has no impact on these properties. Let us add a third property: `Prop3 = ((a2 < 0) => b = a1)`. This time, we kill 16 mutants out of 22. Finally, we need a fourth property: `Prop4 = (((a1 <= 0) and (a2 >= 0)) => b = a2)` to kill all 22 mutants.

```
1   INDUCTIVE VALIDITY CORE: b, d
2   ++++++++++++++++++++++++++++++++++++++++++++++++++
3   MUTATION:
4   KILLED    at 6:3     equal_false            by [Prop2]
5   KILLED    at 6:3     equation_remove        by [Prop2]
6   KILLED    at 6:3     init_false             by [Prop2]
7   KILLED    at 6:11    g2l                    by [Prop2]
8   KILLED    at 6:13    (const int 0 -> 1)     by [Prop2]
9   KILLED    at 7:3     init_-1                by [Prop1, Prop2]
10  KILLED    at 7:3     equal_5                by [Prop1, Prop2]
11  KILLED    at 7:3     equal_-2               by [Prop1, Prop2]
12  KILLED    at 7:3     equation_remove        by [Prop1, Prop2]
13  KILLED    at 7:3     init_5                 by [Prop1, Prop2]
14  KILLED    at 7:7     ifelsethen             by [Prop2]
15  KILLED    at 7:7     ifelse                 by [Prop2]
16  KILLED    at 7:12    or2right               by [Prop2]
17  KILLED    at 7:12    or2xor                 by [Prop2]
18  SURVIVED  at 6:3     init_true
19  SURVIVED  at 6:3     equal_true
20  SURVIVED  at 6:11    g2ge
21  SURVIVED  at 7:7     ifthen
22  SURVIVED  at 7:12    or2left
23  SURVIVED  at 7:19    l2g
24  SURVIVED  at 7:19    l2le
25  SURVIVED  at 7:21    (const int 0 -> 1)
```

Fig. 5. IVCs and Mutation proof results on *demo2* for properties *Prop1* and *Prop2*

4 From Mutation testing to Mutation proof

Mutation testing is used to evaluate the quality of a test suite that is a set of test cases. It consists in modifying the program under test in small ways. Each mutated version of the program is called a *mutant* and test cases are replayed on it to detect whether its behavior is different from the behavior of the original version. This process is called 'killing the mutant'. The more mutants are killed, the better are the test cases. The quality of a test suite is measured as the

percentage of killed mutants. Mutants that are left can be killed by specifying additional test cases or justified as equivalent to the original program. *Mutators* are mutation operators used to generate mutants, and they tend to mimic standard programming errors. A mutation builds a mutant by applying a mutator on some position in the code. Taking ideas from mutation testing, we developed a *mutation proof* framework for standard inductive model checking using incremental SMT solving. In this section, we present our mutators and describe our mutation proof algorithm.

4.1 Mutators

Our mutators directly modify the Lustre code. We implemented classical mutators, but more advanced ones may be easily added to our framework. We present our mutators in Table 2.

Mutator	**Description**
or2xor	OR is mutated to XOR
xor2implies	XOR is mutated to $\Longrightarrow$
implies2and	$\Longrightarrow$ is mutated to AND
and2or	AND is mutated to OR
or2left	$X\ OR\ Y$ is mutated to X
or2right	$X\ OR\ Y$ is mutated to Y
and2left	$X\ AND\ Y$ is mutated to X
and2right	$X\ AND\ Y$ is mutated to Y
rm_not	NOT is removed
eq2neq	$=$ is mutated to $\neq$
neq2eq	$\neq$ is mutated to $=$
g2ge	$>$ is mutated to $\geq$
ge2g	$\geq$ is mutated to $>$
l2le	$<$ is mutated to $\leq$
le2l	$\leq$ is mutated to $<$
g2l	$>$ is mutated to $<$
l2g	$<$ is mutated to $>$
ge2le	$\geq$ is mutated to $\leq$
le2ge	$\leq$ is mutated to $\geq$
plus2minus	$+$ is mutated to $-$
minus2plus	$-$ is mutated to $+$
rm_minus	$-$ is removed
ifthen	IF condition is replaced by TRUE
ifelse	IF condition is replaced by FALSE
ifelsethen	THEN and ELSE statements are reversed
ConstantMutator	Constant is replaced by 1
eq_remove	Removes an entire equation/line of code

Table 2. Mutators for deep coverage measurement

Our first category of mutators are the *boolean mutators*. For example, the *and2or* mutator transforms `a AND b` into `a OR b`. Then we have *relational mutators* such as *ge*2*le*, which transforms a $\geq$ operator into $\leq$. We also have some *arithmetic mutators* such as *plus*2*minus*, which replaces $+$ by $-$. *Branching mutators* act on *if-then-else* statements replacing the condition by TRUE or FALSE or reversing the THEN and ELSE statements. Finally, we have the *constant mutator* that replaces all constants by 1, and the *equation remover mutator* that removes an entire line of code as seen before.

4.2 Our contribution: Mutation proof algorithm

The main contribution of our paper is the mutation proof algorithm that can be applied to modern inductive model checkers. It takes as input the proved properties and the invariants found during the proof process. It uses BMC and k-induction to retry the proof on mutants. Then, it returns a verdict: *KILLED* (proof fails with a counterexample), *SURVIVED* (proof succeeds), or *UNKNOWN* (proof fails with no counterexample). Our quality metrics is the ratio of killed mutants over the total number of mutants. The more mutants are killed, the better is the quality of the specification, because the better is the coverage of the model by the properties in the specification.

Algorithm 1: Mutation proof algorithm

input : M, P
output: *report*

Prove $P : \{P_0, P_1 \ldots\}$ on M
$Invs \leftarrow$ invariants from the proof of P on M
$k_{proof} \leftarrow$ maximum k-depth for proving P on M

foreach *mutation LCM* **do**
  $M_{mut} \leftarrow MUTATE(M, LCM)$
  if $BMC((M_{mut}, \emptyset, \emptyset), P, k_{proof}) = SAT$ **then**
    $M_{SAT} \leftarrow getModel()$
    $report$ += **KILLED**(mut:LCM, KillingProps:$\{P_i \in P \mid M_{SAT} \vDash \neg P_i\}$)
  else
    $SI \leftarrow FilterInvs(Invs, M_{mut})$
    $UP \leftarrow \emptyset$
    $SP \leftarrow P$
    while $KIND((M_{mut}, SI, \emptyset), SP, k_{proof}) = SAT$ **do**
      $M_{SAT} \leftarrow getModel()$
      $UP = UP \cup \{P_i \in SP \mid M_{SAT} \vDash \neg P_i\}$
      $SP = P \setminus UP$
    if $SP = P$ **then**
      $report$ += **SURVIVED**(mut:LCM)
    else
      if $BMC((M_{mut}, SI, SP), UP, k_{kill}) = SAT$ **then**
        $M_{SAT} \leftarrow getModel()$
        $report$ += **KILLED**(mut:LCM, KillingProps:$\{P_i \in UP | M_{SAT} \vDash \neg P_i\}$)
      else
        $report$ += **UNKNOWN**(mut:LCM, SurvivingProps:SP)

Before describing our algorithm, let us define its variables and functions: P are the specification Properties, M is the original Model, M_{mut} is the current mutated Model (Mutant), LCM represents a mutation in the form Line:Column of code and Mutator, function $MUTATE(M, LCM)$ returns the mutant M_{mut} corresponding to LCM applied to M, KP are the Killing Properties, SI are the Surviving Invariants, SP are the Surviving Properties, UP are the Unknown Properties, k_{kill} is a parameter for maximum k-depth to kill a mutant, functions $BMC((Model, Invariants, ValidProperties), Prop, k)$ and $KIND(\dots)$ run respectively BMC and K-INDuction on a model together with its invariants and its valid properties to check new properties $Prop$ at depth k and answer $UNSAT$ (all $Prop$ are valid) or SAT (some of $Prop$ are not valid). When the answer is SAT, the function $getModel()$ gives the counterexample. Finally, function $FilterInvs(invariants, M_{mut})$ filters the $invariants$ of the original Model M using BMC and k-induction to find the ones that survive the mutation and are still invariants of the current mutant M_{mut}.

Starting from the proof of P on M which requires the generation of invariants $Invs$ and induction at depth k_{proof}, our algorithm applies a mutation LCM at each iteration to obtain a mutant M_{mut} and retries the proof of P on M_{mut}. It runs first BMC at depth k_{proof} to verify whether all properties in P hold on M_{mut} for the first k_{proof} steps. If it is not the case, the mutant M_{mut} is already killed by some properties in P reported within the verdict $KILLED$. When all properties in P hold, which means that the base step is valid, the algorithm will try the k-induction step after filtering the invariants $Invs$ of M to keep only those that are still valid for M_{mut}. When the k-induction step succeeds ($UNSAT$), all properties in SP are k-inductive and survive, otherwise we use the counterexample model to find the properties that are not k-inductive, add them to the unknown properties UP, and we try again the k-induction on the remaining properties $SP \setminus UP$. We add the non k-inductive properties to UP because they can be valid but may require a k-induction of a higher depth. The verdict is $SURVIVED$ when the k-induction succeeds at the first iteration and in this case all properties in P hold for M_{mut} (i.e. $P = SP$ and UP is empty). If UP is not empty, we run again BMC at maximum depth k_{kill} to try to kill the current mutant by any property from UP. If this last attempt to kill M_{mut} fails, we return the verdict $UNKNOWN$.

5 Implementation and initial results

5.1 Implementation

We implemented our algorithm on a GitHub fork of JKind[4]. Our algorithm, shown in Figure 6, runs as a separate engine (module) of JKind and starts at the end of the proof process. It retrieves the invariants and k_{proof} used for proving the properties and returns mutations verdicts.

[4] JKind with Mutation on GitHub: `https://github.com/v-todorov/jkind`

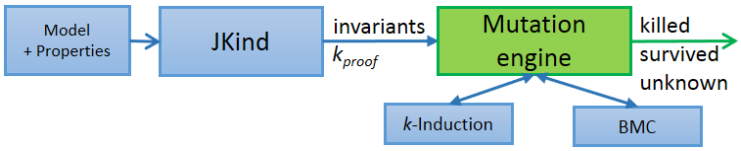


Fig. 6. Mutation engine implementation in JKind

5.2 Optimizations

Our implementation is very efficient because instead of submitting the entire mutated model to the SMT-solver it works in an incremental way, using *pop* and *push* only on the mutated lines. Furthermore, to take maximum advantage of this incremental feature, we group the mutations of the same line of code and run them all on the same SMT-solver instance.

We introduced two major optimizations as parameters in JKind: parallelMutants and ivcMutation. Firstly, unlike IVC, which cannot be parallelized, our mutation algorithm can run each mutation proof on a different thread. We group mutations that affect a given line of code. Different groups can be executed in parallel. The second optimization is intended for large models and runs the mutation only over the resulting minimal core produced by IVC. Thus IVC eliminates the unused part of the model, and mutation runs faster based on the results of IVC. The designer should me informed of the unused part in order to be able to write some additional properties about it.

5.3 Initial results

We used the benchmark of JKind (from GitHub), which provides Lustre files and properties to be proved. We selected 22 example Lustre files with only valid properties, because it is not useful to analyze the coverage of invalid properties. We used a laptop equipped with an Intel Xeon E-2176M CPU and 32GB RAM to run the benchmarks. We applied IVC alone, Mutation with equation removing only, and Mutation with all mutators activated. We activated the *parallelMutants* option to use the 6 cores of our CPU and we did not activate IVC when running Mutation. The results are shown in Figure 7. On the left, we see the results that compare Mutation with only the equation removing mutator (Mut-Eq) to IVC. We notice that in 82% of the use cases we obtained equal times for calculating IVC and Mut-Eq, in 9% of the cases mutation (Mut-Eq) was faster than IVC and in another 9% it was slower. For Mutation using all mutators (Mut-All), we had same execution times in 59% of the cases, mutation (Mut-All) was faster than IVC in 5% of the cases, and it was slower in 36% of the cases.

The unsat cores given by most SMT solvers are not necessarily minimal, IVC needs some backtracking to reduce them to minimal ones. The IVC implementation in JKind is sequential and requires calculation power. On the other hand, our algorithm runs in parallel and uses incremental SMT solving. Thus, we

obtain a greater coverage precision thanks to the mutation, with an equivalent performance most of the time.

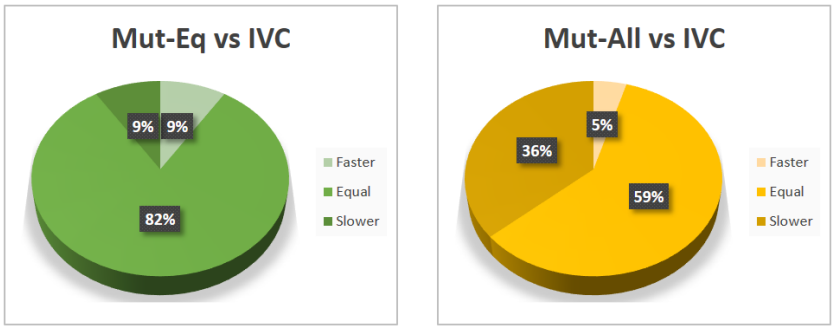


Fig. 7. Comparison between equation remover mutation/full mutation and IVC

5.4 Industrial use case results

We also used a representative industrial use case that is a cruise control function developed in SCADE (1250 lines of Lustre code), with some valid safety properties coming from high level requirements [26]. Using IVC, as well as using mutation with equation removing only, shows that all lines of code were covered and therefore necessary to the specification proof, but when running our mutation proof framework with all mutators activated, we only obtained 39% of killed mutations. This means that we need to strengthen the properties e.g. by adding additional ones to kill the 61% surviving mutations. In particular, we found some interesting mutations of *if-then-else* statements revealing branches that were not covered by the original properties.

6 Conclusions and Future work

In this paper we proposed a new coverage metrics for evaluating the quality of properties (specification) that are proved valid using model checking on a given model (program). The algorithm we used is particularly efficient unlike classical mutation testing techniques. Its efficiency comes from the fact that instead of submitting each mutant to the SMT solver, we only submit the original model once and we iteratively remove (pop) an equation and push its mutated version to check all mutants. The mutation process can also be run in parallel and thus its performance is almost equivalent to IVC, another heuristic algorithm to find the coverage of the properties on a model. The main advantage of our mutation framework over IVC is that we can look inside the lines of code and see the effect of mutating a constant, a variable or an operator.

As a future work, we will develop a link between invariant generation and mutation proof. It consists in finding parts of the code that are not covered by the automatically generated invariants and highlight them to give an immediate feedback to the designer who will need to strengthen the specification on that particular parts of the code. It will improve the provability of the specification and its quality.

References

1. Barrett, C., Stump, A., Tinelli, C., Boehme, S., Cok, D., Deharbe, D., Dutertre, B., Fontaine, P., Ganesh, V., Griggio, A., Grundy, J., Jackson, P., Oliveras, A., Krstić, S., Moskal, M., Moura, L.D., Sebastiani, R., Cok, T.D., Hoenicke, J.: The SMT-LIB Standard: Version 2.0. Tech. rep. (2010)
2. Bendík, J., Ghassabani, E., Whalen, M., Černá, I.: Online Enumeration of All Minimal Inductive Validity Cores. In: Johnsen, E.B., Schaefer, I. (eds.) Software Engineering and Formal Methods. pp. 189–204. Springer International Publishing, Cham (2018)
3. Bendík, J., Černá, I., Beneš, N.: Recursive Online Enumeration of All Minimal Unsatisfiable Subsets. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis. pp. 143–159. Springer International Publishing, Cham (2018)
4. Berryhill, R.: Chasing Minimal Inductive Validity Cores in Hardware Model Checking (Oct 2019)
5. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Aspects of Computing **20**, 379–405 (2008)
6. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: A Declarative Language for Real-time Programming. In: POPL '87. pp. 178–188. ACM (1987)
7. Chockler, H., Kupferman, O., Kurshan, R.P., Vardi, M.Y.: A Practical Approach to Coverage in Model Checking. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. pp. 66–78. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
8. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 Modulo Theories via Implicit Predicate Abstraction. CoRR **abs/1310.6847** (2013)
9. Claessen, K.: A Coverage Analysis for Safety Property Lists. In: Formal Methods in Computer Aided Design (FMCAD'07). pp. 139–145 (Nov 2007)
10. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. pp. 125–134. FMCAD '11, Austin (2011)
11. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind Model Checker. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 20–27. Springer, Cham (2018)
12. Ghassabani, E., Whalen, M., Gacek, A., Heimdahl, M.: Inductive Validity Cores. IEEE Transactions on Software Engineering pp. 1–1 (Jan 2019)
13. Ghassabani, E., Gacek, A., Whalen, M.W.: Efficient Generation of Inductive Validity Cores for Safety Properties. In: Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 314–325. FSE 2016, ACM, New York (2016)
14. Ghassabani, E., Gacek, A., Whalen, M.W., Heimdahl, M.P.E., Wagner, L.: Proof-based Coverage Metrics for Formal Verification. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. pp. 194–199. ASE 2017, IEEE Press, Piscataway, NJ, USA (Nov 2017), event-place: Urbana-Champaign, IL, USA

15. Ghassabani, E., Whalen, M., Gacek, A.: Efficient Generation of All Minimal Inductive Validity Cores. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 31–38. FMCAD '17, FMCAD Inc, Austin, TX (Nov 2017), event-place: Vienna, Austria
16. Große, D., Kühne, U., Drechsler, R.: Estimating Functional Coverage in Bounded Model Checking. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 1176–1181. DATE '07, EDA Consortium, San Jose, CA, USA (2007), event-place: Nice, France
17. Hardin, D., Hiratzka, T.D., Johnson, D.R., Wagner, L., Whalen, M.: Development of Security Software: A High Assurance Methodology. In: Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering. pp. 266–285. ICFEM '09, Springer-Verlag, Berlin, Heidelberg (2009), event-place: Rio de Janeiro, Brazil
18. ISO: Road vehicles – Functional safety (2011)
19. Kahsai, T., Garoche, P.L., Tinelli, C., Whalen, M.: Incremental Verification with Mode Variable Invariants in State Machines. In: Proceedings of the 4th International Conference on NASA Formal Methods. pp. 388–402. NFM'12, Springer-Verlag, Berlin, Heidelberg (2012)
20. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software Model Checking Takes off. Commun. ACM **53**(2), 58–64 (Feb 2010)
21. Murugesan, A., Whalen, M.W., Ghassabani, E., Heimdahl, M.P.E.: Complete traceability for requirements in satisfaction arguments. In: 2016 IEEE 24th International Requirements Engineering Conference (RE). pp. 359–364
22. RTCA DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Washington, DC (December 2011)
23. Rushby, J.: Software Verification and System Assurance. In: 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods. pp. 3–10 (Nov 2009)
24. Sayanlan Das, Ansuman Banerjee, Prasenjit Basu, Pallab Dasgupta, Chakrabarti, P.P., Chunduri Rama Mohan, Fix, L.: Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In: 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design. pp. 201–206 (Jan 2005)
25. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: FMCAD 2000, pp. 127–144
26. Todorov, V., Taha, S., Boulanger, F., Hernandez, A.: Improved invariant generation for industrial software model checking of time properties. In: Proceedings of the 19th IEEE International Conference on Software Quality, Reliability, and Security. pp. 334–341. IEEE, Sofia, Bulgaria (Oct 2019)