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 [4], k-induction [20], and k-liveness [3] 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 [17, 15, 16] and to support automated test case generation [22, 23].

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 [13], incorrect environmental assumptions [21], and errors either in English language requirements or formalization [19] 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 [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 [24] 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 [7, 18, 9, 8] we have explored several different uses of IVCs, including:

Traceability: For functional properties that can be proven with inductive model checkers, inductive validity cores can provide accurate traceability matrices with no user effort. Given multiple IVCs, *rich traceability* matrices [18] can be automatically constructed that provide additional insight about *required* vs. *optional* design elements.

Vacuity detection: The idea of syntactic vacuity detection (checking whether all subformulae within a property are necessary for its validity) has been well studied [13]. 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: Closely related to vacuity detection is the idea of *coverage analysis*, e.g., are all atoms in the model necessary for at least one of the properties proven about the model? Several different notions of coverage have been proposed [2, 12], but these tend to be very expensive to compute.

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: 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 synthesizers 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 [8] for finding all minimal IVCs based on the MARCO algorithm for MUSes [14]. 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.

TODO: Jaroslav and Ivana: Make sure I'm speaking correctly here TODO: Elaheh: fill in experimental data TODO: ALL: talk about 'unknown' results when timeout occurs. TODO: ALL: do we need a running example? I'm not sure. If we do an illustration of it, the Brno team might be best at this.

In this paper, we consider three *online* algorithms for MIVC enumeration. With these algorithms, solutions are produced at a regular rate, and each solution produced is guaranteed to be minimal. Additionally, for models with a large number of IVCs, the proposed algorithms are considerably more efficient than the baseline MARCO algorithm. We demonstrate this via experiment, where the new

The rest of the paper is organized as follows...

2 Motivating Example

3 Preliminaries

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

TODO: Example

4 Existing Techniques

Let us first consider 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 exponential number of 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 [] proposed in the area of constraint processing and applied by Ghassabani et al. [].

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).

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.

Algorithm 1: Shrinking procedure

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

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*. Shrinking procedure iteratively attempts to remove elements from the set that is being shrunk, checking each new set for adequacy and keeping only changes that leave the set adequate (see Algorithm 1 for a pseudocode).

Ghassabani et al. [8] proposed an algorithm for MIVC enumeration which 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. [8] proposed a yet another (but very similar) algorithm which, instead of computing MIVCs in on online manner by using the shrink procedure, 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` []. `IVC_UC` is able to identify IVCs which are often very close to actual MIVCs, yet cheap to be found. Since this MIVC enumeration algorithm computes only approximations of MIVCs, the actual minimal IVCs are identified at the very end of the computation when the adequacy of each subset is already determined. Their experimental evaluation show that the latter algorithm computes all MIVCs much faster than the algorithm based on shrinking. However, it does not enumerate MIVCs in an online manner and thus on hard benchmarks it might produce no MIVC at all within a given time limit.

5 Algorithm

In this section, we propose a novel algorithm for online enumeration of all MIVCs. The algorithm is built out of two basic procedures: *shrink* and *grow*. The grow procedure is symmetric to the shrink one, i.e., given an inadequate set for $(I, T) \vdash P$, the grow procedure finds a maximal inadequate set for $(I, T) \vdash P$. In the following, we first closely describe how these two procedures work, and then we propose a way to combine them to form an efficient online MIVC enumeration algorithm.

Algorithm 2: Approximate grow

```
input  : (I, T) ⊢ P
input  : inadequate U ⊂ T for (I, T) ⊢ P
input  : set Unexplored of unexplored subsets of T
output: approximately maximal inadequate set for (I, T) ⊢ P
M ← a maximal M ∈ Unexplored such that M ⊇ U
while (I, M) ⊢ P do
    M_IVC ← IVC_UC((I, M), P)        // gets approximately minimal IVC
    T_i ← choose T_i ∈ (M_IVC \ U)
    M ← M \ {T_i}
return M
```

5.1 Shrink Procedure

We can effectively use the set *Explored* 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 explored. If so, the status of $U \setminus \{T_i\}$ is known and no test for adequacy is needed.

However, there is one more observation that can be exploited.

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, we are quaranteed that whenever during the shrinking procedure we come across an explored set, this set is inadequate. Therefore 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 grow procedure.

5.2 Grow Procedure

Recall that if a set is determined to be inadequate then all of its subsets are necessarily also inadequate. Therefore, the larger set is determined to be inadequate, the more inadequate sets become 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 $(I, T) \vdash P$. 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 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 procedure `IVC_UC` by Ghassabani et al. [?] is used to find an approximate MIVC M_{IVC} of M which 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. Mike: In the previous sentence, do you mean that we continue to remove elements from $M_{IVC} \setminus U$ until we reach an inadequate set, or something else? 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.

TODO: JB: Perhaps give a simple running example.

5.3 Solve Procedure

Algorithm 3: Solving algorithm

Function `Solve`(I,U,P):
- $res \leftarrow (I, U) \vdash P$
- **if** $res = TIMEOUT$ **then**
  - $approximateWarning \leftarrow true$; `// a global variable`
- **return** $res = VALID$

Mike: Information here about timeouts and their affect on solving

5.4 Complete Algorithm

In this section, we describe, how to combine the shrink and grow methods in order to form an efficient online MIVC enumeration algorithm.

Since knowledge of (approximately) maximal inadequate subsets might be used to speed up shrinking procedures, it might be tempting to first find all maximal inadequate subsets. However, there can be up to exponentially many such subsets with respect to the size of T. Thus, finding first all maximal inadequate subsets is in general intractable. Instead, we propose to alternate both shrinking and growing procedures. Note, that during shrinking, we might determine some subsets to be inadequate, and such subsets can be subsequently used as *seeds* for growing. Dually, we might determine some subsets to be adequate during growing, and such subsets can be used as *seeds* for shrinking procedure. Thus, both these procedures complements each other.

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 it and all of its subsets are marked as explored. Otherwise, if U_{max} is adequate, then the procedure `IVC_UC` turns it into an approximately minimal IVC, and subsequently the procedure `Shrink` shrinks it into a MIVC.

Procedure `Shrink` works as described in Section 5.1. However, besides shrinking the given set into a MIVC, the procedure has also side effect. 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 5.2. The resultant set and all of its subsets are marked as explored. Moreover, every adequate set that is 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`. TODO: JB: describe how we deal with explored subsets (seeds) in the queue .

TODO: correctness

Mike: Made slight wordsmithing changes to algorithms Mike: Made incompleteness explicit in algorithm

6 Symbolic Representation of Unexplored Subsets

Symbolic representation 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

Algorithm 4: AllMIVC algorithm

```
Function Init((I, T) ⊢ P):
    Unexplored ← 𝒫(T)                                   // a global variable
    shrinkingQueue ← empty queue                        // a global variable
    approximateWarning ← false                          // a global variable
    FindMIVCs()

Function FindMIVCs():
    while Unexplored ≠ ∅ do
        U_max ← a maximal set ∈ Unexplored
        if Solve(I, U, P) then
            U_IVC ← IVC_UC((I, U_max), P)
            Shrink(U_IVC)
        else
            Unexplored ← Unexplored \ Sub(U_max)
        while shrinkingQueue is not empty do
            U ← Dequeue(shrinkingQueue)
            TODO: deal with explored seeds in the queue
            Shrink(U)

Function Shrink(U):
    growingQueue ← empty queue
    for T_i ∈ U do
        if U \ {T_i} ∈ Unexplored then
            if Solve(I, U \ {T_i}, P) then
                U ← U \ {T_i}
            else
                Enqueue(growingQueue, U \ {T_i})
    output U;                                           // Output Minimal IVC
    Unexplored ← Unexplored \ Sup(U)
    while growingQueue is not empty do
        V ← Dequeue(growingQueue)
        Grow(V)

Function Grow(V):
    M ← a maximal set ∈ Unexplored such that M ⊇ V
    while Solve(I, M, P) do
        M_IVC ← IVC_UC((I, M), P)
        Enqueue(shrinkingQueue, M_IVC)
        Unexplored ← Unexplored \ Sup(M_IVC)
        T_i ← choose T_i ∈ (M_IVC \ V)
        M ← M \ {T_i}
    Unexplored ← Unexplored \ Sub(M)
```

subsets *Unexplored* using a Boolean formula $f_{Unexplored}$ such that each model of $f_{Unexplored}$ corresponds to an element of *Unexplored*.

Our algorithm uses two types of operations to manage *Unexplored*: it removes from *Unexplored* either supersets of some adequate $U \subseteq T$ or subsets of some inadequate $U \subseteq T$. To remove $U \subseteq T$ and all its supersets from *Unexplored* we add to $f_{Unexplored}$ one clause,

$$f_{Unexplored} \leftarrow f_{Unexplored} \ \wedge \bigvee_{i:T_i \in U} \neg x_i \ .$$

Symmetrically, to remove $U \subseteq T$ and all its subsets from *Unexplored* we add to $f_{Unexplored}$ one clause,

$$f_{Unexplored} \leftarrow f_{Unexplored} \ \wedge \bigvee_{i:T_i \notin U} x_i \ .$$

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 $True \wedge (\neg x_1 \vee \neg x_3) \wedge (x_3)$.

In order to get an element of *Unexplored*, we can ask a SAT solver for a model of $f_{Unexplored}$. However, our algorithm requires specific elements of *Unexplored*: maximal unexplored subsets (these correspond to maximal models), and elements that are supersets of some particular set. One of the SAT solvers that can be used to obtain models corresponding to these specific elements is the solver miniSAT[5]. In miniSAT, the user can fix values of chosen variables and select a default polarity of variables at decision points during solving. For example, in order to find a maximal unexplored superset of $U \subseteq T$, we set the default polarity of variables during solving to *True* and fix the truth assignment to the variables that correspond to elements in U to *True*.

7 Example Execution of Our Algorithm

The following example explains the execution of our algorithm on a simple input 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, in Figure 7 we illustrate 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. In order 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, $U_{max} = 11111$

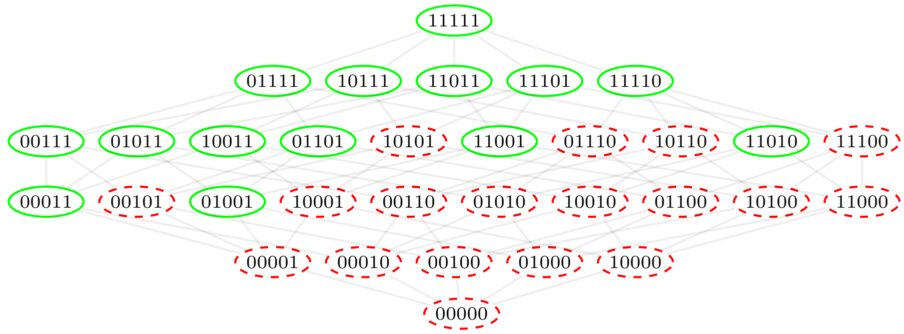

Fig. 1. The power set from the example execution of our algorithm.

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 at the end of the procedure `Shrink`, they are grown.

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 adequate, 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.

8 Implementation

We have implemented the algorithm in an industrial model checker called `JKind` [6], which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [11] as input. The translation of Lustre into a symbolic transition system in `JKind` is straightforward and is similar to what is described in [10]. 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 4 on top of `Z3`. We use the `JKind` IVC generation engine to implement the `IVC_UC` procedure in Algorithm 4.

TODO: ALL: how do timeouts fit into the general framework?

9 Experiment

10 Conclusion

References

1. Cadence JasperGold Formal Verification Platform. https://www.cadence.com/.
2. H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for formal verification. *Correct hardware design and verification methods*, pages 111–125, 2003.
3. K. Claessen and N. Srensson. A liveness checking algorithm that counts. In *FMCAD*, pages 52–59. IEEE, 2012.
4. N. Een et al. Efficient implementation of property directed reachability. In *FMCAD'11*, 2011.
5. N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
6. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The jkind model checker. *arXiv preprint arXiv:1712.01222*, 2017.
7. E. Ghassabani et al. Efficient generation of inductive validity cores for safety properties. In *FSE'16*, 2016.
8. E. Ghassabani, A. Gacek, and M. W. Whalen. Efficient generation of all minimal inductive validity cores. In *FMCAD2017: International Conference on Formal Methods in Computer-Aided Design*, 2017.
9. E. Ghassabani, A. Gacek, M. W. Whalen, M. Heimdahl, and W. Lucas. Proof-based coverage metrics for formal verification. In *ASE2017: 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.
10. G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *FMCAD'08*, 2008.

11. N. Halbwachs et al. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 1991.
12. 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.
13. O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 2003.
14. M. Liffiton et al. Fast, flexible MUS enumeration. *Constraints*, 2016.
15. K. L. McMillan. A methodology for hardware verification using compositional model checking. Technical Report 1999-01, Cadence Berkeley Labs, Berkeley, CA 94704, 1999.
16. S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.
17. A. Murugesan et al. Compositional verification of a medical device system. In *HILT'13*, 2013.
18. A. Murugesan et al. Complete traceability for requirements in satisfaction arguments. In *RE'16 (RE@Next! Track)*, 2016.
19. L. Pike. A note on inconsistent axioms in rushby's "systematic formal verification for fault-tolerant time-triggered algorithms". *TSE*, 2006.
20. M. Sheeran et al. Checking safety properties using induction and a SAT-solver. In *FMCAD'02*, 2000.
21. M. Whalen et al. Integration of formal analysis into a model-based software development process. In *FMICS*, 2007.
22. 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.
23. 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.
24. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *SAT'03*.