Chasing Minimal Inductive Validity Cores in Hardware Model Checking

Ryan Berryhill
Department of E&CE
University of Toronto
Toronto, Ontario, Canada
ryan@eecg.toronto.edu

Andreas Veneris
Depts. of E&CE and CS
University of Toronto
Toronto, Ontario, Canada
veneris@eecg.toronto.edu

***Abstract*—Model checking of safety properties is fundamental in formal verification. When a safety property is found to hold, the model checker provides (at best) a machine-checkable certificate that gives limited insight to users and little confidence that the check passes for the "right" reasons, rather than due to *e.g.*, vacuity or unjustified assumptions. Recently, inductive validity cores (IVCs) have been developed to address this issue. In this paper, we lift several algorithms from the field of UNSAT core extraction in order to compute minimal IVCs of hardware safety checking problems. The MARCO algorithm extracts all minimal cores of an UNSAT formula by efficiently exploring the formula's power set, and has already been applied to compute IVCs in software safety checking. The CAMUS algorithm for UNSAT core extraction exploits a duality between minimal correction subsets (MCSes) of a formula and minimal UNSAT cores. We adapt the algorithms to the hardware IVC context, construct a hybrid algorithm that subsumes both CAMUS and MARCO, and introduce novel domain-specific optimizations. Several instances of the hybrid algorithm are presented (including CAMUS and MARCO themselves, among other novel variants) and evaluated empirically on hardware model checking competition circuits, demonstrating the practicality of the proposed algorithm.**

I. Introduction

Model checking techniques for safety properties (such as IC3/PDR [5], [7]) are fundamental in formal verification. Given a circuit and safety property, IC3 returns either (1) UNSAFE along with a *counter-example trace* that leads from an initial state to an unsafe state; or (2) SAFE along with a *safe inductive invariant* certifying that the circuit cannot reach an unsafe state. The former provides actionable feedback to the user that can be applied to identify the source of the failure. The latter typically does not, and as such, provides the user with little confidence that the check passes for the "right" reasons, rather than due to *e.g.*, unjustified assumptions or vacuity. Boolean satisfiability solvers suffer from a similar lack of feedback regarding unsatisfiable formulas; the solver may be able to return a certificate such as a resolution refutation [22], but this provides limited insight. Instead, *unsatisfiable cores*—subsets of the clauses of the formula that are themselves unsatisfiable—appear more useful. Minimal unsatisfiable cores, also called minimal unsatisfiable subsets (MUSes) are of particular interest. Given an unsatisfiable formula, several algorithms have been developed to extract all MUSes [1], [14], [13], [18] or to extract a smallest MUS [11]. MUSes have seen application in a variety of areas including maximum satisfiability solving [15], vacuity detection [19], automated debugging [21], identifying missing constraints in verification [12], and others [16].

Recently, Ghassabani *et al.* lifted unsatisfiable cores to the domain of safety checking with the concept of *inductive validity cores* (IVCs) and introduced algorithms to compute minimal IVCs (MIVCs) [9], [8] of software safety checking problems. In that context, the software system is viewed as a transition relation consisting of a conjunction of constraints, and an IVC is a subset of those constraints that is sufficient to prove safety. In [9], an algorithm called IVC_UC is used to quickly compute a small (but non-minimal) IVC. Subsequently, a brute-force procedure called IVC_BF is used to minimize the IVC. The combination of these two procedures is called IVC_UCBF. The work of [8] presents an approach based on the MARCO algorithm [13] (originally developed to compute all MUSes of an unsatisfiable formula) that finds all MIVCs of a given software safety checking problem. IVCs appear to have natural applications in areas such as vacuity checking [2] and automated debugging [20]. They have already been used to develop novel coverage metrics [10].

In this paper, we consider the problem of finding all MIVCs of a hardware safety checking instance (AllMIVC), and the problem of computing a smallest MIVC (SMIVC). In this context, a circuit consists of a set of *state elements* (registers) and logic gates, and an IVC is an *abstraction* (defined as a circuit containing a subset of the original gates) over which the given safety property holds. Given a circuit specified at the register-transfer level (RTL), MIVCs can similarly be computed over modules, expressions, lines of RTL code, etc., and are expected to provide the user with significantly more usable feedback than a safe inductive invariant. Continuing in the same vein as previous work on MIVC extraction, we proceed by drawing inspiration from well-known MUS extraction algorithms.

The MARCO algorithm [13] directly explores the power set of the set of clauses in an unsatisfiable formula. It exhibits good anytime behavior as it finds MUSes early and steadily throughout its run. However, the usual form of MARCO cannot find a guaranteed-smallest MUS until it terminates, which is a significant disadvantage when considering the SMIVC problem. In contrast, the CAMUS algorithm [14] computes all MUSes of a given unsatisfiable formula by exploiting a hitting set duality between *minimal correction subsets* (MCSes) and MUSes. It first computes all MCSes and then enumerates MUSes in increasing order of size using the hitting set duality. The MCS enumeration step must end before MUS extraction begins, causing poor anytime behavior, but the first MUS found is guaranteed to be a smallest MUS.

This paper develops variants of CAMUS and MARCO targeted at MIVC extraction. We define MCSes of a circuit, and demonstrate the same hitting set duality exists between MCSes

and MIVCs. We further identify an algorithm to efficiently compute MCSes based on SAT-based debugging of hardware circuits [20], [3]. Using these results, we lift the MUS extraction algorithms to handle MIVCs. Further, we present a parameterized *unified algorithm* and show that CAMUS and MARCO are two special cases of the unified algorithm. Next, we present novel variants of the unified algorithm that outperform the usual instantiations of MARCO and CAMUS in this domain. Finally, considering the different bottlenecks inherent to MIVC extraction, we present a series of optimizations that improve runtime performance significantly. In particular, the MIVC extraction context requires frequent calls to IC3 where the MUS extraction context can instead rely on SAT. As these calls to IC3 are significantly more expensive than satisfiability checks, the optimizations are targeted at eliminating as many of them as possible while accelerating the ones that do occur.

Experiments are presented on circuits from the hardware model checking competition (HWMCC) [4]. Two comprehensive sets of experiments are presented. The first executes a large number of variants of the unified algorithm against a small set of "easy" benchmark circuits in an effort to understand which variants and optimizations perform best. Different variants are found to offer different trade-offs between anytime behavior and overall performance. The second set of experiments executes the best-performing configurations against the entire HWMCC 2017 benchmark set in order to evaluate the approaches against a set of modern, challenging benchmarks. Out of 181 circuits, the configuration with the best anytime performance finds at least one MIVC in 61 instances, and computes a total of 1866 MIVCs across those 61 circuits. In contrast, the configuration with the best overall performance finds at least one MIVC in 49 instances, and finds a total of 6907 MIVCs across those 49 circuits. The same configuration also performs best considering the SMIVC problem, finding a guaranteed-smallest MIVC in all 49 cases.

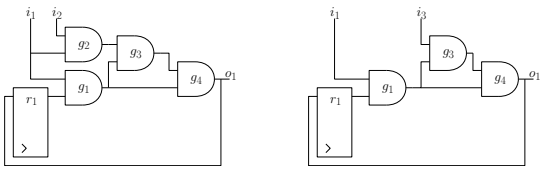
The rest of this paper is organized as follows. Section II presents background material on MUS extraction and MIVC extraction. Section III presents MARCO and CAMUS adapted to MIVC extraction and the unified algorithm. Section IV presents variants of the unified algorithm, while Section V presents a series of performance-driven enhancements. Section VI presents experimental results, while Section VII concludes the paper.

II. Preliminaries

A. Notation and Terminology

The following terminology and notation is used throughout this paper. A *literal* is either a variable or its negation, a *clause* is a disjunction of literals, a *cube* is a conjunction of literals, and a Boolean formula in Conjunctive Normal Form (CNF) is a conjunction of clauses. Where convenient, a CNF formula φ is treated as a set of clauses where $c \in \varphi$ means that clause c appears in φ. A Boolean formula is satisfiable (SAT) if there exists an assignment to its variables such that the formula evaluates to 1. Otherwise it is unsatisfiable (UNSAT).

Given a finite transition system with a set of state variables $\mathcal{V}$, the primed forms $\mathcal{V}' = \{v' | v \in \mathcal{V}\}$ refer to the next-state functions. That is, for every state variable $v \in \mathcal{V}$, v' is a Boolean function of the current state and input defining the next state for v. A safety checking instance is represented by a triple $P = (Init, Tr, Bad)$, where $Init(\mathcal{V})$ and $Bad(\mathcal{V})$ are



(a) (b)

Fig. 1. Circuit with (a) $Tr = \{g_1, g_2, g_3, g_4\}$, (b) abstraction $\{g_1, g_3, g_4\}$

CNF formulas over $\mathcal{V}$ that represent the initial states and the unsafe states, respectively. The transition relation $Tr(\mathcal{V}, \mathcal{V}')$ is encoded as a CNF formula over $\mathcal{V} \cup \mathcal{V}'$ such that $Tr(\vec{v}, \vec{v}')$ is satisfiable if and only if state $\vec{v}$ can transition to state $\vec{v}'$.

B. Model Checking of Safety Properties

A safety property is expressed in Computation Tree Logic (CTL) as $\mathbf{AG}\,\neg Bad$. Hence, an instance $(Init, Tr, Bad)$ is UNSAFE if the formula below is SAT for some value of N:

$$Init(\vec{v_0}) \wedge \Big(\bigwedge_{i=0}^{N-1} Tr(\vec{v_i}, \vec{v_{i+1}}) \Big) \wedge Bad(\vec{v_N}) \tag{1}$$

The satisfying assignment provides a *counter-example trace*: a sequence of input assignments that leads from an initial state to an unsafe state. Conversely, the instance is SAFE if and only if there exists a formula $Inv(\mathcal{V})$ such that:

$$Init(\vec{v}) \Rightarrow Inv(\vec{v}) \tag{2}$$

$$Inv(\vec{v}) \wedge Tr(\vec{v}, \vec{v}') \Rightarrow Inv(\vec{v}') \tag{3}$$

$$Inv(\vec{v}) \Rightarrow \neg Bad(\vec{v}) \tag{4}$$

The properties above are called *initiation* (Eq. 2), *induction* (Eq. 3), and *safety* (Eq. 4). A formula with all three properties is called a *safe inductive invariant* and certifies that the instance is SAFE. Given a safety checking instance, the model checking algorithm IC3 [5] returns either SAFE (along with a safe inductive invariant) or UNSAFE (along with a counter-example trace).

C. MCSes, MSSes, and MUSes

Given an UNSAT formula φ in CNF, any subset $\varphi_1 \subseteq \varphi$ that is itself UNSAT is called an *UNSAT core* of φ. If φ_1 is minimal, meaning that every proper subset of φ_1 is SAT, then φ_1 is a *minimal unsatisfiable subset* (MUS). A subset $C \subseteq \varphi$ is a *minimal correction subset* (MCS) if $\varphi \setminus C$ is SAT, but for every proper subset $D \subsetneq C$, $\varphi \setminus D$ is UNSAT. A *maximal satisfiable subset* (MSS) of φ is the complement of an MCS. In other words, $D \subseteq \varphi$ is an MSS if D is SAT, but for any clause $c \in \varphi \setminus D$, $D \cup \{c\}$ is UNSAT.

Given a set of sets $C = \{C_1, \ldots, C_{|C|}\}$, a *hitting set* of C is a set H such that the intersection of H and C_i is nonempty for every $C_i \in C$. We let $\mathrm{MCSes}(\varphi)$ (resp., $\mathrm{MUSes}(\varphi)$) denote the set of all MCSes (resp., MUSes) of φ. A hitting set duality exists between MUSes and MCSes: a subset C of φ is an MUS if and only if C is a minimal hitting set of $\mathrm{MCSes}(\varphi)$ [14].

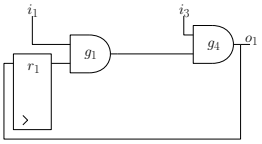


Fig. 2. MIVC $\{g_1, g_4\}$ of $(\overline{r_1}, Tr, o_1)$, where Tr is as shown in Fig. 1

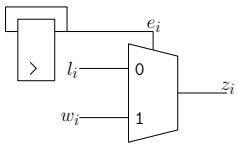


Fig. 3. Multiplexer construction for automated debugging

D. MIVC Extraction

This paper presents algorithms to extract MIVCs of hardware safety checking instances. Where convenient, we treat the transition relation Tr as the set of its logic gates. A subset $S \subseteq Tr$ represents an *abstraction* of Tr in which the output of every gate in $Tr \setminus S$ is replaced by a new primary input. Figure 1 shows an example of this operation. We extend the definition of an MCS to the context of safety checking as follows.

Definition 1 *A **Correction Subset** of a* SAFE *model checking instance $(Init, Tr, Bad)$ is a subset $S \subseteq Tr$ such that $(Init, Tr \setminus S, Bad)$ is* UNSAFE. *A **Minimal Correction Subset** (MCS) is a correction subset for which no proper subset is a correction subset.*

We further define IVCs and MIVCs, which are analogous to UNSAT cores and MUSes, respectively, as follows.

Definition 2 *An **Inductive Validity Core** (IVC) of a* SAFE *model checking instance $(Init, Tr, Bad)$ is an abstraction $S \subseteq Tr$ such that $(Init, S, Bad)$ is* SAFE. *A **Minimal Inductive Validity Core** (MIVC) is an IVC for which every proper subset is not an IVC.*

An IVC is simply a SAFE abstraction of the circuit. Figure 2 shows an MIVC of the circuit in Figure 1(a), with initial state $Init = (\overline{r_1})$ and unsafe state $Bad = o_1$. It can easily be verified that $(\overline{r_1}, \{g_1, g_4\}, o_1)$ is SAFE and therefore $\{g_1, g_4\}$ is an IVC. Further, $\{g_1, g_4\}$ is minimal, as $(\overline{r_1}, \{g_1\}, o_1)$ and $(\overline{r_1}, \{g_4\}, o_1)$ are both UNSAFE. We also define maximal unsafe abstractions (MUAs) analogously to MSSes of unsatisfiable CNF formulas.

Definition 3 *A **Maximal Unsafe Abstraction** (MUA) is an abstraction $S \subseteq Tr$ that is* UNSAFE *and for which $S \cup \{g_i\}$ is* SAFE *for every $g_i \in (Tr \setminus S)$.*

For instance, $\{g_1, g_2, g_3\}$ and $\{g_2, g_3, g_4\}$ are MUAs of the circuit in Figure 1(a). Given an MCS S, the set $Tr \setminus S$ (the complement of S) is an MUA. Likewise, the complement of an MUA is an MCS.

The `IVC_UCBF` algorithm [9] efficiently computes a single MIVC of a safety checking instance. While originally presented for software MIVCs, the same algorithm applies to hardware. It takes as input a safety checking instance $(Init, Tr, Bad)$ that is assumed to be SAFE. The algorithm works in two steps: the first (`IVC_UC`) computes an IVC, while the second (`IVC_BF`) uses brute force to minimize the result to an MIVC.

The first step, `IVC_UC`, uses a model checker to compute a safe inductive invariant[1] Inv, which is assumed to be a CNF formula over $\mathcal{V}$. As a performance optimization, the invariant is then reduced to a subset of its clauses that is also a safe inductive invariant. Applying this optimization may cause the algorithm to find a different MIVC, but it will not change the fact that algorithm finds a *minimal* IVC. Next, using standard UNSAT core techniques, the algorithm computes an abstraction $S \subseteq Tr$ that is sufficient to make the following formula UNSAT:

$$Inv(\vec{v}) \wedge S(\vec{v}, \vec{v}') \wedge \neg Inv(\vec{v}') \tag{5}$$

The unsatisfiability of Eq. 5 is equivalent to the condition for induction described in Eq. 3. Safety and initiation also hold for S by construction, so S is an IVC, but it may not be minimal. The second step, `IVC_BF`, uses brute force to minimize S. It works by repeatedly removing a gate from S and checking the satisfiability of Eq. 5. If it is SAT, the gate is added back to S. The process repeats until every gate has been considered, at which point S is an MIVC.

E. Automated Debugging

This section presents an algorithm that uses SAT-based automated debugging [20] to compute MCSes of a safety checking problem, which is essential to the algorithms presented in this paper. The work of [3] presents an algorithm to diagnose a failing *reachability property*, which is a property that requires a certain state $Good$ to be reachable (*i.e.*, $\mathbf{EF}Good$ in CTL). Equivalently, such a property requires that $(Init, Tr, Good)$ is UNSAFE. The algorithm takes as input an *error cardinality* $n \geq 1$ and a user-provided set of *suspect locations* in the circuit $L = \{l_1, \ldots, l_{|L|}\}$ and returns n-tuples of suspect locations where a change can be made to correct the failure. More formally and assuming $L = Tr$, it returns every n-subset $S \subseteq Tr$ such that $(Init, Tr \setminus S, Good)$ is UNSAFE. That is, it returns all correction subsets of cardinality n.

At a high level, the algorithm works as follows. The enhanced transition relation Tr_{en} is constructed from Tr by adding *error-select lines* $E = \{e_1, e_2, \ldots, e_{|Tr|}\}$ and new primary inputs $W = \{w_1, w_2, \ldots, w_{|Tr|}\}$. When $e_1 = 1$, l_i is replaced by w_i. This operation can be implemented using the multiplexer construction shown in Figure 3. When simulating or model checking Tr_{en}, it behaves like Tr, except with certain locations l_i (those for which $e_i = 1$) replaced by inputs. A formula is constructed representing the enhanced initial states $Init_{en} = Init \wedge \Phi_n$, where Φ_n is a cardinality constraint enforcing that exactly n error-select lines are assigned to `1`. The safety checking instance $(Init_{en}, Tr_{en}, Good)$ is then solved using `IC3`. If there is a counter-example, the n error-select lines that are assigned to `1` indicate a correction subset. As a performance optimization, BMC can be used to find solutions for a bounded number of time-frames, and then `IC3` can be used to find any remaining solutions and conclude that no other solutions exist.

[1] In [9], this is more generally a safe k-inductive invariant. We present the algorithm in terms of safe inductive invariants for simplicity.

III. A Unified Algorithm for MIVC Extraction

This section presents an MIVC enumeration algorithm that unifies CAMUS and MARCO. Section III-A adapts the MARCO algorithm to MIVC extraction based on the work of [8]. Section III-B adapts CAMUS to MIVC extraction. Finally, Section III-C presents an approach that unifies both algorithms.

A. MARCO

Given an unsatisfiable CNF formula, MARCO [13] extracts all MUSes of the formula by directly exploring the power set of the set of clauses in the formula. The algorithm has already been applied to extract MIVCs of software safety checking instances [8]. Algorithm 1 shows pseudocode for the procedure, adapted to the hardware MIVC context. It takes a safety checking problem $(Init, Tr, Bad)$ as input and explores the power set of Tr.

Line 2 initializes a CNF formula called the *map* that tracks which portions of the power set are explored. The map is a CNF formula with one variable s_i for each gate $g_i \in Tr$. Each satisfying assignment of the map corresponds to an abstraction of Tr that includes g_i if and only if s_i is assigned to 1. Lines 3–13 compute MIVCs. Within the loop, line 4 extracts an abstraction of Tr called a *seed* from the map. If the seed is UNSAFE, line 10 calls grow to expand it to an MUA, which is stored in the variable mua. An efficient implementation of grow is discussed in Section V-A. The map is then updated on line 11 by adding a clause (shown in Eq. 6) that blocks the MUA *and all of its subsets*, as any subset of an MUA is UNSAFE. The clause requires any future seeds to contain an element of the MCS $Tr \setminus mua$.

$$\texttt{blockDown}(S) = \bigvee_{g_i \in Tr \setminus S} s_i \tag{6}$$

If the seed is SAFE, line 6 calls shrink to reduce the seed to an MIVC. This is implemented using the IVC_UCBF procedure described in Section II-D. The map is subsequently updated on line 8 by adding a clause (shown in Eq. 7) that blocks the MIVC *and all of its supersets*. A strict superset of an MIVC is indeed an IVC, but is not minimal and therefore can safely be ignored. Blocking all supersets of MIVCs and all subsets of MUAs is key to the algorithm's efficiency.

$$\texttt{blockUp}(S) = \bigvee_{g_i \in S} \neg s_i \tag{7}$$

B. CAMUS

Given an unsatisfiable CNF formula, CAMUS [14] extracts all MUSes of the formula in order from smallest to largest using the hitting set duality between MCSes and MUSes noted in Section II-C. First, it enumerates all MCSes of Φ, which can be done using a variety of approaches [17]. Subsequently, the algorithm computes MUSes as minimal hitting sets of the MCSes. Lifting CAMUS to MIVC extraction requires a way to compute MCSes of safety checking instances and showing the existence of a hitting set duality between MCSes and MIVCs. The former is described in Section II-E, while the latter is demonstrated in Theorem 1 below.

Theorem 1 *Given a safety checking problem $(Init, Tr, Bad)$, a subset $S \subseteq Tr$ is an IVC if and only if S is a hitting set of* $\mathrm{MCSes}(Init, Tr, Bad)$.

Algorithm 1 MARCO for MIVC extraction

Input: safety checking problem $(Init, Tr, Bad)$
Output: set of all MIVCs of $(Init, Tr, Bad)$

1: $MIVCs \leftarrow \emptyset$
2: $map \leftarrow \top$
3: **while** map is SAT **do**
4: $\quad seed \leftarrow \texttt{getUnexplored}(map)$
5: $\quad$ **if** $(Init, seed, Bad)$ is SAFE **then**
6: $\quad\quad mivc \leftarrow \texttt{shrink}(seed)$
7: $\quad\quad MIVCs \leftarrow MIVCs \cup \{mivc\}$
8: $\quad\quad map \leftarrow map \wedge \texttt{blockUp}(mivc)$
9: $\quad$ **else**
10: $\quad\quad mua \leftarrow \texttt{grow}(seed)$
11: $\quad\quad map \leftarrow map \wedge \texttt{blockDown}(mua)$
12: $\quad$ **end if**
13: **end while**
14: **return** $MIVCs$

Proof: Suppose S is an IVC. By Definition 1, there is no MCS Ψ such that S is contained in $Tr \setminus \Psi$ (otherwise S would be UNSAFE). Thus, for every MCS Ψ, $S \cap \Psi \neq \emptyset$, implying that S is a hitting set of $\mathrm{MCSes}(Init, Tr, Bad)$.

Now suppose S is a hitting set of $\mathrm{MCSes}(Init, Tr, Bad)$. For every MCS Ψ, S contains an element of Ψ, so $S \not\subseteq (Tr \setminus \Psi)$. This implies S is SAFE and therefore an IVC. ■

Algorithm 2 shows pseudocode for the MIVC enumeration variant of CAMUS, presented in terms of the same subroutines used by MARCO. The loop beginning on line 3 repeatedly extracts an MCS and updates the map accordingly using blockDown. This step is closely related to lines 10–11 of Algorithm 1 in which MARCO computes an MUA and then refines the map by calling blockDown on the MUA. In contrast, CAMUS directly computes an MCS, the complement of which (*i.e.*, $Tr \setminus mcs$) is an MUA by definition. Ultimately, the clause added to the map in this step requires any satisfying assignment to correspond to a subset of Tr that hits mcs.

After all MCSes have been found, the loop beginning on line 7 extracts MIVCs using a process similar to the main loop of MARCO with several steps removed. First, since every seed is a hitting set of the MCSes, there is no need to check for safety as every seed is guaranteed to be an IVC. Second, instead of calling getUnexplored, CAMUS uses getUnexploredMin, which returns an unexplored seed of minimum cardinality. This ensures IVCs are enumerated from smallest to largest and that each IVC found is minimal. Third, as every seed is of minimum cardinality, there is no need to call shrink upon discovering a SAFE seed.

C. Unified Algorithm

From the descriptions in the preceding subsections, it is clear that MARCO and CAMUS have significant similarities. This gives rise to a *unified algorithm* for MIVC extraction that generalizes both CAMUS and MARCO. Pseudocode for the unified algorithm is presented in Algorithm 3. It takes as input a safety checking problem and a parameter k that controls the behavior of the MCS enumeration step. The loop on lines 3–8 enumerates MCSes of cardinality k or less and updates the map accordingly. The parameter k provides a configurable trade-off between anytime performance (improved by lowering k) and overall performance (improved by increasing k).

Algorithm 2 CAMUS for MIVC extraction

Input: safety checking problem $(Init, Tr, Bad)$
Output: set of all MIVCs of $(Init, Tr, Bad)$

1: $MIVCs \leftarrow \emptyset$
2: $map \leftarrow \top$
3: **while** more MCSes exist **do**
4: $mcs \leftarrow \texttt{FindMCS}(Init, Tr, Bad)$
5: $map \leftarrow map \wedge \texttt{blockDown}(Tr \setminus mcs)$
6: **end while**
7: **while** map is SAT **do**
8: $mivc \leftarrow \texttt{getUnexploredMin}(map)$
9: $MIVCs \leftarrow MIVCs \cup \{mivc\}$
10: $map \leftarrow map \wedge \texttt{blockUp}(mivc)$
11: **end while**
12: **return** $MIVCs$

That is, finding more MCSes upfront reduces the number of iterations required in the second loop, which results in fewer calls to grow, shrink, and the model checker. Subsequently, the loop on lines 9–19 enumerates MIVCs using an approach that is very similar to the main loop of MARCO.

Algorithm 3 UMIVC

Input: safety checking problem $(Init, Tr, Bad), k \in \mathbb{Z} \cup \{\infty\}$
Output: set of all MIVCs of $(Init, Tr, Bad)$

1: $MIVCs \leftarrow \emptyset$
2: $map \leftarrow \top$
3: **for** $i = 1$ to k **do**
4: **while** more MCSes of cardinality i exist **do**
5: $mcs \leftarrow \texttt{FindMCS}(Init, Tr, Bad, i)$
6: $map \leftarrow map \wedge \texttt{blockDown}(Tr \setminus mcs)$
7: **end while**
8: **end for**
9: **while** map is SAT **do**
10: $seed \leftarrow \texttt{getUnexplored}(map)$
11: **if** $k = \infty$ **or** $(Init, seed, Bad)$ is SAFE **then**
12: $mivc \leftarrow \texttt{shrink}(seed)$
13: $MIVCs \leftarrow MIVCs \cup \{mivc\}$
14: $map \leftarrow map \wedge \texttt{blockUp}(mivc)$
15: **else**
16: $mua \leftarrow \texttt{grow}(seed)$
17: $map \leftarrow map \wedge \texttt{blockDown}(mua)$
18: **end if**
19: **end while**
20: **return** $MIVCs$

IV. Algorithm Variants

This section presents several important variants of Algorithm 3, distinguished by different strategies for exploring the search space and different amounts of upfront computation. In particular, different variants pass different values of k to Algorithm 3 and compute unexplored seeds using one of the following approaches:

- getUnexplored, which returns an arbitrary unexplored seed using SAT;
- getUnexploredMin, which returns an unexplored seed of minimum cardinality using MaxSAT;
- getUnexploredMax, which returns an unexplored seed of maximum cardinality using MaxSAT; or
- getUnexploredZZ (zig-zag), which alternates between returning minimum- and maximum-cardinality unexplored seeds on consecutive calls.

TABLE I
Summary of UMIVC variants

Variant	k	Exploration
CAMUS	∞	getUnexploredMin
MARCO	0	getUnexplored
MARCO-DOWN	0	getUnexploredMax
MARCO-UP	0	getUnexploredMin
MARCO-ZZ	0	getUnexploredZZ
k-UMIVC	k	getUnexplored
k-UMIVC-DOWN	k	getUnexploredMax
k-UMIVC-UP	k	getUnexploredMin
k-UMIVC-ZZ	k	getUnexploredZZ

Table I summarizes all of the variants, which are discussed in more detail in the following subsections.

A. CAMUS and MARCO

CAMUS corresponds to the variant where $k = \infty$ and getUnexploredMin is used for exploration. All MCSes are computed upfront and the search space is explored bottom-up *i.e.*, starting from the smallest seeds. In this variant safety checking is unnecessary, grow is never called, and shrink can be replaced with a no-op as seeds are always minimal and therefore any seed returned by getUnexploredMin is guaranteed to be an MIVC.

The simplest form of MARCO described in Section III-A corresponds to a variant where $k = 0$ and getUnexplored is used for exploration. The search space is explored arbitrarily, with neither a bottom-up nor a top-down bias. The primary advantage of this variant is that getUnexplored may require less runtime. Experimentally, we observe that getUnexplored accounts for very little runtime, so optimizing its runtime is not expected to be beneficial.

We also consider several other variants of MARCO. The first, MARCO-DOWN, uses getUnexploredMax to explore the search space in a top-down fashion *i.e.*, starting with the largest seeds. In this variant, grow can be replaced by a no-op as seeds are always maximal, and therefore any UNSAFE seed returned by getUnexploredMax is guaranteed to be an MUA. This variant (with additional optimizations) was referred to as "Optimized MARCO" in the work of [13] and was used to extract MIVCs of software safety checking problems in [8]. A significant advantage of MARCO-DOWN is that the first iteration of the main loop is guaranteed to find an MIVC, which provides excellent anytime performance—the algorithm computes a single MIVC as quickly as IVC_UCBF.

In contrast, MARCO-UP uses getUnexploredMin to explore the search space bottom-up and is presented as a dual variant of MARCO-DOWN in [13]. A closely-related smallest MUS extraction algorithm was presented in [11]. In this variant, seeds are always minimal and calling shrink is unnecessary. CAMUS and MARCO-UP are closely related; both algorithms work by finding minimal hitting sets of the MCSes. The only difference is that CAMUS computes all MCSes upfront, whereas MARCO-UP does not. Instead, it repeatedly computes a seed as a minimal hitting set of the currently-known MCSes, checks if the seed is an MIVC, and if not,

computes a new MCS. A significant advantage of this variant is that the first MIVC it finds is guaranteed to be an SMIVC.

The variant MARCO-ZZ explores the search space in a zig-zagging fashion, alternating between a top-down and bottom-up search on consecutive iterations of the main loop. This variant preserves some advantages of both the bottom-up and top-down approaches. First, in an iteration in which getUnexploredZZ returns a minimum seed, the call to shrink can be eliminated. Likewise, for maximum seeds, the call to grow can be eliminated. Second, in our implementation the first iteration finds a maximum seed, which preserves the fast time-to-first-MIVC of MARCO-DOWN. Finally, this variant can also discover a smallest MIVC before termination, though it is unlikely to do so as quickly as MARCO-UP.

B. Hybrid Variants

The primary difference between the unified algorithm and existing approaches is the potential to combine the advantages of MARCO and CAMUS. By front-loading the computation of all MCSes, CAMUS avoids the need to perform any safety checks in the MIVC extraction loop, including those that result from calling grow or shrink. This comes at the cost of poor anytime behavior, as computing all MCSes is itself often intractable. The hybrid variants front-load some computation of MCSes, potentially achieving a more useful trade-off between overall performance and anytime behavior.

Each of the MARCO variants has a hybrid counterpart: k-UMIVC, k-UMIVC-DOWN, k-UMIVC-UP, and k-UMIVC-ZZ. In these variants, all MCSes of size k or less are computed upfront. Due to an optimization presented in Section V-B, computing MCSes upfront can be less computationally-expensive than computing them on-demand using grow in the second loop. This observation holds true even when using an optimized implementation of grow described in Section V-A. In addition, each MCS computed upfront avoids running one iteration of the MIVC extraction loop, which saves a call to IC3 and potentially avoids a call to grow or shrink.

V. Performance Optimizations

This section presents a series of performance optimizations to the UMIVC algorithm. MARCO and CAMUS were initially developed for MUS extraction, but extracting MIVCs presents a different set of constraints and performance characteristics. In each context, a large number of seeds are generated, and each needs to be checked for either satisfiability (in the MUS context) or safety (in the MIVC context). Naturally, safety checking is significantly more computationally expensive than satisfiability checking, and this drives different implementation decisions in the MIVC context. In particular, the greater expense of safety checking necessitates optimizations that aggressively reduce the number of calls to the model checker and accelerates the calls that do occur.

A. Implementing grow via MCS Extraction

A key reason that MARCO-DOWN performs well in software MIVC extraction [8] is the use of IVC_UCBF to efficiently shrink seeds. The runtime of MARCO-UP and MARCO-ZZ variants also depends on an efficient implementation of grow. A brute-force implementation of grow repeatedly selects a gate $g \in Tr \setminus seed$ and checks if $(Init, seed \cup \{g\}, Bad)$ is UNSAFE. If so, g is added to the seed. The process repeats until every gate of $Tr \setminus seed$ has been considered. The optimized implementation leverages the fact that the complement of an MCS is an MUA. The algorithm simply finds an MCS S over $Tr \setminus seed$ (using the algorithm described in Section II-E with a suspect set equal to $Tr \setminus seed$) and then returns $Tr \setminus S$. Early experiments showed that this optimization gives drastic performance improvements, so it is enabled for all experiments presented in Section VI. Indeed, without this optimization MARCO-UP is rarely able to find any MIVCs.

B. Approximating Upfront MCS Computation

The hybrid UMIVC variants compute a subset of all MCSes upfront using the algorithm described in Section II-E. That is, for each i from 1 to k, BMC is used to find MCSes of cardinality i, and then IC3 is used to find any remaining MCSes of cardinality i and finally prove that they have all been found. This approach is guaranteed to find every MCS of size k or less. However, correctness of the hybrid UMIVC variants does not depend on every such MCS being found; it does not even require the correction sets found in this step to be minimal. If a particular MCS is not found during this step, it will instead be found in the MIVC enumeration loop, and thus the resulting set of MIVCs is unchanged.

By sacrificing completeness and minimality, this step can be implemented entirely using BMC. For each i starting from 1, BMC is used to search for correction subsets of cardinality i. The BMC-based search at each cardinality is terminated after reaching a suitable stopping condition, such as unrolling for a specific number of time-frames. This approximation eliminates the IC3 calls that dominate the runtime of this step. Early experiments showed that this optimization yields drastic performance improvements, so it is enabled for all experiments presented in Section VI.

C. Caching Invariants and Counter-examples

The UMIVC algorithm executes many safety checks against various abstractions of the given circuit. In practice, different abstractions tend to have similar behavior. In particular, a counter-example (resp., safe inductive invariant) that witnesses unsafety (resp., safety) for a particular abstraction is likely to be a witness for other abstractions. As a performance optimization, our implementation maintains least recently used (LRU) caches of counter-examples and safe inductive invariants. When a new counter-example or safe inductive invariant is found for any abstraction, it is put at the front of the LRU cache. If the cache grows beyond a pre-determined size limit, the least recently used element is evicted.

When checking an abstraction for safety and before running BMC or IC3, each cached counter-example is re-simulated on the abstraction. If one witnesses unsafety, it is moved to the front of the LRU cache. If no cached counter-example is a witness, each cached invariant is checked to see if it *contains* a valid safe inductive invariant for the abstraction. For cached invariant Inv, this is accomplished using an invariant finder algorithm [6] that finds the maximum subset of Inv that is an inductive invariant with respect to the abstraction, if any exists. As is the case for counter-examples, when a witness to safety is found, it is moved to the front of the LRU cache. If neither cache contains a witness, a normal safety check is executed, which may involve running BMC and IC3.

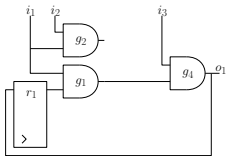


Fig. 4. Abstraction $\{g_1, g_2, g_4\}$ of Tr (Fig. 1(a)), in which g_2 is isolated

D. Biased Safety Checking

Considering the caching optimization presented in the previous subsection, checking if a cached invariant witnesses safety is still an expensive operation. In UMIVC, many safety checks have an "expected" outcome, depending on where in the algorithm the model checker is called. For instance, when running a bottom-up search and calling grow, the algorithm is often starting from a very small UNSAFE seed. In this case, it is probable that many gates need to be added to the seed before it becomes SAFE, and therefore most safety checks will return UNSAFE. Likewise, when searching top-down and starting with a large SAFE seed, it may be the case that many gates need to be removed before reaching an UNSAFE seed and most safety checks will return SAFE.

Based on this intuition, our implementation considers the expected outcome of a safety check when deciding the order in which to execute individual checks. If the expected result is UNSAFE or no bias is expected, the checks are executed in the following order: counter-example cache, BMC, invariant cache, IC3. If the expected result is SAFE, the following order is used instead: invariant cache, BMC, IC3; the counter-example cache is skipped as it was found to rarely yield a result in this case. Safety checks made from grow are expected to be UNSAFE, as are those made on a seed returned from getUnexploredMin. Safety checks made from shrink are expected to return SAFE, as are those made on a seed returned from getUnexploredMax.

E. Exploiting Structure

In order to be *minimal* IVC, an abstraction cannot contain any gates that do not affect its behavior. For example, consider the circuit in Figure 4, where the output of gate g_2 is disconnected. Since its output is ignored, g_2 does not affect the behavior of the circuit and $\{g_1, g_4\}$ is an equivalent circuit. This implies that $\{g_1, g_2, g_4\}$ cannot be an MIVC, since if it is SAFE, so is the subset $\{g_1, g_4\}$. This notion is formalized as the following lemma, where $\text{fanout}(g_i)$ denotes the set of all gates for which g_i is an input.

Lemma 1 *Let $S \subseteq Tr$ be an MIVC of $(Init, Tr, Bad)$. For every gate $g_i \in S$, either: (1) g_i is the single output of the circuit; (2) g_i is the next-state function of a state element; or (3) an element g_j of $\text{fanout}(g_i)$ is in S.*

Proof: Assume towards a contradiction $g_i \in S$ but g_i is neither an output, a next-state function, nor an input of any $g_j \in S$. Thus, the output of g_j is disconnected in S, so $S \backslash \{g_j\}$ is also an IVC, contradicting the minimality of S. ∎

For any gate g_i that is not a next-state function or an output, Lemma 1 allows adding the clause below to the map without blocking any MIVCs.

$$\left(\neg s_i \vee \bigvee_{g_j \in \text{fanout}(g_i)} s_j\right)$$

Adding these clauses can prune large portions of non-solution space (which may contain IVCs, but not MIVCs).

VI. EXPERIMENTS

This section presents empirical results comparing the presented algorithms[2] on benchmark safety checking instances from the Hardware Model Checking Competition (HWMCC). Two comprehensive sets of experiments are presented. The first evaluates several configurations of UMIVC (*i.e.,* different variants with different optimizations from Section V enabled) against a sample of 50 benchmark circuits from the HWMCC 2011 benchmark set. Since MIVC extraction appears more difficult than model checking, using an older benchmark set (with "easier" problem instances) gives more meaningful results with fewer across-the-board timeouts. This set of experiments is intended to evaluate each variant and optimization to determine which configurations perform best. Subsequently, for each of the major variants (k−UMIVC−DOWN, k−UMIVC−UP, k−UMIVC−ZZ, and CAMUS), a configuration that is expected to perform well is executed against all of the SAFE benchmarks in the HWMCC 2017 set. The second set of experiments is intended to evaluate the best-performing configurations against a large set of modern benchmark circuits. All experiments are executed on a Linux workstation with an Intel Core i7-8700 CPU clocked at 3.2 GHz with a memory limit of 16 GB and a time limit of 15 minutes per benchmark circuit. The optimizations to grow (Section V-A) and to upfront MCS computation computation (Section V-B) are enabled in all cases, as is the IVC_UCBF-based shrink procedure.

In the first set of experiments, there are 12 circuits for which no evaluated algorithm could find an MIVC within the time limit. These are removed from further consideration, leaving 38 circuits. Table II presents a summary of the results. Due to space constraints, not all configurations tested are shown. In particular, for hybrid UMIVC only one value of k is shown for each search strategy. Each row in the table refers to a configuration of the UMIVC algorithm. The first column shows the variant, the second shows the value of k, and the next three columns indicate if the (S)tructure optimization (Section V-E), (C)aching optimization (Section V-C), and (B)iased safety (Section V-D) optimizations are present, respectively. The sixth column (MCS) indicates for how many benchmarks the configuration was able to complete the MCS enumeration step (top line) and the number of seconds to do so summed across all benchmarks (bottom line). The next three columns show similar information related to finding a single MIVC, finding all MIVCs, and finding a guaranteed-smallest MIVC, respectively. The tenth column (#MIVC) reports the total number of MIVCs found across all circuits. The final two columns indicate the total number of calls made to IC3 and the total time taken for all IC3 calls, respectively. For clarity, we refer to configurations by their name followed by a plus sign and the optimizations enabled. For instance, the configuration in the bottom row is called 0−UMIVC−ZZ+CB while the top row is ∞−UMIVC−UP (*i.e.*, CAMUS).

[2]Implementations can be found at: https://github.com/ryanberryhill/pme

TABLE II
SUMMARY OF RESULTS (38 HWMCC 2011 CIRCUITS)

Variant	k	S	C	B	MCS	One MIVC	All MIVC	SMIVC	# MIVC	# IC3	IC3 time
UP	∞				16 20941	16 20943	16 21854	16 20943	69	195	1678
UP	∞	Y			16 20944	16 20947	16 20963	16 20947	71	195	1682
UP	0				-	25 13519	17 19438	25 13519	64	135	2828
UP	3				26 11623	23 14365	18 19928	23 14365	362	429	5391
UP	0	Y			-	25 12978	19 18843	25 12978	110	164	1969
UP	0		Y		-	25 13547	18 19463	25 13547	89	99	4597
UP	0		Y	Y	-	25 13552	18 19443	25 13552	89	99	4595
UP	3	Y	Y	Y	26 11630	23 14371	19 19421	23 14371	330	101	6975
DOWN	0				-	34 6168	19 18820	19 18820	287	39180	7438
DOWN	2				32 6996	31 9176	20 18106	20 18106	290	34883	4908
DOWN	0	Y			-	34 6129	22 17032	22 17032	323	41123	7408
DOWN	0		Y		-	37 4071	17 20706	17 20706	171	1659	2169
DOWN	0		Y	Y	-	38 2505	19 19352	19 19352	448	2396	3552
DOWN	2	Y	Y	Y	32 7081	21 8355	23 16929	23 16929	550	901	1572
ZZ	0				-	34 6219	17 19945	19 19039	192	27215	6662
ZZ	3				26 11607	25 13495	17 21466	21 17434	256	20272	3922
ZZ	0	Y			-	34 6188	10 27669	10 27669	56	18482	6448
ZZ	0		Y		-	37 4061	7 29642	7 29642	67	4891	2181
ZZ	0		Y	Y	-	38 2477	7 29593	7 29592	71	4596	2532
ZZ	3	Y	Y	Y	26 11621	26 12600	18 19496	22 15836	428	94	600

The results demonstrate that the top-down, bottom-up, and zig-zagging variants all offer worthwhile trade-offs. Bottom-up variants outperform other variants at finding smallest MIVCs, as expected. The top-down and zig-zagging variants perform best at finding one MIVC, as the first step in these approaches is to compute an MIVC using IVC_UCBF. The zig-zagging variant offers an appealing trade-off as it computes a single MIVC as quickly as the top-down variant while finding SMIVCs faster—though not as fast as the bottom-up variant. The hybrid variants appear to trade off anytime behavior (primarily observed through the "One MIVC" column) for overall performance (primarily observed through the "# MIVC" column). For instance, when comparing 0-UMIVC-UP (*i.e.*, MARCO-UP) to the hybrid variant 3-UMIVC-UP, increasing k appears to slightly degrade anytime performance, as 3-UMIVC-UP finds a single MIVC in only 23 cases, compared to 25 for 0-UMIVC-UP. However, 3-UMIVC-UP has significantly better overall performance, as it finds a total of 362 MIVCs compared to 64 for 0-UMIVC-UP.

The optimizations presented in Section V also appear highly successful. While the optimization of Section V-B is not directly evaluated as it is enabled in every configuration, we note that computing MCSes upfront appears to be much more efficient than computing them during grow. Considering the 3-UMIVC-UP+SCB configuration, a total of 32,048 MCSes are computed upfront, compared to 372 computed during grow. Upfront computation takes a grand total of 11630 seconds (as indicated by the MCS column), whereas grow consumes a total of 3193 seconds. Ultimately, upfront computation computes 86× as many MCSes in only 3.6× as much time. Adding clauses that reflect the circuit structure (Section V-E) offers a modest improvement to overall performance with no effect on anytime behavior. Caching invariants and counter-examples offers a modest improvement in the bottom-up variant, but significantly improves anytime behavior in the other variants. Interestingly, our implementation only caches the single most-recently discovered safe inductive invariant. Despite that, the cache is very often able to witness safety, as evidenced by the significant reduction in calls to IC3. Considering 0-UMIVC-DOWN+C, across all benchmarks where the cache is checked at least once, the median "hit rate" (*i.e.*, fraction of safe queries where the cached invariant witnesses safety) is 93%. Evidently, safe inductive invariants often witness safety for many different abstractions. It can also be seen from the 0-UMIVC-DOWN+CB variant that biased safety checking offers significant additional performance gains.

TABLE III
SUMMARY OF RESULTS (64 HWMCC 2017 CIRCUITS)

Algorithm	MC	MCS	One MIVC	All MIVC	SMIVC	# MIVC
3-UMIVC-UP+SCB	64 481	55 12265	49 16869	32 30075	49 16869	6907
2-UMIVC-DOWN+SCB	64 481	63 3739	61 7100	37 26887	37 26874	1866
3-UMIVC-ZZ+SCB	64 481	55 12260	54 15298	33 30294	46 20457	2434
∞-UMIVC-UP+C	64 481	32 30819	30 31462	30 31472	30 31462	107

Based on these results, the configurations that are expected to perform best are also evaluated against the HWMCC 2017 benchmark set. Of 181 safe circuits in that set, there are 117 for which no algorithm is able to find a single MIVC. This is expected, as the HWMCC 2017 circuits are intended to be challenging for safety checking and MIVC extraction appears to be significantly harder. Those circuits are discarded, leaving 64 circuits under consideration. Table III shows the results of executing several configurations that are expected to perform well against this benchmark set. The first column shows the name of the variant. In order to compare the runtime of MIVC enumeration to that of safety checking, the second column shows the number of instances for which safety checking was completed successfully and the time taken to do so. The remaining columns report similar data to the corresponding columns of Table II. It can be seen that computing all MIVCs is significantly more expensive than safety checking, as expected. The observed trade-offs are similar to those observed in the HWMCC 2011 set. Interestingly, circuits in the HWMCC 2017 set appear to have significantly more MIVCs. The greatest number of MIVCs found for any HWMCC 2011 circuit is 110. Conversely, there are 8 HWMCC 2017 circuits for which at least that many MIVCs were found, and 3 circuits for which over 1000 distinct MIVCs were found.

VII. CONCLUSION

This paper presents algorithms to enumerate MIVCs of hardware safety checking instances, drawing inspiration from well-known MUS enumeration algorithms. It lifts CAMUS and MARCO and presents an algorithm called UMIVC that generalizes both algorithms while preserving the advantages of each. Several variants of the algorithm are introduced and a set of novel performance optimizations are presented. Experiments are presented on standard benchmark circuits demonstrating the applicability of the presented approaches and the trade-offs between them. MIVCs have already found application in coverage analysis, and are expected to have applications in areas such as debugging, vacuity detection, and others.

REFERENCES

[1] Bailey, J., Stuckey, P.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. Practical Aspects of Declarative Languages pp. 174–186 (2005)

[2] Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in actl formulas. In: International Conference on Computer Aided Verification. pp. 279–290. Springer (1997)

[3] Berryhill, R., Veneris, A.G.: Methodologies for diagnosis of unreachable states via property directed reachability. IEEE Trans. on CAD of Integrated Circuits and Systems **37**(6), 1298–1311 (2018)

[4] Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: International Conference on Formal Methods in Computer-Aided Design. pp. 9–9 (2017)

[5] Bradley, A.: Sat-based model checking without unrolling. In: International Conf. on Verification, Model Checking, and Abstract Interpretation. VMCAI'11 (2011)

[6] Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: International Conference on Formal Methods in Computer-Aided Design. pp. 135–143. FMCAD '11, Austin, TX (2011)

[7] Eén, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design. FMCAD '11 (2011)

[8] Ghassabani, E., Whalen, M., Gacek, A.: Efficient generation of all minimal inductive validity cores. In: 2017 Formal Methods in Computer Aided Design (FMCAD) (2017)

[9] Ghassabani, E., Gacek, A., Whalen, M.W.: Efficient generation of inductive validity cores for safety properties. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 314–325. ACM (2016)

[10] Ghassabani, E., Gacek, A., Whalen, M.W., Heimdahl, M.P., Wagner, L.: Proof-based coverage metrics for formal verification. In: Automated Software Engineering (ASE), IEEE/ACM International Conference on. pp. 194–199. IEEE (2017)

[11] Ignatiev, A., Previti, A., Liffiton, M.H., Marques-Silva, J.: Smallest MUS extraction with minimal hitting set dualization. In: Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings. pp. 173–182 (2015)

[12] Keng, B., Qin, E., Veneris, A., Le, B.: Automated debugging of missing assumptions. In: Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific. pp. 732–737. IEEE (2014)

[13] Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (Apr 2016)

[14] Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning **40**(1), 1–33 (Jan 2008)

[15] Marques-Sila, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Advanced Techniques in Logic Synthesis, Optimizations and Applications, pp. 171–182. Springer (2011)

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

[17] Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Twenty-Third International Joint Conference on Artificial Intelligence (2013)

[18] Nadel, A.: Boosting minimal unsatisfiable core extraction. In: International Conference on Formal Methods in Computer-Aided Design. pp. 221–229. FMCAD '10 (2010)

[19] Simmonds, J., Davies, J., Gurfinkel, A., Chechik, M.: Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC. In: FMCAD. pp. 3–12. IEEE Computer Society (2007)

[20] Smith, A., Veneris, A., Ali, M.F., Viglas, A.: Fault diagnosis and logic debugging using boolean satisfiability. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst **24**(10), 1606–1621 (Oct 2005)

[21] Suelflow, A., Fey, G., Bloem, R., Drechsler, R.: Using unsatisfiable cores to debug multiple design errors. In: ACM Great Lakes Symposium on VLSI. pp. 77–82. GLSVLSI '08 (2008)

[22] Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: Design, Automation and Test in Europe Conference and Exhibition, 2003. pp. 880–885. IEEE (2003)