Online Enumeration of All Minimal Inductive Validity Cores

Jaroslav Bendík[1], Elaheh Ghassabani[2], Michael Whalen[2], and Ivana Černá[1]

[1] Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbendik,cerna}@fi.muni.cz
[2] Department of Computer Science & Engineering, University of Minnesota, MN, USA
{ghass013,mwwhalen}@umn.edu

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. Minimal Inductive Validity Cores (MIVCs) trace a property to a minimal set of model elements necessary for constructing a proof, and can help to explain why a property is true of a model. In addition, the traceability information provided by MIVCs can be used to perform a variety of engineering analysis such as coverage analysis, robustness analysis, and vacuity detection. The more MIVCs are identified, the more precisely such analyses can be performed. Nevertheless, a full enumeration of all MIVCs is in general intractable due to the large number of possible model element sets. The bottleneck of existing algorithms is that they are not guaranteed to emit minimal IVCs until the end of the computation, so returned results are not known to be minimal until all solutions are produced.
In this paper, we propose an algorithm that identifies MIVCs in an *online* manner (i.e., one by one) and can be terminated at any time. We benchmark our new algorithm against existing algorithms on a variety of examples, and demonstrate that our algorithm not only is better in intractable cases but also completes the enumeration of MIVCs faster than competing algorithms in many tractable cases.

Keywords: Inductive Validity Cores, SMT-based model checking, Inductive proofs, Traceability, Proof cores

1 Introduction

Symbolic model checking using induction-based techniques such as IC3/PDR [9], k-induction [24], and k-liveness [8] can be used to determine whether properties hold of complex finite or infinite-state systems. Such tools are popular both because they are highly automated (often requiring no user interaction other than the specification of the model and desired properties), and also because, in the event of a violation, the tool provides a counterexample demonstrating a situation in which the property fails to hold. These counterexamples can be used

both to illustrate subtle errors in complex hardware and software designs [22, 21] and to support automated test case generation [26, 27].

If a property is proved, however, most model checking tools do not provide additional information. This can lead to situations in which developers have an unwarranted level of confidence in the behavior of the system. Issues such as vacuity [17], incorrect environmental assumptions [25], and errors either in English language requirements or formalization can all lead to failures of "proved" systems. Thus, even if proofs are established, one must approach verification with skepticism.

Recently, *proof cores* [1] 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 [11] 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 [28] 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. Depending on the model and property to be analyzed, there are many possible MIVCs, and there is often substantial diversity between the IVCs used for proof. In previous work [11, 23, 13, 12] we have explored several different uses of IVCs, including:

Traceability: Inductive validity cores can provide accurate traceability matrices with no user effort. Given multiple IVCs, *rich traceability* matrices [23] can be automatically constructed that provide additional insight about *required* vs. *optional* design elements.

Vacuity detection: Syntactic vacuity detection (checking whether all subformulae within a property are necessary for its validity) has been well studied [17]. IVCs allow a generalized notion of vacuity that can indicate weak or mis-specified properties even when a property is syntactically non-vacuous.

Coverage analysis: Coverage analysis provides a metric as to whether a set of properties is adequate for the model. Several different notions of coverage have been proposed [7, 16], but these tend to be very expensive to compute. IVCs provide an inexpensive coverage metric by determining the percentage of model atoms necessary for proofs of all properties.

Impact Analysis: Given a single (or for more accurate results, 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.

Design Optimization: A practical way of calculating all MIVCs allows synthesis tools to find a minimum set of design elements (optimal implementation) for a certain behavior. Such optimizations can be performed at different levels of synthesis.

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 minimal). In previous work, we have developed an efficient *offline* algorithm [12] for finding all minimal IVCs based on the MARCO algorithm for MUSes [18]. The algorithm is considered *offline* because it is not until all IVCs have been computed that one knows whether the solutions computed are, in fact, minimal. In cases in which models contain many IVCs, this approach can be impractically expensive or simply not terminate.

In this paper, we propose a novel *online* algorithm for MIVC enumeration. With this algorithm, solutions are produced incrementally, and each solution produced is guaranteed to be minimal. Therefore, the algorithm produces at least some MIVCs even in the case of models for which is a complete MIVC enumeration intractable. Moreover, the proposed algorithm is often more efficient then the baseline MARCO also in the case of tractable models. We demonstrate this via an experimental evaluation.

The rest of the paper is organized as follows. In Section 2 we define all the necessary notions. Section 3 summarizes the existing techniques. In Section 4 we present our novel algorithm. Section 5 provides an example execution of our algorithm. Finally, sections 4.6 and 6 cover implementation details and present experimental results.

2 Preliminaries

A transition system (I, T) over a state space S consists of an initial state predicate $I : S \to \mathit{bool}$ and a transition step predicate $T : S \times S \to \mathit{bool}$. The notion of reachability for (I, T) is defined as the smallest predicate $R : S \to \mathit{bool}$ satisfying the following formulae:

$$\forall s \in S : I(s) \Rightarrow R(s)$$
$$\forall s, s' \in S : R(s) \wedge T(s, s') \Rightarrow R(s')$$

A safety property $P : S \to \mathit{bool}$ holds on a transition system (I, T) iff it holds on all reachable states, i.e., $\forall s \in S : R(s) \Rightarrow P(s)$. We denote this by $(I, T) \vdash P$. We assume the transiton step predicate T is equivalent to a conjunction of transition step predicates $T_1, \ldots, T_n$, called top level conjuncts. In such case, T can be identified with the set of its top level conjuncts $\{T_1, \ldots, T_n\}$. By further abuse of notation, we write $T \setminus \{T_i\}$ to denote removal of top level conjunct T_i from T, and $T \cup \{T_j\}$ to denote addition of top level conjunct T_j to T.

Definition 1. *A set of conjuncts $U \subseteq T$ is an* Inductive Validity Core (IVC) *for $(I, T) \vdash P$ iff $(I, U) \vdash P$. Moreover, U is a* Minimal IVC (MIVC) *for $(I, T) \vdash P$ iff $(I, U) \vdash P$ and $\forall T_i \in U : (I, U \setminus \{T_i\}) \nvdash P$.*

Note, that the minimality is with respect to the set inclusion and not wrt cardinality. There can be multiple MIVCs with different cardinalities. For an illustration of the concepts on a particular transition system, please refer e.g. to the Altitude Switch example [12].

3 Existing Techniques

Consider first a naive enumeration algorithm that explicitly checks each subset of T for being an IVC and then finds the minimal IVCs using subset inclusion relation. The main disadvantage of this approach is the large number of checks since there are exponentially many subsets of T. We briefly describe existing techniques that can be used to find all MIVCs while checking only a a small portion of subsets of T for being IVCs. Most of the techniques were inspired by the MUS enumeration techniques [19, 5, 6] proposed in the area of constraint processing and applied by Ghassabani et al. [12, 11].

Definition 2 (Inadequacy). *A set of conjuncts $U \subseteq T$ is an* inadequate *set for $(I, T) \vdash P$ iff $(I, U) \nvdash P$. Especially, $U \subseteq T$ is a* Maximal Inadequate Set (MIS) *for $(I, T) \vdash P$ iff U is inadequate and $\forall T_i \in (T \setminus U) : (I, U \cup \{T_i\}) \vdash P$.*

Inadequate sets are duals to inductive validity cores. Each $U \subseteq T$ is either inadequate set or an inductive validity core. In order to unify the notation, we use notation *inadequate* and *adequate*. Note that especially minimal inductive validity cores can be thus called minimal adequate sets.

The first property used to improve the naive enumeration algorithm is the *monotonicity* of adequacy with respect to the subset inclusion.

Lemma 1 (Monotonicity). *If a set of conjuncts $U \subseteq T$ is an adequate set for $(I, T) \vdash P$ than all its supersets are adequate for $(I, T) \vdash P$ as well:*

$$\forall U_1 \subseteq U_2 \subseteq T : (I, U_1) \vdash P \Rightarrow (I, U_2) \vdash P.$$

Symmetrically, if $U \subseteq T$ is an inadequate set for $(I, T) \vdash P$ than all its subsets are inadequate for $(I, T) \vdash P$ as well:

$$\forall U_1 \subseteq U_2 \subseteq T : (I, U_2) \nvdash P \Rightarrow (I, U_1) \nvdash P.$$

Proof. If $U_1 \subseteq U_2$ then reachable states of (I, U_2) form a subset of the reachable states of (I, U_1).

The monotonicity allows to determine status of multiple subsets of T while using only a single check for adequacy. For example, if a set $U \subseteq T$ is determined to be adequate, than all of its supersets are adequate and do not need to be explicitly checked. Let $Sup(U)$ and $Sub(U)$ denote the set of all supersets and subsets of U, respectively.

Every algorithm for computing MIVCs has to determine status (i.e adequate or inadequate) of every subset of T. In order to distinguish the subsets whose status is already known from those whose status is not known yet, we denote the former subsets as *explored* subsets and the latter as *unexplored* subsets. Moreover, we distinguish *maximal* unexplored subsets:

- U_{max} is a *maximal unexplored subset* of T iff $U_{max} \subseteq T$, U_{max} is unexplored, and each of its proper supersets is explored.

A straightforward way to find a (so far unexplored) MIVC of T is to find an unexplored adequate subset $U \subseteq T$ and turn U into an MIVC by a process called *shrinking*. A shrinking procedure iteratively attempts to remove elements from

Algorithm 1: A näive shrinking algorithm

input : $(I, U) \vdash P$
output: MIVC for $(I, U) \vdash P$
1 **for** $T_i \in U$ **do**
2 | **if** $(I, U \setminus \{T_i\}) \vdash P$ **then** $U \leftarrow U \setminus \{T_i\}$
3 **return** U

the set that is being shrunk, checking each new set for adequacy and keeping only changes that leave the set adequate. A näive example is shown in Algorithm 1.

Ghassabani et al. [12] proposed an algorithm for MIVC enumeration which is based on the MUS enumeration algorithm MARCO [19]. The algorithm iteratively chooses maximal unexplored subsets and tests them for adequacy. Each maximal subset that is found to be adequate is then shrunk into a MIVC. This algorithm enumerates MIVCs in an online manner with a relatively steady rate of the enumeration. However, an evaluation of the algorithm shown that it is rather slow since the shrinking procedure can be extremely time consuming as each check for adequacy is in fact a model checking problem.

Therefore, Ghassabani et al. [12] proposed another algorithm which, instead of computing MIVCs in on online manner, rather computes only *approximately* minimal IVCs. In particular, it iteratively picks maximal unexplored subsets, checks them for adequacy, and turns the adequate subsets into approximately minimal IVCs using the approximation algorithm `IVC_UC` [11]. `IVC_UC` is able to identify IVCs which are often very close to actual MIVCs, yet cheap to compute. This enumeration algorithm computes approximately minimal IVCs, and identifies MIVCs at the very end of the computation. An experimental evaluation shows that the latter algorithm computes all MIVCs much faster than the algorithm based on shrinking. However, it does not enumerate MIVCs online and thus on some benchmarks may produce no MIVCs within a given time limit.

4 Grow-Shrink Algorithm

In this section, we propose a novel algorithm for online MIVC enumeration. The MIVCs are found using an improved shrinking procedure. Moreover, the algorithm uses a procedure *grow*, which is a dual of the shrinking procedure. The algorithm also maintains the set *Unexplored* of unexplored subsets.

We can effectively use the set *Unexplored* for speeding up the shrinking procedure. When testing the set $U \setminus \{T_i\}$ (see line 2 in Algorithm 1) we first check whether $U \setminus \{T_i\}$ is still unexplored. If $U \setminus \{T_i\}$ is already explored, then its status is already known and no test for adequacy is needed.

4.1 Shrink Procedure

In the following observation, we specify which explored subsets can be used to speed up the shrinking procedure.

Algorithm 2: Approximate grow

input : $(I, T) \vdash P$
input : inadequate $U \subset T$ for $(I, T) \vdash P$
input : set *Unexplored* of unexplored subsets of T
output: approximately maximal inadequate set for $(I, T) \vdash P$

```
1 M ← a maximal M ∈ Unexplored such that M ⊇ U
2 while (I, M) ⊢ P do
3 |   M_IVC ← IVC_UC((I, M), P)        // gets approximately minimal IVC
4 |   T_i ← choose T_i ∈ (M_IVC \ U)
5 |   M ← M \ {T_i}
6 return M
```

Observation 1. *Let U_1, U_2 be subsets of T such that U_1 is explored, U_2 is unexplored, and $U_1 \subset U_2$. Then U_1 is inadequate for $(I, T) \vdash P$.*
Symetrically, if U_1, U_2 are subsets of T such that U_2 is explored, U_1 is unexplored, and $U_1 \subset U_2$. Then U_2 is adequate for $(I, T) \vdash P$.

Proof. If U_1 is adequate, then all of its supersets are necessarily adequate. Thus, if U_1 is determined to be adequate, then not just U_1 but also all of its supersets becomes explored. Since U_1 is explored and U_2 is unexplored, then U_1 is necessarily an inadequate subset of T.

In other words, during the shrinking procedure, we are guaranteed that whenever we find an explored set, this set is inadequate. Thus, as a further optimization in our algorithm we try to identify as many inadequate sets as possible before starting the shrinking procedure. The search for inadequate sets is done with the help of the grow procedure.

4.2 Grow Procedure

Recall that if a set is determined to be inadequate then all of its subsets are necessarily also inadequate. Therefore, the larger is the set that is determined to be inadequate, the more inadequate sets are explored. To identify inadequate sets as quickly as possible we search for maximal inadequate sets (MISes).

In order to find a MIS, we can find an inadequate set $U \subset T$ and use a process called *grow* which turns U to a MIS for $(I, T) \vdash P$. The grow procedure iteratively attempts to add elements from $T \setminus U$ to U, checking each new set for adequacy and keeping only changes that leave the set inadequate. Same as in the case of shrink procedure, we can use the set *Explored* to avoid checking sets whose status is already known. However, such grow procedure might still perform too many checks for adequacy and thus be very inefficient.

Instead, we propose to use a different approach. Algorithm 2 shows a procedure that, given an inadequate set U for $(I, T) \vdash P$, finds an *approximately* maximal inadequate set. It first finds some maximal unexplored set M such that $M \supseteq U$ and checks it for adequacy. If M is inadequate, then it is necessarily a

Algorithm 3: Solving algorithm

```
1 Function Solve(I,U,P):
2 |   res ← CheckAdq(I, U, P)
3 |   if res = UNKNOWN then
4 |   |   approximateWarning ← true                    // a global variable
5 |   return (res = ADEQUATE)
```

MIS (this is a straightforward consequence of Observation 1.) Otherwise, if M is adequate then it is iteratively reduced until an inadequate set is found.

In particular, whenever M is found to be adequate, the approximative algorithm `IVC_UC` by Ghassabani et al. [11] is used to find an approximately minimal IVC M_{IVC} of M. M_{IVC} succinctly explains M's adequacy. In order to turn M into an inadequate set, it is reduced by one element from $M_{IVC} \setminus U$ and checked for adequacy. If M is still adequate then the approximate growing procedure continues with a next iteration. Otherwise, if M is inadequate, the procedure finishes.

Proposition 1. *Given an unexplored inadequate set U for $(I, T) \vdash P$ and a set Unexplored of unexplored subsets of T, Algorithm 2 returns an* unexplored *inadequate subset M of T.*

Proof. Let us denote initial M as M_{init}. Since $M_{init} \supseteq U$ and M is recursively reduced only by elements that are not contained in U, then in every iteration holds that $U \subseteq M \subseteq M_{init}$. Since both U, M_{init} are unexplored, then M is necessarily also unexplored.

4.3 Solve Procedure

Determining whether a particular subset of elements $U \subset T$ can prove a property of interest P is as hard as model checking ([11], Theorem 1). Thus, in the general case, determining whether a set of model elements is an MIVC may not be possible for model checking problems that are in general undecidable, such as those involving infinite theories. We assume there is a function `CheckAdq` that checks whether or not P is provable for some (I, U). `CheckAdq` can return UNKNOWN (after a user-defined timeout) as well as ADEQUATE or INADEQUATE. For a given set U, if our implementation is unable to prove the property, we conservatively assume that the property is falsifiable and set a global warning flag *approximateWarning* to the user that the results produced may be approximate.

4.4 Complete Algorithm

In this section, we describe, how to combine the shrink and grow methods to form an efficient online MIVC enumeration algorithm. We call the algorithm Grow-Shrink algorithm. Since knowledge of (approximately) maximal inadequate subsets can be exploited to speed up the shrinking procedure, it might be tempting

Algorithm 4: The Grow-Shrink algorithm

```
1  Function Init((I, T) ⊢ P):
2      Unexplored ← 𝒫(T)                                  // a global variable
3      shrinkingQueue ← empty queue                       // a global variable
4      approximateWarning ← false                         // a global variable
5      FindMIVCs()
1  Function FindMIVCs():
2      while Unexplored ≠ ∅ do
3          U_max ← a maximal set ∈ Unexplored
4          if Solve(I, U_max, P) then
5              U_IVC ← IVC_UC((I, U_max), P)
6              Shrink(U_IVC)
7          else
8              Unexplored ← Unexplored \ Sub(U_max)
9          while shrinkingQueue is not empty do
10             U ← Dequeue(shrinkingQueue)
11             Shrink(U)
1  Function Shrink(U):
2      growingQueue ← empty queue
3      for T_i ∈ U do
4          if U \ {T_i} ∈ Unexplored then
5              if Solve(I, U \ {T_i}, P) then  U ← U \ {T_i}
6              else  Enqueue(growingQueue, U \ {T_i})
7      output U                                           // Output Minimal IVC
8      UpdateShrinkingQueue(U)
9      Unexplored ← Unexplored \ Sup(U)
10     while growingQueue is not empty do
11         V ← Dequeue(growingQueue)
12         Grow(V)
1  Function Grow(V):
2      M ← a maximal set ∈ Unexplored such that M ⊇ V
3      while Solve(I, M, P) do
4          M_IVC ← IVC_UC((I, M), P)
5          UpdateShrinkingQueue(M_IVC)
6          Enqueue(shrinkingQueue, M_IVC)
7          Unexplored ← Unexplored \ Sup(M_IVC)
8          T_i ← choose T_i ∈ (M_IVC \ V)
9          M ← M \ {T_i}
10     Unexplored ← Unexplored \ Sub(M)
1  Function UpdateShrinkingQueue(U):
2      for V ∈ shrinkingQueue do
3          if U ⊆ V then remove V from shrinkingQueue
```

to first find all MISes. However, this is in general intractable since there can be up to exponentially many MISes (w.r.t. the size of T). Instead, we propose to alternate both the shrinking and growing procedures. Note that during shrinking,

we might determine some subsets to be inadequate. Such subsets can be subsequently used as *seeds* for growing. Dually, adequate subsets that are explored during growing can be later used as *seeds* for the shrinking procedure.

The pseudocode of our algorithm is shown in Algorithm 4. The computation of the algorithm starts with an initialisation procedure `Init` which creates a global variable *Unexplored* for maintaining the unexplored subsets and a global shrinking queue *shrinkingQueue* for storing seeds for the shrinking procedure. Then the main procedure `FindMIVCs` of our algorithm is called.

Procedure `FindMIVCs` works iteratively. In each iteration, the procedure picks a maximal unexplored subset U_{max} and checks it for adequacy. If U_{max} is inadequate, then U_{max} and all of its subsets are marked as explored. Otherwise, if U_{max} is adequate, then the algorithm `IVC_UC` [11] is used to reduce U_{max} into an approximately minimal IVC, and subsequently the procedure `Shrink` is used to shrink it into a MIVC.

Procedure `Shrink` works as described in Section 4.1. However, besides shrinking the given set into a MIVC, the procedure has also another purpose. Every inadequate set that is found during the shrinking is stored in a queue *growingQueue*. At the end of the procedure, all of these inadequate sets are grown into approximately maximal inadequate sets using the procedure `Grow`.

Procedure `Grow` turns a given inadequate set V into an approximately maximal inadequate set M as described in Section 4.2. The resultant set and all of its subsets are marked as explored. Moreover, every adequate set found during the growing is marked as explored and enqueued into *shrinkingQueue*. The queue *shrinkingQueue* is dequeued at the end of each iteration of the main procedure `FindMIVCs` and the sets that were stored in the queue are shrunk to MIVCs.

We need to ensure that each result of the shrinking procedure is a *fresh* MIVC, i.e. that each MIVC is produced only once. We shrink two kinds of inadequate sets in our algorithm: those that result from the inadequate maximal unexplored subsets, and those that are stored in *shrinkingQueue*. In the former case, we always shrunk an unexplored subset U_{IVC} which guarantees that the resultant MIVC U_{MIVC} is unexplored and thus fresh (if U_{MIVC} is already explored, then U_{IVC} would be necessarily also explored). However, in the latter case, all the sets stored in *shrinkingQueue* are already explored. To guarantee that shrinking of the sets from *shrinkingQueue* result only in fresh MIVCs, we maintain the following invariants of the queue:

I1) For each already produced MIVC M holds that there is no U in the queue such that $M \subseteq U$.
I2) There are no two U, V in the queue such that $U \subseteq V$.

To ensure that the invariants hold, we use the procedure `UpdateShrinkingQueue` which given an adequate set U removes from *shrinkingQueue* all supersets of U. We call the procedure every time a MIVC is found and every time a set is added to the queue.

Correctness: The algorithm produces only the MIVCs found by the shrinking procedure and all of them are *fresh*, i.e. produced only once. Only subsets whose

status is known are removed from the set *Unexplored*, thus no MIVC is excluded from the computation. The algorithm terminates and all MIVCs are found since the size of *Unexplored* is reduced after every iteration.

4.5 Symbolic Representation of Unexplored Subsets

Since there are exponentially many subsets of T, it is intractable to represent the set *Unexplored* explicitly. Instead, we use a symbolic representation that is based on a well known isomorphism between finite power sets and Boolean algebras. We encode $T = \{T_1, T_2, \ldots, T_n\}$ by using a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. Each valuation of X then corresponds to a subset of T. This allows us to represent the set of unexplored subsets *Unexplored* using a Boolean formula $f_{Unexplored}$ such that each model of $f_{Unexplored}$ corresponds to an element of *Unexplored*. The formula is maintained as follows:

- Initially, $f_{Unexplored} = True$ since all of $\mathcal{P}(T)$ are unexplored.
- To remove an adequate set $U \subseteq T$ and all its supersets from the set *Unexplored* we add to $f_{Unexplored}$ the clause $\bigvee_{i:T_i \in U} \neg x_i$.
- To remove an inadequate set $U \subseteq T$ and all its subsets from the set *Unexplored* we add to $f_{Unexplored}$ the clause $\bigvee_{i:T_i \notin U} x_i$.

In order to get an element of *Unexplored*, we ask a SAT solver for a model of $f_{Unexplored}$. In particular, to get a maximal unexplored subset, we ask a SAT solver for a *maximal model* of $f_{Unexplored}$. To get a maximal unexplored superset of $U \subseteq T$, we fix the truth assignment to the Boolean variables that correspond to elements in U to *True* and ask for a maximal model of $f_{Unexplored}$.

Example 1. Let us illustrate the symbolic representation on $T = \{T_1, T_2, T_3\}$. If all subsets of T are unexplored then $f_{Unexplored} = True$. If $\{T_1, T_3\}$ is classified as an MIVC and $\{T_1, T_2\}$ as a inadequate set, then $f_{Unexplored}$ is updated to $True \wedge (\neg x_1 \vee \neg x_3) \wedge (x_3)$.

4.6 Implementation

We have implemented the Grow-Shrink algorithm in an industrial model checker called `JKind` [10], which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [15] as input. The translation of Lustre into a symbolic transition system in `JKind` is straightforward and is similar to what is described in [14]. Verification is supported by multiple "proof engines" that execute in parallel, including K-induction, property directed reachability (PDR), and lemma generation. During verification, `JKind` emits SMT problems using the theories of linear integer and real arithmetic, and can use 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 [11] to generate an (approximately) minimal IVC. To implement our method, we have extended `JKind` with a new engine that implements Algorithm 4 on top of `Z3`. We use the `JKind` IVC generation engine to implement the `IVC_UC` procedure in Algorithm 4.

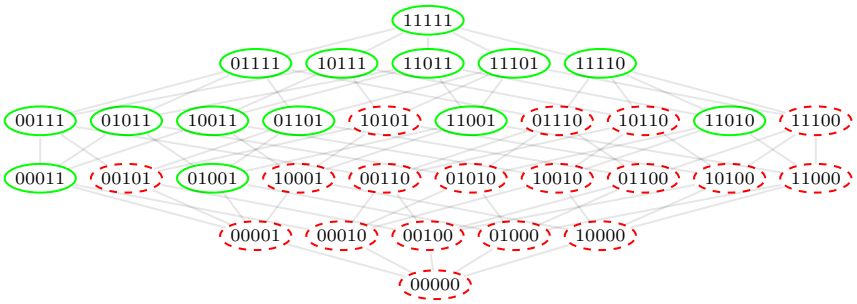


Fig. 1: The power set from the example execution of our algorithm.

5 Example Execution of the Grow-Shrink Algorithm

The following example explains the execution of our algorithm on a simple instance where the transition step predicate T is given as a conjunction of five sub-predicates $\{T_1, T_2, T_3, T_4, T_5\}$. We do not exactly state what are the predicates and what is the safety property of interest. Instead, Figure 1 illustrates the power set of $\{T_1, T_2, T_3, T_4, T_5\}$ together with an information about adequacy of individual subsets. The subsets with solid green border are the adequate subsets, and the subsets with dashed red border are the inadequate ones. To save space, we encode subsets as bitvectors, for example the subset $\{T_1, T_2, T_4\}$ is written as 11010. There are three MIVCs in this example: 00011, 01001, and 11010.

We illustrate the first iteration of the main procedure `FindMIVCs` of our algorithm. Initially, all subsets are unexplored, i.e. $f_{Unexplored} = True$ and the queue *shrinkingQueue* is empty. The procedure starts by finding a maximal unexplored subset and checking it for adequacy. In our case, $U_{max} = 11111$ is the only maximal unexplored subset and it is determined to be adequate. Thus, the algorithm `IVC_UC` is used to compute an approximately minimal IVC $U_{IVC} = 01101$ which is then shrunk to a MIVC 01001.

During the shrinking, sets 00101, 01001, and 01000 are subsequently checked for adequacy and determined to be inadequate, adequate, and inadequate, respectively. The set 01001 is the resultant MIVC, thus the formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5)$. The other two sets, 00101 and 01000, are enqueued to the *growingQueue* and grown at the end of the procedure.

We first grow the set 00101. Initially, the procedure `Grow` picks $M = 10111$ as the maximal unexplored superset of 00101, and checks it for adequacy. It is adequate and thus, an approximately minimal IVC $M_{IVC} = 00011$ is computed, enqueued to *shrinkingQueue*, and formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5)$. Then, M is (based on M_{IVC}) reduced to $M = 10101$ and checked for adequacy. It is found to be inadequate, thus formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4)$, and the procedure terminates.

The growing of the set 01000 results into an approximately maximal inadequate subset 01110. Moreover, an approximately minimal IVC 11110 is found

during the growing and enqueued into *shrinkingQueue*. The formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_5)$.

After the second grow, the procedure `Shrink` terminates and the main procedure `FindMIVCs` continues. The queue *shrinkingQueue* contains two sets: 00011, 11110, thus the procedure now shrinks them. During shrinking the set 00011, the algorithm would attempt to check the sets 00001 and 00010 for adequacy, however since both these are already explored, the set 00011 is identified to be a MIVC without performing any adequacy checks. The procedure `FindMIVCs` would now shrink also the set 11110, thus empty the queue *shrinkingQueue*, and continue with a next iteration.

6 Experiment

We are interested in examining the performance of algorithms to compute minimal IVCs. We examine three algorithms: **Offline MARCO**, the algorithm from [12], **Online MARCO**, a variant of the algorithm from [12] that performs a shrink step prior to returning a solution to ensure minimality, and **Grow-Shrink**, the algorithm described in this paper. We investigate the following research questions: (RQ1:) For the large models where the complete MIVC enumeration is intractable, how many MIVCs are found within the given time limit? (RQ2:) For the tractable models, i.e. models in which all MIVCs are found, how much time is required to complete the enumeration of MIVCs? Finally, we are interested in how many solver calls are necessary for the enumeration. Thus, we add (RQ3:) What is the (average) number of solver calls with result adequate/inadequate required by evaluated online algorithms to produce individual MIVCs?

Experimental Setup : We start from a benchmark suite that is a superset of the benchmarks used in [12]. This suite contains 660 models, and includes all models that yield a valid result (530 in total) from previous Lustre model checking papers [14, 20] and 130 industrial models yielding valid results derived from an infusion pump system [22] and other sources [20, 4]. As this paper is concerned with analysis problems involving multiple MIVCs, we include only models that had more than 4 MIVCs (46 models in total). To consider problems with many IVCs, we took those models and mutated them, constructing 20 mutants for each model. We added the mutants that still yielded valid results and have more than 5 MIVCs (384 in total) back to the benchmark suite. Thus, the final suite contains 430 Lustre models. The original benchmarks and our augmented benchmark are available from [3].

For each test model, we configured `JKind` to use 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 a 3.50GHz Intel(R) i5-4690 processor 16 GB memory machine running Linux with a 30 minute timeout. All experimental data is available online [2].

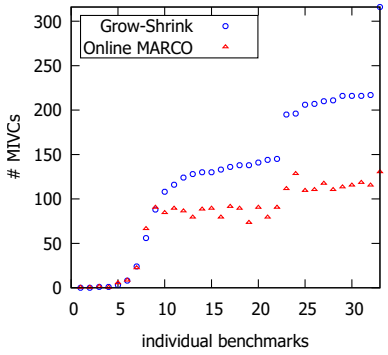


Fig. 2: Number of MIVCs produced by online algorithms.

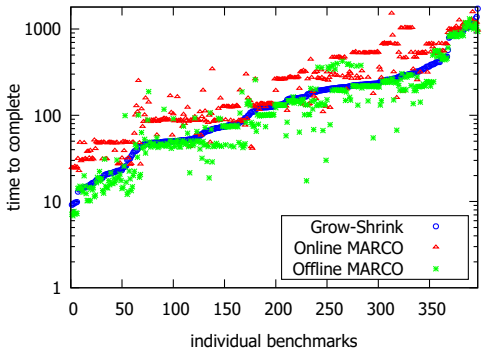


Fig. 3: Runtime for tractable benchmarks for all algorithms in a log scale.

6.1 Experimental Results

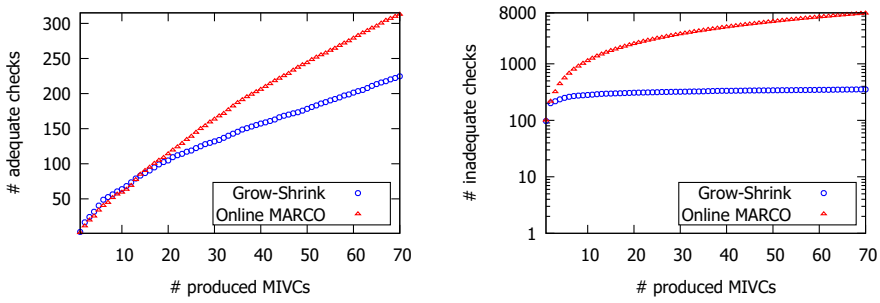
In this section, we examine the experimental results to address the research questions.

RQ1 and RQ2: Data related to the first two research questions is shown in Figures 2 and 3. Figure 2 describes the number of MIVCs found be the two online algorithms in the intractable benchmarks, i.e. the benchmarks where the algorithms did not complete the computation within the time limit. There are 33 such benchmarks. The Grow-Shrink algorithm substantially outperforms Online MARCO in the majority of the benchmarks, finding an average of 55% additional MIVCs.

Figure 3 describes the time for each algorithm needed to complete the computation in the case of 397 tractable benchmarks. We see that the performance of the Grow-Shrink algorithm is very similar to Offline MARCO, but as previously discussed, has the advantage of returning guaranteed MIVCs, rather than approximate MIVCs. It is much faster than the Online MARCO algorithm.

RQ3: For RQ3, we examined the number of required calls to the solver per MIVC. For this question, we used the 33 models that contained a large number of MIVCs (>70) in order to show the solver efficiency as the number of MIVCs increased. A point with coordinates (x, y) states that the algorithm needed to perform y solver calls (on average) in order to produce (find) the first x MIVCs. We grouped the calls in terms of the number of calls that returned *adequate* vs. *inadequate* results. It is evidenced by the results in Figure 4, the new algorithm improves upon Online MARCO as the number of MIVCs becomes larger.

The improvement in the number of *inadequate* calls is due the novel shrinking and growing procedures. Each (approximately) maximal inadequate subset found by the growing procedure allows to save (up to exponentially) many inadequate



(a) Checks with result "adequate". (b) Checks with result "inadequate".

Fig. 4: Average number of performed adequacy checks required to produce individual MIVCs. Note that Figure (b) is in a log scale.

calls during subsequent executions of the shrinking procedure. Indeed, the Grow-Shrink algorithm performed on average only 353 inadequate calls to output the first 70 MIVCs, whereas the online MARCO needed to perform 7775 calls to output the same number of MIVCs.

The improvement in the number of *adequate* calls is not so significant as in the case of inadequate calls. Yet, since the adequate calls are usually much more time consuming than inadequate ones, even a slight saving in the number of adequate calls might significantly speed up the whole computation. The Grow-Shrink algorithm saves adequate calls due to the usage of the shrinking queue and due to the invariants that are maintained by the queue. In particular, shall two comparable sets appear in the queue, only the smaller is left. Thus, the algorithm avoids shrinking of relatively large sets and saves some adequate calls.

7 Conclusion

In this paper, we have presented an *online* algorithm, called *Grow-Shrink algorithm*, for computation of minimal Inductive Validity Cores (MIVCs). The new algorithm substantially outperforms previous approaches. As opposed to the *Offline MARCO* algorithm in [12], it is guaranteed to produce minimal IVCs. As opposed to a näive extension *Online MARCO*, the new algorithm is substantially faster and requires fewer solver calls as the number of MIVCs increases. We believe that this new algorithm will substantially increase the applicability of software engineering tasks that require MIVCs. In the future, we hope to examine parallel computation of MIVCs using a variant of this algorithm to further increase scalability.

References

1. Cadence JasperGold Formal Verification Platform. https://www.cadence.com/.

2. IVCs repository. https://github.com/jar-ben/online-mivc-enumeration.
3. Lustre Benchmarks. https://github.com/elaghs/benchmarks.
4. J. Backes et al. Requirements analysis of a quad-redundant flight control system. In *NFM 2015*, 2015.
5. J. Bendík, N. Benes, J. Barnat, and I. Cerná. Finding boundary elements in ordered sets with application to safety and requirements analysis. In *SEFM 2016*, 2016.
6. J. Bendík, N. Benes, I. Cerná, and J. Barnat. Tunable online MUS/MSS enumeration. In *FSTTCS 2016*, pages 50:1–50:13, 2016.
7. H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for formal verification. *Correct hardware design and verification methods*, pages 111–125, 2003.
8. K. Claessen and N. Srensson. A liveness checking algorithm that counts. In *FM-CAD*, pages 52–59. IEEE, 2012.
9. N. Een et al. Efficient implementation of property directed reachability. In *FM-CAD'11*, 2011.
10. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The jkind model checker. *arXiv preprint arXiv:1712.01222*, 2017.
11. E. Ghassabani et al. Efficient generation of inductive validity cores for safety properties. In *FSE'16*, 2016.
12. E. Ghassabani, A. Gacek, and M. W. Whalen. Efficient generation of all minimal inductive validity cores. In *FMCAD 2017*, 2017.
13. E. Ghassabani, A. Gacek, M. W. Whalen, M. Heimdahl, and W. Lucas. Proof-based coverage metrics for formal verification. In *ASE 2017*, 2017.
14. G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *FMCAD'08*, 2008.
15. N. Halbwachs et al. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 1991.
16. O. Kupferman, W. Li, and S. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *FMCAD 2008*, page 25, 2008.
17. O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 2003.
18. M. Liffiton et al. Fast, flexible MUS enumeration. *Constraints*, 2016.
19. M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible mus enumeration. *Constraints*, 21(2):223–250, 2016.
20. A. Mebsout and C. Tinelli. Proof certificates for smt-based model checkers for infinite-state systems. In *FMCAD'16*, 2016.
21. S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.
22. A. Murugesan et al. Compositional verification of a medical device system. In *HILT'13*, 2013.
23. A. Murugesan et al. Complete traceability for requirements in satisfaction arguments. In *RE'16 (RE@Next! Track)*, 2016.
24. M. Sheeran et al. Checking safety properties using induction and a SAT-solver. In *FMCAD'02*, 2000.
25. M. Whalen et al. Integration of formal analysis into a model-based software development process. In *FMICS*, 2007.
26. M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *ICSE 2013*. ACM, 2013.
27. D. You, S. Rayadurgam, M. Whalen, and M. Heimdahl. Efficient observability-based test generation by dynamic symbolic execution. In *ISSRE 2015*, 2015.
28. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *SAT'03*, 2003.