Synthesizing Safe Bit-Precise Invariants*

Arie Gurfinkel, Anton Belov and Joao Marques-Silva

Abstract. Bit-precise software verification is an important and difficult problem. While there has been an amazing progress in SAT solving, Satisfiability Modulo Theory of Bit Vectors, and bit-precise Bounded Model Checking, proving bit-precise safety, i.e. synthesizing a safe inductive invariant, remains a challenge. Although the problem is decidable and is reducible to propositional safety by bit-blasting, the approach does not scale in practice. The alternative approach of lifting propositional algorithms to bit-vectors is difficult. In this paper, we propose a novel technique that uses unsound approximations (i.e., neither over- nor under-) for synthesizing sound bit-precise invariants. We prototyped the technique using Z3/PDR engine and applied it to bit-precise verification of benchmarks from SVCOMP'13. Even with our preliminary implementation we were able to demonstrate significant (orders of magnitude) performance improvements with respect to bit-precise verificaton using Z3/PDR directy.

1 Introduction

The problem of program safety (or reachability) verification is to decide whether a given program can violate an assertion (i.e., can reach a bad state). The problem is reducible to finding either a finite counter-example, or a safe inductive invariant that certifies unreachability of a bad state. The problem of bit-precise program safety, Safety(BV), further requires that the program operations are represented soundly relative to low-level bit representation of data. Arguably, verification techniques that are not bit-precise are unsound, and do not reflect the actual behavior of a program. Unlike many other problems in software verification, bit-precise verification (without memory allocation and concurrency) is decidable. However, in practice it appears to be more challenging that verification of programs relative to integers or rationals (both undecidable).

The recent decade has seen an amazing progress in SAT solvers, in Satisfiability Modulo Theory of Bit-Vectors, SMT(BV), and in Bounded Model Checkers (BMC) based on these techniques. A SAT solver decides whether a given propositional formula is satisfiable. Current solvers can handle very large problems and are routinely used in many industrial applications (including Hardware and Software verification). SMT(BV) extends SAT-solver techniques to the theory of

* 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-0000869

bit-vectors – that is propositional formulas whose atoms are predicates about bit-vectors. Most successful SMT(BV) solvers (e.g., Boolector [6], STP [17], Z3 [12], MathSAT [9]) are based on reducing the problem to SAT via *pre-processing* and *bit-blasting*. The bit-blasting step takes a BV formula φ and constructs an equivalent propositional formula ψ, where each propositional variable of ψ corresponds to a bit of some bit-vector variable of φ. The, more important, pre-processing step typically consists of equisatisfiable reductions that reduce the size of the input formula. While the pre-processor is not as powerful as the SAT-solver (typically pre-processor is required to run in polynomial time), it does not maintain equivalence. The pre-processing phase of SMT(BV) solvers is crucial for their performance. For example, in our experiments with Boolector, the different between straight forward bit-blasting and pre-processing is several order of magnitude.

There has also been a tremendous progress in applying those techniques to program verification. In particular, there are several mature Bounded Model Checkers, including CBMC [10], LLBMC [31], and ESBMC [11], that decide existence of a bounded bit-precise counterexamples of C programs. These tools are based on ultimate reduction of BMC to SAT, either via their own custom bit-blasting and pre-processing steps (e.g., CBMC) or by leveraging SMT(BV) solvers described above (e.g., LLBMC). While BMC tools are great at finding counterexamples (even in industrial applications), *proving* bit-precise safety, i.e., synthesizing a bit-precise invariant, remains a challenge. For example, none of the tools submitted to the Software Verification Competition in 2013 (SVCOMP'13) are both bit-precise and effective at invariant synthesis.

As we described above, Safety(BV) is decidable. In fact, it is reducible to safety problem over propositional logic, Safety(Prop), via the simple bit-blasting mentioned above. Thus, the naive solution is to reduce Safety(BV) to Safety(Prop) and decide it using tools for propositional verification. This, however, does not scale. Our experiments with Z3/PDR (the Model Checker of Z3), show that the approach is ineffective for almost all benchmarks in SVCOMP'13. The main issue is that the reduction of Safety(BV) to Safety(Prop) is incompatible with the pre-processing techniques that make bit-blasting for SMT(BV) so effective.

An alternative approach of lifting effective Model Checking technique from propositional level to BV appears to be difficult, with only a few somewhat successful attempts (e.g., [25, 19]). For example, techniques based on interpolation (e.g., [30, 26, 1]) require world-level interpolation for BV [24, 19] that satisfies additional properties (e.g., sequence and tree properties) [20]. While, techniques based on PDR [21], require novel world-level inductive generalization strategies. Both are difficult problems in themselves.

Thus, instead of lifting existing techniques, we are interested in finding a way to use existing verification engines to improve scalability of the naive bit-blasting-based solution. Our key insight is based on the observation that most program verifiers abstract program arithmetic by integer (or rational) arithmetic. This is unsound in the presence of overflows (see [19] for an example), but the results

are often "almost" correct. More importantly, they are useful to the users. Thus, we are interested in how to reuse such unsound invariants in a sound way.

Our procedure is based on an iterative *guess-and-check* loop. Given a Safety(BV) problem P, we begin by trying to solve P using a Safety(BV) solver. If this takes too long, we abort it, and construct an approximation (neither over- nor under-) P_T of P in another theory T (e.g., Linear Rational Arithmetic), decide the safety of P_T using a solver for Safety(T), and obtain an inductive safe invariant Inv_T. We then *port* Inv_T in a sound way to P, strengthen P with it, and repeat bit-blasting-based verification. In the best case, the ported version of Inv_T is a safe and inductive invariant for P and the process terminates immediately. In the worst case, Inv_T contributes facts that might help the next verification effort.

We make the following contributions. First, we formally define a framework that allows to use unsound invariants soundly in a verification loop. Second, we instantiate the framework for the theories of Bit Vectors and Linear Arithmetic. In particular, we describe an algorithm for computing Maximal Inductive Subformula for SMT(BV) and show how it interacts with the pre-processing step. Third, we have implemented the proposed framework using Z3/PDR for Safety(Prop) and Boolector for SMT(BV) and have evaluated it on the benchmarks from SVCOMP'13. Even with our preliminary implementation, we are able to synthesize safe invariants for most programs.

Related work. The use of over- and under-approximation and relaxation of a problem from one theory into another is common in both SMT-solving and Model Checking. For example, Bryant et al. [7], use over- and under-approximation to decide formulas in SMT(BV). Komuravelli et al. [23] similarly use over- and under-approximations for Software Model Checking. While we do not require our approximations to be sound, we employ similar techniques to lift proof certificates (inductive invariants in our case) are in principle similar.

Computing Maximal Inductive Subformula (MIS) is similar to mining an inductive invariant from a set of possible annotations, as for example in [16, 22]. The key novelty in our approach is in the reduction from MIS problem to a Minimal Unsatisfiable Subformula (MUS) problem that allows the use of efficient MUS extractors for SAT.

The works conceptually closest to ours are in the area of Upgrade Checking [15], Multi-Property Verification [8], and Regression Verification [18, 27, 4]. A common theme in the above approaches is that they attempt to lift a safety invariant from one given program P_1 to another, related but not equivalent, program P_2. The key difference is that we do not assume existence of a proven program P_1, but, instead, synthesize P_1 and its safety proof automatically.

2 Preliminaries

We assume some familiarity with program verification, logic, SMT and SAT.

Safety verification. A transition system P is a tuple $(\mathcal{V}, Init, Tr, Bad)$, where $\mathcal{V}$ is a set of variables, $Init$, Bad, and Tr are formulas (with free variables in $\mathcal{V}$) denoting the initial and the bad states, and the transition relation, respectively.

A transition system P is UNSAFE iff there exists a natural number N such that the following formula is satisfiable:

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

When P is UNSAFE and $s \in Bad$ is the reachable state, the path from $s_0 \in Init$ to $s \in Bad$ is called a counterexample (CEX).

A transition system P is SAFE if and only if there exists a formula Inv, called *a safe invariant*, that satisfies the following conditions:

$$Init(v) \rightarrow Inv(v) \qquad Inv(v) \wedge Tr(v, u) \rightarrow Inv(u) \qquad Inv(v) \rightarrow \neg Bad(v) \tag{2}$$

A formula Inv that satisfies the first two conditions is called an *invariant* of P, while a formula Inv that satisfies the third condition is called *safe*. A *safety* verification problem is to decide whether a transition system P is SAFE or UNSAFE. Thus, a safety verification problem is equivalent to the problem of establishing an existence of a safe invariant. In SAT-based Model Checking, the verification problem is decided by iteratively synthesizing an invariant Inv or finding a CEX.

Minimal Unsatisfiability. A CNF formula F, viewed as a set of clauses, is *minimal unsatisfiable (MU)* if (i) F is unsatisfiable, and (ii) for any clause $C \in F$, $F \setminus \{C\}$ is satisfiable. A CNF formula F' is a *minimal unsatisfiable subformula (MUS)* of a formula F if $F' \subseteq F$ and F' is MU. Motivated by several applications, minimal unsatisfiability and related concepts have been extended to CNF formulas where clauses are partitioned into disjoint sets called *groups*.

Definition 1. *[32] Given an explicitly partitioned unsatisfiable CNF formula $G = G_0 \cup G_1 \cup \cdots \cup G_k$ (a* group-MUS instance *or a* group-CNF formula*), where G_i's are pair-wise disjoint sets of clauses called* groups*, a* group-MUS *of G is a subset $\mathcal{G}$ of $\{G_1, \ldots, G_k\}$ such that (i) $G_0 \cup \bigcup \mathcal{G}$ is unsatisfiable, and (ii) for any group $G \in \mathcal{G}$, $G_0 \cup \bigcup (\mathcal{G} \setminus \{G\})$ is satisfiable.*

Notice that group-0, G_0, plays the special role of a "background" subformula, with respect to which the set of groups $\{G_1, \ldots, G_k\}$ is minimized. In particular, if G_0 is unsatisfiable, the group-MUS of G is $\emptyset$.

3 Synthesizing Safe Bit-Precise Invariants

3.1 High-level description of the approach

Given a transition system $P = (\mathcal{V}, Init, Tr, Bad)$, let the *target* theory $\mathcal{T}_T$ be the theory[1], or a combination of theories, that define the formulas in P. Let $\mathcal{T}_W$ be another theory, referred to as a *working* theory, with the intention that

[1] The term "theory" is used as in the context of Satisfiability Modulo Theories.

reasoning in $\mathcal{T}_W$ is easier in practice than reasoning in $\mathcal{T}_T$. Our approach relies on a mapping $M_{T\to W}$ that translates formulas over $\mathcal{T}_T$ to formulas over $\mathcal{T}_W$. Although the correctness of the approach is not affected by the choice of $M_{T\to W}$, its effectiveness is. We would like to map between formulas that are somewhat close to each other semantically. Thus, we assume that $M_{T\to W}$ maps the *terms* and the *atomic formulas* of $\mathcal{T}_T$ to those of $\mathcal{T}_W$ and is an identity mapping for the symbols shared between the two theories. The mapping is extended to all formulas of $\mathcal{T}_T$ by structural induction, i.e., given a formula $F(v)$ over $\mathcal{T}_T$, the corresponding formula $F_W(v)$ over $\mathcal{T}_W$ is constructed by inductively applying $M_{T\to W}$ on the structure of $F(v)$. Similarly, to translate formulas from $\mathcal{T}_W$ to $\mathcal{T}_T$, we work with a mapping $M_{W\to T}$ from the terms and the atomic formulas of the working theory $\mathcal{T}_W$ to those of $\mathcal{T}_T$, extended to all formulas of $\mathcal{T}_W$.

Example 1. Let $\mathcal{T}_T = \mathrm{BV}^*(32)$ — a sub-theory of the quantifier-free fragment of the first-order theory of 32 bit bit-vector arithmetic (cf., [7]) obtained by removing all the non-arithmetic functions and predicates, as well as the multiplication and the division on bit-vectors. Let $\mathcal{T}_W = \mathrm{LA}$ — the quantifier-free fragment of the first order-theory of linear arithmetic, together with the propositional logic. The mapping $M_{T\to W}$ is defined as follows: (i) the propositional fragment of $\mathcal{T}_T$ maps to the propositional fragment of $\mathcal{T}_W$ as is; (ii) bit-vector variables map to LA variables; (iii) the arithmetic functions and predicates of BV(32) map to their natural counterparts in LA, e.g., $+^{[32]}$ to $+$, $<^{[32]}$ to $<$, etc. Then, if

$$\mathit{Init}(x^{[32]}, y^{[32]}, z) = (x^{[32]} +^{[32]} y^{[32]} >^{[32]} 0^{[32]}) \wedge z,$$

where $x^{[32]}$ and $y^{[32]}$ are bit-vector and z propositional variables, the corresponding LA formula $\mathit{Init}_W(x, y, z)$ is

$$\mathit{Init}(x, y, z) = (x + y > 0) \wedge z.$$

The inverse mapping $M_{W\to T}$ from LA to BV(32) is constructed in a similar manner, with the slight complication related to LA constants, which might be non-integer, too large to fit into the required bit-width, or negative. One possibility to deal with non-integer constants is to truncate the fractional digits, i.e., map 0.5 to $0^{[32]}$. Other options include rounding up the constants when possible, e.g., by translating $(x > 0.5)$ to $(x^{[32]} \geq^{[32]} 1^{[32]})$, but $(x < 0.5)$ to $(x^{[32]} \leq^{[32]} 0^{[32]})$. For this paper, we adopt the former, simpler, approach, and leave the investigation of more sophisticated translations to future work. To convert an integer LA constant to BV(32) we take the lower 32 bit of its 2s-complement representation.

Remark 1. Clearly, our sub-theory $\mathrm{BV}^*(32)$ of the full theory BV(32) was chosen to simplify the construction of the mapping to and from LA. Generally, such restriction of the original target theory might not be necessary if the working theory $\mathcal{T}_W$ supports uninterpreted functions.

The pseudocode in Algorithm 1 provides the high-level description of our verification framework (MISper). Given a transition system $P = (\mathcal{V}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$

Algorithm 1: MISper — safety verification framework

Input : $P = (\mathcal{V}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$ — a transition system over theory $\mathcal{T}_T$
Output: $\mathsf{st} \in \{\mathsf{SAFE}, \mathsf{UNSAFE}, \mathsf{UNKNOWN}\}$

1 **forever do**
2 **under resource limits do**
3 $(\mathsf{st}, \mathit{Inv}, \mathit{Cex}) \leftarrow \mathtt{Safety}(\mathcal{T}_T)(P)$ `// solve in the target theory`
4 **if** $\mathsf{st} \neq \mathsf{UNKNOWN}$ **then return** st
5 $(\mathcal{T}_W, M_{T\to W}, M_{W\to T}) \leftarrow$ pick a working theory and mappings
6 $P_W \leftarrow M_{T\to W}(P)$ `// translate` P `to the working theory`
7 $(\mathsf{st}, \mathit{Inv}_W, \mathit{Cex}_W) \leftarrow \mathtt{Safety}(\mathcal{T}_W)(P_W)$
8 **if** $\mathsf{st} \neq \mathsf{SAFE}$ **then**
9 **return** UNKNOWN `// options: deal with CEX; try another` $\mathcal{T}_W$
10 $\mathit{Cand} \leftarrow M_{W\to T}(\mathit{Inv}_W)$ `// get the candidate invariant for` P
11 **if** *Cand is safe invariant for* P **then**
12 **return** SAFE
13 $\mathit{Cand}_I \leftarrow \mathtt{ComputeMIS}(\mathit{Cand})$
14 $\mathit{Tr}(u,v) \leftarrow \mathit{Cand}_I(u) \wedge \mathit{Tr}(u,v) \wedge \mathit{Cand}_I(v)$ `// strengthen tr. rel.`

over the target theory $\mathcal{T}_T$ (e.g., BV*(32) from Example 1), we first attempt to solve P with a solver for Safety($\mathcal{T}_T$) under heuristically chosen resource limits[2]. If the solver fails to prove or disprove the safety of P, we pick a working theory $\mathcal{T}_W$, and a pair of corresponding mappings $M_{T\to W}$ and $M_{W\to T}$ (e.g., $\mathcal{T}_W$ = LA and the mappings are as in Example 1). Then, we attempt to verify the safety of $P_W = M_{T\to W}(P) = (\mathcal{U}, M_{T\to W}(\mathit{Init}), M_{T\to W}(\mathit{Tr}), M_{T\to W}(\mathit{Bad}))$, where $\mathcal{U}$ are the fresh variables introduced by $M_{T\to W}$, using a solver for Safety($\mathcal{T}_W$). Since P_W is in general neither under- nor over- approximation of P, the (un)safety of the former does not imply the (un)safety of the latter. Since the focus of this paper is on synthesis of invariants for verification, we omit the detailed discussion of how to handle the UNSAFE status of P_W. One option is to simply return UNKNOWN, as in Algorithm 1. Alternatively, the CEX for P_W can be mapped to $\mathcal{T}_T$ via $M_{W\to T}$ and checked on P — if the mapped CEX is also a CEX for P, return UNSAFE. Otherwise, the mapping can be refined to eliminate the CEX, and the safety verification of P_W under the new mapping repeated. If, on the other hand, P_W *is* safe, we take the safe invariant Inv_W of P_W, and translate it back to the target theory $\mathcal{T}_T$ to obtain a *candidate-invariant* formula $\mathit{Cand} = M_{W\to T}(\mathit{Inv}_W)$. If Cand is a safe invariant of P, then the safety of P is established, and the algorithm returns SAFE. Otherwise, we attempt to compute a subformula Cand_I of Cand that is an invariant of P — this is done in the function `ComputeMIS` on line 13 of Algorithm 1, which we describe in detail in Section 3.2. Once an invariant of P is obtained, we restrict the transition relation of P by replacing the formula $\mathit{Tr}(u,v)$ in P with the formula $\mathit{Cand}_I(u) \wedge \mathit{Tr}(u,v) \wedge \mathit{Cand}_I(v)$, and attempt to verify the safety of the new

[2] This step is optional on the first iteration of the main loop of Algorithm 1.

transition system (the next iteration of the main loop). Since $Cand_I$ is the actual invariant of P, the (un)safety of strengthened transition system implies the (un)safety of the input system P.

This verification framework can be instantiated in numerous ways and leaves a number of open heuristic choices. We postpone the description of an instantiation of the framework used in our experiments to Section 4.

3.2 Computing invariants

Given a candidate invariant $Cand$ for a transition system $P = (\mathcal{V}, Init, Tr, Bad)$, obtained as described in Section 3.1, we are interested in computing a subformula $Cand_I$ of $Cand$ that is an invariant with respect to P, that is, $Cand_I(u) \wedge Tr(u, v) \models Cand_I(v)$. Similarly to the previous work on invariant extraction (e.g., [8, 23]), we proceed under the assumption that the candidate invariant $Cand(u)$ is given as a *conjunction* of formulas $Cand(u) = L_1(u) \wedge \cdots \wedge L_n(u)$. We refer to the conjuncts L_i of $Cand$ as *lemmas*. Then, the invariant $Cand_I$ can be always be constructed as a (possibly empty) conjunction of some of the lemmas in $Cand$. In our setting, this assumption is justified by the fact that many verification tools, particularly those based on PDR [5, 13] and its extensions (e.g., Z3/PDR [21]) do indeed produce invariants in this form. In the worst case, $Cand$ itself can be treated as the (only) conjunct, which, while affecting the effectiveness of our approach, does not affect its correctness. We note that the ideas discussed in this section can be extended to candidate invariants of arbitrary structure, though such extension is outside of the scope of this paper.

For notational convenience we treat $Cand$ as a *set* of lemmas $\{L_1, \ldots, L_n\}$, and formalize the invariant computation problem as follows:

Definition 2. *Given a set of lemmas $\mathcal{L} = \{L_1, \ldots, L_n\}$ and a transition relation $Tr(u, v)$, a subset $\mathcal{L}' \subseteq \mathcal{L}$ is* inductive *if $(\bigwedge_{L \in \mathcal{L}'} L(u)) \wedge Tr(u, v) \models \bigwedge_{L \in \mathcal{L}'} L(v)$. An inductive subset $\mathcal{L}' \subseteq \mathcal{L}$ is* maximal *if no strict superset of $\mathcal{L}'$ is inductive. Finally, an inductive subset $\mathcal{L}' \subseteq \mathcal{L}$ is* maximum *if the cardinality of $\mathcal{L}'$ is maximum among all inductive subsets of $\mathcal{L}$.*

It is not difficult to see that a union of two inductive subsets is inductive, and so any set of lemmas $\mathcal{L}$ has a *unique* maximal, and hence a *unique* maximum, inductive subset $\mathcal{L}'$. We refer to $\mathcal{L}'$ as the *MIS* (maximal/maximum inductive subset) of $\mathcal{L}$. Thus, in our framework, given a candidate invariant $Cand$ of transition system P, the actual invariant $Cand_I$ of P is obtained by computing the MIS of $Cand$ — this is motivated by the fact that we aim to strengthen the transition relation as much as possible prior to the next iteration of the algorithm.

Approaches to MIS computation. The existing approaches to computation of MISes can be categorized into *eager* and *lazy*. Given a set of lemmas $\mathcal{L} = \{L_1, \ldots, L_n\}$ and the transition relation Tr, the eager approach (taken, for example, in [8]) starts by checking whether $\mathcal{L}(u) \wedge Tr(u, v) \models \mathcal{L}(v)$. This is typically done by testing the unsatisfiability of the formula $\mathcal{L}(u) \wedge Tr(u, v) \wedge \neg\mathcal{L}(v)$ with an SMT (or a SAT) solver. If the formula satisfiable, i.e., $\mathcal{L}$ is not inductive,

the model returned by the solver must falsify one or more lemmas in $\mathcal{L}(v)$. These lemmas are then removed *both* from $\mathcal{L}(u)$ and from $\mathcal{L}(v)$, and the test is repeated. The process continues until for some subset $\mathcal{L}' \subseteq \mathcal{L}$, $\mathcal{L}'(u) \wedge Tr(u,v) \models \mathcal{L}'(v)$. The final subset $\mathcal{L}'$ is obviously inductive. Furthermore, for any set of lemmas $\mathcal{L}'' \subseteq \mathcal{L} \setminus \mathcal{L}'$ there must have been a point in the execution of the algorithm where it obtained a model for a formula $\mathcal{L}'(u) \wedge \mathcal{L}''(u) \wedge Tr(u,v)$ that falsifies at least one lemma in $\mathcal{L}''(v)$, as otherwise this lemma would be included in $\mathcal{L}'$. Hence, $\mathcal{L}'$ is maximal, and therefore is a MIS of $\mathcal{L}$.

In the lazy approach to MIS computation (e.g., [16, 23]), when the set $\mathcal{L}$ is not inductive, the lemmas in the consequent $\mathcal{L}(v)$ that are falsified by the model of $\mathcal{L}(u) \wedge Tr(u,v)$ are initially removed *only* from $\mathcal{L}(v)$. The process continues until for some $\mathcal{L}' \subseteq \mathcal{L}$, $\mathcal{L}(u) \wedge Tr(u,v) \models \mathcal{L}'(v)$ — notice that the premise still contains all of the lemmas of $\mathcal{L}$. We refer to such sets $\mathcal{L}'$ as *semi-inductive* with respect to $\mathcal{L}$ and Tr. Observe that the semi-inductive subset $\mathcal{L}'$ obtained in this manner is *maximal* and also *maximum*, by the argument analogous to that used to establish the uniqueness of MISes. Once the maximum semi-inductive subset $\mathcal{L}'$ of $\mathcal{L}$ is computed, the lemmas excluded from $\mathcal{L}'$ are removed from $\mathcal{L}(u)$, and the algorithm checks whether $\mathcal{L}'(u) \wedge Tr(u,v) \models \mathcal{L}'(v)$, i.e., whether $\mathcal{L}'$ is inductive. If not, the algorithm repeats the process, by first computing a maximum semi-inductive subset of $\mathcal{L}'$, then checking its inductiveness, and so on. The, eventually obtained, inductive subset of $\mathcal{L}$ is the MIS of $\mathcal{L}$ — this can be justified in essentially the same way as for the eager approach.

One potential advantage of the lazy approach is that, since, compared to the eager approach, there are often more lemmas in the premises, the SMT/SAT solver is likely to work with stronger formulas. Furthermore, if a solver retains information between invocations — for example, derived facts and history-based heuristic parameters, as in *incremental* SAT solvers — more information can be reused between iterations, thus speeding-up the MIS computation.

One additional feature of the lazy approach, pointed out and used in [23], is that the computation of semi-inductive subsets can be reduced to the computation of Minimal Unsatisfiable Subformulas (MUSes), or, more precisely, to the computation of group-MUSes (recall Definition 1). This observation is particularly useful in cases when satisfiability problem in the theory that defines the invariants can be soundly reduced to propositional satisfiability, as it allows to leverage the large body of recent work and tools for the computation of MUSes (e.g., [2, 29, 33]). We take advantage of this observation in the implementation of our framework since, in our case, the invariants are quantifier-free formula over (a sub-theory of) the theory of bit-vectors, and the satisfiability of such formulas can be soundly reduced to SAT via bit-blasting. The reduction to group-MUS computation and the overall MIS extraction flow are presented below.

Computing MISes with group-MUSes. For a set of lemmas $\mathcal{L} = \{L_1, \ldots, L_n\}$ and a transition relation formula Tr, we first rewrite the formula $\mathcal{L}(u) \wedge Tr(u,v) \wedge \neg\mathcal{L}(v)$, used to check the inductiveness of $\mathcal{L}$, as a formula $A_{\mathcal{L},Tr}$ defined in the

following way:

$$A_{\mathcal{L},Tr} = \left(\bigwedge_{L_i \in \mathcal{L}} (pre_i \rightarrow L_i(u)) \right) \wedge Tr(u,v) \wedge \left(\bigvee_{L_i \in \mathcal{L}} (post_i \wedge \neg L_i(v)) \right), \quad (3)$$

where pre_i and $post_i$ for $i \in [1,n]$ are fresh propositional variables, one for each lemma $L_i \in \mathcal{L}$. One of the purposes of these variables is similar to that of the indicator variables used in assumption-based incremental SAT solving (cf. [14]) — the variables can be used to emulate the removal of lemmas from formulas $\mathcal{L}(u)$ and $\mathcal{L}(v)$. Setting pre_i to true (resp. false) causes the lemma L_i to be included (resp. excluded) from $\mathcal{L}(u)$, while setting $post_i$ to true (resp. false) has the same effect on the lemma L_i in $\mathcal{L}(v)$. The names of the variables reflect the fact that they control either the "*pre*condition" or the "*post*condition" lemmas. With this in mind, a computation of the MIS of $\mathcal{L}$ with respect to Tr can be implemented on top of an incremental SMT solver by loading the formula $A_{\mathcal{L},Tr}$ into the solver, and checking the satisfiability of the formula under a set of assumptions. For example, the set $\mathcal{L}$ is inductive if and only if the formula is unsatisfiable under assumptions $\bigcup_{i\in[1,n]}\{pre_i, post_i\}$. When a lemma $L_i \in \mathcal{L}$ needs to be removed from $\mathcal{L}(u)$ and/or $\mathcal{L}(v)$, we simply assert the formula $(\neg pre_i)$ and/or $(\neg post_i)$ to the solver.

However, as explained above, our intention is to take advantage of propositional MUS extractors, using the fact that quantifier-free bit-vector formulas can be soundly converted to propositional logic. The *pre* and *post* variables serve a purpose in this context as well. Assume that we have a polytime computable function *B2P*, which given a quantifier-free formula F_{BV} over the theory BV, and a set of *propositional* variables $X = \{x_1, \ldots, x_k\}$ that occur in F_{BV} returns a propositional formula $F_{Prop} = B2P(F_{BV}, X)$, in CNF, with the following property: for any assignment τ to the variables in X, the formula $F_{BV}[\tau]$ is satisfiable if and only if so is the formula $F_{Prop}[\tau]$. Following [28], we say that the formulas F_{BV} and F_{Prop} are *var-equivalent* on X in this case. Note that var-equivalence of F_{BV} and F_{Prop} on X does not imply F_{Prop} contains all variables of X — for example, $F_{Prop} = \top$ is var-equivalent to F_{BV} if $F_{BV}[\tau]$ is satisfiable for every assignment τ for X.

Now, for a set of lemmas $\mathcal{L} = \{L_1, \ldots, L_n\}$ and a transition relation Tr over BV, let $A_{\mathcal{L},Tr}$ be the formula defined in (3), let $Pre = \{pre_i \mid i \in [1,n]\}$, $Post = \{post_i \mid i \in [1,n]\}$. Consider the group-CNF formula $G_{\mathcal{L},Tr}$ constructed in the following way:

$$\begin{aligned} &G_{\mathcal{L},Tr} = G_0 \cup G_1 \cup \cdots \cup G_n, \text{ where:} \\ &\quad G_0 = C_{\mathcal{L},Tr} \cup \{(pre_i) \mid i \in [1,n]\}, \text{ with } C_{\mathcal{L},Tr} = B2P(A_{\mathcal{L},Tr}, Pre \cup Post) \\ &\quad G_i = \{(\neg post_i)\} \text{ for } i \in [1,n] \end{aligned}$$

That is, the group G_0 of $G_{\mathcal{L},Tr}$ is the formula $C_{\mathcal{L},Tr}$ — a CNF formula var-equivalent to $A_{\mathcal{L},Tr}$ on the set $Pre \cup Post$ — together with the positive unit clauses for *pre* variables. Each group G_i in $G_{\mathcal{L},Tr}$ consists of a single negative unit clause for the variable $post_i$.

Proposition 1. *Let $\mathcal{G}$ be a group-MUS of the group-CNF formula $G_{\mathcal{L},Tr}$. Then, the set of lemmas $\mathcal{L}' = \{L_k \mid k \in [1,n] \text{ and } G_k \notin \mathcal{G}\}$ is the maximum semi-inductive subset of $\mathcal{L}$ with respect to Tr. Furthermore, $\mathcal{G} = \emptyset$ iff $\mathcal{L}$ is inductive.*

Intuitively, Proposition 1 follows from the fact that the function *B2P* preserves var-equivalence. The formulas $A_{\mathcal{L},Tr}$ and $C_{\mathcal{L},Tr}$ are var-equivalent on the variables $Pre \cup Post$. Thus, any group-MUS $\mathcal{G}$ of the group-CNF formula $G_{\mathcal{L},Tr}$ is exactly a group-MUS of the "group-BV" formula obtained by taking $A_{\mathcal{L},Tr}$ together with the appropriate unit clauses as group-0 and the rest of groups as in $G_{\mathcal{L},Tr}$. Furthermore, whenever a group G_i is included in $\mathcal{G}$, the corresponding variable $post_i$ is forced to 0, and so the lemma $L_i(v)$ is disabled in $A_{\mathcal{L},Tr}$. Since $G_0 \cup \bigcup \mathcal{G}$ is unsatisfiable, so is the formula $A_{\mathcal{L},Tr}$ with the rest of the post-lemmas (i.e., the set $\mathcal{L}'$) enabled, thus implying the semi-inductiveness of $\mathcal{L}'$. The maximality of the latter is implied by the minimality of $\mathcal{G}$.

Proof. First, observe that the formula $G_{\mathcal{L},Tr}$ is unsatisfiable. This is because $G_{\mathcal{L},Tr} \equiv C_{\mathcal{L},Tr}[\tau]$, where $\tau = \{pre_i \rightarrow 1, post_i \rightarrow 0 \mid i \in [1,n]\}$ is the assignment entailed by the unit clauses in $G_{\mathcal{L},Tr}$. Since *B2P* preserves var-equivalence on $Pre \cup Post$, the formula $C_{\mathcal{L},Tr}[\tau]$ is equisatisfiable with the formula $A_{\mathcal{L},Tr}[\tau]$ (cf. (3)), which, in turn, is unsatisfiable since τ sets all *post* variables to 0.

Let now $\mathcal{G}$ be a group-MUS of $G_{\mathcal{L},Tr}$. Since $G_0 \cup \bigcup \mathcal{G}$ is unsatisfiable (recall Definition 1), so is the formula $C_{\mathcal{L},Tr}[\tau_{\mathcal{G}}]$, where $\tau_{\mathcal{G}} = \{pre_i \rightarrow 1 \mid i \in [1,n]\} \cup \{post_j \rightarrow 1 \mid G_j \notin \mathcal{G}\} \cup \{post_k \rightarrow 0 \mid G_k \in \mathcal{G}\}$, and, therefore, the formula $A_{\mathcal{L},Tr}[\tau_{\mathcal{G}}]$. Note, however, that the latter is equivalent to $\mathcal{L}(u) \wedge Tr(u,v) \wedge \neg\mathcal{L}'(v)$, where $\mathcal{L}'$ is as defined in the statement of the proposition. Hence, $\mathcal{L}'$ is semi-inductive.

Finally, w.l.o.g. take any $\mathcal{G}' \subset \mathcal{G}$. Since $\mathcal{G}$ is a group-MUS of $G_{\mathcal{L},Tr}$, the formula $G_0 \cup \bigcup \mathcal{G}'$ is satisfiable. Following the previous argument with the assignment $\tau_{\mathcal{G}'}$ we have that the formula $A_{\mathcal{L},Tr}[\tau_{\mathcal{G}'}]$ is satisfiable, and so is the formula $\mathcal{L}(u) \wedge Tr(u,v) \wedge \neg\mathcal{L}''(v)$, where $\mathcal{L}'' = \mathcal{L} \cup \{L_k \mid G_k \in \mathcal{G} \setminus \mathcal{G}'\}$. We conclude that any $\mathcal{L}'' \supset \mathcal{L}'$ is not semi-inductive, and so $\mathcal{L}'$ is maximal.

The "only-if" part of the second claim of the proposition follows immediately from the first claim. For the "if" part, assume that $\mathcal{L}$ is inductive, and let τ be the assignment that enables all lemmas of $\mathcal{L}$, i.e., $\tau = \{pre_i \rightarrow 1, post_i \rightarrow 1 \mid i \in [1,n]\}$. Then, the formula $A_{\mathcal{L},Tr}[\tau]$ is unsatisfiable. Since the *post* variables appear in $A_{\mathcal{L},Tr}$ only in positive polarity, changing the value of any of the *post* variables to 0 cannot make the formula satisfiable. Thus, for $\tau' = \{pre_i \rightarrow 1 \mid i \in [1,n]\}$ the formula $A_{\mathcal{L},Tr}[\tau']$ is also unsatisfiable, and since *B2P* preserves var-equivalence, so is the CNF formula $C_{\mathcal{L},Tr}[\tau']$. But, $C_{\mathcal{L},Tr}[\tau'] \equiv G_0$, and so the group-MUS of $G_{\mathcal{L},Tr}$ is $\emptyset$. □

The MIS computation algorithm. Based on Proposition 1, we can compute the maximum semi-inductive subset of the set of lemmas $\mathcal{L}$ by invoking any off-the-shelf group-MUS extractor (e.g., MUSer2 [3]). The *post* variables are essential for this reduction, as the translation function *B2P* can, and in practice does, significantly modify the structure of the input BV formula through the application of various BV-specific preprocessing techniques. The purpose of *pre*

Algorithm 2: `ComputeMIS` for invariants in BV

Input : $(\mathcal{L}, Tr)$ — a set of lemmas and a transition relation, in BV
Output: $\mathcal{L}' \subseteq \mathcal{L}$ — the MIS of $\mathcal{L}$ with respect to Tr

```
1 construct A_{L,Tr}                                  // the BV formula defined in eq. (3)
2 C_{L,Tr} ← B2P(A_{L,Tr}, Pre ∪ Post)                // compute a var-equivalent CNF
3 L' ← L
4 forever do
5     construct G_{L,L',Tr}                           // the group-CNF defined in eq. (4)
6     G ← ComputeGMUS(G_{L,L',Tr})                    // compute a group-MUS
7     if G = ∅ then                                   // L' is inductive, cf. Prop. 1
8         return L'
9     L' = {L_k | k ∈ [1,n] and G_k ∉ G}              // remove lemmas included in G
```

variables is slightly more technical. Assume that in the first iteration of the lazy MIS computation algorithm a maximal semi-inductive set $\mathcal{L}'$ of $\mathcal{L}$ is computed, and that $\mathcal{L}' \subset \mathcal{L}$. At this point, some of the lemmas $\mathcal{L}(u)$ (i.e., the precondition lemmas) have to be removed from $\mathcal{L}$. One possibility is to build a new formula $A_{\mathcal{L}', Tr}$ analogously to that in equation (3), apply the function *B2P* to it, and proceed with the computation of the maximum semi-inductive subset of $\mathcal{L}'$. An alternative is to re-use the CNF formula $C_{\mathcal{L}, Tr}$, obtained by translating the original formula $A_{\mathcal{L}, Tr}$ via *B2P*, and simply add negative unit clauses $(\neg pre_i)$ and $(\neg post_i)$ for each of the lemmas removed from $\mathcal{L}$. This way we avoid re-invoking *B2P*, and open up the possibility of reusing more information between the invocations of the group-MUS extractor[3]. As the group-CNF formula $G_{\mathcal{L}, Tr}$ does need to be modified between iterations by taking into account removal of some of the lemmas, for a set $\mathcal{L}' \subseteq \mathcal{L}$ of *remaining* lemmas we define the group-CNF formula $G_{\mathcal{L},\mathcal{L}', Tr}$ as follows:

$$
\begin{aligned}
G_{\mathcal{L},\mathcal{L}', Tr} &= G_0 \cup \{G_i \mid L_i \in \mathcal{L}'\}, \text{ where:} \\
G_0 &= C_{\mathcal{L}, Tr} \cup \{(pre_i) \mid L_i \in \mathcal{L}'\} \cup \{(\neg pre_j), (\neg post_j) \mid L_j \in \mathcal{L} \setminus \mathcal{L}'\} \quad (4) \\
G_i &= \{(\neg post_i)\} \text{ for } L_i \in \mathcal{L}'.
\end{aligned}
$$

The pseudocode of the MIS computation algorithm based on the ideas presented above is presented in Algorithm 2. Given a set of BV lemmas $\mathcal{L}$ and a transition relation formula Tr, the algorithm constructs the formula $A_{\mathcal{L}, Tr}$, defined in (3), and converts the formula to CNF using a var-equivalence preserving function *B2P*. The set $\mathcal{L}'$ that will eventually represent the resulting MIS is initialized to $\mathcal{L}$. The main loop of the algorithm reflects the outer loop of the lazy MIS computation approach. On every iteration, the maximum semi-inductive subset of $\mathcal{L}'$ is computed via the reduction to group-MUS computation, as justified by Proposition 1. If the group-MUS is empty, then, according to Proposition 1, the set $\mathcal{L}'$ itself is inductive, and, therefore, based on the correctness of

[3] This assumes an *incremental* group-MUS extractor.

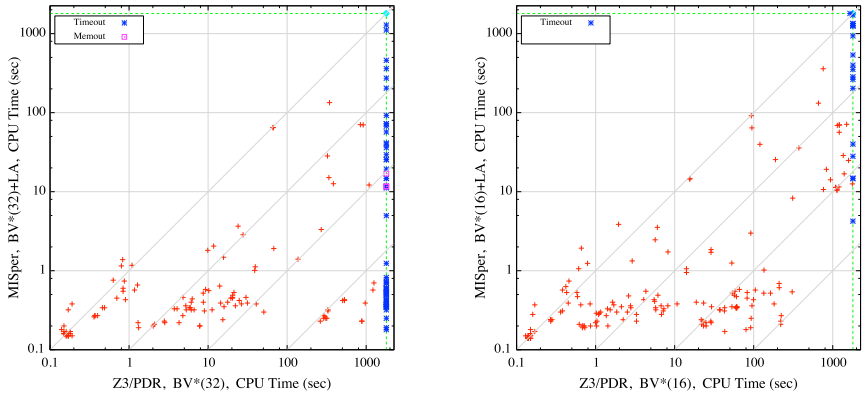

Fig. 1. Performance of Z3/PDR and MISper for the target theories BV*(32) (left) and BV*(16) (right) in terms of CPU runtime. Timeout of 1800 seconds is represented by the dashed (green) lines; orders of magnitude are represented by diagonals.

the lazy MIS computation algorithm, is the MIS of $\mathcal{L}$. Otherwise, $\mathcal{L}'$ is updated to the computed maximum semi-inductive subset represented by the extracted group-MUS (line 9). Note that the removal of the lemmas from the premise formula $\mathcal{L}(u)$ performed at this stage during the lazy MIS computation is implicit in the construction of the group-CNF formula $G_{\mathcal{L},\mathcal{L}',Tr}$ in the next iteration of the main loop (cf. (4)). The termination of the algorithm is guaranteed by the fact that on every iteration at least one lemma is removed from $\mathcal{L}'$, and so, in the worst case, there will be an iteration of the main loop with $\mathcal{L}' = \emptyset$. Since, in this case, $\mathcal{L}'$ is inductive, by Proposition 1 the computed group-MUS will be $\emptyset$, and the algorithm terminates.

4 Implementation and Empirical Evaluation

Our prototype implementation of MISper framework was instantiated with the restriction BV* of the theory of bit-vectors, described in Example 1, as the target theory $\mathcal{T}_T$, and the theory of linear arithmetic LA as working theory $\mathcal{T}_W$. The mappings $M_{T\to W}$ and $M_{W\to T}$ between the theories are as described in Example 1. We used Z3/PDR engine for the implementation of the Safety(BV) and Safety(LA) procedures. The Horn SMT systems used as an input to Z3/PDR were obtained from the UFO framework. To check the safety and the inductiveness of the candidate invariants *Cand* in BV we used the bit-vector engine of Z3. To perform var-equivalent translation of BV formulas to CNF during invariant extraction (function *B2P* in Algorithm 2) we used the front-end of the SMT(BV) solver Boolector [6]. Though we were unable to formally establish the var-equivalence of the translation, we validated the inductiveness of computed invariants independently. Finally, we used the MUS extractor MUSer2 to compute group-MUSes in Algorithm 2.

Experimental setup and results. To evaluate the performance of the proposed framework empirically we selected 214 bit-precise verification benchmarks from the set of SAFE benchmarks used in 2013 Competition on Software Verification, SVCOMP'13[4]. We translated the benchmarks to Horn SMT formulas over the theories $\mathrm{BV}^*(32)$ and $\mathrm{BV}^*(16)$ (recall Example 1), after replacing the unsupported bit-vector operations by fresh variables — hence, the resulting systems are an over-approximation of the original programs[5]. We compared the performance of Z3/PDR engine with that of MISper, instantiated with the theory of linear arithmetic (LA) as a working theory $\mathcal{T}_W$. All experiments were performed on Intel Xeon X3470, 32 GB, running Linux 2.6. For each experiment, we set a CPU time limit of 1800 seconds, and a memory limit of 4 GB.

The scatter plots in Figure 1, complemented by Table 1, summarize the results of our experiments. In 32-bit experiments, MISper solved all 116 instances solved by Z3/PDR, and additional 58 on which Z3/PDR exceeded the allotted resources (174 in total). Furthermore, judging from the scatter plot (left), on the vast majority of instances MISper was at least one order of magnitude faster than Z3/PDR, and, in some cases, the performance improvement exceeded three orders of magnitude. The 16-bit benchmarks were, not surprisingly, easier for Z3/PDR than 32-bit, and so it succeeded to solve quite significantly more problems (165). Nevertheless, MISper significantly outperforms Z3/PDR in this setting as well, solving 17 more benchmarks, and still demonstrating multiple orders of magnitude performance improvements. We found only one instance solved by Z3/PDR, but unsolved by MISper (exceeded time limit). To summarize, the results clearly demonstrate the effectiveness of the proposed framework.

A number of interesting additional observations can be made by analyzing the data in Table 1. Consider the 58 instances unsolved by Z3/PDR and solved by MISper in the 32-bit experiments (second row of Table 1). In 52 of these the safe invariants obtained in LA were transferred to directly to BV. Thus, in many practical cases, while the safety of the program can be easily established without taking into account its bit-precise semantics, the BV-based engine seems to get bogged down by discovering information that, in the end, is mostly irrelevant. In these situations, our approach allows to "find needles in the haystack", and quickly. In the remaining 6 cases, the bit-precise semantics do come into play. However, the MIS-based invariant synthesis allows to transfer information that is useful for bit-precise reasoning — this is evidenced by at least 3x average speed-up of bit-precise reasoning on the strengthened system, with close to 6x speed-up on 3 instances out of 6. The 16-bit experiments confirm the usefulness of the partially transferred invariants further: out of 18 instances unsolved by Z3/PDR, only on 6 the LA invariant could be transferred directly to BV, while on remaining 12 the partial information allowed to speed-up the verification by at least 2x on average.

[4] `http://sv-comp.sosy-lab.org/2013`.
[5] The benchmarks are available at `http://bitbucket.org/arieg/misp`.

Table 1. Performance of Z3/PDR and MISper for the target theories BV*(32) and BV*(16). Within each horizontal section, the first row (all) presents the data for all 214 instances, while the second row (unsol.) presents the data for those instances that were not solved by Z3/PDR. "Solved" means that the tool returned SAFE within the timeout/memout of 1800 sec/4 GB. Column Z3/PDR shows the data for Z3/PDR — each cell contains the number of solved instances (#sol), followed by the average and the median of the CPU times on the solved instances (avg/med). Column MISper displays the same data for MISper. Column MISper:*Cand* displays the data for instances solved by MISper by proving the safety of the candidate invariant *Cand* (Alg. 1, line 12). Column MISper:MIS displays the data for instances solved by MISper by computing MIS of *Cand*, and invoking Z3/PDR on strengthened system (Alg. 1, lines 13-14). For example, the first row in the table shows that out of 214 instances, Z3/PDR solved 116, while MISper solved 174, out of which 165 were solved immediately after the conversion of LA invariant to BV*(32), and 9 were solved after extracting invariants.

bit width	inst.	count	Z3/PDR #sol(avg/med)	MISper #sol(avg/med)	MISper:*Cand* #sol(avg/med)	MISper:MIS #sol(avg/med)
32	all	214	116(127.54/8.27)	**174**(28.34/0.43)	165(8.50/0.42)	9(391.95/133.94)
	unsol.	98	—	58(75.90/1.03)	52(21.89/0.70)	6(544.05/366.18)
16	all	214	165(176.69/8.20)	**182**(69.32/0.38)	165(8.37/0.36)	17(660.91/399.32)
	unsol.	49	—	18(624.79/376.24)	6(50.80/21.45)	12(911.78/1094.58)

5 Conclusion

In this paper, we introduced a bit-precise program verification framework MISper. The key idea behind the framework is to transfer, at least partially, information obtained during the verification of an unsound approximation of the original program in the form of bit-precise invariants. We describe a novel approach to computing such invariants that allows to take advantage of the state-of-the-art propositional MUS extractors. The results of the experiments with our prototype implementation of the framework suggest that the proposed approach is promising.

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. In *TACAS*, 2012.
2. A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, 25(2), 2012.
3. A. Belov and J. Marques-Silva. MUSer2: An Efficient MUS Extractor. *JSAT*, 8(1/2), 2012.
4. D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *ESEC/SIGSOFT FSE*, 2013.
5. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, 2011.
6. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS*, 2009.
7. R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *TACAS*, 2007.

8. H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *FMCAD*, 2011.
9. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, 2013.
10. E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, 2004.
11. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Trans. Software Eng.*, 38(4), 2012.
12. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
13. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, 2011.
14. N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
15. G. Fedyukovich, O. Sery, and N. Sharygina. Function Summaries in Software Upgrade Checking. In *Haifa Verification Conference*, 2011.
16. C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*, 2001.
17. V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, 2007.
18. B. Godlin and O. Strichman. Regression verification. In *DAC*, 2009.
19. A. Griggio. Effective word-level interpolation for software verification. In *FMCAD*, 2011.
20. A. Gurfinkel, S. F. Rollini, and N. Sharygina. Interpolation properties and sat-based model checking. In *ATVA*, 2013.
21. K. Hoder and N. Bjørner. Generalized Property Directed Reachability. In *SAT*, 2012.
22. T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-Based Invariant Discovery. In *NASA Formal Methods*, 2011.
23. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*, 2013.
24. D. Kroening and G. Weissenbacher. Lifting Propositional Interpolants to the Word-Level. In *FMCAD*, 2007.
25. D. Kroening and G. Weissenbacher. Interpolation-Based Software Verification with Wolverine. In *CAV*, 2011.
26. V. Kuncak and A. Rybalchenko, editors. *VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *LNCS*. Springer, 2012.
27. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *CAV*, 2012.
28. J. Lang, P. Liberatore, and P. Marquis. Propositional Independence: Formula-Variable Independence and Forgetting. *J. Artif. Intell. Res. (JAIR)*, 18, 2003.
29. J. Marques-Silva, M. Janota, and A. Belov. Minimal Sets over Monotone Predicates in Boolean Formulae. In *CAV*, 2013.
30. K. L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, 2006.
31. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *VSTTE*, 2012.
32. A. Nadel. Boosting minimal unsatisfiable core extraction. In *FMCAD*, 2010.
33. A. Nadel, V. Ryvchin, and O. Strichman. Efficient MUS Extraction with Resolution. In *FMCAD*, 2013.