Efficient Generation of Inductive Validity Cores for Safety Properties

Elaheh Ghassabani
Department of Computer Science and Engineering
University of Minneapolis
200 Union Street
Minneapolis, MN, 55455, USA
ghass013@umn.edu

Andrew Gacek
Rockwell Collins
Advanced Technology Center
400 Collins Rd. NE
Cedar Rapids, IA, 52498, USA
andrew.gacek@rockwellcollins.com

Michael W. Whalen
Department of Computer Science and Engineering
University of Minneapolis
200 Union Street
Minneapolis, MN, 55455, USA
whalen@cs.umn.edu

ABSTRACT

Symbolic model checkers can construct proofs of properties over very complex models. However, the results reported by the tool when a proof succeeds do not generally provide much insight to the user. It is often useful for users to have traceability information related to the proof: which portions of the model were necessary to construct it. This traceability information can be used to diagnose a variety of modeling problems such as overconstrained axioms and underconstrained properties, and can also be used to measure *completeness* of a set of requirements over a model. In this paper, we present a new algorithm to efficiently compute the *inductive validity core* (IVC) within a model necessary for inductive proofs of safety properties for sequential systems. The algorithm is based on the UNSAT core support built into current SMT solvers and a novel encoding of the inductive problem to try to generate a minimal inductive validity core. We prove our algorithm is correct, and describe its implementation in the JKind model checker for Lustre models. We then present an experiment in which we benchmark the algorithm in terms of speed, diversity of produced cores, and minimality, with promising results.

CCS Concepts

•Theory of computation → Verification by model checking; Automated reasoning; •Software and its engineering → Requirements analysis; Formal software verification;

Keywords

Traceability, Requirements Completeness, k-Induction, IC3/PDR

1. INTRODUCTION

Symbolic model checking using induction-based techniques such as IC3/PDR [17] and k-induction [44] can often determine whether safety properties hold of complex finite or infinite-state systems. Model checking tools are attractive both because they are automated, requiring little or no interaction with the user, and if the answer to a correctness query is negative, they provide a counterexample to the satisfaction of the property. These counterexamples can be used both to illustrate subtle errors in complex hardware and software designs [33,35,38] and to support automated test case generation [48, 49]. In the event that a property is proved, however, it is not always clear what level of assurance should be invested in the result. Given that these kinds of analyses are performed for safety- and security-critical software, this can lead to overconfidence in the behavior of the fielded system. It is well known that issues such as vacuity [28] can cause verification to succeed despite errors in a property specification or in the model. Even for non-vacuous specifications, it is possible to over-constrain the specification of the *environment* in the model such that the implementation will not work in the actual operating environment.

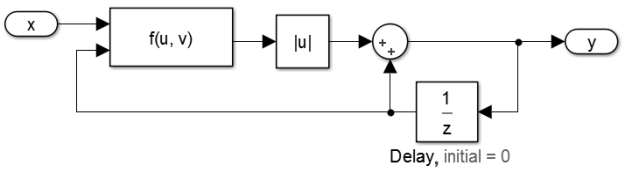
At issue is the level of feedback provided by the tool to the user. In most tools, when the answer to a correctness query is positive, no further information is provided. What we would like to provide is traceability information, an *inductive validity core* (IVC), that explains the proof, in much the same way that a counterexample explains the negative result. This is not a new idea: UNSAT cores [50] provide the same kind of information for individual SAT or SMT queries, and this approach has been lifted to bounded analysis for Alloy in [46]. What we propose is a generic and efficient mechanism for extracting supporting information, similar to an UNSAT core, from the proofs of safety properties using inductive techniques such as PDR and k-induction. Because many properties are not themselves inductive, these proof techniques introduce lemmas as part of the solving process in order to strengthen the properties and make them inductive. Our technique allows efficient, accurate, and precise extraction of inductive validity cores even in the presence of such auxiliary lemmas.

Once generated, the IVC can be used for many purposes in the software verification process, including at least the following:

Vacuity detection: The idea of syntactic vacuity detection (checking whether all subformulae within a property are necessary for its satisfaction) has been well studied [28]. However, even if a property is not syntactically vacuous, it may not require substantial portions of the model. This in turn may indicate that either a.) the model is incorrectly constructed or b.) the property is weaker than expected. We have seen several examples of this mis-specification in our verification work, especially when variables computed by the model are used as part of antecedents to implications.

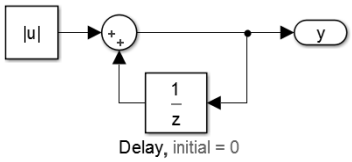
Completeness checking: Closely related to vacuity detection is the idea of *completeness checking*, e.g., are all atoms in the model necessary for at least one of the properties proven

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950346




```
node filter(x : real) returns (a, b, y : real);
let
  a = f(x, 0.0 -> pre y);
  b = if a >= 0.0 then a else -a;
  y = b + (0.0 -> pre y);
tel;
```

Figure 1: Model with property $y \geq 0$, before IVC analysis




```
node filter(x, a : real) returns (b, y : real);
let
  b = if a >= 0.0 then a else -a;
  y = b + (0.0 -> pre y);
tel;
```

Figure 2: Model with property $y \geq 0$, after IVC analysis

about the model? Several different notions of completeness checking have been proposed [9, 27], but these are very expensive to compute, and in some cases, provide an overly strict answer (e.g., checking can only be performed on non-vacuous models for [27]).

Traceability: Certification standards for safety-critical systems (e.g., [36, 41]) usually require *traceability matrices* that map high-level requirements to lower-level requirements and (eventually) leaf-level requirements to code or models. Current traceability approaches involve either manual mappings between requirements and code/models [32] or a heuristic approach involving natural language processing [25]. Both of these approaches tend to be inaccurate. For functional properties that can be proven with inductive model checkers, inductive validity cores can provide accurate traceability matrices with no user effort.

Symbolic Simulation / Test Case Generation: Model checkers are now often used for symbolic simulation and structural-coverage-based test case generation [31, 48]. For either of these purposes, the model checker is supposed to produce a witness trace for a given coverage obligation using a "trap property" which is expected to be falsifiable. In systems of sufficient size, there is often "dead code" that cannot ever be reached. In this case, a proof of non-reachability is produced, and the IVC provides the reason why this code is unreachable.

Nevertheless, to be useful for these tasks, the generation process must be efficient and the generated IVC must be accurate and precise (that is, sound and close to minimal). The requirement for accuracy is obvious; otherwise the "minimal" set of model elements is no longer sufficient to produce a proof, so it no longer meets our IVC definition. Minimality is important because (for traceability) we do not want unnecessary model elements in the trace matrix, and (for completeness) it may give us a false level of confidence that we have enough requirements.

In addition, we are also interested in *diversity*: how many different IVCs can be computed for a given property and model? Requirements engineers often talk about "the traceability matrix" or "the satisfaction argument". If proofs are regularly diverse, then there are potentially many equally valid traceability matrices, and this may lead to changes in traceability research.

In the remainder of this paper, we present an algorithm for efficient generation of IVCs for induction-based model checkers. Our contributions, as detailed in the remainder of the paper, are as follows:

- We present a technique for extracting inductive validity cores from an inductive verification of a safety property over a sequential model involving lemmas.
- We formalize this technique and present an implementation of it in the JKind model checker [1].
- We present an experiment over our implementation and measure the efficiency, minimality, and robustness of the IVC generation process.

The rest of this article is organized as follows. In Section 2, we present a motivating example. In Section 3, we present the required background for our approach. In Sections 4 and 5, we present our approach and our implementation in JKind. Sections 6 and 7 present an evaluation of our approach on a set of benchmark examples. Finally, Section 8 discusses related work and Section 9 concludes.

2. MOTIVATING EXAMPLE

Consider the model shown both graphically and textually in Figure 1. This model takes an input, combines it with the previous output in some way, takes the absolute value, and then adds this to an accumulating value. This model has the property that the output is always non-negative, i.e., $y \geq 0$. Moreover, it happens that this property holds regardless of the way the input is combined with the previous output, i.e., the function f in the model. Formally, we say that the minimal inductive validity core (IVC) does not contain that part of the model. The model reduced to a minimal IVC is shown in Figure 2. Note that traditional static dependency analysis (i.e., a *backward static slice*) would not be able to remove f from the original model. In our experiments in Section 7, we demonstrate that IVCs are much smaller and more precise than static slices.

3. PRELIMINARIES

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

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

A safety property $P : S \rightarrow \mathit{bool}$ is a state predicate. A safety property P holds on a transition system (I, T) if it holds on all reachable states, i.e., $\forall s.\ R(s) \Rightarrow P(s)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$.

$$I(s_0) \Rightarrow P(s_0)$$
$$\vdots$$
$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-2}, s_{k-1}) \Rightarrow P(s_{k-1})$$
$$P(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge P(s_{k-1}) \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k)$$

Figure 3: k-induction formulas: k base cases and one inductive step

For an arbitrary transition system (I, T), computing reachability can be very expensive or even impossible. Thus, we need a more effective way of checking if a safety property P is satisfied by the system. The key idea is to over-approximate reachability. If we can find an over-approximation that implies the property, then the property must hold. Otherwise, the approximation needs to be refined.

A good first approximation for reachability is the property itself. That is, we can check if the following formulas hold:

$$\forall s.\ I(s) \Rightarrow P(s) \tag{1}$$
$$\forall s, s'.\ P(s) \wedge T(s, s') \Rightarrow P(s') \tag{2}$$

If both formulas hold then P is *inductive* and holds over the system. If (1) fails to hold, then P is violated by an initial state of the system. If (2) fails to hold, then P is too much of an over-approximation and needs to be refined.

One way to refine our over-approximation is to add additional lemmas to the property of interest. For example, given another property $L : S \rightarrow \mathit{bool}$ we can consider the extended property $P'(s) = P(s) \wedge L(s)$, written as $P' = P \wedge L$ for short. If P' holds on the system, then P must hold as well. The hope is that the addition of L makes formula (2) provable because the antecedent is more constrained. However, the consequent of (2) is also more constrained, so the lemma L may require additional lemmas of its own. Finding and proving these lemmas is the means by which property directed reachability (PDR) strengthens and proves a safety property.

Another way to refine our over-approximation is to use use k-*induction* which unrolls the property over k steps of the transition system. For example, 1-induction consists of formulas (1) and (2) above, whereas 2-induction consists of the following formulas:

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

That is, there are two base step checks and one inductive step check. In general, for an arbitrary k, k-induction consists of k base step checks and one inductive step check as shown in Figure 3 (the universal quantifiers on s_i have been elided for space). We say that a property is k-inductive if it satisfies the k-induction constraints for the given value of k. The hope is that the additional formulas in the antecedent of the inductive step make it provable.

In practice, inductive model checkers often use a combination of the above techniques. Thus, a typical conclusion is of the form "P with lemmas $L_1, \ldots, L_n$ is k-inductive".

4. INDUCTIVE VALIDITY CORES

Given a transition system which satisfies a safety property P, we want to know which parts of the system are necessary for satisfying the safety property. One possible way of asking this is, "What is the most general version of this transition system that still satisfies the property?" The answer is disappointing. The most general system is $I(s) = P(s)$ and $T(s, s') = P(s')$, i.e., you start in any state satisfying the property and can transition to any state that still satisfies the property. This answer gives no insight into the original system because it has no connection to the original system. In this section we introduce the notion of *inductive validity cores* (IVC) which looks at generalizing the original transition system while preserving a safety property.

Algorithm 1: IVC_BF: Brute-force algorithm for computing a minimal IVC

input : $(I, T) \vdash P$
output: Minimal inductive validity core for $(I, T) \vdash P$

```
S ← T
for x ∈ S do
    if (I, S \ {x}) ⊢ P then
        S ← S \ {x}
return S
```

In order to talk about generalizing a transition system, we assume the transition relation of the system has the structure of a top-level conjunction. This assumption gives us a structure that we can easily manipulate as we generalize the system. Given $T(s, s') = T_1(s, s') \wedge \cdots \wedge T_n(s, s')$ we will write $T = T_1 \wedge \cdots \wedge T_n$ for short. By further abuse of notation we will identify T with the set of its top-level conjuncts. Thus we will write $x \in T$ to mean that x is a top-level conjunct of T. We will write $S \subseteq T$ to mean that all top-level conjuncts of S are top-level conjuncts of T. We will write $T \setminus \{x\}$ to mean T with the top-level conjunct x removed. We will use the same notation when working with sets of invariants.

Definition 1. Inductive Validity Core: Let (I, T) be a transition system and let P be a safety property with $(I, T) \vdash P$. We say $S \subseteq T$ is an *inductive validity core* for $(I, T) \vdash P$ iff $(I, S) \vdash P$. When I, T, and P can be inferred from context we will simply say S is an inductive validity core.

Definition 2. Minimal Inductive Validity Core: An inductive validity core S for $(I, T) \vdash P$ is minimal iff there does not exist $M \subset S$ such that M is an inductive validity core for $(I, T) \vdash P$.

Note that minimal inductive validity cores 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 minimal inductive validity cores for $(I, T) \vdash P$. However, inductive validity cores do have the following monotonicity property.

Lemma 1. Let (I, T) be a transition system and let P be a safety property with $(I, T) \vdash P$. Let $S_1 \subseteq S_2 \subseteq T$. If S_1 is an inductive validity core for $(I, T) \vdash P$ then S_2 is an inductive validity core for $(I, T) \vdash P$.

PROOF. From $S_1 \subseteq S_2$ we have $S_2 \Rightarrow S_1$. Thus the reachable states of (I, S_2) are a subset of the reachable states of (I, S_1). □

This lemma gives us a simple, brute-force algorithm for computing a minimal inductive validity core, Algorithm IVC_BF (1). The resulting set of this algorithm is obviously an inductive validity core for $(I, T) \vdash P$. The following lemma shows that it is also minimal.

Lemma 2. The result of Algorithm 1 is a minimal inductive validity core for $(I, T) \vdash P$.

Algorithm 2: IVC_UC: Efficient algorithm for computing a nearly minimal inductive validity core from UNSAT cores

input : P with invariants Q is k-inductive for (I, T)
output: Inductive validity core for $(I, T) \vdash P$

```
1 k ← MINIMIZEK(T, P ∧ Q)
2 R ← REDUCEINVARIANTS_k(T, Q, P)
3 return MINIMIZEIVC_k(I, T, R)
```

$$\text{BASEQUERY}_1(I, T, P) \equiv \forall s_0.\ I(s_0) \Rightarrow P(s_0)$$

$$\text{BASEQUERY}_{k+1}(I, T, P) \equiv \text{BASEQUERY}_k(I, T, P) \wedge (\forall s_0, \ldots, s_k.\ I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k))$$

$$\text{INDQUERY}_k(T, Q, P) \equiv (\forall s_0, \ldots, s_k.\ Q(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge Q(s_{k-1}) \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k))$$

$$\text{FULLQUERY}_k(I, T, P) \equiv \text{BASEQUERY}_k(I, T, P) \wedge \text{INDQUERY}_k(T, P, P)$$

Figure 4: k-induction queries

PROOF. Let the result be R. Suppose towards contradiction that R is not minimal. Then there is an inductive validity core M with $M \subset R$. Take $x \in R \setminus M$. Since $x \in R$ it must be that during the algorithm $(I, S \setminus \{x\}) \vdash P$ is not true for some set S where $R \subseteq S$. We have $M \subset R \subseteq S$ and $x \notin M$, thus $M \subseteq S \setminus \{x\}$. Since M is an inductive validity core, Lemma 1 says that $S \setminus \{x\}$ is an inductive validity core, and so $(I, S \setminus \{x\}) \vdash P$. This is a contradiction, thus R must be minimal. □

This algorithm has two problems. First, checking if a safety property holds is undecidable in general thus the algorithm may never terminate even when the safety property is easily provable over the original transition system. Second, this algorithm is very inefficient since it tries to re-prove the property multiple times.

The key to a more efficient algorithm is to make better use of the information that comes out of model checking. In addition to knowing that P holds on a system (I, T), suppose we also know something stronger: P with the invariant set Q is k-inductive for (I, T). This gives us the broad structure of a proof for P which allows us to reconstruct the proof over a modified transition system. However, we must be careful since this proof structure may be more than is actually needed to establish P. In particular, Q may contain unneeded invariants which could cause the inductive validity core for $P \wedge Q$ to be larger than the inductive validity core for P. Thus before computing the inductive validity core we first try to reduce the set of invariants to be as small as possible. This operation is expensive when k is large so as a first step we minimize k. This is the motivation behind Algorithm IVC_UC (2).

To describe the details of Algorithm 2 we define queries for the base and inductive steps of k-induction (Figure 4). Note, in $\text{INDQUERY}(T, Q, P)$ we separate the assumptions made on each step, Q, from the property we try to show on the last step, P. We use this separation when reducing the set of invariants.

We assume that our queries are checked by an SMT solver. That is, we assume we have a function $\text{CHECKSAT}(F)$ which determines if F, an existentially quantified formula, is satisfiable or not. In order to efficiently manipulate our queries, we assume the ability to create *activation literals* which are simply distinguished Boolean variables. The call $\text{CHECKSAT}(A, F)$ holds the activation literals in A true while checking F. When F is unsatisfiable, we assume we have a function UNSATCORE() which returns a minimal subset of the activation literals such that the formula is unsatisfiable with those activation literals held true. In practice, SMT solvers often return a non-minimal set, but we can minimize the set via repeated calls to CHECKSAT. We assume both CHECKSAT and UNSATCORE are always terminating.

Algorithm 3: $\text{MINIMIZEK}(T, P)$

```
1 k' ← 1
2 while CHECKSAT(¬INDQUERY_{k'}(T, P, P)) = SAT do
3     k' ← k' + 1
4 return k'
```

Algorithm 4: $\text{REDUCEINVARIANTS}_k(T, \{Q_1, \ldots, Q_n\}, P)$

```
1  R ← {P}
2  Create activation literals A = {a_1, ..., a_n}
3  C ← (a_1 ⇒ Q_1) ∧ ··· ∧ (a_n ⇒ Q_n)
4  while true do
5      CHECKSAT(A, ¬INDQUERY_k(T, C, R))
6      if UNSATCORE() = ∅ then
7          return R
8      for a_i ∈ UNSATCORE() do
9          R ← R ∪ {Q_i}
10         C ← C \ {a_i ⇒ Q_i}
```

The function $\text{MINIMIZEK}(T, P)$ is defined in Algorithm 3. This function assumes that P is k-inductive for (I, T). It returns the smallest k' such that P is k'-inductive for (I, T). We start checking at $k' = 1$ since smaller values of k' are much quicker to check than larger ones. The checking must eventually terminate since P is k-inductive. We also only check the inductive query since we know the base query will be true for all $k' \leq k$. Although we describe each query in Algorithm 3 separately, in practice they can be done incrementally to improve efficiency.

The function $\text{REDUCEINVARIANTS}_k(T, \{Q_1, \ldots, Q_n\}, P)$ is defined in Algorithm 4. This function assumes that $P \wedge Q_1 \wedge \cdots \wedge Q_n$ is k-inductive for (I, T). It returns a set $R \subseteq \{P, Q_1, \ldots, Q_n\}$ such that R is k-inductive for (I, T) and $P \in R$. Like MINIMIZEK, this function only checks the inductive query since each element of R is an invariant and therefore will always pass the base query. A significant complication for reducing invariants is that some invariants may mutually need each other, even though none of them are needed to prove P. Thus in Algorithm 4 we find a minimal set of invariants needed to prove P, then we find a minimal set of invariants to prove those invariants, and so on. We terminate when no more invariants are needed to prove the properties in R. Algorithm 4 is guaranteed to terminate since R gets larger in every iteration of the outer loop and it is bounded above by $\{P, Q_1, \ldots, Q_n\}$. As with Algorithm 3, we describe each query in Algorithm 4 separately, though in practice large parts of the queries can be re-used to improve efficiency.

This iterative lemma determination does not guarantee a minimal result. For example, we may find P requires just Q_1, that Q_1 requires just Q_2, and that Q_2 does not require any other invariants. This gives the result $\{P, Q_1, Q_2\}$, but it may be that Q_2 alone is enough to prove P thus the original result is not minimal. Also note, we do not care about the result of CHECKSAT, only the UNSATCORE that comes out of it. Since $P \wedge Q_1 \wedge \cdots \wedge Q_n$ is

Algorithm 5: MINIMIZEIVC$_k(I, \{T_1, \ldots, T_n\}, P)$

```
1 Create activation literals A = {a_1, ..., a_n}
2 T ← (a_1 ⇒ T_1) ∧ ··· ∧ (a_n ⇒ T_n)
3 CHECKSAT(A, ¬FULLQUERY_k(I, T, P))
4 R ← ∅
5 for a_i ∈ UNSATCORE() do
6     R ← R ∪ {T_i}
7 return R
```

k-inductive, we know the CHECKSAT call will always return UNSAT.

The function MINIMIZEIVC$_k(I, \{T_1, \ldots, T_n\}, P)$ is defined in Algorithm 5. This function assumes that P is k-inductive for (I, T). It returns a minimal inductive validity core $R \subseteq \{T_1, \ldots, T_n\}$ such that P is k-inductive for (I, R). It is trivially terminating. Since Algorithms 3, 4, and 5 are terminating, Algorithm 2 is always terminating.

Our full inductive validity core algorithm in Algorithm 2 does not guarantee a minimal inductive validity core. One reason is that REDUCEINVARIANTS does not guarantee a minimal set of invariants. A larger reason is that we only consider the invariants that the algorithm is given at the outset. It is possible that there are other invariants which could lead to a smaller inductive validity core, but we do not search for them. In Sections 6 and 7, we show that in practice our algorithm is nearly minimal and much more efficient than the naive algorithm. The following theorem shows that minimality checking is at least as hard as model checking and therefore undecidable in many settings.

Theorem 1. Determining if an IVC is minimal is as hard as model checking.

PROOF. Consider an arbitrary model checking problem $(I, T) \vdash^? P$ where P is not a tautology. We will construct an IVC for a related model checking problem which will be minimal if and only if $(I, T) \nvdash P$. Let x and y be fresh variables. Construct a transition system with initial predicate $I \wedge \neg x$ and transition predicate $(x' \Rightarrow y') \wedge ((y' \Rightarrow P') \wedge T)$. The constructed system clearly satisfies the property $x \Rightarrow P$. Thus $S = \{x' \Rightarrow y', (y' \Rightarrow P') \wedge T\}$ is an IVC. S is minimal if and only if neither $\{x' \Rightarrow y'\}$ nor $\{(y' \Rightarrow P') \wedge T\}$ is an IVC. Since x and y are fresh and P is not a tautology, $\{x' \Rightarrow y'\}$ is not an IVC. Since x and y are fresh, $\{(y' \Rightarrow P') \wedge T\}$ is an IVC for the property $x \Rightarrow P$ if and only if $(I, T) \vdash P$. Therefore, S is minimal if and only if $(I, T) \nvdash P$. □

When minimality is a necessity, we can combine IVC_BF and IVC_UC into a single algorithm which aims to efficiently guarantee minimality. The hybrid algorithm, IVC_UCBF, consists of running IVC_UC to generate an initial nearly minimal IVC which is then run through IVC_BF to guarantee minimality. The resulting algorithm is not guaranteed to terminate since IVC_BF is not guaranteed to terminate.

5. IMPLEMENTATION

We have implemented the inductive validity core algorithms in the previous section in two tools: *JKind*, which performs the IVC_UC algorithm, and *JSupport*, which can compute either the IVC_BF or the IVC_UCBF algorithm (using JKind as a subprocess). Moreover, our implementation of IVC_UCBF uses an additional feature of JKind to store and re-use discovered invariants between separate runs. This reduces some of the cost of attempting to re-prove a property multiple times. These tools operate over the Lustre language [22], which we briefly illustrate below.

5.1 Lustre and IVCs

Lustre [22] is a synchronous dataflow language used as an input language for various model checkers. The textual models in Figures 1 and 2 are written in Lustre. We will use model in Figure 1 as a running example in this section. For our purposes, a Lustre program consists of 1) input variables, `x` in the example, 2) output variables, `a`, `b`, and `y` in the example, and 3) an equation for each output variable. A Lustre program runs over discrete time steps. On each step, the input variables take on some values and are used to compute values for the output variables on the same step. In addition, equations may refer to the previous value of a variable using the `pre` operator. This operator is underspecified in the first step, so the arrow operator, `->`, is used to guard the `pre` operator. In the first step the expression `e1 -> e2` evaluates to `e1`, and it evaluates to `e2` in all other steps.

We interpret a Lustre program as a model specification by considering the behavior of the program under all possible input traces. Safety properties over Lustre can then be expressed as Boolean expressions in Lustre. A safety property holds if the corresponding expression is always true for all input traces. For example, the property for Figure 1 is `y >= 0`, which is a valid property.

It is straightforward to translate this interpretation of Lustre into the traditional initial and transition relations. We will show this by continuing with the example in Figure 1. First we introduce a new Boolean variable $init$ into the state space to denote when the system is in its initial state, the state of the system prior to initialization. In the initial state, all other variables are completely unconstrained which models the underspecification of the pre operator during the first step. Then we define,

$$
\begin{aligned}
&I((x, a, b, y, init)) = init \\
&T((x, a, b, y, init), (x', a', b', y', init')) = \\
&\qquad (a' = f(x', \mathit{if\ init\ then\ 0\ else\ } y)) \wedge \\
&\qquad (b' = \mathit{if\ } a' \geq 0 \mathit{\ then\ } a' \mathit{\ else\ } -a') \wedge \\
&\qquad (y' = b' + (\mathit{if\ init\ then\ 0\ else\ } y)) \wedge \\
&\qquad \neg init'
\end{aligned}
$$

Note that f is unspecified in Figure 1 and so also in T. In a real system, f would be defined in the Lustre model and expanded in T. A safety property such as `y >= 0` is translated into $init \vee (y \geq 0)$. Nested uses of arrow and pre operators are handled by introducing new output variables for nested expressions, though such details are unimportant for our purposes.

Each equation in the Lustre program is translated into a single top-level conjunct in the transition relation. This is very convenient as the IVC of a Lustre property can be reported in terms of the output variables whose equations are part of the IVC. Equivalently, the interpretation of an IVC for a Lustre property is that any output variable that is not part of the IVC can be turned into an input variable, its equation thrown away, while preserving the validity of the property. Thus the granularity of the IVC analysis is determined by the granularity of the Lustre equations and can be adjusted by introducing auxiliary variables for subexpressions if desired.

5.2 JKind

JKind [1] is an infinite-state model checker for safety properties. JKind proves safety properties using multiple cooperative engines in parallel including k-induction [44], property directed reachabil-

ity [17], and template-based lemma generation [24]. JKind accepts Lustre programs written over the theory of linear integer and real arithmetic. In the back-end, JKind uses an SMT solver such as Z3 [15], Yices [16], MathSAT [13], or SMTInterpol [12].

JKind works on multiple properties simultaneously. When a property is proven and IVC generation is enabled, an additional parallel engine executes Algorithm 2 to generate a nearly minimal IVC.

JKind accepts an annotation on its input Lustre program indicating which outputs variables to consider for IVC generation. Output variables not mentioned in the annotation are implicitly included in all IVCs. This allows the implementation to focus on the variables important to the user and ignore, for example, administrative equations. This is even more important for tools which generate Lustre as they often create many such administrative equations which simply wire together more interesting expressions.

6. EXPERIMENT

We would like to investigate both the *efficiency* and *minimality* of our three algorithms: the naive brute-force algorithm (IVC_BF), the UNSAT core-based algorithm (IVC_UC), and the combined UNSAT core followed by brute-force minimization algorithm (IVC_UCBF). Efficiency is computed in terms of wall-clock time: how much overhead does the IVC algorithm introduce? Minimality is determined by the size of the IVC: cores with a smaller number of variables are preferred to cores with a larger number of variables. Finally, we are interested in the *diversity* of solutions: how often do different tools/algorithms generate different minimal IVCs?

The use of JKind allows additional dimensions to our investigation: it supports two different inductive algorithms: k-induction and PDR, and a "fastest" mode, that runs both algorithms in parallel. In addition, JKind supports multiple back-end SMT solvers including Z3 [15], Yices [16], MathSAT [13], and SMTInterpol [12]. We would like to determine whether the choice of inductive algorithm affects the size of the IVC, whether different solvers are more or less efficient at producing IVCs, and whether running different solvers/algorithms leads to *diversity* of IVC solutions.

Therefore, we investigate the following research questions:

- **RQ1:** How expensive is it to compute inductive validity cores using the IVC_BF, IVC_UC, and IVC_UCBF algorithms?
- **RQ2:** How close to minimal are the IVC sets computed by IVC_UC as opposed to the (guaranteed minimal) IVC_UCBF? How do the sizes of IVCs compare to static slices of the model?
- **RQ3:** How much *diversity* exists in the solutions produced by different solver/induction algorithm configurations?

6.1 Experimental Setup

In this study, we started from a suite of 700 Lustre models developed as a benchmark suite for [21]. We augmented this suite with 81 additional models from recent verification projects including avionics and medical devices [4, 38]. Most of the benchmark models from [21] are small (10kB or less, with 6-40 equations) and contain a range of hardware benchmarks and software problems involving counters. The additional models are much larger: around 80kB with over 300 equations. We added the new benchmarks to better check the scalability for the tools, especially with respect to the brute force algorithm. Each benchmark model has a single property to analyze. For our purposes, we are only interested in models with a *valid* property (though it is perhaps worth noting that there is no additional computation—and thus no overhead—using the JKind IVC options for *invalid* properties). In our benchmark set, 295 models yield counterexamples, and 10 additional models are neither provable nor yield counterexamples in our test configuration (see next paragraph for configuration information). The benchmark suite therefore contains 476 models with valid properties, which we use as our test subjects.

For each test model, we computed IVC_UC in 12+1 configurations: the twelve configurations were the cross product of all solvers {Z3, Yices, MathSAT, SMTInterpol} and inductive algorithms {k-induction, PDR, fastest}, and the remaining (+1) configuration was an instance of IVC_BF run on Yices, which is the default solver in JKind. In addition, for each of the 12 configurations, we ran an instance of JKind without IVC to examine overhead. The experiments were run on an Intel(R) i5-2430M, 2.40GHz, 4GB memory machine, with a 1 hour timeout for each analysis on any model. The data gathered for each configuration of each model included the time required to check the model without IVC, with IVC, and also the set of elements in the computed IVC.[1]

Note that not all analysis problems were solvable with all algorithms: for all solvers, k-induction (without IVC) was unable to solve 172 of the examples. When comparing minimality of different solving algorithms, we only considered cases where both algorithms provided a solution (as will be discussed in more detail in Section 7.2).

7. RESULTS

In this section, we examine our experimental results from three perspectives: performance, minimality of IVC_UC results, and diversity.

7.1 Performance

In this subsection, we examine the performance of our inductive validity core algorithms (research question **RQ1**). First we examine the performance overhead of the IVC_UC algorithm over the time necessary to find a proof using inductive model checking. To examine this question, we use the default *fastest* option of JKind which terminates when either the k-induction or PDR algorithm finds a proof. To measure the performance overhead of the IVC_UC algorithm, we execute it over the proof generated by the *fastest* option.

Since the IVC_UC algorithm uses the UNSAT core facilities of the underlying SMT solver, the performance is dependent on the efficiency of this part of the solver. Looking at Tables 1 and 2, it is possible to examine both the computation time for analysis using the four solvers under evaluation and the overhead imposed by the IVC_UC algorithm. Figure 7.1 allows a visualization of the runtime for the IVC_UC algorithm running different solvers. The data suggests that Yices (the default solver in JKind) and Z3 are the most performant solvers both in terms of computation time and overhead.

The IVC_UC algorithm using the Z3 and Yices SMT solvers adds a modest performance penalty to the time required for inductive proofs.

Next, we consider the overhead of IVC_UC vs. IVC_BF. Recall from Section 4 that IVC_BF requires n model checking runs, where n is the number of conjuncts in the transition relation. As expected, the performance is approximately a linear multiple of the

[1]The benchmarks, all raw experimental results, and computed data are available on [2].

Table 1: IVC_UC runtime with different solvers

runtime (sec)	min	max	mean	stdev
Z3	0.005	2.335	0.192	0.355
Yices	0.014	13.297	0.589	1.473
SMTInterpol	0.029	19.254	1.396	2.991
MathSAT	0.011	86.421	3.071	10.403

Table 2: Overhead of IVC_UC computations using different solvers

solver	min	max	mean	stdev
Z3	0.73%	84.13%	17.38%	16.92%
Yices	0.17%	351.47%	52.20%	54.50%
SMTInterpol	1.46%	175.75%	46.81%	37.35%
MathSAT	0.78%	955.52%	80.21%	112.92%

size of the model, so larger models yield substantially lower performance.[2] We run the brute-force algorithm using Yices as it is the default solver for JKind and is close to Z3 in terms of computation time. For 19 models, IVC_BF times out after 1 hour. Figure 6 shows the overhead of IVC_BF in comparison to IVC_UC with multiple solvers.

The brute-force algorithm IVC_BF adds a substantial performance penalty to inductive proofs in all cases and is not scalable enough to compute a minimal core for large analysis problems.

Finally, we consider the combined IVC_UCBF algorithm, in which we first run the IVC_UC to determine a close-to-minimal IVC, then run IVC_BF on the remaining set. The overhead of this algorithm is considered in Tables 3 and 4. While considerably slower than IVC_UC, this approach can still be used for reasonably sized models.

7.2 Minimality

In this section, we examine the minimality of the cores computed by the IVC_UC and IVC_UCBF algorithms using different inductive proof methods, and we compare both algorithms against a *backward static slice* [45] of the Lustre program starting with the property of interest. There are three interesting aspects to be examined related to this research question. First (**RQ2.1**), does the choice of SMT solver or algorithm used to produce a proof (k-induction or PDR) matter in terms of the minimality of the inductive core? As mentioned in Section 4, the IVC_UC algorithm is not guaranteed to produce a minimal core due in part to the role of invariants used in producing a proof; as k-induction and PDR use substantially different invariant generation algorithms, it is likely that the set of necessary invariants for proofs are dissimilar, and that this would in turn affect the number of model elements required for the proof. It is possible that one or the other algorithm is more likely to yield smaller invariant sets. In addition, differences in the choice of the UNSAT core algorithms in the different solvers could affect the size of the generated core. However, our algorithm already performs a minimization step on UNSAT cores, and thus the only differences would be due to one algorithm leading to a different minimal core than another.

[2]for Lustre models, the number of conjuncts is equivalent to the number of equations in the Lustre model.

Table 3: IVC_UCBF runtime

runtime (sec)	min	max	mean	stdev
Yices	0.68	3600.0	91.59	490.01
Z3	0.66	3600.0	93.01	490.27

Table 4: Overhead of IVC_UCBF algorithm

solver	min	max	mean	stdev
Yices	122.50%	30092.78%	3195.90%	3896.05%
Z3	101.70%	28114.07%	3190.18%	4119.14%

Table 5: Aggregate IVC sizes produced by IVC_UC using different inductive algorithms and solvers

solver	PDR	k-induction	**total**
Z3	2378	2379	4757
Yices	2384	2376	4760
MathSAT	2375	2369	4744
SMTInterpol	2378	2368	4746
total	9515	9492	

As discussed in Section 6, k-induction is unable to solve all of the analysis problems; therefore we include only models that are solvable using *both* k-induction and PDR by *all solvers*, 304 models in all. Examining the aggregate data in Table 5, we can see the sizes of cores produced by different algorithms and solvers.

Neither PDR nor k-induction yields a smaller inductive validity core in general. The choice of underlying SMT solver does not substantially affect the size of the inductive validity cores.

The next question (**RQ2.2**) asks how close to minimal are the cores produced by IVC_UC vs. the (guaranteed minimal) cores produced by the IVC_UCBF algorithm? Note that we cannot measure the distance on all models because the combined algorithm times out on 9 of the larger models. We therefore examine the distance from minimal cores produced by the combined algorithm for models in which it completes within the one hour timeout. For comparison, we run the IVC_UC algorithm using Z3 and Yices with JKind's default *fastest* algorithm, which will use the result of either k-induction or PDR. A graph showing the size of the IVCs for each model produced using the Yices solver is shown in Figure 7. In the figure, the models are ranked along the x-axis by the size of the core produced by IVC_UCBF. The figure demonstrates that while on average there is a modest change in minimality, there can be substantial variance on the sizes of the cores produced by the IVC_UC algorithm. Summary statistics are shown in Table 6.

The IVC_UC algorithm computes cores that are on average 21% larger than those produced by IVC_UCBF, with substantial variance in some cases.

The final question (**RQ2.3**) asks how well the approach compares to *backwards static slicing* [45], since slicing also reduces

Table 6: Increase in IVC Size for IVC_UC vs. IVC_UCBF

solver	min	max	mean	stdev
Yices	0.0%	725.0%	20.54%	50.47%
Z3	0.0%	725.0%	20.81%	50.34%

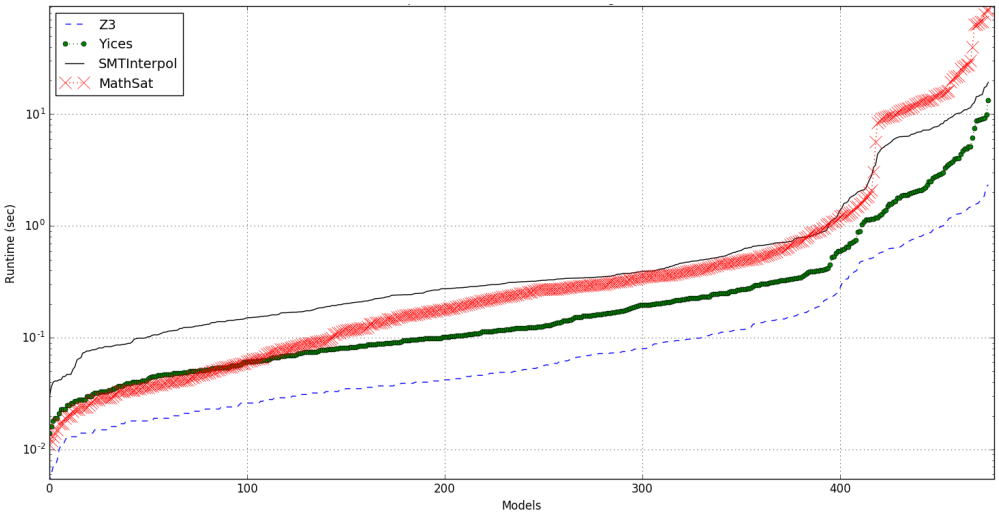


Figure 5: IVC_UC performance on different solvers

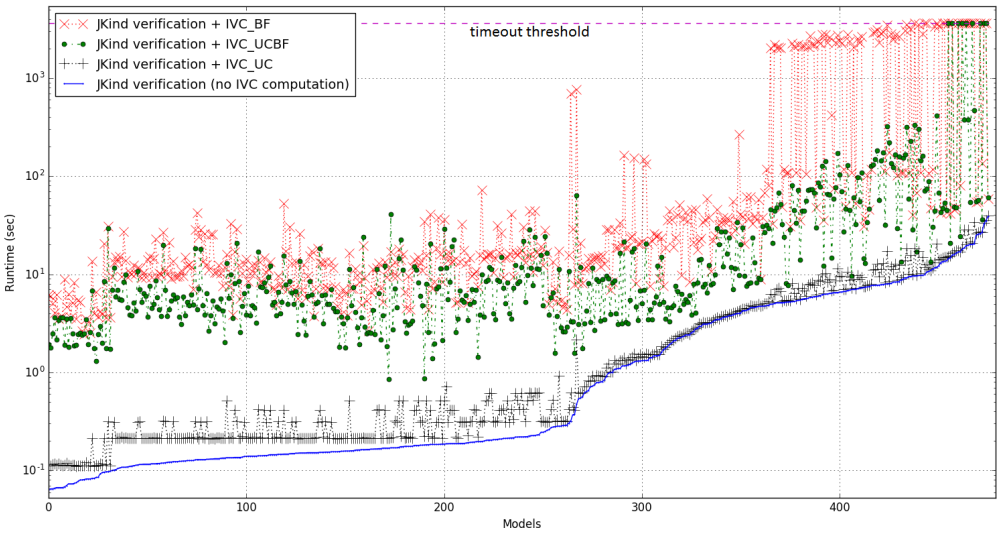


Figure 6: Runtime of IVC_BF, IVC_UCBF, IVC_UC algorithms for Yices

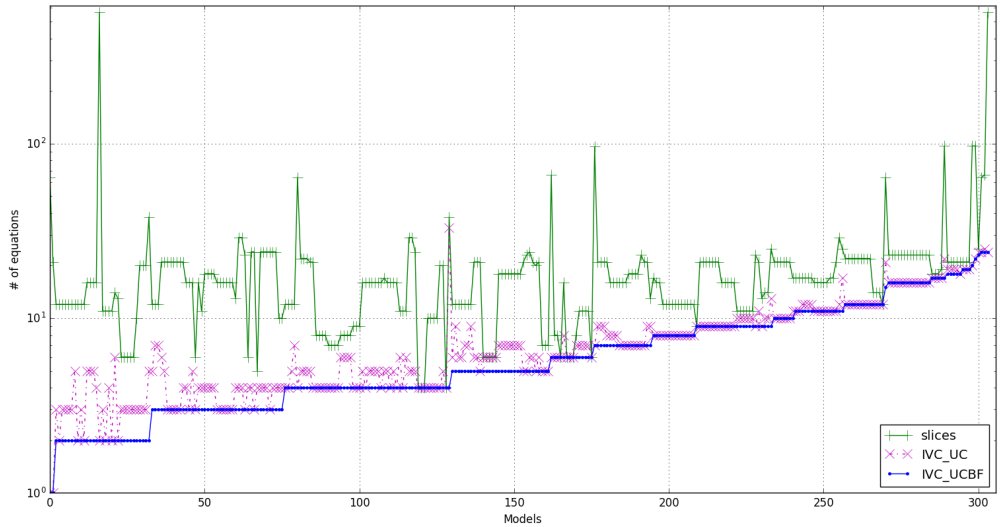


Figure 7: IVC sizes produced by IVC_UC, IVC_UCBF for Yices vs. static slices

Table 7: Pairwise Jaccard distances among all models

min	max	mean	stdev
0.0	0.878	0.026	0.059

the set of model elements necessary to construct a proof. We start the slice from the equation defining the property of interest, and use the usual approach [18] that performs an iterative backward traversal from the variables used within an equation to their defining equations. We expect the IVC mechanism to be more precise, because the slice overapproximates of the set of equations necessary for *any* proof. This claim is demonstrated in Figure 7; slices are (mean) 406% larger than the IVCs produced by our IVC_UC algorithm and 465% larger than those produced by IVC_UCBF algorithm.

Both IVC algorithms compute cores that are usually much smaller than backwards static slices.

Comparing the sizes of the IVC_UC IVCs to the original models, the original 395 benchmark models from [21] already had applied slicing, so there is no difference between the sliced size and the original model size. For the remaining 81 benchmarks, the number of equations is (mean) 2500% larger than the IVC_UC IVCs. We note, however, that comparison of IVC size against the original model size can be misleading, as the improvement can easily be "gamed" by adding equations that are irrelevant to the property.

7.3 Diversity

Recall from Section 4 that a *minimal* IVC set is any set leading to a proof such that if you remove any of its elements, it no longer produces a proof. For certain models and properties, it is possible that there are many minimal cores that will lead to a proof. In this section, we examine the issue of diversity: do different solvers and algorithms lead to *different* minimal cores? This is both a function of the models and the solution algorithms: for certain models, there is only one possible minimal IVC set, whereas other models might have many. Given that there are multiple solutions, the interesting question is whether using different solvers and algorithms will lead to different solutions. The reason diversity is considered is that it has substantial relevance to some of the uses of the tool, e.g., for constructing multiple traceability matrices from proofs (see Section 9). Note that our exploration in this experiment is not exhaustive, but only exploratory, based on the IVCs returned by different algorithms and tools; we leave exhaustive exploration of IVCs for future work.

To measure diversity of IVCs, we use Jaccard distance:

Definition 3. Jaccard distance: $d_J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$, $0 \leq d_J(A,B) \leq 1$

Jaccard distance is a standard metric for comparing finite sets (assuming that both sets are non-empty) by comparing the size of the intersection of two sets over its union. For each model in the benchmark, the experiments generated 13 potentially different IVCs. Therefore, we obtained $\binom{13}{2} = 78$ combinations of pairwise distances per model. Then, minimum, maximum, average, and standard deviation of the distances were calculated (Figure 8), by which, again, we calculated these four measures among all models. As seen in Table 7, on average, the Jaccard distance between different solutions is small, but the maximum is close to 1, which indicates that even for our exploratory analysis, there are models for which the tools yield substantially diverse solutions. The diversity between solutions is represented graphically in Figure 8, where for each model, we present the min, max, and mean pairwise Jaccard distance of the solutions produced by algorithm IVC_UC for each model, ranked by the mean distance.

7.4 Discussion

In the previous section, we presented three algorithms for determining inductive validity cores. The brute-force algorithm is guaranteed minimal, but is often very slow. The other two algorithms, the UNSAT core algorithm IVC_UC and the combined algorithm IVC_UCBF, represent interesting trade-offs. The IVC_UC algorithm is much faster, but is not guaranteed to be minimal; the result of this algorithm can be further, and sometimes quickly, refined by the combined algorithm. Thus, we can choose to trade off speed for guaranteed minimality using these two algorithms; the combined algorithm can be viewed as a refinement algorithm that we can terminate either at completion or after a fixed time bound.

Although our experiment does not ask statistical questions, it is still worth examining threats towards generalizing our results. First, are the models and properties that we chose representative? We started from an existing benchmark from another research group suite to try to assuage this concern, but most of these models were small, so we extended the benchmark suite with 81 of our own models. It is possible that our additions skew the results, though these models are immediately derived from previously published work and not modified for our analysis here. Second, our models and tools use the Lustre language, which is equational, rather than conjuncted transition systems; it is possible (though, in our opinion, unlikely) that arbitrary conjuncts rather than equations will yield different performance or minimality characteristics.

Our approach is limited by the capabilities of the SMT solvers and inductive model checking algorithms that are used. For example, it is difficult, given state of the art SMT solvers, to produce proofs involving complex models involving non-linear floating-point arithmetic. However, given an inductive proof produced by an UNSAT-core-producing SMT solver, we feel confident that the IVC_UC algorithm can produce an IVC. Our approach is theory and invariant-generator agnostic, so as inductive model checking algorithms evolve and SMT solvers add support for new theories, the IVC algorithm should be able to work without modification.

8. RELATED WORK

Our work builds on top of a substantial foundation building Minimally Unsatisfiable Subformulas (MUSes) from UNSAT cores [14], including [6, 7, 30, 40, 42]. Recent algorithms can handle very large problems, but computing MUSes is still a resource-intensive task. While some work is aimed at providing a set of minimal unsatisfiable formulae, minimality is usually defined such that given a set of clauses $\mathbb{M}$, removing any member of $\mathbb{M}$ makes it satisfiable [6]. The step of producing minimal invariants for proofs has been investigated in depth by Ivrii et al. in [23].

UNSAT cores and MUSes are used for many different activities within formal verification. Gupta et al. [19] and McMillan and Amla [34] introduced the use of unsatisfiable cores in proof-based abstraction engines. Their goal is to shrink the abstraction size by omitting the parts of the design that are irrelevant to the proof of the property under verification. Torlak et al. in [46] finds MUSes of Alloy specifications, and considers semantic vacuity, which we consider in Section 1. Alloy models are only analyzed up to certain size bounds, however, and in general are unable to prove properties for arbitrary models. Also, because we are extracting information from proofs, it is possible to use IVCs for additional purposes (proof explanation and completeness checking).

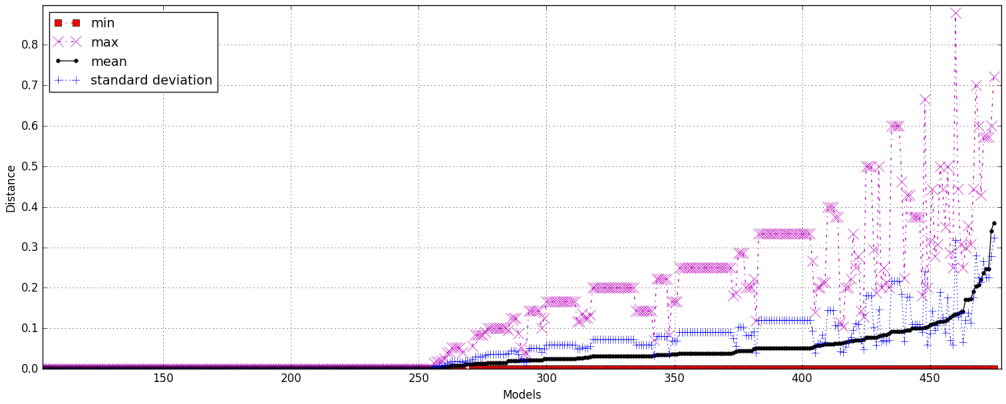


Figure 8: Pairwise Jaccard distance between IVCs

If we view Lustre as a programming language, our work can be viewed as a more accurate form of program slicing [45]. We perform *backwards slicing* from the formula that defines the property of interest of the model. The slice produced is smaller and more accurate than a static slice of the formula [47], but guaranteed to be a sound slice for the formula for all program executions, unlike dynamic slicing [3]. Predicate-based slicing has been used [29] to try to minimize the size of a dynamic slice. Our approach may have utility for some concerns of program slicing (such as model understanding) by constructing simple "requirements" of a model and using the tool to find the relevant portions of the model.

Another potential use of our work is for "semantic" vacuity detection. A standard definition of vacuity is syntactic and defined as follows [26]: *A system K satisfies a formula ϕ vacuously iff $K \vdash \phi$ and there is some subformula ψ of ϕ such that ψ does not affect ϕ in K*. Vacuity has been extensively studied [5, 8, 10, 11, 20, 26] considering a range of different temporal logics and definitions of "affect". On the other hand, our work can be used to consider a broader definition of vacuity. Even if all subformulae are required (the property is not syntactically vacuous), it may not require substantial portions of the model, and so may be provable for vacuous reasons. The problem is exacerbated when the modeling and property language are the same (as in JKind), because whether a subformula is considered part of the model or part of the property, from the perspective of checking tools, can be unclear.

Determining completeness of properties has also been extensively studied. Certification standards such as DO-178C [41] require that requirements-derived tests achieve some level of structural coverage (MC/DC, decision, statement) depending on the criticality level of the software, in order to approximate completeness. If coverage is not achieved, then additional requirements and tests are added until coverage is achieved. Chockler [9] defined the first completeness metrics directly on formal properties based on mutation coverage. Later work by Kupferman et al. [26] defines completeness as an extension of vacuity to elements in the model. We present an alternative approach that uses the proof directly, which we expect to be considerably less expensive to compute. Recent work by Murugesan [39] and Schuller [43] attempts to combine test coverage metrics with requirements to determine completeness.

9. CONCLUSIONS & FUTURE WORK

In this paper, we have defined the notion of inductive validity core (IVC) which appears to be a useful measure in relation to a valid safety property for inductive model checking. We have presented a novel algorithm for computing IVCs that are nearly minimal and have shown that full minimality is undecidable in many settings. Our algorithm is applicable to all forms of inductive SAT/SMT-based model checking including k-induction, PDR, and interpolation-based model checking. We have implemented our IVC algorithm as part of the open source model checker JKind. We have shown that the algorithm requires only a moderate overhead and produces nearly minimal IVCs in practice. Moreover, the produced IVCs are fairly stable with respect to underlying proof engines (k-induction and PDR) and back-end SMT solvers (Yices, Z3, MathSAT, SMTInterpol).

Our work has recently been integrated into the AADL/AGREE tool suite [4, 38], which supports compositional reasoning about system architectures. First, IVCs are used to to automatically compute traceability information between high- and low-level requirements in compositional proofs. Second, IVCs are used by the AGREE symbolic simulator to explain conflicts when the simulator is not able to compute a "next state" for a set of chosen constraints. A pilot project at Rockwell Collins is using the traceability information produced by the IVC support in the AGREE tool.

In future work, we will compare the traceability matrices generated by IVCs with those produced by human experts and and by automated heuristic approaches. Our expectation is that the traceability information produced by IVCs will be both more accurate and closer to minimal than other approaches. We also will examine the impact of multiple distinct IVCs on traceability research. An initial paper on this work, which we call *complete traceability* has been accepted to the RE@Next! track of the Requirements Engineering conference [37]. We are interested in diversity both in terms of regression analysis for testing and proof, as well as examining the underlying sources of diversity in our analysis models. We suspect that in some cases, it indicates fault tolerance in the architecture under analysis, and in other cases it may indicate redundancy in requirements specifications for subcomponents. To support a systematic investigation of diversity, we plan to investigate algorithms for exploring the space of IVCs, e.g., finding a minimum, rather than minimal IVC, or finding all IVCs.

Finally, we are in the process of comparing our approach against other approaches measuring completeness of requirements (such as those in [9, 26, 27]).

Acknowledgments: This work was supported by DARPA under contract FA8750-12-9-0179 (Secure Mathematically-Assured Composition of Control Models) and by NASA under contract NNA13AA21C (Compositional Verification of Flight Critical Systems).

10. REFERENCES

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

[2] Set of Support. https://github.com/elaghs/Working/tree/master/support/experiments.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[4] J. Backes, D. Cofer, S. Miller, and M. W. Whalen. Requirements analysis of a quad-redundant flight control system. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 9058 of *Lecture Notes in Computer Science*, pages 82–96. Springer International Publishing, 2015.

[5] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *9th International Conference on Computer Aided Verification (CAV'97)*, pages 279–290, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[6] A. Belov, M. Janota, I. Lynce, and J. Marques-Silva. On computing minimal equivalent subformulas. In *Principles and Practice of Constraint Programming*, pages 158–174. Springer, 2012.

[7] A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient MUS extraction. *AI Communications*, 25(2):97–116, Apr. 2012.

[8] S. Ben-David and O. Kupferman. A framework for ranking vacuity results. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 148–162, 2013.

[9] H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for formal verification. *Correct hardware design and verification methods*, pages 111–125, 2003.

[10] H. Chockler and O. Strichman. Easier and more informative vacuity checks. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[11] H. Chockler and O. Strichman. Before and after vacuity. *Formal Methods in System Design*, 34(1):37–58, 2008.

[12] J. Christ, J. Hoenicke, and A. Nutz. Smtinterpol: An interpolating smt solver. In *Proceedings of the 19th International Conference on Model Checking Software*, SPIN'12, pages 248–254, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 smt solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 93–107, Berlin, Heidelberg, 2013. Springer-Verlag.

[14] A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, SAT'07, pages 334–339, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[16] B. Dutertre and L. D. Moura. The YICES SMT solver. Technical report, SRI, 2006.

[17] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, TX, 2011. FMCAD Inc.

[18] F. Gaucher. Slicing lustre programs. Technical report, VERIMAG, Grenoble, February 2003.

[19] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using sat-based bmc with proof analysis. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 416. IEEE Computer Society, 2003.

[20] A. Gurfinkel and M. Chechik. Robust vacuity for branching temporal logic. *ACM Trans. Comput. Logic*, 13(1):1:1–1:32, Jan. 2012.

[21] G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08*, pages 1–9, Nov 2008.

[22] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[23] A. Ivrii, A. Gurfinkel, and A. Belov. Small inductive safe invariants. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 2014*, pages 115–122, October 2014.

[24] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 192–206, 2011.

[25] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. Huffman Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1375–1378, Piscataway, NJ, USA, 2012. IEEE Press.

[26] O. Kupferman. Sanity checks in formal verification. In *Proceedings of the 17th International Conference on Concurrency Theory*, CONCUR'06, pages 37–51, Berlin, Heidelberg, 2006. Springer-Verlag.

[27] O. Kupferman, W. Li, and S. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Proceedings of the 2008 Int'l Conf. on Formal Methods in Computer-Aided Design*, page 25, 2008.

[28] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *Journal on Software Tools for Technology Transfer*, 4(2), February 2003.

[29] H. F. Li, J. Rilling, and D. Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. *Automated Software Engineering*, 11(1):63–89, 2004.

[30] J. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications. In *Multiple-Valued Logic (ISMVL), 2010 40th IEEE International Symposium on*, pages 9–14. IEEE, 2010.

[31] MathWorks Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier, 2015.

[32] MathWorks Inc. Simulink Requirements Traceability. http://www.mathworks.com/discovery/requirements-traceability.html, 2016.

[33] K. L. McMillan. A methodology for hardware verification using compositional model checking. Technical Report 1999-01, Cadence Berkeley Labs, Berkeley, CA 94704, 1999.

[34] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 2–17. Springer, 2003.

[35] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[36] *Requirements for Safety Related Software in Defence Equipment, Issue 2*. UK Ministry of Defence, 1997.

[37] A. Murugesan, M. W. Whalen, E. Ghassabani, and M. P. Heimdahl. Complete traceability for requirements in satisfaction arguments. In *Proceedings of the International Conference on Requirements Engineering (RE@Next! Track)*. IEEE, September 2016.

[38] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl. Compositional verification of a medical device system. In *ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013*. ACM, November 2013.

[39] A. Murugesan, M. W. Whalen, N. Rungta, O. Tkachuk, S. Person, M. P. Heimdahl, and D. You. Are we there yet? Determining the adequacy of formalized requirements and test suites. In *NASA Formal Methods*, pages 279–294. Springer, 2015.

[40] A. Nadel. Boosting minimal unsatisfiable core extraction. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 221–229. IEEE, 2010.

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

[42] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Theory and Applications of Satisfiability Testing-SAT 2011*, pages 174–187. Springer, 2011.

[43] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the Fourth IEEE Int'l Conf. on Software Testing, Verification and Validation*, pages 90–99, 2011.

[44] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 108–125, 2000.

[45] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[46] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[47] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[48] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM, May 2013.

[49] D. You, S. Rayadurgam, M. Whalen, and M. Heimdahl. Efficient observability-based test generation by dynamic symbolic execution. In *26th International Symposium on Software Reliability Engineering (ISSRE 2015)*, November 2015.

[50] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *6th International Conference on Theory and Applications of Satisfiability Testing: SAT 2003*, May 2003.