Int J Softw Tools Technol Transfer (2009) 11:95–104
DOI 10.1007/s10009-009-0101-x

SPECIAL SECTION ON TACAS07

An abstraction-based decision procedure for bit-vector arithmetic

Randal E. Bryant · Daniel Kroening · Joël Ouaknine · Sanjit A. Seshia · Ofer Strichman · Bryan Brady

Published online: 12 February 2009

Abstract We present a new decision procedure for finite-precision bit-vector arithmetic with arbitrary bit-vector operations. Such decision procedures are essential components of verifications systems, whether the domain of interest is hardware, such as in word-level bounded model-checking of circuits, or software, where one must often reason about programs with finite-precision datatypes. Our procedure alternates between generating under- and over-approximations of the original bit-vector formula. An under-approximation is obtained by a translation to propositional logic in which some bit-vector variables are encoded with fewer Boolean variables than their width. If the under-approximation is unsatisfiable, we use the unsatisfiable core to derive an over-approximation based on the subset of predicates that participated in the proof of unsatisfiability. If this over-approximation is satisfiable, the satisfying assignment guides the refinement of the previous under-approximation by increasing, for some bit-vector variables, the number of Boolean variables that encode them. We present experimental results that suggest that this abstraction-based approach can be considerably more efficient than directly invoking the SAT solver on the original formula as well as other competing decision procedures.

Keywords Bit-vector · Decision-procedures

B. Brady, R. E. Bryant, and S. A. Seshia were supported in part by SRC contract 1355.001. This research was also supported in part by the MARCO Gigascale Systems Research Center and by NSF grant CNS-0627734.

R. E. Bryant
Carnegie Mellon University, Pittsburgh, USA

D. Kroening · J. Ouaknine
Oxford University Computing Laboratory, Oxford, UK

S. A. Seshia · B. Brady
University of California, Berkeley, USA

O. Strichman (✉)
The Technion, Haifa, Israel
e-mail: ofers@ie.technion.ac.il

1 Introduction

Decision procedures for quantifier-free fragments of first-order theories find widespread use in hardware and software verification. Current uses of decision procedures fall into one of two extremes. At one end, a Boolean satisfiability solver is directly employed as the decision procedure, with systems modeled at the bit-level. Sample applications of this kind include bounded model checking [5,10] and SAT-based program analysis [28]. At the other extreme, verifiers use decision procedures that reason over arbitrary-precision abstract types such as the integers and reals ($\mathbb{Z}$ and $\mathbb{R}$).

In reality, system descriptions are best modeled with a level of precision that is somewhere in between. System descriptions are usually at the *word-level*; i.e., they use finite-precision arithmetic and bit-wise operations on bit-vectors. Of course, reasoning about hardware designs or programs written in languages that support finite-precision arithmetic, such as C, are naturally modeled (or treated directly) at the word-level. Ignoring the finiteness of the represented numbers can make a reasoning system unsound.

The following formula, for example, obviously holds over the integers:

$$(x - y > 0) \iff (x > y). \tag{1}$$

If x and y are interpreted as finite-width bit-vectors, however, this equivalence no longer holds, due to possible overflow of the subtraction operation. As another example, consider the

following small C program:

```
unsigned char number = 200;
number = number + 100;
printf("Sum : %d\n", number);
```

The program may return a surprising result, as most architectures use 8 bits to represent variables with type `unsigned char`:

$$\begin{array}{l} \texttt{11001000} = 200 \\ +\ \texttt{01100100} = 100 \\ \hline = \texttt{00101100} = 44 \end{array}$$

When represented with 8 bits by a computer, 200 is stored as 11001000. Adding 100 results in an overflow, as the ninth bit of the result is discarded.

The direct use of a SAT solver as cited earlier (also known as "bit-blasting") is the conceptually simplest way to implement a bit-vector decision procedure, even though it ignores higher-level structure present in the original decision problem.

Naïve bit-blasting is used, for example, in Microsoft, for verifying device drivers. Cook et al. [11] report experimental results that quantify the impact of replacing ZAPATO, a decision procedure for a fragment of linear arithmetic, with Cogent, a bit-vector decision procedure based on bit-blasting. The increased precision of Cogent not only improved the performance of SLAM, it also resulted in the discovery of a previously unknown bug in a Windows device driver.

However, the bit-blasting approach can be too computationally expensive in practice (see, for example, [1]). There is therefore a pressing need for better decision procedures for bit-vector arithmetic.

What is this article about? We present a decision procedure for quantifier-free bit-vector arithmetic that uses automatic abstraction-refinement. This procedure is now implemented in the verification system UCLID, and we shall call it by this name from hereon. Given an input bit-vector formula ϕ, UCLID first builds an under-approximation $\underline{\phi}$ from ϕ by restricting the number of Boolean variables used to encode each bit-vector variable (see details of this encoding in Sect. 3.1). The reduced formula is typically much smaller and easier to solve. If $\underline{\phi}$ is satisfiable, so is ϕ, and the algorithm terminates. In case the Boolean formula is found to be unsatisfiable, the SAT solver is able to output a resolution proof of this fact, from which the unsatisfiable core used in this proof can be extracted. Using this core, an over-approximation $\overline{\phi}$ is built. This over-approximation uses the full set of bits of the original vectors, but only a subset of the constraints. This subset is determined by examining the unsatisfiable core of $\underline{\phi}$. If $\overline{\phi}$ is unsatisfiable, so is ϕ, and UCLID terminates. Otherwise, the algorithm refines the under-approximation $\underline{\phi}$ by increasing, for at least one bit-vector variable, the number of Boolean variables encoding it. Specifically, the new size is implied by the value of this variable in the satisfying assignment to $\overline{\phi}$. This process is repeated until the original formula is shown to be either satisfiable or unsatisfiable. The algorithm is trivially guaranteed to terminate due to the finite domain.

This approach has the potential of being practically efficient in one of the following two scenarios:

1. The bit-vector formula is satisfiable, and there exists a numerically 'small' solution, i.e., a solution that can be represented with a small number of bits.
2. The bit-vector formula is unsatisfiable, and a relatively small number of terms in this formula participate in the proof (i.e., the proof still holds after replacing the other terms with new inputs).

Whether this potential is fulfilled depends on one's ability to find such small solutions and small unsatisfiable cores[1] efficiently: for the former, we search for gradually increasing solutions in terms of the number of bits that are needed in order to represent them, and hence are guaranteed to find a small one if it exists; for the latter, modern SAT solvers are quite apt at finding small cores when they exist. In practice, as our experiments show, one of these conditions frequently holds and we are able to detect it with our tool faster than analyzing the formula head-on without any approximations.

Our approach can be seen as an adaptation to bit-vector formulas of our previous work [18] on abstraction-refinement of quantifier-free Presburger Arithmetic, which, in turn, was inspired by the proof-based abstraction-refinement approach to model checking that was proposed by McMillan and Amla [20]. Other than the different problem domain (bit-vectors vs. Presburger formulas), we also extend the theoretical framework to operate on an arbitrary circuit representation of the formula, rather than on a CNF representation. We also employ optimizations specific to bit-vector arithmetic. On the applied side, we report experimental results on a set of benchmarks generated in both hardware and software verification. Our experiments suggest that our approach can be considerably more efficient than directly invoking the SAT solver on the original formula as well as other state-of-the-art decision procedures.

This article extends the proceedings version [8] mainly by adding more technical background and motivation, elaborating more on the over-approximation technique and clarifying various issues.

Related Work Current decision procedures for bit-vector arithmetic fall into one of three categories:

[1] A small unsatisfiable core of the CNF encoding of a formula does not necessarily correspond to a small number of terms from the original formula, but obviously the two measures are correlated.

1) Bit-blasting and its variants: Many current decision procedures are based on bit-blasting the input formula to SAT, with a variety of methods for encoding the various bit-vector operations. Most of the modern tools apply various simplifications and high-level reasoning before the bit-blasting phase. The Cogent [11] procedure mentioned earlier belongs to this category. The most current version of CVC-Lite [14] pre-processes the input formula using a normalization step followed by equality rewrites before finally bit-blasting to SAT. Wedler et al. [26] have a similar approach wherein they normalize bit-vector formulas in order to simplify the generated SAT instance. STP [15] is a decision procedure for both bit-vector arithmetic and the theory of arrays; it performs a lazy instantiation of array axioms as well as arithmetic and Boolean simplifications on the bit-vector formula before bit-blasting to MiniSat. Yices [13] applies bit-blasting to all bit-vector operators except for equality. The tool Spear [2], by Babić and Hu, is based on bit-blasting and a fast SAT solver with numerous optimization parameters.

2) Canonizer-based procedures: Earlier work on deciding bit-vector arithmetic centered on using a Shostak-like approach of using a canonizer and solver for that theory. The work by Cyrluk et al. [12] and by Barrett et al. on the Stanford Validity Checker [4] fall into this category; the latter differs from the former in the choice of a canonical representation. These approaches are very elegant, but are restricted to a subset of bit-vector arithmetic comprising concatenation, extraction, and linear equations (not inequalities) over bit-vectors.

3) Procedures for modular and bounded arithmetic: The third category of systems focuses on techniques to handle (linear and non-linear) modular arithmetic. The most recent work in this area is by Babić and Musuvathi [3], who encode non-linear operations as non-linear congruences and make novel use of Newton's p-adic method for solving them. However, this approach does not treat some of the operations that we handle such as integer division, and seems harder to extend to new operations. Brinkmann and Drechsler [6] use an encoding of linear bit-vector arithmetic into integer linear programming with bounded variables in order to decide properties of RTL descriptions of circuit data-paths, but do not handle any Boolean operations. Parthasarathy et al. [23] build on this approach by using a lazy encoding with a modified DPLL search, but non-linear bit-vector arithmetic is not supported. Huang and Cheng [17] give an approach to solving bit-vector arithmetic based on combining ATPG with a solver for linear modular arithmetic. This approach is limited in its treatment of non-linear operations which it handles by heuristically rewriting them as linear modular arithmetic constraints.

McMillan and Amla [20] use a technique related to ours in order to accelerate model checking algorithms over finite Kripke structures. Specifically, they invoke a bounded model checker to decide which state variables should be made visible in order to generate a 'good' abstraction for the next iteration of model checking. Gupta et al. [16] propose a similar model-checking framework, which among others makes greater use of counterexamples and uses abstract models for both validation and falsification attempts. Our approach differs from both of these in the following respects: we use a bit-vector decision procedure instead of a model checker, and we seek to eliminate constraints rather than variables (or gates or latches, as the case may be).

Lahiri et al. [19] present an algorithm for deciding satisfiability of quantifier-free Presburger arithmetic that is based on alternating between an under- and an over-approximation. The under-approximation is constructed as in [18]. The over-approximation uses a Craig Interpolant.

2 Preliminaries

2.1 Boolean satisfiability

We begin by recalling some well-known terms and observations concerning Boolean formulas and satisfiability (SAT). A *literal* is either a variable or its negation. A *clause* is a disjunction of zero or more literals, with the empty clause denoting *False*. A formula is said to be in conjunctive normal form (CNF) if it is a conjunction of clauses.

Linear conversion to CNF: Let β be a Boolean formula with variables $b_1, \ldots, b_n$. It is possible to construct a Boolean formula $\mathsf{cnf}(\phi)$ with variables $b_1, \ldots, b_{n+p}$ (where the $b_{n+1}, \ldots, b_{n+p}$ are fresh Boolean variables), such that

- $\mathsf{cnf}(\beta)$ is in CNF:

$$\mathsf{cnf}(\beta) = \bigwedge_{j=1}^{m} B_j,$$

  where each B_j is a Boolean clause,
- $\mathsf{cnf}(\beta)$ is satisfiable iff β is satisfiable; more precisely,

$$\exists b_{n+1}, \ldots, b_{n+p}.\ \mathsf{cnf}(\beta)$$

  is tautologically equivalent to β, and
- The number of variables and the number of clauses in $\mathsf{cnf}(\beta)$ are both linear in the size of β.

Linear-time algorithms for computing $\mathsf{cnf}(\beta)$ are well-known since Tseitin [24].

Tseitin suggested to add one new variable for every *logical gate* in the original formula, and several clauses (e.g., three for 'and' and 'or' nodes, two for 'xor') to constrain the value of this variable to be equal to the gate it represents, in terms of the inputs to this gate. The original formula is satisfiable if and only if the conjunction of these clauses together

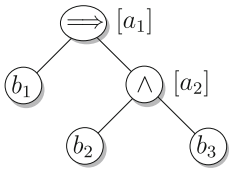


Fig. 1 Tseitin's encoding. Assigning an auxiliary variable to each logical gate (here in *square brackets*) enables us to translate each propositional formula to CNF, while increasing the size of the formula only linearly

with the new variable associated with the top-most operator, is satisfiable. This is best illustrated with an example.

Example 1 Given a propositional formula

$$b_1 \implies (b_2 \wedge b_3), \tag{2}$$

with Tseitin's encoding we assign a new variable to each subexpression, or, in other words, to each logical gate, e.g., 'and' ($\wedge$), 'or' ($\vee$), 'not' ($\neg$) etc.

For this example, let us assign the variable a_2 to the 'and' gate (corresponding to the subexpression $b_2 \wedge b_3$) and a_1 to the 'implication' gate (corresponding to $b_1 \implies a_2$), which is also the top-most operator of this formula. Figure 1 illustrates the *derivation tree* of our formula, together with these auxiliary variables in square brackets. We need to satisfy a_1, together with two equivalences:

$$\begin{array}{l} a_1 \iff (b_1 \implies a_2) \\ a_2 \iff (b_2 \wedge b_3). \end{array} \tag{3}$$

The first equivalence can be rewritten in CNF as:

$$\begin{array}{ll} (a_1 \vee b_1) & \wedge \\ (a_1 \vee \neg a_2) & \wedge \\ (\neg a_1 \vee \neg b_1 \vee a_2), & \end{array} \tag{4}$$

and the second equivalence can be rewritten in CNF as:

$$\begin{array}{ll} (\neg a_2 \vee b_2) & \wedge \\ (\neg a_2 \vee b_3) & \wedge \\ (a_2 \vee \neg b_2 \vee \neg b_3). & \end{array} \tag{5}$$

Thus, the overall CNF formula is the conjunction of (4), (5) and the unit clause

$$(a_1), \tag{6}$$

which represents the top-most operator. □

SAT solvers A CNF SAT solver is an algorithm that determines, given a Boolean formula β in CNF, whether β is satisfiable. If so, it outputs a satisfying assignment for β. If, on the other hand, β is unsatisfiable, modern SAT solvers can also generate a *proof of unsatisfiability* [20,29] based on the *binary resolution* inference-rule. The leafs of such proofs (the assumptions) constitute an *unsatisfiable core*, i.e., an unsatisfiable subset of the clauses of β. In practice, SAT solvers tend to find small unsatisfiable cores if they exist. Indeed, in many cases in practice, formulas contain a large number of redundant constraints.

2.2 Bit-vector arithmetic

The quantifier-free fragment of the first-order theory of *bit-vector arithmetic* that we consider here includes finite-precision integer arithmetic with linear and non-linear operators, as well as standard bit-wise operators, such as left shift, logical and arithmetic right shifts, extraction, concatenation, and so forth. In fact, the approach we use in this paper is orthogonal to the the set of operators, since it only relies on the given finite width for each variable, as well as on the existence of a propositional encoding of the formula.

At present, UCLID supports all bit-vector arithmetic constructs defined in the grammar that appears in Fig. 2. Standard notation has been used in describing the above grammar, and we only point out certain aspects of the notation. *Terms* denote bit-vectors while *formulas* are Boolean-valued expressions. The expression

$$\mathit{formula}\ ?\ \mathit{term}\ :\ \mathit{term}$$

is an "if-then-else" expression that selects between two terms on the basis of the value of its Boolean first argument. The expression $\mathit{term}[i : j]$ denotes the extraction of bits i through j of the bit-vector expression *term*. The operator % denotes the integer mod operator, while @ denotes concatenation.

Each bit-vector expression is associated with a type. The type is the width of the expression in bits and whether it is signed (two's complement encoding) or unsigned (binary encoding). Assigning semantics to this language is straightforward, e.g., as done in [6].

Note that all arithmetic operators (addition: $+$, subtraction: $-$, multiplication: $*$, division: $\div$, modulo: %) are finite-precision, and come with an associated operator width.

Note also that the relational operators $>, <, \leq, \geq$, the non-linear arithmetic operators ($*, \div$, %) and the right-shift depend on whether an unsigned, binary encoding is used or a two's complement encoding is used. We assume that the type of the expression is clear from the context.

This paper addresses the *satisfiability problem* for bit-vector formulas: given a bit-vector formula ϕ, is there an assignment to the bits in ϕ under which ϕ evaluates to *True*? It is easy to see that this problem is NP-complete.

3 The decision procedure

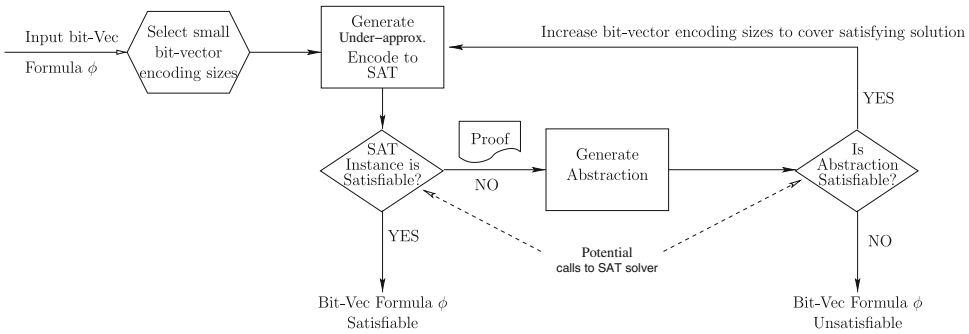
We now present the main contribution of this paper, a SAT-based decision procedure that operates by generating increasingly precise *abstractions* of bit-vector formulas.

Notation Formulas in bit-vector arithmetic are denoted by $\phi, \phi', \phi_1, \phi_2, \ldots,$ and Boolean formulas by $\beta, \beta_1, \beta_2, \ldots$. We denote by ϕ the input bit-vector arithmetic formula and by $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_n$ its bit-vector variables. Each such variable $\mathbf{v}_i$ has an associated bit-width w_i.

Fig. 2 Supported grammar: UCLID supports all bit-vector arithmetic operators defined in the above grammar

$$
\begin{aligned}
\mathit{formula} &: \mathit{formula} \vee \mathit{formula} \mid \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid \mathit{atom} \\
\mathit{atom} &: \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \\
\mathit{rel} &: = \mid \neq \mid \leq \mid \geq \mid < \mid > \\
\mathit{term} &: \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \sim \mathit{term} \mid \mathit{constant} \mid \mathit{term}[i:j] \mid \mathit{formula}\ ?\ \mathit{term} : \mathit{term} \\
\mathit{op} &: + \mid - \mid * \mid \div \mid \% \mid << \mid >> \mid @ \mid \& \mid \mid \mid \hat{}
\end{aligned}
$$

Fig. 3 Abstraction-based approach to solving bit-vector arithmetic



3.1 Overview

We first give a broad overview of our decision procedure, which is illustrated in Fig. 3. Details of design decisions are described later in this section.

The decision procedure performs the following steps:

1. *Initialization:* For each variable $\mathbf{v}_i$, we select a corresponding number s_i of Boolean variables to encode it with, where $0 \leq s_i \leq w_i$.
   We will call s_i the *encoding size* of bit-vector variable $\mathbf{v}_i$.
2. *Under-approximation and encoding to SAT:* An under-approximation, denoted $\underline{\phi}$, is generated by restricting the values of each $\mathbf{v}_i$ to range over a set of cardinality 2^{s_i}. Thus, the Boolean encoding of $\mathbf{v}_i$ will comprise s_i Boolean variables; note, however, that the length of the vector of Boolean variables replacing $\mathbf{v}_i$ remains w_i.
   There are several ways to generate such an under-approximation and its Boolean encoding. One option is to encode $\mathbf{v}_i$ using Boolean variables on its s_i low order bits and then zero-extend it to be of length w_i. The other is to "sign-extend" it instead. For example, if $s_i = 2$ and $w_i = 4$, the latter would generate the Boolean vector $[\mathbf{v}_{i1}, \mathbf{v}_{i1}, \mathbf{v}_{i1}, \mathbf{v}_{i0}]$ (where $\mathbf{v}_{ij}$ are Boolean variables). Our implementation currently uses the latter encoding, as it enables searching for solutions at both ends of the ranges of bit-vector values (e.g., in the example above the possible values are 0000, 0001, 1110 and 1111). This is especially useful in cases where the formula contains comparisons to high values (such as MAXINT in C). Further exploration of this aspect is left to future work.
   A Boolean formula β is then computed from $\underline{\phi}$ using standard circuit encodings for bit-vector arithmetic operators. The width of the operators is left unchanged. The formula β is passed to an off-the-shelf SAT solver. The only feature required of this SAT solver is that its response on unsatisfiable formulas be accompanied by an unsatisfiable core.
   If the SAT solver reports that β is satisfiable, then the satisfying assignment is an assignment to the original formula ϕ, and the procedure terminates. However, if β is unsatisfiable, we continue on to the next step.
3. *Over-approximation from the unsatisfiable core:* The SAT solver extracts an unsatisfiable core $\mathcal{C}$ from the proof of unsatisfiability of β. We use $\mathcal{C}$ to generate an over-approximating *abstraction* $\overline{\phi}$ of ϕ. The formula $\overline{\phi}$ is also a bit-vector formula, but typically much smaller than ϕ. The algorithm that generates $\overline{\phi}$ is described in Sect. 3.2.
   The key property of $\overline{\phi}$ is that its translation into SAT, using the same sizes s_i as those that were used for $\underline{\phi}$, would also result in an unsatisfiable Boolean formula.
   The satisfiability of $\overline{\phi}$ is then checked using a *sound and complete decision procedure* $\mathcal{P}$ for bit-vector arithmetic, e.g., a bit-blasting approach.
   If $\overline{\phi}$ is unsatisfiable, we can conclude that so is ϕ. On the other hand, if $\overline{\phi}$ is satisfiable, it must be the case that at least one variable $\mathbf{v}_i$ is assigned a value that is not representable with s_i Boolean variables (recall the key property enjoyed by $\overline{\phi}$ cited earlier). This larger satisfying assignment indicates the necessary increase in the encoding size s_i for $\mathbf{v}_i$. Proceeding thus, we increase s_i for all relevant i, and go back to Step 2.

Remark 1 Note that in this step it would be permissible to merely use a *sound*, but not necessarily *complete*, bit-vector arithmetic decision procedure $\mathcal{P}$. In other words, we require that the outcome of $\mathcal{P}$ be correct whenever

this outcome is 'Unsat', but we can tolerate spurious satisfying assignments. Indeed, in cases where $\mathcal{P}$ provides a satisfying assignment that is not a satisfying assignment for $\overline{\phi}$, we can simply increase s_i by 1 for each i such that $s_i < w_i$, and go back to Step 2. Of course, bit-blasting is both sound and complete.

Since s_i increases for at least one i in each iteration of this loop, this procedure is guaranteed to terminate in $O(n \cdot w_{\max})$ iterations, where $w_{\max} = \max_i w_i$. Each such iteration involves a call to a SAT solver and a decision procedure for bit-vector arithmetic.

One of the main theoretical advances we make over the earlier work on Presburger arithmetic [18] is a different method for generating the abstraction. We describe this in the following section.

3.2 Generating an over-approximating abstraction

The earlier work assumed that ϕ was in CNF, whereas our procedure works with an arbitrary directed acyclic graph (DAG) or circuit-based representation, which is the format in which the input problems are typically given. While ϕ can be transformed to CNF (in two different ways, listed below), we argue below that neither of those approaches is desirable, primarily due to the presence of nested if-then-else (ITE) expressions at arbitrary locations in ϕ.

1) Eliminating ITE using new variables: By giving each ITE expression in the formula a fresh bit-vector variable name, we can eliminate all ITEs with just a linear increase in the number of bit-vector variables and formula size.

Introducing such new bit-vector variables restricts the amount of cheap simplifications one can perform on the formula before passing it to a SAT solver. For many formulas, such simplifications (corresponding, for example, to standard Boolean identities) are essential for scalability. For example, consider the ITE-based formula $\text{ITE}(b, T_1, T_2) = v$ where T_1 and T_2 are bit-vector terms. Using fresh variables v_1 and v_2 for T_1 and T_2 respectively, we would obtain the transformed ITE-free expression as $(b \implies v = v_1) \wedge (\neg b \implies v = v_2) \wedge (v_1 = T_1) \wedge (v_2 = T_2)$. Now, if owing to simplifications v always differs from T_1 and T_2 in the least significant bit, the ITE-based formula will simplify to *False*. However, the ITE-free representation will require the SAT solver to propagate implications to be able to perform the same simplification. This inability to adequately simplify the formula before passing it to SAT can result in an unnecessarily large SAT problem with a resultant slowdown.

Note that the number of input bit-vector variables ($\mathbf{v}_i$'s) is usually a few orders of magnitude smaller than the size of the formula ϕ. As a result, when treating the new variables as inputs, the SAT solver's performance has been observed to suffer a great deal. Since the value of each such new variable is implied by the values of the original variables, it may seem that one way to deal with the above problem is to restrict the SAT solver from splitting on the bit-encodings of the new ITE variables. However, such restrictions have also been found to severely adversely affect the run-time of current SAT engines. (It is rarely the case that changing the generic decision heuristic of a modern SAT solver due to high-level information improves performance).

2) Direct elimination of ITE: Another way of eliminating ITEs is to expand out the cases without introducing new variables. However, this leads to a worst-case exponential blow-up in formula size, which is commonly witnessed in practice.

We have therefore devised an abstraction-generation algorithm $\mathcal{A}$ that operates directly on the DAG representation of ϕ, denoted D_ϕ. The inputs to $\mathcal{A}$ are D_ϕ, the root node, and the unsatisfiable core $\mathcal{C}$. The output is a DAG $D_{\overline{\phi}}$ representing $\overline{\phi}$, which is an over-approximation of ϕ. Let N^ϕ and $N^{\overline{\phi}}$ be the set of nodes in D_ϕ and $D_{\overline{\phi}}$, respectively.

Before describing the algorithm, we need to describe the process of transforming the Boolean encodings of ϕ and $\overline{\phi}$ into CNF. It can be seen as a generalization of Tseitin's encoding (which introduces fresh variables for internal nodes, as described in Sect. 2) to the case of bit-vector formulas. Each internal node $\mathsf{n} \in N^\phi$ is annotated with a set of CNF clauses $c(\mathsf{n})$ that relate the output of that node $o(\mathsf{n})$ to its inputs, according to the operator in the node. These output variables then appear as input to the parent nodes of n. Then a conjunction of the clauses in $\{c(\mathsf{n}) | \mathsf{n} \in N^\phi\}$ and one more unit clause with the variable encoding the top node, is the CNF representation of ϕ. A subset of these clauses constitutes the UNSAT core $\mathcal{C}$. These definitions and notations also apply to $D_{\overline{\phi}}$, and we will use them for both DAGs when the meaning is clear from the context. For a formula (or equivalently, a set of clauses) C, we denote by $vars(C)$ the set of variables that appear in C.

Procedure $\mathcal{A}$ (see Algorithm 1) recurses down the structure of D_ϕ and creates $D_{\overline{\phi}}$. It replaces a Boolean node n with a new variable and backtracks, if and only if none of the variables in $vars(c(\mathsf{n}))$ are present in the unsatisfiable core $\mathcal{C}$[2]. It uses the functions left-child(D_ϕ, n) and right-child(D_ϕ, n) to return the left and right child of n on D_ϕ, respectively.

The replacement of Boolean nodes with new variables can be further optimized using the "pure-literal rule": if n_ϕ is a Boolean-valued node and only appears unnegated, replace it by *True*; likewise, if n_ϕ only appears negated, replace it by *False*. In other words, in such cases no new Boolean variable is needed.

[2] The same replacement criterion can be applied to bit-vector-valued nodes, which can then be replaced with fresh bit-vector variables. Our implementation ignores this option, however, and we shall therefore also ignore this possibility in the proof.

Algorithm 1 An algorithm for abstracting an NNF formula ϕ such that only subformulas that do not contribute to the UNSAT core $\mathcal{C}$ are replaced with a new variable

```
procedure A(DAG D_φ, node n, unsat-core C)
    if n is a leaf then return ;
    end if
    if n is Boolean and vars(c(n)) ∩ vars(C) = ∅ then
        Replace n in D_φ with a new Boolean variable;
        return ;
    end if
    A(D_φ, left-child(D_φ, n), C);
    A(D_φ, right-child(D_φ, n), C);
end procedure
```

Note that the resulting DAG $D_{\overline{\phi}}$ can be embedded into D_ϕ. For each node $\mathsf{n} \in N^\phi$ we will denote by $\bar{\mathsf{n}}$ its counterpart in $D_{\overline{\phi}}$ before the abstraction process begins (after the abstraction some of them can be eliminated by simplifications).

The correctness of our abstraction technique is formalized by the following two theorems:

Theorem 1 *$\overline{\phi}$ is an over-approximating abstraction of ϕ.*

Proof Let α be a satisfying assignment of ϕ. We show how to construct $\bar{\alpha}$, a satisfying assignment for $\overline{\phi}$. First, for each variable $v \in vars(\phi)$ such that the (leaf) node representing v is still present in $D_{\overline{\phi}}$, define $\bar{\alpha}(v) = \alpha(v)$. Second, for each Boolean variable $b \in \{vars(\overline{\phi}) \backslash vars(\phi)\}$ (i.e., the new abstracting variables) represented by node $\mathsf{n} \in D_{\overline{\phi}}$, define $\bar{\alpha}(b)$ to be equal to the Boolean value of the corresponding node in D_ϕ, as implied by α. For example, if $\alpha(b_1) = True$, $\alpha(b_2) = False$ and the node $b_1 \vee b_2$ was replaced with a new variable b, then $\bar{\alpha}(b) = True \vee False = True$. Clearly, $\bar{\alpha}$ satisfies $\overline{\phi}$, since every node in $D_{\overline{\phi}}$ is evaluated the same as its counterpart in D_ϕ. Hence, if ϕ is satisfiable, then so is $\overline{\phi}$, which implies the correctness of the Theorem. □

Next, we have to prove termination. Termination is implied if we can show that any satisfying assignment to $\overline{\phi}$ requires width larger than the current one s_i (i.e., the width with which the unsatisfiable core $\mathcal{C}$ was derived), or, equivalently:

Theorem 2 *The SAT encoding of $\overline{\phi}$ with encoding sizes s_i is unsatisfiable.*

Proof We will prove that the CNF encoding of $\overline{\phi}$ with sizes s_i contains the clauses of the UNSAT core $\mathcal{C}$.

Three observations about this encoding are important for our proof:

1. First, for an internal node n that represents a Boolean operator, each clause in $c(\mathsf{n})$ contains the output variable of its node. For example, the CNF of an 'and' node $o = a \wedge b$ is $(o \vee \neg a \vee \neg b)$, $(\neg o \vee a)$, $(\neg o \vee b)$, and indeed o, the output variable of this node, is present in all three clauses. The same applies to the other Boolean operators. Hence, we can write $o(cl)$ for a clause cl to mean the output variable of the node that cl annotates (hence, $o(cl) \in cl$).
2. Second, the same observation applies to predicates over bit-vectors. For simplicity, we concentrate only on the bit-vector equality predicate. In such a node, each clause contains either the output variable or an auxiliary variable present only in this node. For example, the CNF of the node $o = (\mathbf{v_1} = \mathbf{v_2})$ for 2-bit bit-vectors $\mathbf{v_1}$ and $\mathbf{v_2}$, is the following:

   $(x \vee \mathbf{v_1}[0] \vee \mathbf{v_2}[0])$, $(x \vee \neg\mathbf{v_1}[0] \vee \neg\mathbf{v_2}[0])$,
   $(\neg x \vee \mathbf{v_1}[0] \vee \neg\mathbf{v_2}[0])$, $(\neg x \vee \neg\mathbf{v_1}[0] \vee \mathbf{v_2}[0])$,
   $(o \vee \neg x \vee \neg\mathbf{v_1}[1] \vee \neg\mathbf{v_2}[1])$, $(o \vee \neg x \vee \mathbf{v_1}[1] \vee \mathbf{v_2}[1])$,
   $(\neg o \vee \mathbf{v_1}[1] \vee \neg\mathbf{v_2}[1])$, $(\neg o \vee \neg\mathbf{v_1}[1] \vee \mathbf{v_2}[1])$,
   $(\neg o \vee x)$

   (the first four clauses encode $x = (\mathbf{v_1}[0] = \mathbf{v_2}[0])$, the other clauses encode $o = x \wedge (\mathbf{v_1}[1] = \mathbf{v_2}[1])$ where x is the local auxiliary variable).
3. Finally, observe that resolution among clauses that relate the output and input of a node using the output variable as the resolution variable, results in a tautology. For example, recall the CNF representation of the 'and' node above: resolving on the output variable o of that node results in a tautology. The same observation applies to other Boolean operators and equality between bit-vectors.

We use these observations for analyzing the three possible cases for a node n in D_ϕ: either it is retained in $D_{\overline{\phi}}$, replaced with a new variable, or eliminated. Our goal, recall, is to show that despite the abstraction implied by these changes to the DAG, the set of clauses that encode the new DAG $D_{\overline{\phi}}$ still contains the UNSAT core $\mathcal{C}$ of ϕ.

Claim 1: for each node $\mathsf{n} \in N^\phi$ for which the corresponding $\bar{\mathsf{n}} \in N^{\overline{\phi}}$ is retained in the abstraction process, $c(\mathsf{n}) \cap \mathcal{C} = c(\bar{\mathsf{n}}) \cap \mathcal{C}$.

Proof Since n and $\bar{\mathsf{n}}$ encode the same operator and receive the same *type* of input (e.g., if n and $\bar{\mathsf{n}}$ represent a bit-vector operator, then their respective inputs are bit-vectors of the same width), then $c(\mathsf{n})$ and $c(\bar{\mathsf{n}})$ are equivalent up to renaming of variables. Such a renaming can occur if the abstraction process replaced one of the inputs (or both) with a new variable. But this means that none of these inputs are in $\mathcal{C}$, hence those clauses in $c(\bar{\mathsf{n}})$ that contain renamed literals, are not in $\mathcal{C}$. Hence, $c(\mathsf{n}) \cap \mathcal{C} = c(\bar{\mathsf{n}}) \cap \mathcal{C}$.

Claim 2: for each node $\mathsf{n} \in N^\phi$ that was replaced with a new variable in $N^{\overline{\phi}}$, $c(\mathsf{n}) \cap \mathcal{C} = \emptyset$.

Proof This is trivial by the construction of the abstraction: if any of the clauses in $c(\mathsf{n})$ were in $\mathcal{C}$, then this node would not be replaced with a new variable.

Claim 3: For each node $\mathsf{n} \in N^{\phi}$ whose corresponding node $\bar{\mathsf{n}} \in N^{\overline{\phi}}$ was eliminated (i.e., the paths of this node to the root were all 'cut' by the abstraction), $c(\mathsf{n}) \cap \mathcal{C} = \emptyset$.

Proof On each path from n to the root node, there exists one or more nodes other than n that were replaced with free variables. For simplicity of the proof, we will consider one such path and denote the closest node to n that was replaced with a new variable by n_c.
We will now prove the claim by induction on the distance (in terms of number of DAG operators) from n to n_c. In the base case n is a direct child of n_c. Falsely assume that there exists a clause $cl \in c(\mathsf{n})$ such that $cl \in \mathcal{C}$. $c(\mathsf{n}_c)$ contain $o(\mathsf{n})$, the output variable of n, and cl also contains $o(\mathsf{n})$ (see observations 1 and 2 above). Hence, if $cl \in \mathcal{C}$, then $o(\mathsf{n}) \in vars(\mathcal{C})$ which contradicts the condition for abstracting n_c with a new variable.
For the induction step falsely assume that there exists a clause $cl \in c(\mathsf{n})$ such that $cl \in \mathcal{C}$. By the induction hypothesis, none of the clauses in the parent node of n are in $\mathcal{C}$. Hence, only clauses from $c(\mathsf{n})$ can contain the output variable of cl in $\mathcal{C}$. This means that $o(cl)$ can only be resolved-on among $c(\mathsf{n})$ clauses. By noting that this kind of resolution can only result in a tautology (see observation 3 above), this resolution step cannot be on the path to the empty clause in the resolution proof. This contradicts, however, the requirement that any variable in every clause that participates in a proof of the empty clause must be resolved on in order to eliminate it.

Thus, the set of clauses annotating $D_{\overline{\phi}}$ contains $\mathcal{C}$ and hence $\overline{\phi}$ is unsatisfiable. □

In comparison with our previous CNF-based abstraction scheme [18], we note that, for ITE-free formulas, that approach can generate more compact abstractions, as they do not introduce new variables. However, for real-world benchmarks from both hardware and software verification, such as those discussed in the following section, we found that elimination of ITEs leads to significant space and time overheads. The approach of this paper allows us to extend the abstraction-based approach to operate on arbitrary DAG-like formulas. Moreover, we found that the Boolean structure in the original bit-vector formula is typically not the primary source of difficulty; it is the bit-vector constraints that are the problem.

3.3 Abstraction with partially-interpreted functions

It is well-known that certain bit-vector arithmetic operators, such as integer multiplication of two variables (of adequately large width), are extremely hard for a procedure based on bit-blasting. However, for many problems involving these operators, it is unnecessary to reason about all of the operators' properties in order to decide the formula. Instead, using a set of rules (based on well-known rewrite rules) allows us to perform fine-grained abstractions of functions, which often suffices. Such (incomplete) abstractions can be used in the over-approximation phase of our procedure, while maintaining the overall procedure sound and complete (see Remark 1 in Sect. 3.1). This is a major advantage, because these rules can be very powerful in simplifying the formula.

Therefore, UCLID invokes a preprocessing step before calling Algorithm $\mathcal{A}$. In this step, it replaces a subset of "hard" operators by lambda expressions that *partially interpret* those operators. The resulting formula is then bit-blasted to SAT.

For example, UCLID replaces the multiplication operator $*_w$ of width w (for $w > 4$, chosen according to the capacity of current SAT engines) by the following lambda expression involving the freshly introduced uninterpreted function symbol $\mathtt{mul}_w$:

$$\lambda x.\, \lambda y.\, ITE(x = 0 \vee y = 0, 0, ITE(x = 1, y, ITE(y = 1, x, \mathtt{mul}_w(x, y)))). \quad (7)$$

This expression can be seen as partially interpreting multiplication, as it models precisely the behavior of this operator when one of the arguments is 0 or 1, but is uninterpreted otherwise.

4 Experimental results

The new procedure is now incorporated within the UCLID verification system [25], which is implemented in Moscow ML [22] (a dialect of Standard ML). MiniSat [21] was used as the SAT solver to solve over-approximations, while Booleforce (written by Armin Biere) was used as a proof-generating SAT solver for under-approximations. The initial value of s_i is set to $\min(4, w_i)$ for benchmarks not involving hard operators (like multiplication) while it is set to $\min(2, w_i)$ otherwise.

Table 1 shows experimental results obtained on a set of bit-vector formulas. We compare the run-time of UCLID against bit-blasting to MiniSat, and the STP [9] and Yices [13] decision procedures. (The latter two procedures jointly won the bit-vector division of the SMT-COMP'06 competition, and we compare against the versions that were entered to the competition.) All results were obtained on a system with a 2.8 GHz Xeon processor and 2 GB RAM. The benchmarks are drawn from a wide range of sources, arising from verification and testing of both hardware and software:[3]

– Verification of word-level versions of an x86-like processor model [7] (Y86-std, Y86-btnft);

[3] All benchmarks that we have permission to make publicly available have been submitted to the SMT-LIB repository at http://goedel.cs.uiowa.edu/smtlib/benchmarks/QF_BV.

Table 1 Comparison of run-time of abstraction-based approach (UCLID) with bit-blasting, STP, and Yices

Formula	Ans.	Bit-blasting			UCLID			STP (s)	Yices (s)
		Run-time (s)			Run-time (s)				
		Enc.	SAT	Total	Enc.	SAT	Total		
Y86-std	UNSAT	17.91	TO	TO	23.51	987.91	**1011.42**	2083.73	TO
Y86-btnft	UNSAT	17.79	TO	TO	26.15	1164.07	**1190.22**	err	TO
s-40-50	SAT	6.00	33.46	39.46	106.32	10.45	116.77	**12.96**	65.51
BBB-32	SAT	37.09	29.98	67.07	19.91	1.74	**21.65**	38.45	183.30
rfunit_flat-64	SAT	121.99	32.16	154.15	19.52	1.68	**21.20**	873.67	1312.00
C1-P1	SAT	2.68	45.19	47.87	2.61	0.58	**3.19**	err	err
C1-P2	UNSAT	0.44	TO	TO	2.24	2.12	**4.36**	TO	TO
C3-OP80	SAT	14.96	TO	TO	14.54	349.41	**363.95**	TO	3242.43
egt-5212	UNSAT	0.064	0.003	0.067	0.163	0.001	0.164	0.018	**0.009**

The best run-time is highlighted in bold font. A "TO" indicates that a timeout of 3,600 s was reached. An "err" indicates that the solver could not handle bit-vectors of width as wide as those in the benchmark or quit with an exception. Bit-blasting used MiniSat. UCLID used Booleforce for proof generation and MiniSat on the abstraction. STP is based on MiniSat. "Ans" indicates whether the formula was satisfiable (SAT) or not (UNSAT). "Enc" indicates time for translation to SAT, and "SAT" indicates the time taken by the SAT solver (both calls)

Table 2 Statistics on the abstraction-based approach (UCLID)

Formula	Ans.	$\max_i\ s_i$	$\max_i\ w_i$	Num. Iter	$\max \frac{\lvert\bar{\phi}\rvert}{\lvert\phi\rvert}$	Speedup
Y86-std	UNSAT	4	32	1	0.18	2.06
Y86-btnft	UNSAT	4	32	1	0.20	>3.01
s-40-50	SAT	32	32	8	0.12	0.11
BBB-32	SAT	4	32	1	–	1.78
rfunit_flat-64	SAT	4	64	1	–	7.27
C1-P1	SAT	2	65	1	–	15.00
C1-P2	UNSAT	2	14	1	1.00	>825.69
C3-OP80	SAT	2	9	1	–	8.91
egt-5212	UNSAT	8	8	1	0.13	0.06

"$\max_i\ s_i$" indicates the maximum value of s_i generated in the entire run. "Num. Iter" indicates the number of iterations of the abstraction-refinement loop where an iteration is counted if at least one of the SAT solver calls is made. The second to last column compares the size of the largest abstraction $\bar{\phi}$ created as a fraction of the size of the original formula ϕ, where sizes are measured as the number of nodes in the DAG representations of the formulas. "Speedup" indicates the factor by which the abstraction-based approach is faster than its nearest competitor, or slower than the best solver

- Detection of format-string vulnerabilities in C programs [27] (s-40-50);
- Hardware verification benchmarks obtained from Intel, slightly modified (BBB-32, rfunit_flat-64);
- Word-level combinational equivalence checking benchmarks obtained from a CAD company[4] (C1-P1, C1-P2, C3-OP80); and
- Directed random testing of programs [9] (egt-5212). This represents the set of benchmarks used in SMT-COMP'06, which are easily solved within a fraction of a second.[5]

The first three sets of benchmarks involve only (finite-precision) linear arithmetic. The combinational equivalence checking benchmarks involve finite-precision multiplication with large widths (e.g., C1-P1 and C1-P2 involve 65-bit, 49-bit, and 30-bit multiplication), apart from bitwise operations including extraction and concatenation. The last set includes linear arithmetic and bitwise operations.

An analysis of UCLID's performance on the benchmarks is given in Table 2. We observe the following: 1) Only very few iterations of the abstraction-refinement loop are required (just 1 in most cases); 2) The abstractions generated are small in most cases; and 3) UCLID yields a speed-up in all but one case when the number of iterations is 1. In the 2 other cases, where some s_i reached the maximum w_i, it performs worse.

[4] Name withheld on their request.

[5] As the run-times on this benchmark is very small, we state them to three decimal places, unlike the others.

We look more closely at two benchmarks. UCLID's performance is orders of magnitude better than the other solvers on the C1-P2 benchmark: this involves multiplication as noted earlier, and the abstraction described in Sect. 3.3 was particularly effective. However, on the benchmark s-40-50, it is 10 times worse than STP, with most of the time spent in encoding. This problem is mainly due to re-generation of the SAT instance in each step, which an incremental implementation can fix.

The results indicate a complementarity amongst the solvers with respect to this set of benchmarks: either bit-blasting (with rewrites as explained in Sect. 3.3) is effective, or the problem is unsatisfiable with a small UNSAT core, or there is a satisfying solution within a small range at the high and low ends of the bit-vector's value domain. In the latter two cases, our abstraction-based approach is effective.

5 Conclusion

We have demonstrated the utility of an abstraction-based approach for deciding the satisfiability of finite-precision bit-vector arithmetic. The speed-ups we have obtained, especially on benchmarks involving non-linear arithmetic operations, indicate the promise of the proposed approach. The algorithm is applicable in many areas in formal verification (e.g., word-level bounded model checking) and can be extended to handle floating-point arithmetic. Ongoing and future work includes generalizing the form of over- and under-approximations beyond those we have proposed herein, and making the encoding to SAT incremental.

References

1. Arons, T., Elster, E., Fix, L., Mador-Haim, S., Mishaeli, M., Shalev, J., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zuck, L.D.: Formal verification of backward compatibility of microcode. In: Proceedings of the Computer-Aided Verification (CAV'05). LNCS, vol. 2404, pp. 185–198 (2005)
2. Babic, D., Spear, F.H.: Proceedings of the SAT 2007 competition (2007)
3. Babić, D., Musuvathi, M.: Modular Arithmetic Decision Procedure. Technical report, Microsoft Research, Redmond (2005)
4. Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Proceedings of DAC'98, pp. 522–527. ACM Press, New York (1998)
5. Biere, A., Cimatti, A., Clarke, E., Yhu, Y.: Symbolic model checking without BDDs. In: TACAS, pp. 193–207 (1999)
6. Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: Proceedings of VLSI Design, pp. 741–746. IEEE (2002)
7. Bryant, R.E.: Term-Level Verification of a Pipelined CISC Microprocessor. Technical Report CMU-CS-05-195, Computer Science Department, Carnegie Mellon University (2005)
8. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07), pp. 358–372 (2007)
9. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: 13th ACM Conference on Computer and Communications Security (CCS '06), pp. 322–335. ACM, New York (2006)
10. Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Proceedings of ASP-DAC 2003, pp. 308–311. IEEE Computer Society Press, Washington (2003)
11. Cook, B., Kroening, D., Sharygina, N.: Cogent: accurate theorem proving for program verification. In: Proceedings of CAV 2005, pp. 296–300. Springer, Berlin (2005)
12. Cyrluk, D., Möller, M.O., Rueß, H.: An efficient decision procedure for the theory of fixed-sized bit-vectors. In: Computer-Aided Verification (CAV '97), pp. 60–71 (1997)
13. Dutertre, B., de Moura, L.: The Yices SMT solver. Available at http://yices.csl.sri.com/tool-paper.pdf (2006)
14. Ganesh, V., Berezin, S., Dill, D.: A decision procedure for fixed-width bit-vectors. Technical Report, Computer Science Department, Stanford University (2005)
15. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Computer Aided Verification (CAV '07), Berlin, Germany, July 2007. Springer, Berlin (2007)
16. Gupta, A., Ganai, M., Yang, Z., Ashar, P.: Iterative abstraction using SAT-based BMC with proof analysis. In: ICCAD (2003)
17. Huang, C.-Y., Cheng, K.-T.: Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques. In: Proceedings of DAC, pp. 118–123 (2000)
18. Kroening, D., Ouaknine, J., Seshia, S., Strichman, O.: Abstraction-based satisfiability solving of Presburger arithmetic. In: Alur R., Peled D. (eds.) Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04). LNCS, vol. 3114, pp. 308–320, Boston, MA, July 2004. Springer, Berlin (2004)
19. Lahiri, S., Mehra, K.: Interpolant Based Decision Procedure for Quantifier-Free Presburger Arithmetic. Technical Report 2005-121, Microsoft Research (2005)
20. McMillan, K., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS'03. Lect. Notes in Comp. Sci., vol. 2619 (2003)
21. MiniSat. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/
22. Moscow, M.L.: http://www.dina.dk/~sestoft/mosml.html
23. Parthasarathy, G., Iyer, M.K., Cheng, K.-T., Wang, L.-C.: An efficient finite-domain constraint solver for circuits. In: Design Automation Conference (DAC), pp. 212–217 (2004)
24. Tseitin, G.: On the complexity of proofs in poropositional logics. In: Siekmann, J., Wrightson, G. (eds.) *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer-Verlag, 1983. Originally published 1970
25. UCLID verification system. http://www.cs.cmu.edu/~uclid
26. Wedler, M., Stoffel, D., Kunz, W.: Normalization at the arithmetic bit level. In: Proceedings of DAC, pp. 457–462. ACM Press, New York (2005)
27. Wisconsin Safety Analyzer Project. http://www.cs.wisc.edu/wisa
28. Xie, Y., Aiken, A.: Scalable error detection using Boolean satisfiability. In: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL), pp. 351–363 (2005)
29. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In: In Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003), S. Margherita Ligure (2003)