Small Inductive Safe Invariants

Alexander Ivrii
IBM Research
alexi@il.ibm.com

Arie Gurfinkel
Software Engineering Institute
http://arieg.bitbucket.org

Anton Belov
Synopsys
http://anton.belov-mcdowell.com

***Abstract*—Computing minimal (or even just small) certificates is a central problem in automated reasoning and, in particular, in automated formal verification. For example, Minimal Unsatisfiable Subsets (MUSes) have a wide range of applications in verification ranging from abstraction and generalization to vacuity detection and more. In this paper, we study the problem of computing minimal certificates for safety properties. In this setting, a certificate is a set of clauses Inv such that each clause contains initial states, and their conjunction is safe (no bad states) and inductive. A certificate is minimal, if no subset of Inv is safe and inductive. We propose a two-tiered approach for computing a Minimal Safe Inductive Subset (MSIS) of Inv. The first tier is two efficient approximation algorithms that under- and over-approximate MSIS, respectively. The second tier is an optimized reduction from MSIS to a sequence of computations of Maximal Inductive Subsets (MIS). We evaluate our approach on the HWMCC benchmarks and certificates produced by our variant of IC3. We show that our approach is several orders of magnitude more effective than the naïve reduction of MSIS to MIS.**

I. Introduction

Computing minimal (or even just small) certificates is a central problem in automated reasoning, and, in particular, in Model Checking. For reachability, the certificates take the form of counterexamples. It is widely believed that small counterexamples are the key to success of Model Checking in practice, as they increase user comprehension and provide better fault localization. In SAT-based Bounded Model Checking (BMC), the certificates for bounded safety (i.e., absence of counterexamples bounded by a given fixed length) correspond to unsatisfiable subsets. Minimal Unsatisfiable Subsets (MUSes) have a wide range of applicability. For example, they are a key ingredient in Proof-Based Abstraction [1], and have also been used to improve user's comprehension of verification results through vacuity [2]. For Unbounded Model Checking (or unreachability) the certificates are represented by *safe inductive invariants*. A recent trend, borrowing from the breakthroughs in Incremental Inductive Verification (such as IMC [3], IC3 [4], and PDR [5]), is to represent such invariants by a set of simple lemmas. In this paper, we study the problem of efficiently minimizing the set of such lemmas, and especially constructing a *minimal safe inductive subset* of a given safe inductive invariant. We focus on the algorithmic aspects of the problem and on empirical evaluation, and leave exploring the numerous potential applications for future work.

Throughout the paper, we assume that all formulas are in CNF and that a safe inductive invariant is represented by a set of clauses Inv such that each clause contains the initial states, and their conjunction is invariant under the transition relation and does not contain any bad states. The set Inv is minimal, called *Minimal Safe Inductive Invariant (MSIS)*, if, in addition to being safe and inductive, no subset of Inv is safe and inductive.

In this paper, we make the following contributions. First, in Section III, we show that computing an MSIS is reducible to a sequence of computations of Maximal Inductive Subset (MIS). While this yields a simple-to-implement algorithm, we show that it is not efficient. Second, we propose a two-tiered algorithm. The first tier, described in Section IV, consists of two approximation algorithms. The first algorithm under-approximates an MSIS by identifying the necessary clauses that are shared between all MSISs. The second, uses a sequence of MUS computations to over-approximate an MSIS. While these algorithms do not guarantee minimality, they can be used as an effective pre-processing step. The second tier, described in Section V, consists of two alternative optimized reductions from MSIS to MIS. The key idea is to combine the basic MSIS to MIS reduction with some of the pre-processing techniques to reduce the number of redundant SAT calls in each MIS computation. Third, we evaluate all of the algorithms on the benchmarks from the Hardware Model Checking Competition. We show that our ultimate algorithm that combines pre-processing and optimizations is several order of magnitude faster than the naïve approach. Furthermore, we show that the technique is extremely effective at reducing the size of the certificate, compared to the certificate produced by our custom variant of IC3.

To our knowledge, the problem of computing MSIS is not widely studied in SAT-based Model Checking (as opposed to computing minimal counterexamples or minimal unsatisfiable subsets). The only alternative solution is proposed by Bradley et al. [6] in the context of `FAIR` algorithm, which is similar to our base algorithm in Section III. However, we show that it does not scale in our context. On the other hand, we believe that efficient algorithms for computing MSIS are just as important as efficient algorithms for computing minimal unsatisfiable subsets, and they are necessary for extending many of the applications (in particular vacuity and abstraction) from BMC to Unbounded Model Checking. We believe that our work lays the foundation for numerous applications of small safety certificates in SAT-based Model Checking.

II. Preliminaries

Let $\mathcal{V}$ be a set of variables. A *literal* is either a variable $b \in \mathcal{V}$ or its negation $\neg b$. A *clause* is a disjunction of literals. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. It is often convenient to treat a clause

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0001309.

Input: $P = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, CNF $\mathcal{L}$
Output: $\mathit{Inv} \subseteq \mathcal{L}$ the MIS of $\mathcal{L}$ relative to P

```
1 $Inv \leftarrow \mathcal{L}$
2 forever do
3     let $R = \{d \in Inv \mid (Inv \wedge Tr) \not\Rightarrow d'\}$
4     if $R \neq \emptyset$ then
5         $Inv \leftarrow Inv \setminus R$
```

Fig. 1. A generic MIS algorithm.

Input: $(\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, safe inductive invariant Inv_o
Output: A minimal safe inductive subset $\mathit{Inv} \subseteq \mathit{Inv}_o$

```
1 $Inv \leftarrow Inv_o$ ; $\mathcal{W} \leftarrow Inv_o$
2 while $\mathcal{W} \neq \emptyset$ do
3     $c \leftarrow$ a clause from $\mathcal{W}$ ; $\mathcal{W} \leftarrow \mathcal{W} \setminus \{c\}$
4     $X \leftarrow \texttt{MIS}((Init, Tr, Bad), Inv \setminus \{c\})$
5     if $(X \Rightarrow \neg Bad)$ then $Inv \leftarrow X$
```

Fig. 2. A naïve `MSIS` algorithm for Minimal Safe Inductive Subset.

as a set of literals, and a CNF as a set of clauses. For example, given a CNF formula F, a clause c and a literal ℓ, we write $\ell \in c$ to mean that ℓ occurs in c, and $c \in F$ to mean that c occurs in F.

A variable assignment is a map $\sigma : \mathcal{V} \rightarrow \{\top, \bot\}$ that assigns $\top$ or $\bot$ to every variable in $\mathcal{V}$. A clause c is satisfied by an assignment σ if $\sigma(\ell) = \top$ for a literal $\ell \in c$. A CNF formula F is satisfied by σ if σ satisfies every clause in F. A CNF formula is SAT if there exists an assignment that satisfies it and is UNSAT otherwise.

A SAT-solver is a complete decision procedure for propositional formulas in CNF. We assume that the reader is familiar with the basic interface of an incremental solver. We use the following API: (a) `Sat_Add`(φ) adds clauses corresponding to the formula φ to the solver; (b) `Sat_Checkpoint`$()$ saves the current state of the solver; (c) `Sat_Rollback`$()$ restores the solver to the previously saved state.

Let F be an UNSAT CNF formula. A *minimal unsatisfiable subset (MUS)* of F is a subset of clauses $U \subseteq F$ such that U is UNSAT, and for every clause $c \in U$, $U \setminus \{c\}$ is SAT. There are many efficient algorithms for computing an MUS [7]–[9]. In the paper, we write `Sat_Mus`(F) for a call to an unspecified MUS algorithm. We assume that the MUS is always computed relative to the clauses already added to the solver using `Sat_Add`.

Let $\mathcal{V}$ be a set of variables and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. A safety verification problem is a tuple $P = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\mathit{Init}(\mathcal{V})$ and $\mathit{Bad}(\mathcal{V})$ are formulas with free variables in $\mathcal{V}$ denoting initial and bad states, respectively, and $\mathit{Tr}(\mathcal{V}, \mathcal{V}')$ is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ denoting the transition relation. Without loss of generality, we assume that Init and Tr are in CNF, and that Bad is a single literal.

The verification problem P is SAT (or UNSAFE) iff there exists a natural number N such that the following formula is SAT:

$$\mathit{Init}(\vec{v}_0) \wedge \left(\bigwedge_{i=0}^{N-1} \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1}) \right) \wedge \mathit{Bad}(\vec{v}_n) \tag{1}$$

P is UNSAT (or SAFE) iff there exists a formula $\mathit{Inv}(\mathcal{V})$, called *a safe invariant*, that satisfies the following conditions:

$$\mathit{Init}(\vec{v}) \Rightarrow \mathit{Inv}(\vec{v}) \qquad \mathit{Inv}(\vec{v}) \wedge \mathit{Tr}(\vec{v}, \vec{v}') \Rightarrow \mathit{Inv}(\vec{v}') \tag{2}$$
$$\mathit{Inv}(\vec{v}) \Rightarrow \neg \mathit{Bad}(\vec{v}) \tag{3}$$

A formula Inv that satisfies (2) is called an *invariant*, while a formula Inv that satisfies (3) is called *safe*. Without loss of generality, we assume that $\neg \mathit{Bad} \in \mathit{Inv}$.

Throughout the paper, we fix a problem $P = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$. Let $\mathcal{L}$ be a formula in CNF. A *maximal inductive subset (MIS)* of $\mathcal{L}$ relative to P is the largest subset $\mathit{Inv} \subseteq \mathcal{L}$ that satisfies (2). There are several algorithms for computing MIS [10]–[12]. A generic MIS algorithm is shown in Fig. 1, in which we first set Inv to $\mathcal{L}$, and then we repeatedly remove those clauses $R \subseteq \mathit{Inv}$ that fail to be inductive relative to Inv. We write `MIS`$(\mathcal{L})$ for a call to an MIS algorithm.

III. Minimal Safe Inductive Subset

Fix a safety verification problem $P = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, and let Inv be a safe inductive invariant of P in CNF. A subset of clauses $S \subseteq \mathit{Inv}$ is called a *safe inductive* subset of Inv relative to P if S is inductive and safe. S is *minimal* if any subset of S is either not safe or not inductive. In this section, we give a basic algorithm to compute a minimal safe inductive subset (MSIS) of a safe inductive invariant in CNF.

The algorithm is shown in Fig. 2. It works by a repeated application of the MIS algorithm. The input is a safety problem and a safe inductive invariant Inv_o. The algorithm keeps a work-set $\mathcal{W}$ of yet unprocessed elements of Inv_o. In each iteration of the loop, a clause $c \in \mathcal{W}$ from the work-set (line 3) is removed, and an MIS algorithm is used to compute the maximal inductive subset X of $\mathit{Inv} \setminus \{c\}$ (line 4). If X is also safe, then X represents a smaller safe inductive invariant of Inv (not containing c and possibly some additional clauses), and so Inv is replaced by X (line 5). Otherwise, c must belong to an MSIS of Inv_o. The algorithm terminates when there are no more unprocessed clauses, at which point we claim that Inv is an MSIS of Inv_o. The fact that Inv remains safe follows from the fact that the initial invariant Inv_o was safe and that each update of Inv maintains that. For the sake of contradiction, suppose that Inv is not minimal, i.e. that there is a minimal safe inductive invariant $\mathit{Inv}_m \subsetneq \mathit{Inv}$. Take any clause $c \in \mathit{Inv} \setminus \mathit{Inv}_m$. Consider the iteration of the loop corresponding to the removal of c from $\mathcal{W}$ and let Inv_1 represent Inv on that iteration. Since c was not removed from Inv_1, the maximal inductive subset of $\mathit{Inv}_1 \setminus \{c\}$ is not safe. On the other hand, Inv_m is a safe inductive subset of $\mathit{Inv} \setminus \{c\}$ and hence of $\mathit{Inv}_1 \setminus \{c\}$, leading to a desired contradiction.

While this naïve algorithm is simple, it is not efficient. In our experience, the calls to MIS are the bottleneck. Furthermore, the algorithm makes a lot of redundant calls because it does not take into account the dependency between clauses. Often, the inductive clauses occur in a group such that removing any one of the clauses makes the MIS of the result unsafe. In the rest of the paper, we propose two significant improvements. First, in Section IV, we give efficient algorithms to under- and over-approximate MSIS. While these algorithms

```
Input: $(Init, Tr, Bad), Inv, \mathcal{N}_o \subseteq Inv$ s.t. $\neg Bad \in \mathcal{N}_o$
Output: safe necessary set $\mathcal{N}$ s.t. $\mathcal{N}_o \subseteq \mathcal{N} \subseteq Inv$
1  $\varphi \leftarrow (\bigwedge_{c \in \mathcal{N}_o} c) \wedge (\bigwedge_{c \in Inv \setminus \mathcal{N}_o} a_c \Leftrightarrow c) \wedge$
2      $(\sum_{c \in Inv \setminus \mathcal{N}_o} \bar{a}_c \leq 1) \wedge Tr$
3  Sat_Add($\varphi$)
4  $\mathcal{N} \leftarrow \mathcal{N}_o$ ; $\mathcal{W} \leftarrow \mathcal{N}_o$
5  while $\mathcal{W} \neq \emptyset$ do
6      $d \leftarrow$ a clause from $\mathcal{W}$ ; $\mathcal{W} \leftarrow \mathcal{W} \setminus \{d\}$
7      while $\varphi \wedge \neg d'$ is SAT (with model $M$) do
8          let $c \in Inv \setminus \mathcal{N}$ be a clause s.t. $M \models (a_c = 0)$
9          Sat_Add($a_c$)
10         $\mathcal{N} \leftarrow \mathcal{N} \cup \{c\}$ ; $\mathcal{W} \leftarrow \mathcal{W} \cup \{c\}$
```

Fig. 3. NEC algorithm.

do not necessarily compute a minimal set, they are used as effective pre-processing steps. Second, in Section V, we give a more efficient variant of MSIS that attempts to minimize the amount of wasted work in each iteration and show how to combine it with the over- and under-approximating algorithms. Our results in Section VI show that this achieves orders of magnitude improvements in performance.

IV. Approximating SIS

In this section, we present two algorithms to approximate a MSIS of a given inductive invariant *Inv*. The first algorithm, called NEC, under-approximates an MSIS by identifying a set of clauses that must be included in any safe inductive subset. The second algorithm, called FEAS, over-approximates an MSIS by removing clauses that do not belong to some SIS. Throughout the section, we fix a verification problem $P = (Init, Tr, Bad)$ and let *Inv* be a safe inductive invariant of P.

A. Necessary Under-Approximation

A clause $c \in Inv$ is called *safe necessary* (or *necessary* for short) if c is included in *every* MSIS of *Inv*. While computing all necessary clauses is expensive, they can be approximated by the set *NEC* defined as the smallest subset of *Inv* that satisfies the following recursive definition:

$$\neg Bad \in NEC$$
$$\forall c \in Inv, d \in NEC \cdot (Inv \setminus \{c\} \wedge d \wedge Tr \not\Rightarrow d') \Rightarrow c \in NEC$$

That is, *NEC* contains the $\neg Bad$ clause, and all clauses that are necessary to ensure that other clauses in *NEC* remain inductive. It is easy to show by induction that if a clause $c \in NEC$ then c is safe necessary. However, *NEC* does not contain all necessary clauses. For example, consider the problem $P_1 = (Init_1, Tr_1, Bad_1)$ and Inv_1, where

$$Init_1 = Inv_1 \equiv x \wedge y \wedge z \tag{4}$$
$$Tr_1 \equiv x' = y \wedge y' = x \wedge z' = x \vee y \tag{5}$$
$$Bad_1 \equiv \neg z \tag{6}$$

Inv_1 is a MSIS of itself. Thus, all of its clauses are necessary. However, $NEC_1 = \{z\}$ because

$$x \wedge z \wedge Tr_1 \Rightarrow z' \qquad y \wedge z \wedge Tr_1 \Rightarrow z'$$

```
Input: $(Init, Tr, Bad), Inv_o, \mathcal{N} \subseteq Inv_o$ s.t. $\neg Bad \in \mathcal{N}$
Output: A safe inductive set $Inv$ s.t. $\mathcal{N} \subseteq Inv \subseteq Inv_o$
1  $Inv \leftarrow \mathcal{N}$, $\mathcal{W} \leftarrow \mathcal{N}$
2  while $\mathcal{W} \neq \emptyset$ do
3      Sat_Checkpoint()
4      $\varphi \leftarrow Inv \wedge Tr \wedge (\bigvee_{c \in \mathcal{W}} \neg c')$
5      Sat_Add($\varphi$)
6      $\mathcal{W} \leftarrow$ Sat_Mus($Inv_o \setminus Inv$)
7      $Inv \leftarrow Inv \cup \mathcal{W}$
8      Sat_Rollback()
```

Fig. 4. FEAS algorithm.

The set *NEC* can be computed efficiently using an incremental SAT solver, as shown in the algorithm in Fig. 3. The algorithm takes as input a verification problem, an inductive invariant *Inv*, and a starting subset $\mathcal{N}_o$ of *Inv* of safe necessary clauses. We require that $\neg Bad$ is in $\mathcal{N}_o$, but it is possible for $\mathcal{N}_o$ to include additional clauses as well (the value of this will become clear in Section V-C). The output of the algorithm is a possibly enlarged safe necessary subset $\mathcal{N}$ of *Inv*.

The algorithm starts by creating a Boolean formula φ (line 1) consisting of the following components:

- The clauses $c \in \mathcal{N}_o$.
- For each clause $c \in Inv \setminus \mathcal{N}_o$, we introduce a new variable a_c and clauses for $c \Leftrightarrow a_c$ (so that c is satisfied if and only if a_c evaluates to 1).
- Clauses for the at-most-one constraint over the negations of the variables a_c. In practice, we implement such constraints using a sequential counter construction [13].
- The transition relation clauses Tr.

It maintains a work-set of yet unprocessed elements of $\mathcal{N}$ in $\mathcal{W}$. In each iteration of the outermost loop (line 5), a clause $d \in \mathcal{N}$ is selected and tested using the SAT query shown on line 7. Note that $\neg d'$ is passed via assumptions interface. Suppose that this query is satisfiable. We claim that in the satisfying assignment exactly one of the variables a_c is assigned to 0. Indeed, since $Inv \wedge Tr \Rightarrow d'$, not all a_c can be 0. On the other hand, assigning more than one a_c to 0 is prohibited by the at-most-one constraint. Letting $c \in Inv \setminus \mathcal{N}$ be the corresponding clause, we obtain that $Inv \setminus \{c\} \wedge d \wedge Tr \not\Rightarrow d'$, and c can be added to *NEC*. This is accomplished on lines 8–10 by marking c as necessary and permanently setting a_c to 1. The algorithm terminates when all of the necessary clauses have been processed. The algorithm makes at most $2|\mathcal{N}|$ SAT queries: one satisfiable query for each new clause in $\mathcal{N}$ and one unsatisfiable query for each clause in $\mathcal{N}$.

B. Feasible Over-Approximation

Given two subsets $C, D \subseteq Inv$, D is *inductively supported* (*supported* for short) by C iff C is a set such that $C \wedge D \wedge Tr \Rightarrow D'$. That is, D is inductive relative to C. If D is supported by C, then C inductively supports D. Given a safe inductive invariant Inv_o, it is possible construct a SIS *Inv* of Inv_o by first adding $\neg Bad$ to *Inv*, and then, repeatedly, adding to *Inv* supporting clauses of *Inv* until fix-point. Note that the fix-point always exists. In the worst case, $Inv = Inv_o$.

An optimized implementation of this idea is shown in Fig. 4. In addition to Inv, we maintain a work-set $\mathcal{W} \subseteq \mathit{Inv}$ of clauses which are not yet supported. On line 1, we initialize both Inv and $\mathcal{W}$ to $\mathcal{N}$ (which includes $\neg \mathit{Bad}$ and possibly some additional clauses as well). Let us consider one iteration of the loop (lines 3-8). Our goal is to support $\mathcal{W}$ by including in Inv as few additional clauses as possible and we achieve it by a reduction to `Sat_Mus`. Let $\varphi = \mathit{Inv} \wedge Tr \wedge \neg(\bigvee_{c \in \mathcal{W}} \neg c')$. Since $\mathcal{W}$ can be supported by including *all* of the clauses in $\mathit{Inv}_0 \setminus \mathit{Inv}$, the formula $(\mathit{Inv}_0 \setminus \mathit{Inv}) \wedge \varphi$ is unsatisfiable. Thus, the set of clauses required to support $\mathcal{W}$ can be computed as `Sat_Mus`$(\mathit{Inv}_0 \setminus \mathit{Inv})$ (with respect to φ). After this set is found, we include it in Inv (line 7) and by induction this set represents exactly the set of clauses of Inv not known to be supported. If empty, then Inv is already a SIS, and the algorithm terminates. The algorithm makes at most $|F|$ queries to `Sat_Mus` (and much fewer in practice).

We remark that even though we always choose a minimal set of clauses to be added to Inv, the overall algorithm does not necessary produce a MSIS. We illustrate this using the following example. Consider the problem $P_2 = (\mathit{Init}_2, \mathit{Tr}_2, \mathit{Bad}_2)$ and Inv_2, where

$$\mathit{Init}_2 = \mathit{Inv}_2 \equiv x \wedge y \wedge z \quad (7)$$

$$\mathit{Tr}_2 \equiv x' = y \wedge y' = y \wedge z' = x \vee y \quad (8)$$

$$\mathit{Bad} \equiv \neg z \quad (9)$$

It is easy to see that $\{z\}$ is not inductive, but can be supported by either x or y. Suppose that x is chosen and is included to F. Since $\{x\}$ itself is not supported, the next iteration will include y as well, ending up with $F = \mathit{Inv}_2$. However, the MSIS of Inv_2 is $\{y, z\}$.

We conclude this section with several observations on the interaction between `NEC` and `FEAS` algorithms. First, we have found that running `FEAS` after `NEC` produces tighter over-approximations and takes less time on average then running `FEAS` alone. This can be explained, as illustrated by the example above, by the fact that `FEAS` heavily depends on the order in which clauses are added to Inv. On the other hand, `NEC` marks the necessary clauses that must be eventually included in any SIS. Thus, `FEAS` makes better choices when started with those clauses upfront and is faster on average. Second, for a similar reason, we have found that the effort spent on finding a *minimal* set $\mathcal{W}$ to be incrementally added to Inv also pays off – both in terms of the quality of the final over-approximation and the time spent by the algorithm. Finally, Bradley et al. [6] suggest to over-approximate a SIS by computing a “global” unsatisfable core of $\mathit{Inv}_o \wedge Tr \wedge \neg \mathit{Inv}'_o$ by minimizing the set of clauses of Inv_o required for unsatisfiability. We have not found this approach useful, even with the MUS version of the computation. In fact, on our benchmarks it seems that there are large sets of clauses which can be removed from Inv_o but which are required to support themselves. In such a case, the global approach keeps these clauses in the over-approximation, while the iterative approach has a good chance for removing them.

V. Minimal Inductive Safe Invariant

In this section, we present two algorithms for finding a minimal inductive safe subset of a given safe inductive invariant Inv. Our first algorithm (Section V-A) is a simple yet powerful optimization of the basic algorithm from Fig. 2 which identifies the necessary clauses as soon as possible. Our second algorithm (Section V-B) additionally exploits the support dependency between different clauses in a MSIS and avoids performing redundant computations as much as possible.

```
Input: $(\mathit{Init}, \mathit{Tr}, \mathit{Bad}), \mathit{Inv}_o, \mathcal{N}_o \subseteq \mathit{Inv}_o$ s.t.
       $\neg \mathit{Bad} \in \mathcal{N}_o$
Output: An MSIS $\mathit{Inv} \subseteq \mathit{Inv}_o$
1  $\mathit{Inv} \leftarrow \mathcal{N}_o$ ; $\mathcal{W} \leftarrow \mathit{Inv}_o \setminus \mathcal{N}_o$
2  while $\mathcal{W} \neq \emptyset$ do
3      $c \leftarrow$ a clause from $\mathcal{W}$
4      $\mathcal{W} \leftarrow \mathcal{W} \setminus \{c\}$ ; $\mathcal{U} \leftarrow \mathcal{W}$
5      forever do
6          let $R = \{d \in \mathit{Inv} \cup \mathcal{U} \mid (\mathit{Inv} \wedge \mathcal{U} \wedge \mathit{Tr}) \not\Rightarrow d'\}$
7          if $R = \emptyset$ then
8              $\mathcal{W} \leftarrow \mathcal{U}$ ; break
9          else if $R \cap \mathit{Inv} \neq \emptyset$ then
10             $\mathit{Inv} \leftarrow \mathit{Inv} \cup \{c\}$ ; break
11         else $\mathcal{U} \leftarrow \mathcal{U} \setminus R$
```

Fig. 5. An optimized SIS algorithm (`OptMSIS`).

A. *Optimized `MSIS` algorithm*

The `OptMSIS` algorithm is shown in Fig. 5. As before, the input is a verification problem, an initial safe inductive invariant Inv_o, and a subset $\mathcal{N}_o$ of safe necessary clauses of Inv_o. The output is a minimal safe inductive invariant Inv. We maintain two sets of clauses Inv and $\mathcal{W}$ such that the while-loop satisfies the following invariants: (1) $\mathit{Inv} \cup \mathcal{W}$ is a safe inductive invariant, and (2) Inv is safe necessary for $\mathit{Inv} \cup \mathcal{W}$. Initially, $\mathit{Inv} = \mathcal{N}_o$ and $\mathcal{W} = \mathit{Inv}_o \setminus \mathit{Inv}$. Intuitively, the algorithm proceeds by selecting a clause $c \in \mathcal{W}$ and either deducing that c is safe necessary (adding it to Inv) or finding a safe inductive subset of $\mathit{Inv} \wedge \mathcal{W}$ that does not contain c (shrinking $\mathcal{W}$ accordingly). The algorithm terminates when $\mathcal{W} = \emptyset$ at which point Inv is indeed a minimal safe inductive invariant.

In more details, on each iteration of the while-loop we select a clause $c \in \mathcal{W}$, remove it from $\mathcal{W}$, and denote the resulting set by $\mathcal{U}$ (lines 3-4). Next, on each iteration of the inner loop, we compute the set of clauses $R \subseteq \mathit{Inv} \cup \mathcal{U}$ that are no longer supported. On one hand, if $R = \emptyset$, then $\mathit{Inv} \cup \mathcal{U}$ remains a SIS, which means that we have succeeded in removing c (and possibly some other clauses) from $\mathcal{W}$, in which case we update $\mathcal{W}$ and proceed with the next unprocessed clause (lines 7-8). On the other hand, if $R \cap \mathit{Inv} \neq \emptyset$, then one of the necessary clauses in Inv becomes unsupported, in which case we conclude that c must be included in any MSIS of $\mathit{Inv} \cup \mathcal{W}$, mark c as necessary, and proceed with the next unprocessed clause (lines 9-10). Finally, if all of the necessary clauses in Inv remain supported but $R \neq \emptyset$, then the clauses in R cannot be part of any SIS of $\mathit{Inv} \cup \mathcal{U}$, and so we remove these clauses from $\mathcal{U}$ and make another iteration of the inner loop (line 11).

In our implementation, we compute the clauses in R incrementally, making a separate SAT query for each clause $d \in \mathit{Inv} \cup \mathcal{U}$. This computation is aborted as soon as a clause

```
Input: $(Init, Tr, Bad), Inv_o, \mathcal{N}_o \subseteq Inv_o$ s.t.
    $\neg Bad \in \mathcal{N}_o$
Output: An MSIS $Inv \subseteq Inv_o$
1  $Inv \leftarrow \mathcal{N}_o$ ; $\mathcal{W} \leftarrow Inv_o \setminus \mathcal{N}_o$
2  while $\mathcal{W} \neq \emptyset$ do
3      $c \leftarrow$ a clause from $\mathcal{W}$
4      $S \leftarrow \langle \{c\} \rangle$
5      forever do
6          Suppose that $S = \langle C_1, \ldots, C_n \rangle$
7          let $R = \{d \in Inv \cup (\mathcal{W} \setminus C_n) \mid$
8                $(Inv \wedge (\mathcal{W} \setminus C_n) \wedge Tr) \not\Rightarrow d'\}$
9          if $R = \emptyset$ then
10             $\mathcal{W} \leftarrow \mathcal{W} \setminus C_n$
11             $S \leftarrow \langle C_1, \ldots, C_{n-1} \rangle$
12             if S is empty then break
13         else if $R \cap Inv \neq \emptyset$ then
14             $Inv \leftarrow Inv \cup C_1 \cup \cdots \cup C_n$
15             $\mathcal{W} \leftarrow \mathcal{W} \setminus (C_1 \cup \cdots \cup C_n)$
16             break
17         else if $R \cap C_i \neq \emptyset$ for some $i \leq n$ then
18             $S \leftarrow \langle C_1, \ldots, C_{i-1}, C_i \cup \cdots \cup C_n \rangle$
19         else
20             $d \leftarrow$ a clause from $R$
21             $S \leftarrow \langle C_1, \ldots, C_n, \{d\} \rangle$
```

Fig. 6. Binary Implication Graph MSIS algorithm (BigMSIS).

from *Inv* is added to R. Furthermore, the clauses in *Inv* are checked before the remaining clauses in $\mathcal{U}$. In other words, the OptMSIS corresponds to the naïve MSIS algorithm in which (1) the safe necessary clauses are marked as soon as they are discovered, (2) computing an MIS is aborted as soon as one of the necessary clauses becomes unsupported, and (3) necessary clauses are checked first. In practice, this significantly reduces the number of SAT queries done by the algorithm.

B. B.I.G. MSIS algorithm

Our ultimate algorithm for finding MSIS exploits the dependency of including some clauses to a SIS based on the inclusion of other clauses. We say that a clause c is *necessary* for a clause d if c is included in *every* MSIS of *Inv* that contains d. In particular, if $Inv \setminus \{c\} \wedge Tr \not\Rightarrow d'$, then c is necessary for d. From the definition, the necessary relation is transitive: if c is necessary for d and d is necessary for e, then c is necessary for e as well.

Consider a directed graph G on the clauses of *Inv* so that there is an edge from a clause $c \in Inv$ to $d \in Inv$ if and only if c is necessary for d. Them, for every strongly connected component C of G and every MSIS *Inv* of Inv_o either all of the clauses of C are included in *Inv*, or none of the clauses of C are included in *Inv*.

The BigMSIS algorithm shown in Fig. 6 makes use of these observations by incrementally learning and exploiting the underlying graph structure. It has the same input and output as the OptMSIS algorithm, and similarly keeps two sets *Inv* and $\mathcal{W} \subseteq Inv_o$ such that $Inv \cup \mathcal{W}$ is safe and inductive and *Inv* is safe necessary for $Inv \cup \mathcal{W}$. In addition, we use a vector of sets $\langle C_1, \ldots, C_n \rangle$, with the following properties: **(1)** The sets C_i are pairwise disjoint and contained in $\mathcal{W}$; **(2)** For any $i \leq j$, any clause $c \in C_i$, and any clause $d \in C_j$ the clause c is necessary for d. In particular, for every i all of the clauses in C_i belong to the same connected component of the graph.

```
Input: $(Init, Tr, Bad)$, safe inductive invariant $Inv_0$
Output: minimal safe inductive invariant $Inv$ s.t.
        $\neg Bad \in Inv \subseteq Inv_0$
1  $Inv \leftarrow Inv_0$; $\mathcal{N} \leftarrow \{\neg Bad\}$
2  $\mathcal{N} \leftarrow$ NEC($(Init, Tr, Bad), Inv, \mathcal{N}$)
3  $Inv \leftarrow$ FEAS($(Init, Tr, Bad), Inv, \mathcal{N}$)
4  $\mathcal{N} \leftarrow$ NEC($(Init, Tr, Bad), Inv, \mathcal{N}$)
5  $Inv \leftarrow$ MSIS($(Init, Tr, Bad), Inv, \mathcal{N}$)
```

Fig. 7. Combined MSIS algorithm.

In the outermost while-loop of the algorithm we pick the next unprocessed clause $c \in \mathcal{W}$ and initialize S to consist of a single component $\{c\}$. We always focus on the last component C_n of S. On each iteration of the inner loop we compute the set R of clauses that become unsupported if C_n is removed from $Inv \cup \mathcal{W}$. Let us analyze the possible outcomes of this query in detail.

- (lines 9-12) The set $Inv \cup (\mathcal{W} \setminus C_n)$ is inductive. Since $\neg Bad \in Inv$, it is also safe. In this case, we tighten *Inv* by removing all of the clauses $c \in C_n$ (and focus on C_{n-1}, or proceed with the next unprocessed clause if $n = 1$).
- (lines 13-16) A safe necessary clause in *Inv* is no longer supported. It follows that *every* clause in $C_1 \cup \cdots \cup C_n$ is safe necessary as well. In this case we update the set *Inv* by including all of the clauses in S (and proceed with the next unprocessed clause).
- (lines 17-18) A clause $d \in C_i$ is no longer supported. In this case all of the clauses in $C_i \cup \cdots \cup C_n$ belong to the same connected component, and we replace the sets $C_i, \ldots, C_n$ in S by a single set $C_i \cup \cdots \cup C_n$ (and focus on this new set).
- (lines 19-21) A clause $d \in \mathcal{W}$ is no longer supported. Moreover, d is not one of the clauses in S. In this case, we add a new component $C_{n+1} = \{d\}$ to S (and focus on C_{n+1}).

As before, in our implementation R is computed incrementally. Moreover, we have found it beneficial to abort the computation as soon as the first unsupported clause $d \in Inv \cup (\mathcal{W} \setminus C_n)$ is found, and executing the corresponding branch (13-16, 17-18 or 19-21) right away. In this respect, BigMSIS is highly customizable: we can prioritize checking first the known necessary clauses (*Inv*), or the clauses already visited (S), or the clauses not yet explored.

Even though the high-level descriptions of OptMSIS and BigMSIS are rather similar, there is an important theoretical difference between the two algorithms: in the worst case, OptMSIS executes its inner-loop *a quadratic number of times*, while BigMSIS executes its inner-loop only *a linear number of times*. We illustrate this using the following example. Consider the problem $P_3 = (Init_3, Tr_3, Bad_3)$ and an invariant

Inv_3, where

$$\mathit{Init}_3 = \mathit{Inv}_3 \equiv x_1 \wedge \cdots \wedge x_n \quad (10)$$

$$\mathit{Tr}_3 \equiv x_1' = x_n \wedge x_2' = x_1 \wedge \cdots \wedge x_n' = x_{n-1} \quad (11)$$

$$\mathit{Bad}_3 \equiv \neg x_n \quad (12)$$

Further, suppose that initially $\mathcal{N}_o = \{x_n\} \equiv \{\neg \mathit{Bad}\}$. Suppose that `OptMSIS` picks the clause x_1 for removal. Then, after one iteration of the inner loop, the clause x_2 will be removed, after another iteration – the clause x_3, and so on, in total requiring n iterations to detect that $\neg \mathit{Bad}$ is removed. However, this only allows to deduce that x_1 is safe necessary (and can be included in *Inv*) giving no information about the other clauses. So `OptMSIS` would then proceed to remove x_2, leading to yet another $n-1$ iterations of the inner loop, and so on, leading to a quadratic number of iterations. The `BigMSIS` algorithm, on the other hand, requires one iteration of the inner loop to detect that x_1 cannot be removed unless x_2 is removed, another iteration to detect that x_2 cannot be removed unless x_3 is removed, and so on, overall requiring only n iterations to detect that none of $x_1, \ldots, x_n$ can be removed.

C. The combined algorithm

In practice even our best MSIS algorithm is slow due to a large number of required SAT queries. Fortunately, we achieve a significant improvement in runtime by suitably combining the computation of an MSIS with the the under- and over-approximating approaches. The combined algorithm is shown in Fig. 7. The algorithm takes as input a verification problem and an initial safe inductive invariant Inv_0. In the rest of this section, we analyze the suggested approach in detail.

- (Line 1) We mark the $\neg \mathit{Bad}$ clause as necessary and we set *Inv* to Inv_0.
- (Line 2) We run `NEC` to detect additional clauses that must be included in any MSIS of *Inv*. We emphasize the number of SAT calls performed by `NEC` is proportional to the number of necessary clauses detected.
- (Line 3) We run `FEAS` to prune the set of clauses in *Inv*. As discussed in Section IV-B, `FEAS` uses the necessary clauses found `NEC` for better overall choices of the algorithm.
- (Line 4) After some of the clauses were removed, we have new opportunities to mark additional clauses as necessary, and indeed we have found it beneficially to do so. The second run of `NEC` reuses the necessary clauses found in the first run.
- (Line 5) Finally we call an MSIS algorithm.

VI. Experiments

In this section, we present our experimental results. All experiments were performed on a 2.0 GHz Linux-based machine with Intel Xeon E7540 processor and 4 GB of RAM. We consider the unsatisfiable single property benchmarks from the 2011 and 2013 Hardware Model Checking Competitions [14], [15]. To obtain initial invariants, we preprocessed each of the benchmarks using a combinatorial logic optimization, and, if needed, ran (our implementation of) IC3 with 3 hours time limit. Altogether, IC3 successfully completed verification (and produced safe inductive invariants) of 305 benchmark instances. We use these to evaluate the techniques presented in this paper.

We denote by NAIVE the naïve `MSIS` algorithm from Fig. 2 (Section III) with the additional optimization described by Bradley et al. [6]: the computation of the maximal inductive subset of $\mathit{Inv} \setminus \{c\}$ (see Fig. 1) aborts as soon as the current subset becomes unsafe. We denote by OPT the `OptMSIS` algorithm from Fig. 5 (Section V-A), and by BIG the `BigMSIS` algorithm from Fig. 6 (Section V-B). We denote by NAIVE+NFN, OPT+NFN, and BIG+NFN (respectively) the combination of each of these algorithms with preprocessing (computing under-approximations using `NEC`, and over-approximations using `FEAS`), as described in Fig. 7 (Section V-C). We have run each of these 6 algorithms on each of the 305 testcases with a time limit of 1 hour.

The cactus plot in Fig. 8 presents a comparison between the algorithms. Note that preprocessing has a huge impact on any of the three MSIS algorithms, both in terms of instances solved and the total time (for example, NAIVE is able to solve 258 problems without preprocessing, and 293 problems with preprocessing). The best algorithm is BIG+NFN (solving 294 problems). The effectiveness of preprocessing is further corroborated by the fact that on average the initial `NEC` pass identifies about 70% of the final MSIS clauses as necessary; the following `FEAS` pass on average over-approximates the MSIS by only 4%; finally, after the second `NEC` pass, over 90% of the final MSIS clauses are marked as necessary. Thus, the final MSIS pass has to deal with only about 10% of the MSIS clauses on average.

We emphasize that when searching for small (and not necessary minimal) inductive invariants, the preprocessing stage alone (or, more precisely, `NEC` + `FEAS`) produces an almost optimal invariant in most cases, and as such the final MSIS stage can be skipped. Furthermore, any of the MSIS algorithms can be adapted to run with a resource limit, providing a safe and inductive over-approximation in case this limit is reached (such as the set $\mathcal{W}$ in the naïve `MSIS` algorithm, and the set $\mathit{Inv} \cup \mathcal{W}$ in the `OptMSIS` and `BigMSIS` algorithms).

The scatter plot on the left of Fig. 9 demonstrates that BIG+NFN is 2 to 3 orders of magnitude more effective than NAIVE, and hence represents the overall improvement of our best algorithm over prior work. The scatter plot in the center compares BIG and BIG+NFN, and serves to highlight that preprocessing is a crucial technique for most of the problems. Finally, the scatter plot on the right shows that even when preprocessing is used, BIG is an order of magnitude better than NAIVE. It should be noted the most of the points on the diagonal correspond to the problems solved by preprocessing alone.

It is also interesting to compare the number of clauses in the MSIS with the number of clauses in the original invariant computed by IC3. In this aspect there is very little variance between different algorithms, so we provide the data only for BIG+NFN, see Fig. 10. On average, the reduction is the more pronounced the larger is the initial invariant. In fact, this says something about IC3: even when IC3 learns a lot of invariants (and takes a long time to solve a problem), it does not mean that these invariants are useful for the final proof. Finally, we note that on our benchmarks MSIS on average removes 20%

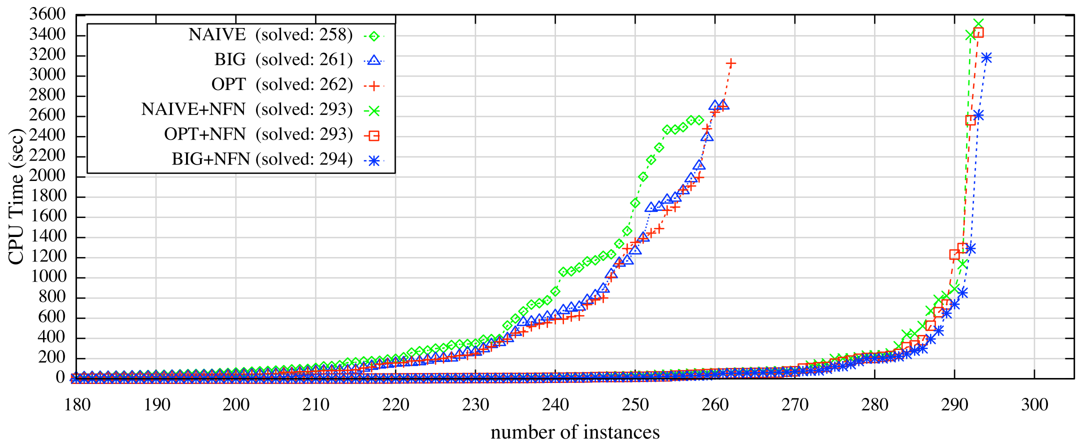

Fig. 8. Comparison between various algorithms, with and without preprocessing.

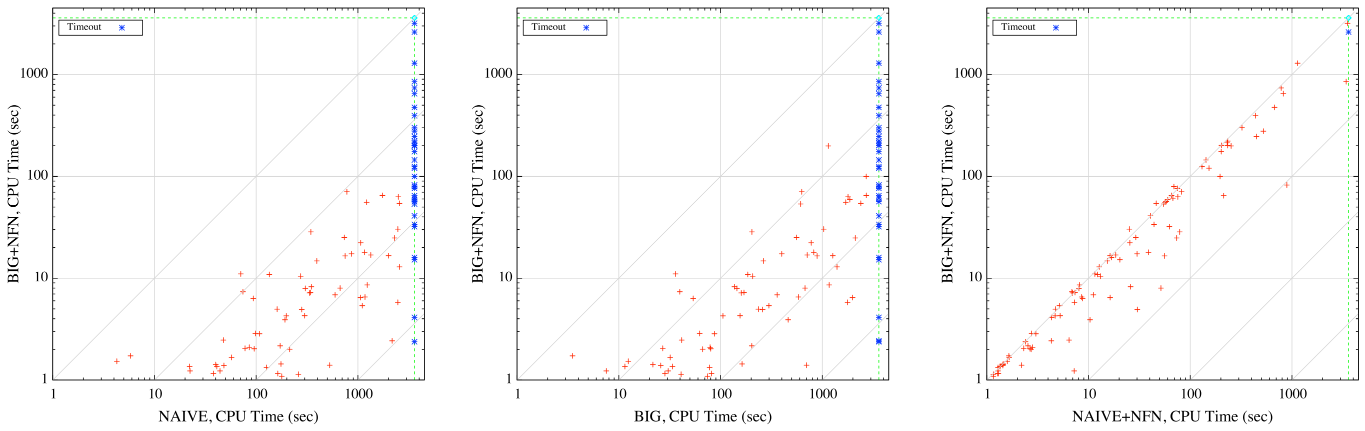

Fig. 9. CPU time comparison between selected algorithms.

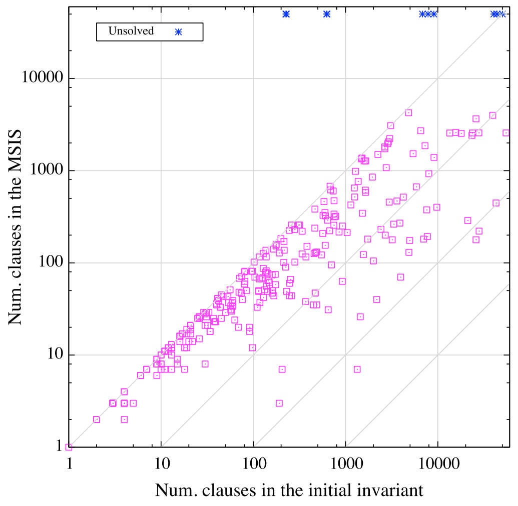

Fig. 10. Number of clauses in the MSIS (computed by BIG+NFN) vs. the initial invariant.

of variables.

VII. Conclusion

In this paper, we advocate for the problem of computing small inductive certificates in the context of unbounded model checking. We believe that this problem is as fundamental as computing minimal unsatisfiable subsets, and that it has just as wide of a variety of applications.

We propose an efficient algorithm for finding minimal safe inductive invariants, which combines: (1) An algorithm to under-approximate an MSIS by identifying necessary clauses that must be included in any safe inductive subset; (2) An algorithm to over-approximate an MSIS by removing clauses that do not belong to some safe inductive subset; (3) Two alternative algorithms to compute an MSIS via an optimized reduction to a series of computations of a maximal inductive subset. We show that on the benchmarks from the Hardware Model Checking Competition, our combined algorithm is several orders of magnitude more efficient than a naïve approach.

References

[1] K. L. McMillan and N. Amla, “Automatic Abstraction without Counterexamples,” in *TACAS*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 2–17.

[2] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik, “Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC,” in *FMCAD*. IEEE Computer Society, 2007, pp. 3–12.

[3] K. L. McMillan, “Interpolation and SAT-Based Model Checking,” in *CAV*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13.

[4] A. R. Bradley, “SAT-Based Model Checking without Unrolling,” in *VMCAI*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87.

[5] N. Eén, A. Mishchenko, and R. K. Brayton, “Efficient implementation of property directed reachability,” in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 125–134.

[6] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, “An incremental approach to model checking progress properties,” in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 144–153.

[7] A. Belov and J. Marques-Silva, “MUSer2: An Efficient MUS Extractor,” *JSAT*, vol. 8, no. 1/2, pp. 123–128, 2012.

[8] J. Marques-Silva, M. Janota, and A. Belov, “Minimal sets over monotone predicates in boolean formulae,” in *CAV*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 592–607.

[9] A. Nadel, V. Ryvchin, and O. Strichman, “Efficient MUS extraction with resolution,” in *FMCAD*. IEEE, 2013, pp. 197–200.

[10] A. Gurfinkel, A. Belov, and J. Marques-Silva, “Synthesizing Safe Bit-Precise Invariants,” in *TACAS*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 93–108.

[11] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, “Automatic Abstraction in SMT-Based Unbounded Software Model Checking,” in *CAV*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 846–862.

[12] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, “Incremental formal verification of hardware,” in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 135–143.

[13] C. Sinz, “Towards an Optimal CNF Encoding of Boolean Cardinality Constraints,” in *CP*, ser. Lecture Notes in Computer Science, P. van Beek, Ed., vol. 3709. Springer, 2005, pp. 827–831.

[14] “Hardware Model Checking Competition 2011,” http://fmv.jku.at/hwmcc11.

[15] “Hardware Model Checking Competition 2013,” http://fmv.jku.at/hwmcc13.

[16] P. Bjesse and A. Slobodová, Eds., *International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11, Austin, TX, USA, October 30 - November 02, 2011*. FMCAD Inc., 2011.

[17] N. Sharygina and H. Veith, Eds., *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013.