# Efficient Generation of All Minimal Inductive Validity Cores

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

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

***Abstract*—Symbolic model checkers can construct proofs of safety properties over complex models, but when a proof succeeds, the results do not generally provide much insight to the user. Recently, proof cores (alternately, for inductive model checkers, Inductive Validity Cores (IVCs)) were introduced to trace a property to a minimal set of model elements necessary for proof. Minimal IVCs facilitate several engineering tasks, including performing traceability and analyzing requirements completeness, that usually rely on the minimality of IVCs. However, existing algorithms for generating an IVC are either expensive or only able to find an approximately minimal IVC.**

**Besides minimality, computing *all* minimal IVCs of a given property is an interesting problem that provides several useful analyses, including regression analysis for testing/proof, determination of the minimum (as opposed to minimal) number of model elements necessary for proof, the diversity examination of model elements leading to proof, and analyzing fault tolerance.**

**This paper proposes an efficient method for finding *all minimal* IVCs of a given property proving its correctness and completeness. We benchmark our algorithm against existing IVC-generating algorithms and show, in many cases, the cost of finding all minimal IVCs by our technique is similar to finding a single minimal IVC using existing algorithms.**

***Keywords*-Inductive Validity Cores; UNSAT-core generation; SMT-based model checking; Inductive proofs;**

## I. Introduction

Most modern sequential model checking techniques for safety properties, including IC3/PDR [1] and $k$-induction [2], use a form of induction to establish proof. These techniques are very powerful, and can often reason successfully over very large or even infinite state spaces. The proofs provided by these tools can provide rigorous evidence that a software or hardware system works as intended.

On the other hand, there are many situations in which properties can be proved, but systems still will not perform as intended. Issues such as vacuity [3], incorrect environmental assumptions [4], and errors either in English language requirements or formalization [5] can all lead to failures of "proved" systems. Thus, even if proofs are established, one must approach verification with skepticism.

Recently, *proof cores* [6] have been proposed as a mechanism to determine which elements of a model are used when constructing a proof. This idea is formalized by Ghassabani et al. for inductive model checkers in [7] as *Inductive Validity Cores* (IVCs). IVCs offer proof explanation as to why a property is satisfied by a model in a formal and human-understandable way. The idea lifts UNSAT cores [8] to the level of sequential model checking algorithms using induction. Informally, if a model is viewed as a conjunction of constraints, a minimal IVC (MIVC) is a set of constraints that is sufficient to construct a proof such that if any constraint is removed, the property is no longer valid. IVCs and MIVCs can be used for several purposes, including performing traceability between specification and design elements [9], assessing model coverage [10], and explaining unsatisfiable test obligations when using model checkers for test case generation. Ghassabani et al. [7] presented two algorithms: one that computes an approximately minimal IVC using UNSAT cores (`IVC_UC`) that is computationally inexpensive, and a more accurate algorithm that usually produces a minimal IVC using a brute-force post-processing step (`IVC_UCBF`) that is considerably more expensive to compute.[1]

The IVC and proof core ideas share many similarities with approaches for computing minimal invariant sets for inductive proofs (such as is performed for inductive proof certificates [11], [12]), and in fact the `IVC_UC` algorithm performs a minimal lemma set computation. However, there is a substantive difference: to find a minimal set of constraints, it is usually necessary to find new proofs involving *new lemmas* not used in the original proof, which accounts for the expense of the `IVC_UCBF` algorithm.

It is often the case that there are multiple MIVCs for a given property. In this case, computing a single IVC provides, at best, an incomplete picture of the traceability information associated with the proof. Depending on the model and property to be analyzed, there is often substantial diversity between the IVCs used for proof, and there can also be a substantive difference in the size of a *minimal* IVC and a *minimum* IVC, which is the (not necessarily unique) smallest MIVC. If *all* MIVCs can be found, then several additional analyses can be performed:

- Coverage Analysis: MIVCs can be used to define cov-

[1] In [7] it is shown that minimization is as hard as model checking, so for model checking problems that generally undecidable, the minimization process is also generally undecidable, so the `IVC_UCBF` algorithm may time out and return an approximate result.

erage metrics by examining the percentage of model elements required for a proof. However, since MIVCs are not unique, there are multiple, equally legitimate coverage scores possible. Having *all* MIVCs allows one to define additional metrics: coverage of MAY elements, coverage of MUST elements, as well as policies for the existing MIVC metric: e.g., choose the smallest MIVC [10].
- Optimizing Logic Synthesis: synthesis tools can benefit from MIVCs in the process of transforming an abstract behavior into a design implementation. A practical way of calculating all MIVCs allows to find a minimum set of design elements (optimal implementation) for a certain behavior. Such optimizations can be performed at different levels of synthesis.
- Impact Analysis: Given all MIVCs, it is possible to determine which requirements may be falsified by changes to the model. This analysis allows for selective regression verification of tests and proofs: if there are alternate proof paths that do not require the modified portions of the model, then the requirement does not need to be re-verified [9].
- Robustness Analysis: It is possible to partition the model elements into MUST and MAY sets based on whether they are in every MIVC or only some MIVCs, respectively. This may allow insight into the relative importance of different model elements for the property. For example, if the MUST set is empty, then the requirement has been implemented in multiple ways, such as would be expected in a fault-tolerant system [9].

In addition, the Requirements Engineering community is keenly interested in approaches to manage requirements traceability. In most cases, it is assumed that there is a single "golden" set of trace links that describes how requirements are implemented in software [13]–[15]. However, if there are multiple MIVCs, then it is possible that there are several equally valid sets of trace links. Examining the diversity of all MIVCs could lead to changes in how traceability is performed for critical systems.

In this paper, we propose a new method for computing *all* IVCs that is *always* minimal for decidable model checking problems and *usually* (and detectably) minimal for model-checking problems that are generally undecidable. In recent years, a number of efficient algorithms for extracting minimal UNSAT subformulae (MUSes) have been proposed [16], most of which are focused on computing a single MUS [17]–[21]. In this paper, we adapt the recent work by Liffiton et al. [22] from the generation of MUSes from UNSAT-cores to all IVCs for inductive model checking. This requires changing the underlying mechanisms that are used to construct candidate solutions and also changing the structure of the proof of correctness. In addition, we demonstrate that the approach can terminate with all minimal IVCs even if the witness generator only generates approximately minimal IVCs (utilizing the "fast" `IVC_UC` algorithm from [7]). In our empirical results, this allows our algorithm to be quite efficient to the extent

```
node asw (alt1, alt2: int) returns (doi_on: bool);
   var
        a1_below, a2_below, a1_above, a2_above,
        one_below, both_above, on_p : bool;
   let
(1)     a1_below = (alt1 < THRESHOLD);
(2)     a2_below = (alt2 < THRESHOLD);
(3)     a1_above = (alt1 >= T_HYST);
(4)     a2_above = (alt2 >= T_HYST);
(5)     one_below = a1_below or a2_below;
(6)     both_above = a1_above and a2_above;
(7)     doi_on = if one_below then true
                 else if both_above then false
                 else (false -> pre(doi_on));
(8)     on_p = ((alt1 < THRESHOLD) and
               (alt2 < THRESHOLD)) => doi_on;
   tel;
```

Fig. 1. Altitude Switch Model

that in many cases, the cost of extracting all minimal IVCs is similar to the cost of finding a single guaranteed-minimal IVC, and on average is approximately 1.6x the cost of determining a single minimal IVC. The contributions of the work are therefore as follows:

- An algorithm for computing all minimal IVCs.
- A proof of correctness and completeness of the algorithm.
- An evaluation of the algorithm for performance and diversity of result sets against a benchmark suite.

Several commercial tools produce *proof-cores* [6], [23], which we believe to be similar to IVCs/MIVCs, but are not presented at a level of formality to perform a precise comparison. However, to the best of our knowledge, none of these tools offer to calculate *all* proof-cores. Our work can also be useful towards the support of this capability in future editions of these tools.

The rest of the paper is organized as follows. Section II introduces a running example used to illustrate concepts and our method. Section III covers the formal preliminaries for the approach. In Section IV, we present our method for enumerating all minimal IVCs, which is illustrated in Section V. In Sections VI and VII we talk about implementation and evaluation of our method. Finally, Section VIII mentions conclusions and future work.

## II. Running Example

We will use a very simple system from the avionics domain to illustrate our approach. An Altitude Switch (ASW) is a hypothetical device that turns power on to another subsystem, the Device of Interest (DOI), when the aircraft descends below a threshold altitude, and turns the power off again after the aircraft ascends over the threshold plus some hysteresis factor. An implementation of an ASW containing two altimeters written in the Lustre language (simplified and adapted from [24]) is shown in Fig. 1. If either altimeter is below the constant `THRESHOLD`, then it turns on the DOI; else, if the system is inhibited or both altimeters are above the threshold plus the hysteresis factor `T_HYST`, then the DOI is turned off, and if

neither condition holds, then in the initial computation it is false and thereafter retains its previous value. The notation `(false -> pre(doi_on))` in equation (7) describes an initialized register in Lustre: in the first step, the expression is `false`, and thereafter it is the previous value of `doi_on`. A simple property `on_p` states that if both altimeters are under the threshold, then the DOI is turned on. This property can easily be proved over the model using a $k$-induction based verifier such as `JKind` [25].

## III. Preliminaries

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

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

A safety property $P : U \rightarrow bool$ is a state predicate. A safety property $P$ holds on a transition system $(I, T)$ if it holds on all reachable states, i.e., $\forall u.\ R(u) \Rightarrow P(u)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$. We assume the transition relation has the structure of a top-level conjunction. Given $T(u, u') = T_1(u, u') \wedge \cdots \wedge T_n(u, u')$ we will write $T = \bigwedge_{i=1..n} T_i$ for short. By further abuse of notation, $T$ is identified with the set of its top-level conjuncts. Thus, $T_i \in T$ means that $T_i$ is a top-level conjunct of $T$, and $S \subseteq T$ means all top-level conjuncts of $S$ are top-level conjuncts of $T$. When a top-level conjunct $T_i$ is removed from $T$, we write $T \setminus \{T_i\}$. Such a transition system can easily encode our example model in Section II, where each equation defines a conjunct within $T$ that we will denote by the variable assigned; so, $T = \{$ `a1_below, a2_below, a1_above, a2_above, one_below, both_above, doi_on, on_p` $\}$.

The idea behind finding an IVC for a given property $P$ [7] is based on inductive proof methods used in SMT-based model checking, such as $K$-induction and IC3/PDR [1], [26], [27]. Generally, an IVC computation technique aims to determine, for any subset $S \subseteq T$, whether $P$ is provable by $S$. Then, a minimal subset that satisfies $P$ is seen as a minimal proof explanation called a minimal Inductive Validity Core. Theorem 1 demonstrates that the minimization process is as hard as model checking, so finding a minimal inductive validity core may not be possible for some model checking problems.

**Definition 1.** Inductive Validity Core (*IVC*) [7]: *$S \subseteq T$ for $(I, T) \vdash P$ is an Inductive Validity Core, denoted by IVC$(P, S)$, iff $(I, S) \vdash P$.*

**Definition 2.** Minimal Inductive Validity Core (*MIVC*) [7]: *$S \subseteq T$ is a minimal Inductive Validity Core, denoted by MIVC$(P, S)$, iff IVC$(P, S) \wedge \forall T_i \in S.\ (I, S \setminus \{T_i\}) \nvdash P$.*

**Theorem 1.** *Determining if an IVC is minimal is as hard as model checking.*
*Proof: see [7].* □

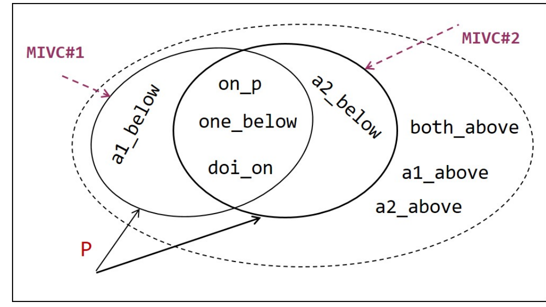


Fig. 2. Graphical representation of *MIVC*s for the model in Fig. 1 with $P = (\text{on_p})$

Note that, given $(I, T) \vdash P$, $P$ always has at least one *MIVC*, and it may also have many distinct *MIVC*s corresponding to different proof paths. To capture the latter, the *all MIVCs ($AIVC$)* relation has been introduced in [9].

**Definition 3.** All *MIVC*s ($AIVC$): *Given $(I, T) \vdash P$, $AIVC(P)$ is an association to all MIVCs for $P$:*

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

Fig. 2 illustrates these notions by a graphical representation of IVCs for property $P = (\text{on_p})$ in the example presented in Section II. As shown in the picture, this property has two distinct *MIVC*s, which means the model satisfies $P$ in two different ways: {{`a1_below, one_below, doi_on, on_p`}, {`a2_below, one_below, doi_on, on_p`}}, This is because in the implementation, the DOI is turned on when either of the altimeters is below the threshold, while our property states that they both must be below. Note that there is a subset of model elements, {`a1_above, a2_above, both_above`}, that does not show up in $AIVC(P)$. Elements in such a subset do not affect the satisfaction of $P$. In the complete ASW model in [24] there are additional properties that use these elements, but they are not necessary for the discussion in this paper.

## IV. Method

Considering the definition of a *MIVC*, a brute-force technique for enumerating all *MIVC*s would be the same as exploring the power set of $T$ (denoted by $\mathcal{P}(T)$). Basically, the algorithm needs to explore the provability of a given property by any subset of $T$, which would be computationally expensive. Our approach is an adaptation of the the work of MARCO for generating all minimal unsatisfiable subsets (MUSes) in [22], and only needs to explore a (small) portion of $\mathcal{P}(T)$ in order to compute $AIVC$. In fact, it can be viewed as an instantiation of the MARCO proof schema for the richer theory of sequential model checking. We begin by introducing several additional notions and definitions, most of which are analogous or equivalent to those in [22].

**Definition 4.** Maximal Inadequate Set (*MIS*): *$S \subset T$ for $(I, T) \vdash P$ is a Maximal Inadequate Set (MIS) iff $(I, S) \nvdash P$ and $\forall T_i \in T \setminus S.\ (I, S \cup \{T_i\}) \vdash P$.*

Given $(I,T) \vdash P$, for every $S \in \mathcal{P}(T)$, we have either $(I,S) \vdash P$ or $(I,S) \nvdash P$. In the former case, we say $S$ is **adequate** for $P$; in the latter, we say that $S$ is **inadequate** for the proof of $P$. Note that every *IVC* is an adequate set for $P$, and every *MIS* is an inadequate set.

**Lemma 1.** *For $(I,T) \vdash P$, if $S \subseteq T$ is adequate for property $P$, then all of its supersets are adequate for $P$ as well:*

$$\forall S_1 \subseteq S_2 \subseteq T.\ (I,S_1) \vdash P \Rightarrow (I,S_2) \vdash P$$

*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 $(I,S_1)$. ∎

**Corollary 1.** *For $(I,T) \vdash P$, if a given subset S is inadequate, then all of its subsets are inadequate as well:*

$$\forall S_1 \subseteq S_2 \subseteq T.\ (I,S_2) \nvdash P \Rightarrow (I,S_1) \nvdash P$$

*Proof:* Immediate from Lemma 1. ∎

The basic idea behind an algorithm for computing $AIVC(P)$ is the same as exploration of $\mathcal{P}(T)$, with two major performance improvements. First, Lemma 1 and Corollary 1 are used to block large portions of $\mathcal{P}(T)$ from consideration. For example, if a set $S \in \mathcal{P}(T)$ is found to be inadequate, then all subsets of $S$ are also inadequate and do not need to be explicitly considered. Second, if a set $S \in \mathcal{P}(T)$ is found to be adequate, then a fast algorithm (such as `IVC_UC` from [7]) is used to find a smaller $S' \subseteq S$ which is still adequate. This feeds into the first optimization since now all supersets of $S'$ rather than $S$ are blocked from future consideration.

To guide our algorithm, we now introduce a way of exploring $\mathcal{P}(T)$ which allows us to eliminate all subsets or supersets of any given set. We use a Boolean expression called $map$, which is in conjunctive normal form (CNF) and built gradually as the algorithm proceeds. Satisfying assignments for $map$ correspond to elements of $\mathcal{P}(T)$. For each $S \in \mathcal{P}(T)$ that the algorithm determines to be adequate or inadequate, a corresponding clause is added to $map$ which blocks $S$ and all supersets or subsets, respectively, from consideration. When a clause is added to $map$, the corresponding $S \in \mathcal{P}(T)$ is called *explored*. The supersets or subsets of $S$ which are blocked from consideration are called *excluded*. The remaining elements of $\mathcal{P}(T)$ are *unexplored*.

More precisely, given $T$ with $n$ top-level conjuncts, we define an ordered set of activation literals $\mathcal{A} = \{a_1, \dots, a_n\}$, where each $a_i$ has type Boolean. We assume the function $\textsc{ActLit} : T \rightarrow \mathcal{A}$ is a bijection assigning every $T_i \in T$ to an $a_i \in \mathcal{A}$ and vice versa. Then, a $map$ for $AIVC(P)$ is a CNF formula built over the elements of $\mathcal{A}$ such that:

- Initially $map$ is $\top$ since all of $\mathcal{P}(T)$ is unexplored.
- When $map$ is satisfiable, a model of it is a set $M \in \mathcal{P}(\mathcal{A})$ consisting of those $a \in \mathcal{A}$ which are assigned $true$.
- Every model $M$ of $map$ corresponds to a set $S \in \mathcal{P}(T)$ such that $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ and $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$.
- For every explored set $S \in \mathcal{P}(T)$:
  - if $S$ is adequate for $P$, then $map$ contains a clause $\bigvee_{T_i \in S} \neg\textsc{ActLit}(T_i)$. This clause blocks all supersets of $S$ from future consideration which is consistent with Lemma 1.
  - if $S$ is inadequate for $P$, then $map$ contains a clause $\bigvee_{T_i \in (T \setminus S)} \textsc{ActLit}(T_i)$. This clause blocks all subsets of $S$ from future consideration which is consistent with Corollary 1.

**Lemma 2.** *When $map$ is satisfiable with model $M$, set $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ is not equal to any adequate or inadequate explored set, nor a subset (superset) of any inadequate (adequate) explored set in $\mathcal{P}(T)$.*

*Proof:* Proof by contradiction. Case 1: Suppose there is an adequate set $Ex \subseteq S$ that has been already explored. Therefore, according to the definition, $map$ contains a clause $C = \bigvee_{T_i \in Ex} \neg\textsc{ActLit}(T_i)$, and since $Ex \subseteq S$, it is impossible for the model $M = \bigcup_{T_i \in Ex} \textsc{ActLit}(T_i)$ to satisfy $C$; hence, the assumption is false.

Case 2: Suppose there is an inadequate set $Ex$ such that $S \subseteq Ex$ and $Ex$ has been already explored. Therefore, according to the definition, $map$ contains a clause $C = \bigvee_{T_i \in (T \setminus S)} \textsc{ActLit}(T_i)$, and since $S \subseteq Ex$, it is impossible for the model $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$ to satisfy $C$; so, the assumption is false.

From Case 1 and Case 2, there is no model of $map$ whose corresponding set in $\mathcal{P}(T)$ is a non-strict subset (superset) of any inadequate (adequate) explored set. ∎

**Lemma 3.** *For $(I,T) \vdash P$, $map$ is satisfiable iff at least one $S \in AIVC(P)$ or one MIS of $T$ is unexplored.*

*Proof:* Let $map$ is satisfiable with a model $M$, and let $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ be the corresponding set of $\mathcal{P}(T)$. If $S$ is adequate, then it contains a *MIVC*. That *MIVC* must not be explored since otherwise $S$ would have been blocked from consideration. The *MIVC* must not be excluded since it is not a strict superset of any adequate set (by minimality) nor a subset of any inadequate set (by Corollary 1). Thus the *MIVC* must be unexplored. The case where $S$ is inadequate is symmetric.

In the other direction, let $S \subseteq T$ be an unexplored *MIVC*. Then consider the model $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$. We will show that each clause of $map$ is satisfied by $M$. There are two types of clauses to consider. A clause $\bigvee_{T_i \in S'} \neg\textsc{ActLit}(T_i)$ is in $map$ only if $S'$ is adequate. $M$ would falsify this clause only if $S' \subseteq S$ which is impossible by minimality of $S$. A clause $\bigvee_{T_i \in (T \setminus S')} \textsc{ActLit}(T_i)$ is in $map$ only if $S'$ is inadequate. $M$ would falsify this clause only if $S \subseteq S'$ which is imposssible by Corollary 1. Thus $M$ is a model for $map$. The case for an unexplored *MIS* is symmetric. ∎

**Corollary 2.** *For $(I,T) \vdash P$, $map$ is unsatisfiable iff every $S \in \mathcal{P}(T)$ has been explored or excluded.*

*Proof:* Immediate from the definition of $map$ and Lemma 3. ∎

Algorithm 1 shows the process of capturing all *MIVC*s,

which are kept in set $A$, along with a warning flag, explained below. In line 2, we create the set of activation literals used by function ACTLIT. Line 3 initializes $map$ with $\top$ over the set of literals we have. The main loop of state exploration starts at line 4 and continues until $map$ becomes UNSAT which means all the *MIVC*s have been found. We assume we have a function CHECKSAT that determines if an existentially quantified formula is satisfiable or not.[2] As long as $map$ is satisfiable, the algorithm computes a *maximal* SAT model for it (line 5). In this context, a maximal SAT model is a model with as many $true$ assignment as possible without violating a clause; this problem, is equivalent to the MaxSAT problem, which has been well studied in the literature [29], [30].[3] So, we assume there is a method by which we are able to have a maximal model of $map$. Line 6 extracts a set $M \in \mathcal{P}(\mathcal{A})$ of literals assigned to $true$ in the model. Then, we need to obtain the corresponding set of $S$ in $\mathcal{P}(T)$, which is done with function $\text{ACTLIT}^{-1}$ in line 7.

We also assume there is a function CHECKADQ that checks whether or not $P$ is provable by a given subset of $T$. Note that from Theorem 1, finding a minimal is undecidable if the original checking problem is undecidable. Thus, for undecidable model checking problems, CHECKADQ can return UNKNOWN (after a user-defined timeout) as well as ADEQUATE or INADEQUATE. For a given set $S$, if our implementation is unable to prove the property, we conservatively assume that the property is falsifiable and set a warning flag $w$ to the user that the results may be approximate. if $S$ is adequate, a *MIVC* is computed by GETIVC and added to set $A$ (lines 10-11).[4] In this case $map$ is constrained by a new clause in a way described before and shown in line 12. However, in the case that $S$ is inadequate or unknown, $map$ is constrained by the corresponding literals from $T \setminus S$ in line 14. Finally, if $S$ is unknown, the warning flag $w$ is set to true, as the results may be approximate (lines 15-16).

**Theorem 2.** *Algorithm 1 will terminate.*

*Proof:* We assume that CHECKADQ has a finite timeout, so all operations within the loop require finite time. Each iteration of the while loop in Algorithm 1 blocks at least one element of $\mathcal{P}(T)$ which was not previously blocked. Since $\mathcal{P}(T)$ is finite, the algorithm terminates. ■

**Theorem 3.** *If no approximation warning is returned ($w$ is* FALSE*), Algorithm 1 enumerates all MISes and MIVCs.*

[2]We assume readers are familiar with the Boolean satisfiability problem, which is the problem of determining whether there exists an assignment that satisfies a given propositional formula. For more information, refer to [28].

[3]MaxSAT is defined as the problem of satisfying as many (weighted) clauses as possible in a SAT instance. For $N$ variables, similar to the MaxSAT problem, each clauses is weighted at $N+1$ and extra unit-weight clauses are added forcing each variable to 1.

[4]Note that CHECKADQ can be any method that verifies a safety property, such as K-induction, and the GETIVC function can be any function that returns an (approximately) minimal IVC, such as the IVC_UC or IVC_UCBF algorithms from [7]. The only requirement is that it follows the definition of an inductive validity core, that is: $S' \leftarrow \text{GETIVC}(P,S)$ implies that $S' \subseteq S$ and $(I,S') \vdash P$.

**Algorithm 1:** Algorithm All_IVCs for computing $AIVC$

```
input : (I,T) ⊢ P
output: AIVC(P), Approximation warning flag w
1  A ← ∅; w ← FALSE
2  Create activation literals {a_1, ..., a_n}
3  map ← ⊤
4  while CHECKSAT(map) do
5      model ← build a maximal model of map
6      M ← extract the set of variables assigned true in
         model
7      S ← ⋃_{a_i ∈ M} ACTLIT^{-1}(a_i)
8      res ← CHECKADQ(P,S)
9      if res = ADEQUATE then
10         S' ← GETIVC(P,S)
11         A ← A ∪ {S'}
12         map ← map ∧ (⋁_{T_i ∈ S'} ¬ACTLIT(T_i))
13     else
14         map ← map ∧ (⋁_{T_i ∈ (T\S)} ACTLIT(T_i))
15         if res = UNKNOWN then
16             w ← TRUE
17 return A, w
```

*Proof:* By Theorem 2 the algorithm terminates. This means $map$ is eventually unsatisfiable. If $w =$ FALSE then all model checking problems are solved definitively (no UNKNOWN results), so by Lemma 3, all *MIS*es and *MIVC*s are either explored or excluded. However, by maximality and Lemma 1, an *MIS* can never be excluded. Similarily, by minimality and Corollary 1, a *MIVC* can never be excluded. Thus all *MIS*es and *MIVC*s are explored and are elements of $A$ by the end of the algorithm. ■

Note that none of the proofs above require that GETIVC returns a minimal IVC. From [7], it is computationally cheap to find an approximately minimal *IVC* using the algorithm IVC_UC; however, using the better, usually minimal *IVC* using the IVC_UCBF algorithm is computationally expensive. For efficiency reasons, it is much better to use the approximate IVC_UC algorithm to compute the set of all *MIVC*s. The IVC_UCBF algorithm attempts to repeatedly prove the property by brute-force removing elements (BF = “brute force”), so does much of the work of Algorithm 1 in a way that is not effective towards finding other IVCs. The overhead of the IVC_UC algorithm is on average 10% over the baseline proof, as opposed to 882% for the IVC_UCBF algorithm. In addition, the average increase in size of IVCs returned by IVC_UC is approximately 10% of the IVC_UCBF algorithm.

On the other hand, if GETIVC does not return minimal adequate sets, at the end of the process, set $A$ may contain both *MIVC*s and some supersets of *MIVC*s. To make sure that the algorithm only returns the minimal adequate sets (*MIVC*s), all we need is to remove any supersets of other sets in $A$. We

can do this "on the fly" by changing line 11 to the following: $A \leftarrow A \cup \{S'\} \setminus \{S \mid S \in A \wedge S' \subset S\}$. Obviously, the closer to minimal the results of GETIVC are, the fewer iterations are required for Algorithm 1 to terminate. Each non-minimal adequate set returned by GETIVC will induce an additional iteration for Algorithm 1.

## V. ILLUSTRATION

To illustrate the `All_IVCs` algorithm we use the example presented in Section II with $P = (\texttt{on_p})$ . For better description, we view $T$ as an ordered set of its top-level conjuncts; i.e. $T = \{$ `a1_below, a2_below, a1_above, a2_above, one_below, both_above, doi_on, on_p` $\}$. The algorithm starts with creating activation literals for each $T_i \in T$. Let the ordered set of Boolean variables $\{a_1, \ldots, a_8\}$ be the corresponding literals to the elements of $T$ (e.g. ACTLIT(`a1_below`) $= a_1$ and ACTLIT(`on_p`) $= a_8$). Then, line 3 initializes *map* with $\top$.

In the first iteration of the `while` loop, since *map* is empty, it is satisfiable, and a model for it can be any subset of literals. So obviously, the first maximal model of *map* contains all the literals, which means, in line 6, $M = \{a_1, \ldots, a_8\}$, and in line 7, $S = T$. Since $S$ is adequate for $P$, the GETIVC module is called in line 10. Suppose the returned *MIVC* by this function is $S' = \{$`a1_below, one_below, doi_on, on_p`$\}$; this set is added to $A$ in line 11, and thus it comes to adding a new clause to *map* (line 12), which makes $map = (\neg a_1 \vee \neg a_5 \vee \neg a_7 \vee \neg a_8)$. As discussed, this constraint marks all the supersets of $S'$ as blocked and prunes them off the search space.

For the second iteration, *map* is still satisfiable, so the algorithm gets to find a maximal model of it in line 5. Suppose this time, the maximal model makes $M = \{a_1, \ldots, a_7\}$, which leads to $S = T \setminus \{\texttt{on_p}\}$ in line 7. Since $S$ is inadequate for $P$, the algorithm jumps to line 12 updating *map* as $map \leftarrow map \wedge a_8$. Adding this new clause removes all the subsets of $T \setminus \{\texttt{on_p}\}$ from the search space. Similarly, in the third iteration, if the maximal model of *map* yields $M = \{a_1, \ldots, a_4, a_6, \ldots, a_8\}$, then $S = T \setminus \{\texttt{one_below}\}$ will be another inadequate set that makes *map* become $map \leftarrow map \wedge a_5$ in line 14.

Suppose, in the fourth iteration, the maximal model leads to $M = \{a_2, \ldots, a_8\}$ and $S = T \setminus \{\texttt{a1_below}\}$ in lines 6 and 7. Since this $S$ is adequate for $P$, GETIVC computes a new *MIVC* in line 10. Let the new *MIVC* be $S' = \{$`a2_below, one_below, doi_on, on_p`$\}$; after adding this set to $A$, it is time to constrain *map* by a new clause in line 11, which results in $map \leftarrow map \wedge (\neg a_2 \vee \neg a_5 \vee \neg a_7 \vee \neg a_8)$.

After these iterations, *map* is still satisfiable, and the maximal model is $S = T \setminus \{\texttt{a1_below}, \texttt{a2_below}\}$ in line 7. In this case, $S$ is inadequate, so we update *map* as $map \leftarrow map \wedge (a_1 \vee a_2)$ (line 14). After adding this new clause to *map*, all the subsets of $T \setminus \{\texttt{a1_below}, \texttt{a2_below}\}$ will be blocked. The algorithm continues similar to the forth iteration leading to $S$ (in line 7) and *map* (in line 14) to be as $S = T \setminus \{\texttt{doi_on}\}$ and $map \leftarrow map \wedge a_7$.

Finally, after the sixth iteration, *map* becomes `UNSAT` and the algorithm terminates. Note that $MIS$es and $IVC$s may be discovered in different orders from what explained here. The order by which sets are explored is quite dependent on the maximal model returned in line 5 as well as the *MIVC*s returned in line 10 because there could be several distinct maximal models (*MIS*es) and *MIVC*s. For this example with a $|T| = 8$ and $|\mathcal{P}(T)| = 2^8$, a brute force approach of power set exploration needs to look into 256 cases. However, the `All_IVCs` algorithm only explored 6 cases to cover the entire power set.

## VI. IMPLEMENTATION

We have implemented the `All_IVCs` algorithm in an industrial model checker called `JKind` [25], which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [31] as input. The translation of Lustre into a symbolic transition system in `JKind` is straightforward and is similar to what is described in [32]. Verification is supported by multiple "proof engines" that execute in parallel, including K-induction, property directed reachability (PDR), and lemma generation engines that attempt to prove multiple properties in parallel. To implement the engines, `JKind` emits SMT problems using the theories of linear integer and real arithmetic. `JKind` supports the `Z3`, `Yices`, `MathSAT`, `SMTInterpol`, and `CVC4` SMT solvers as back-ends. When a property is proved and IVC generation is enabled, an additional parallel engine executes the `IVC_UC` algorithm [7] to generate an (approximately) minimal IVC. To implement our method, we have extended `JKind` with a new engine that implements Algorithm 1 on top of `Z3`. We use the `JKind` IVC generation engine to implement the GETIVC procedure in Algorithm 1.

As mentioned in Section IV the CHECKADQ procedure may not terminate. In our implementation, we measure the time required to prove the property and the initial given the full model (*proof-time*), and the time required to calculate the first (approximate) IVC using `IVC_UC` (*IVC_UC-time*). We then set a timeout for each iteration of the `All_IVCs` algorithm to (30 sec + 5 × (*proof-time* + *IVC_UC-time*)). In almost all cases in our experiment and our use of the tools, this timeout is sufficient to ensure exact results. In the experiment, only 15 of 475 models (3%) had potentially approximate results. It is important to note that by increasing the timeout, it is possible that in some cases smaller IVCs can be generated, but the general problem will remain due to the undecidability of the model checking problem.

## VII. EXPERIMENT

We are interested in examining the *efficacy* and *efficiency* of generating all minimal IVCs, as compared to algorithms for computing a *single approximately minimal* IVC, and a *minimal IVC* as implemented in [7] using the `IVC_UC` and `IVC_UCBF` algorithms, respectively. We would also like to know how performance is affected by the size of models and number of minimal IVCs. Finally, we are also interested in determining whether the `All_IVCs` algorithm generates *smaller* cores than

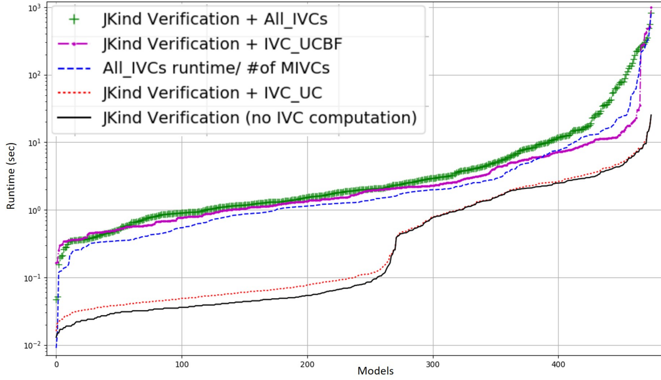


Fig. 3. Runtime of All_IVCs, IVC_UCBF, and IVC_UC algorithms

are generated by the IVC_UCBF algorithm that generates a single MIVC. Therefore, we investigate the following research questions:

- **RQ1:** How expensive is it to compute the All_IVCs algorithm for determining all minimal IVCs when compared to the IVC_UC and IVC_UCBF algorithms, which find a single approximately minimal and guaranteed minimal IVC?
- **RQ2:** How is the verification time of the All_IVCs algorithm affected by the baseline proof time and the number of IVCs that can be found for a property?
- **RQ3:** How large are the IVCs produced by the All_IVCs algorithm compared to those of IVC_UC and IVC_UCBF?

### A. Experimental Setup

The benchmark contains 475 Lustre models, 395 from [32] and 80 industrial models derived from [33] and other sources. Most of the benchmark models from [32] are small (10kB or less, with 6-40 equations) and include a range of hardware benchmarks and software problems involving counters that are difficult to solve inductively. The 80 industrial models each contain over 600 equations and are each ≥80kB in size.

We selected only benchmark problems consisting of a Lustre model with properties that JKind could prove with an hour timeout. For each test model, we computed All_IVCs, IVC_UC, and IVC_UCBF algorithms in a configuration with the Z3 solver and the “fastest” mode of JKind (which involves running the $k$-induction and PDR engines in parallel and terminating when a solution is found). The experiments were run on an Intel(R) i5-4690, 3.50GHz, 16 GB memory machine running Linux, and are available at [34].

### B. Experimental Results

In this section, we examine our experimental results to address the research questions defined in the experiment.

*1) RQ1:* To address RQ1, we measured the performance overhead of the various IVC algorithms against the baseline time necessary to find a proof using inductive model checking. Fig. 3 provides an overview of the overhead of the All_IVCs algorithm in comparison with the IVC_UC and IVC_UCBF algorithms. In the figure, each curve is ranked along the x-axis according to the time required for the algorithm to terminate for each analysis problem. Table I provides a summary of the computation time and the overhead of different algorithms. The IVC_UC algorithm imposes a 1.25x overhead to the baseline proof time, whereas both the IVC_UCBF and All_IVCs algorithms add a substantial time penalty: IVC_UCBF and All_IVCs add a (mean) 18.8x and 31.3x overhead, respectively, to the proof time. For small models, much of this penalty is due to starting many instances of the SMT solver; if we examine models that require $\geq 1s$ of analysis time, the mean overhead of All_IVCs over the baseline analysis drops from 31.3x to 9.7x.

TABLE I
RUNTIME AND OVERHEAD OF DIFFERENT COMPUTATIONS

| runtime (sec) | min | max | mean | stdev |
|---|---|---|---|---|
| *proof-time* | 0.016 | 25.489 | 1.250 | 2.381 |
| All_IVCs | 0.009 | 792.01 | 16.457 | 64.491 |
| IVC_UCBF | 0.163 | 996.734 | 11.987 | 68.525 |
| IVC_UC | 0.003 | 1.126 | 0.078 | 0.158 |

*2) RQ2:* For this question, we examine how the proof time of the original model and the number of MIVCs associated with the property affects the analysis time of the All_IVCs algorithm. Fig. 4 provides an overview of this data. The data in Fig. 4 is sorted twice along the x-axis: the major axis is the number of MIVCs that exist for the model, and the minor axis is the analysis time of the baseline model. In this graph, the graph shows how both factors effect the performance of the All_IVCs algorithm. Note that there are two scales for the y-axis: the scale on the left is a logarithmic scale of performance in terms of the run time; the scale on the right is a linear scale based on the number of minimal IVCs discovered.

Fig. 4 shows two distinct trends. First, for models whose baseline proofs are inexpensive and that only have a single MIVC, the All_IVCs is roughly equivalent in performance to the IVC_UCBF. However, as proofs become more expensive for a single MIVC, the All_IVCs begins to underperform the IVC_UCBF, this is the case for the properties with one MIVC. In the cases where several MIVCs are found, the performance of the All_IVCs is driven to a large degree by the number of MIVCs that exist: the more MIVCs associated with a property, the higher the expense of All_IVCs as compared to the IVC_UCBF algorithm.

*3) RQ3:* For this research question, we analyzed the minimality of the discovered IVC by each algorithm (Figure 5). Since 394 of the models had only one MIVC, for these models, the size of the minimum model produced by the All_IVCs algorithm should be the same as the IVC_UCBF algorithm. For the remainder, even when multiple MIVCs were produced, in only 12 cases did the All_IVCs produce smaller minimal IVCs. For these 12 models, the smallest MIVC was 16% the size of the MIVC produced by IVC_UCBF, and in the most dramatic case, the number of elements shrank from 30 to 5.

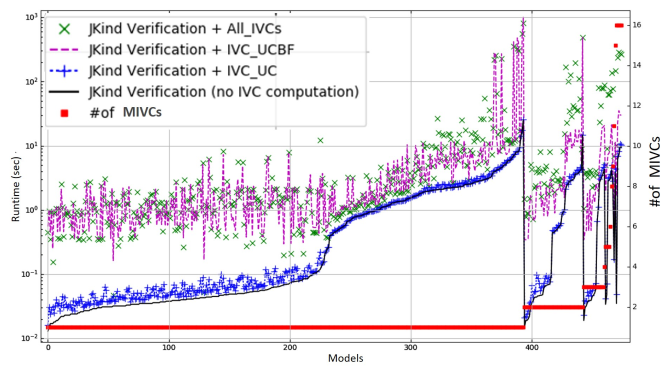


Fig. 4. Runtime of different computations along with the number of MIVCs

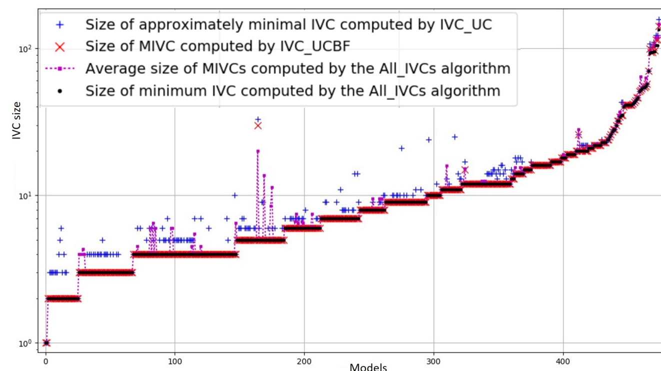


Fig. 5. Size of the IVC sets produced by different algorithms

## VIII. Conclusions & Future Work

The idea of extracting a minimal IVC for a given property and its applications was recently introduced in [7]. However, a single IVC often does not provide a complete picture of the traceability from a property to a model. In this paper, we have addressed the problem of extracting *all minimal* IVCs. We have shown the correctness and completeness of our method and algorithm. In addition, we have a substantial evaluation that shows that the practicality and efficiency of our technique.

Our method is inspired by a recent work in the domain of satisfiability analysis [22]. One interesting future direction is to devise similar MIVC enumeration algorithms based on other studies on MUSes extraction such as [21]. We are also looking into improving our implementation by using more efficient methods for the CheckAdq and GetIVC modules used by our algorithm. Another interesting direction is to parallelize the enumeration process: it is certainly possible to ask for multiple distinct maximal models to be solved in parallel.

We also plan to investigate additional applications of the idea. When performing *compositional verification*, the All-IVCs technique may be able to determine *minimal component sets* within an architecture that can satisfy a given set of requirements, which may be helpful for design-space exploration and synthesis. Finally, we are interested in adapting the notion of (all) validity cores for *bounded* model checking for quantifying how much of models have been explored by bounded analysis.

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

## References

[1] N. Een *et al.*, "Efficient implementation of property directed reachability," in *FMCAD'11*.
[2] M. Sheeran *et al.*, "Checking safety properties using induction and a SAT-solver," in *FMCAD'02*, 2000.
[3] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *STTT*, 2003.
[4] M. Whalen *et al.*, "Integration of formal analysis into a model-based software development process," in *FMICS*, 2007.
[5] L. Pike, "A note on inconsistent axioms in rushby's "systematic formal verification for fault-tolerant time-triggered algorithms"," *TSE*, 2006.
[6] "Cadence JasperGold Formal Verification Platform," https://www.cadence.com/.
[7] E. Ghassabani *et al.*, "Efficient generation of inductive validity cores for safety properties," in *FSE'16*, 2016.
[8] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formula," in *SAT'03*.
[9] A. Murugesan *et al.*, "Complete traceability for requirements in satisfaction arguments," in *RE'16 (RE@Next! Track)*, 2016.
[10] E. Ghassabani *et al.*, "Proof-based coverage metrics for formal verification," in *ASE'17*, 2017.
[11] A. Mebsout and C. Tinelli, "Proof certificates for smt-based model checkers for infinite-state systems," in *FMCAD'16*, 2016.
[12] A. Ivrii *et al.*, "Small inductive safe invariants," in *FMCAD'14*, 2014.
[13] "Center of Excellence for Software Traceability," http://www.coest.org, 2016.
[14] J. H. Hayes *et al.*, "Improving requirements tracing via information retrieval," in *RE'03*, 2003.
[15] J. Cleland-Huang *et al.*, "Best practices for automated traceability," *Computer*, 2007.
[16] M. H. Liffiton *et al.*, "From MaxSAT to MinUNSAT: Insights and applications," *Ann Arbor*, 2005.
[17] F. Bacchus and G. Katsirelos, "Using minimal correction sets to more efficiently compute minimal unsatisfiable sets," in *CAV'15*, 2015.
[18] A. Belov and J. Marques-Silva, "Muser2: An efficient mus extractor," *JSAT journal*, 2012.
[19] A. Belov *et al.*, "Core minimization in sat-based abstraction," in *DATE'13*, 2013.
[20] A. Belov *et al.*, "Towards efficient MUS extraction," *AI Communications*, 2012.
[21] A. Nadel *et al.*, "Accelerated deletion-based extraction of minimal unsatisfiable cores," *JSAT journal*, 2014.
[22] M. Liffiton *et al.*, "Fast, flexible MUS enumeration," *Constraints*, 2016.
[23] Z. Hanna *et al.*, "Formal verification coverage metrics for circuit design properties," 2015. [Online]. Available: https://www.google.com/patents/US20150135150
[24] M. Heimdahl *et al.*, "Deviation analysis via model checking," in *ASE'02*, 2002.
[25] "JKind," http://loonwerks.com/tools/jkind.html.
[26] T. Kahsai *et al.*, "Incremental verification with mode variable invariants in state machines," in *NFM'12*, 2012.
[27] N. Amla *et al.*, "An analysis of sat-based model checking techniques in an industrial environment," in *CHARME'05*, 2005.
[28] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC*, 1971.
[29] J. Davies and F. Bacchus, "Solving MAXSAT by solving a sequence of simpler sat instances," in *CP'11*, 2011.
[30] A. Morgado *et al.*, "Iterative and core-guided MaxSAT solving: A survey and assessment," *Constraints*, 2013.
[31] N. Halbwachs *et al.*, "The synchronous dataflow programming language Lustre," *Proceedings of the IEEE*, 1991.
[32] G. Hagen and C. Tinelli, "Scaling up the formal verification of lustre programs with smt-based techniques," in *FMCAD'08*, 2008.
[33] A. Murugesan *et al.*, "Compositional verification of a medical device system," in *HILT'13*, 2013.
[34] "All IVCs repository," https://github.com/elaghs/Working/tree/master/all_ivcs/experiments.