A Lazy and Layered SMT($\mathcal{BV}$) Solver for Hard Industrial Verification Problems ⋆

Roberto Bruttomesso[1], Alessandro Cimatti[1], Anders Franzén[1,2], Alberto Griggio[2], Ziyad Hanna[3], Alexander Nadel[3], Amit Palti[3], and Roberto Sebastiani[2]

[1] FBK-irst, Povo, Trento, Italy. {bruttomesso,cimatti,franzen}@itc.it
[2] DIT, Università di Trento, Italy. {griggio,rseba}@dit.unitn.it
[3] Logic and Validation Technologies, Intel Architecture Group of Haifa, Israel.
{ziyad.hanna,alexander.nadel,amit.palti}@intel.com

Abstract. Rarely verification problems originate from bit-level descriptions. Yet, most of the verification technologies are based on *bit blasting*, i.e., reduction to boolean reasoning.
In this paper we advocate reasoning at higher level of abstraction, within the theory of bit vectors ($\mathcal{BV}$), where structural information (e.g. equalities, arithmetic functions) is not blasted into bits. Our approach relies on the *lazy* Satisfiability Modulo Theories (SMT) paradigm. We developed a satisfiability procedure for reasoning about bit vectors that carefully leverages on the power of boolean SAT solver to deal with components that are more naturally "boolean", and activates bit-vector reasoning whenever possible. The procedure has two distinguishing features. First, it relies on the on-line integration of a SAT solver with an incremental and backtrackable solver for $\mathcal{BV}$ that enables dynamical optimization of the reasoning about bit vectors; for instance, this is an improvement over static encoding methods which may generate smaller slices of bit-vector variables. Second, the solver for $\mathcal{BV}$ is *layered* (i.e., it privileges cheaper forms of reasoning), and it is based on a flexible use of term rewriting techniques.
We evaluate our approach on a set of realistic industrial benchmarks, and demonstrate substantial improvements with respect to state-of-the-art boolean satisfiability solvers, as well as other decision procedures for SMT($\mathcal{BV}$).

1 Introduction

Historically, algorithmic verification has been based on efficient reasoning engines, such as Binary Decision Diagrams [7], and more recently on SAT procedures [15], reasoning at the *boolean level*. However, the source of verification problems has increasingly moved from the boolean level to higher levels: most designers work at least at Register Transfer Level (or even higher levels). Thus, the mapping to verification engines is typically based on some form of synthesis to the boolean level. With this process, hereafter referred to as *bit blasting*, boolean representations are generated for

⋆ This work has been partly supported by ORCHID, a project sponsored by Provincia Autonoma di Trento, and by a grant from Intel Corporation. The last author would also like to thank the EU project S3MS "Security of Software and Services for Mobile System" contract n. 27004 for supporting part of his research.

structured constructs (e.g., arithmetic operators), and even simple assignments result in fairly large formulae (e.g., conjunctions of equivalences between the bits in the words).

This impacts verification in several ways. For instance, high-level structural information is not readily available for a solver to exploit (and arithmetic is typically not handled efficiently by boolean reasoners). Furthermore, the hardness of the verification exhibits a dependence of the width of the data path.

The importance of avoiding (or controlling) the use of bit blasting has been strongly advocated by [18], where the theory of bit vectors is identified as a suitable representation formalism for practical industrial problems from many application domains, and the development of effective solvers for SMT($\mathcal{BV}$) is highlighted as an important goal for the research community.

In this paper we take on this challenge and propose a new, scalable approach to SMT($\mathcal{BV}$) based on the *lazy* SMT paradigm. We have developed a satisfiability procedure for reasoning in the theory of bit vectors, that leverages the power of boolean SAT solver to deal with components that are more naturally "boolean", and activates reasoning on bit vectors whenever possible.

The procedure has two distinguishing features. First, it is based on the *lazy* SMT paradigm, that is, it relies on the *on-line* integration of a SAT solver with an *incremental and backtrackable* solver for $\mathcal{BV}$ ($\mathcal{BV}$-solver), that allows us to dynamically optimize the reasoning about bit vectors. For instance, this has the advantage that word chunks are kept as large as possible, since the splitting is carried out according to the control path currently activated; this addresses one of the drawbacks of static encoding methods [4, 2], which may result in an unnecessary slicing of bit vector variables.

Second, the $\mathcal{BV}$-solver makes aggressive use of *layering*, i.e., subsolvers for cheaper theories are invoked first, and more expensive ones are called only when required, and on simplified subproblems. The cheapest levels are implemented by means of flexible use of *term rewriting* techiques.

Our approach also relies on a preprocessor, aiming at simplifying the problem before the search, and on a novel boolean enumeration algorithm for circuits that generates *partial* satisfying assignments.

We evaluate our approach experimentally on a set of realistic industrial benchmarks. We analyze the impact of the proposed optimizations, showing that they all contribute to gaining efficiency. We then compare our solver with several state of the art approaches, including MiniSat 2.0, the winners of the last SMT competition on bit vectors, and BAT [13]. The results indicate that our approach, despite the preliminary status of the implementation, has a great potential: on the hardest instances it is often able to largely outperform the other approaches.

This paper is structured as follows. In §2 we describe the problem of Satisfiability Modulo the theory of bit vectors, and in §3 we describe the previous approaches. In §4 we overview our approach. In §5 we discuss the details of the preprocessing, and in §6 we present the $\mathcal{BV}$-solver. In §7 we experimentally evaluate our approach. Finally, in §8 we draw some conclusions and outline directions for future research.

2 SMT($\mathcal{BV}$): Satisfiability Modulo the Theory of Bit Vectors

A bit vector of a given width *n* is an array of bits and hence it may assume (decimal) values in the range $[0, 2^n - 1]$, accordingly to the binary value expressed by its individual bits. From now on we will implicitly assume bit vector variables to have a fixed width, whose maximal value *N* is determined a priori. In the remainder of the paper we shall use the notation $\mathbf{x}^n$ to represent a bit vector variable of width *n* or simply $\mathbf{x}$ when the width is not important or it can be deduced from the context. Constants will be denoted either with their decimal or binary value (in the latter case a subscript "b" is added).

The easiest possible theory of bit vectors (here denoted as $\mathcal{BV}(\varepsilon)$) includes only bit vector variables, constants, and the equality ($=$) as predicate symbol. Notice that, since we are dealing with fixed-width bit vectors , any interpretation of the theory must satisfy implicit finite domain constraints; for instance it is not possible to satisfy formulae of the kind $\bigwedge_{i=1}^{2^n+1} \bigwedge_{j=i+1}^{2^n+1} \mathbf{x_i}^n \neq \mathbf{x_j}^n$, because only 2^n different values may be represented with *n* bits.

More interesting theories may be obtained with the addition of other operators to $\mathcal{BV}(\varepsilon)$. The most common ones can be divided into three main sets:

core operators $\{[i:j], ::\}$, named selection (or extraction) and concatenation respectively; *i* is the most significant bit in the selection ($i \geq j$). The result of a selection is a bit vector of size $i - j + 1$ whose *k*-th bit is equivalent to the $k + j$-th bit of the selected term, for $k \in [0, i - j + 1]$. Concatenation returns a bit vector resulting from the justapposition of the bits of its arguments;

arithmetic operators (and relations) $\{+, -, *, <\}$, i.e., plus, minus, multiplication by constant, and less than. The intended semantic is the one of arithmetic modulo 2^n, *n* being the width of the arguments of the operators;

bitwise operators { **AND** , **OR** , **NOT** } that apply basic logical functions to correspondent bits of the arguments.

In [9] it is shown a polynomial algorithm to solve $\mathcal{BV}(\varepsilon)$ augmented with core operators ($\mathcal{BV}(\mathbf{C})$). As soon as other operators are added to the theory, either arithmetic ($\mathcal{BV}(\mathbf{CA})$) or bitwise ($\mathcal{BV}(\mathbf{CB})$) or both ($\mathcal{BV}(\mathbf{CAB})$ or $\mathcal{BV}$), the problem of deciding a conjunction of atoms becomes NP-Hard [3]. In the following, a bit vector *term* is defined to be either a constant, a variable, or the application of an operator to a term. A bit vector *atom* is an application of a relation ($=$, $<$) to two terms.

Given a decidable first-order theory $\mathcal{T}$, we call the *decision problem* on $\mathcal{T}$ (DEC($\mathcal{T}$)) the problem of deciding the satisfiability in $\mathcal{T}$ of sets/conjunctions of ground atomic formulas ($\mathcal{T}$ *-atoms*) and their negations in the language of $\mathcal{T}$. We call a $\mathcal{T}$ *-solver* any tool able to decide DEC($\mathcal{T}$). *Satisfiability Modulo (the) Theory* $\mathcal{T}$ (*SMT*($\mathcal{T}$)) is the problem of deciding the satisfiability of *boolean combinations* of propositional atoms and theory atoms. (Consequently, DEC($\mathcal{BV}$) and SMT($\mathcal{BV}$) represent respectively the decision and the SMT problem in $\mathcal{BV}$.) We call an *SMT*($\mathcal{T}$) *solver* any tool able to decide SMT($\mathcal{T}$). Notice that, unlike with DEC($\mathcal{T}$), SMT($\mathcal{T}$) involves handling also boolean connectives.

3 An Analysis of Previous Approaches

In this section we overview and analyze the main approaches for the verification of an RTL circuit design.

Eager encoding into SAT (bit blasting). The traditional approach (*bit blasting*) is that of encoding the problem into a boolean formula, which is then fed to a boolean solver: words are encoded into arrays of boolean atoms, and $\mathcal{BV}$ operators are decomposed into their gate-level representation, each gate being a boolean connective. Pre- and post-processing steps can further enhance the performance (see, e.g., [11, 13, 12, 10]). Notice that the winners of SMT-COMP'06 for SMT($\mathcal{BV}$) were all based on bit blasting.

A variant of this approach is followed in [17, 2]: abstract representations of an RTL circuit are generated by abstracting away information on the data path, and the resulting encoding is then fed into a propositional SAT solver. The approach in [2] is subject to loss of information, and iterative refinement may be required.

Eager encodings into DEC($\mathcal{BV}$). The approaches proposed in [9, 14, 3] encode the problem into a set of atomic formulas in (fragments of) $\mathcal{BV}$, which is fed to a $\mathcal{BV}$-solver. Novel and optimized $\mathcal{BV}$-solvers have been introduced there: particular attention is paid in optimizing the partitioning of bit vectors due to core operators [14] and in handling modulo-arithmetic operators [3].

Eager encodings into DEC($\mathcal{LA}(\mathbb{Z})$). The approaches proposed in [19, 6] encode the problem into a set of literals in the theory of linear arithmetic on the integers ($\mathcal{LA}(\mathbb{Z})$) which is then fed to a $\mathcal{LA}(\mathbb{Z})$-solver: bit-vector variables $\mathbf{x_i}^n$ are encoded as integer variables $x_i \in [0, 2^n - 1]$, RTL constructs [19]) are encoded, into $\mathcal{LA}(\mathbb{Z})$ constraints.

Eager encoding into SMT($\mathcal{LA}(\mathbb{Z})$). The approach we proposed in [4] encodes the problem into an SMT($\mathcal{LA}(\mathbb{Z})$) formula, which is fed to an SMT($\mathcal{LA}(\mathbb{Z})$)-solver. The design is partitioned into *control-path* and *data-path* components: control lines are encoded as boolean atoms, and control constructs into boolean combinations of control variables and predicates over data-path variables; data-path bit-vector variables and linearizable data-path constructs are encoded similarly to the DEC($\mathcal{LA}(\mathbb{Z})$) approach; non-linearizable data-path constructs are encoded by bit-blasting, or by means of uninterpreted functions. Some other constraints are introduced to represent the interface between the control and data-path lines.

Generalizing a standard terminology of the SMT community, we call the approaches above, *eager approaches*, because the encodings into SAT, DEC($\mathcal{BV}$), DEC($\mathcal{LA}(\mathbb{Z})$) and SMT($\mathcal{LA}(\mathbb{Z})$) respectively are performed eagerly at the beginning of the process, before starting any form of search.

We now discuss the above approaches. The key issue of the approaches based on bit-blasting is that they encode bits into boolean *atoms*, and consequently words into arrays of boolean atoms and gates into *boolean connectives*. On the one hand, this allows for a straightforward encoding of all constructs of the $\mathcal{BV}$ language; moreover, as all the search is demanded to an external SAT solver, it allows for selecting the SAT solver off-the-shelf; more importantly, these approaches allows for exploiting the full power of modern boolean solvers in handling the search due to the control logic. On the other hand, a predominant part of the computational effort is wasted in performing

useless boolean search on the bitwise encoding of data-path variables and arithmetical operations (e.g., up to a 2^{32} factor in the amount of boolean search for a 32-bit integer value). In particular, notice that boolean solvers are typically "bad at mathematics", in the sense that reasoning on the boolean encoding of arithmetical operations causes a blowup of the computational effort. To this extent, we say that the bit-blasting approaches are "*control-path oriented*", in the sense that they are well-suited for problems where the control-path component dominates, but may suffer when the data-path component dominates, in particular when lots of arithmetic is involved.

The key issue of the encoding-into-DEC($\mathcal{T}$) approaches is that they encode words into *terms* in some first order theory $\mathcal{T}$ (typically some fragment of either $\mathcal{BV}$ or $\mathcal{LA}(\mathbb{Z})$), and consequently gates and RTL operators into *function symbols* of $\mathcal{T}$. On the one hand, these approaches allows and ad-hoc $\mathcal{T}$-solver for handling each word as a single term in $\mathcal{T}$, preventing the bit-blasting of the world itself and the consequent potential blowup in boolean search; moreover, arithmetic operators can be handled directly and efficiently by an ad-hoc solver. On the other hand, the fact that control bits and gates are encoded into terms and function symbols respectively prevents from exploiting the full power of modern boolean solvers in handling the search due to the control logic. To this extent, we say that the encoding-into-DEC($\mathcal{T}$) approaches are "*data-path oriented*", in the sense that they are well-suited for problems where the data-path part dominates, in particular when lots of arithmetic is involved, but may suffer when the control-path part dominates.

The key issue of the encoding-into-SMT($\mathcal{LA}(\mathbb{Z})$) approach is that control bits and gates are encoded into boolean atoms and connectives respectively, whilst words and RTL operators are encoded into terms, function and predicate symbols in $\mathcal{LA}(\mathbb{Z})$ respectively. Remarkably, some bits may have both a control-path and a data-path role, and have a double encoding. On the one hand, this approach allows for exploiting the power of the boolean solver embedded in the SMT($\mathcal{LA}(\mathbb{Z})$) solver in handling the search due to the control logic, preventing the blowup in boolean search due to the bit-blasting of data-path words.

On the other hand, it suffers from other important weaknesses: first, some constructs (e.g., bitwise operators) cannot be encoded into $\mathcal{LA}(\mathbb{Z})$, and must bit-blasted anyway; second, the $\mathcal{LA}(\mathbb{Z})$ constraints resulting from the encoding of core $\mathcal{BV}$ operations, like selection and concatenation, turns out to be very expensive to handle by $\mathcal{LA}(\mathbb{Z})$-solvers; third, many $\mathcal{LA}(\mathbb{Z})$ constraints resulting from the encoding of some $\mathcal{BV}$ constructs prevent and efficient propagation of integer values and boolean values, corresponding to unit-propagation in the equivalent bit-blasted encoding. (These problems are shared also by the encoding-to-DEC($\mathcal{LA}(\mathbb{Z})$) approach.) Overall, from our experience the approach turned out to be less efficient than expected, mostly due to too many and too expensive calls to $\mathcal{LA}(\mathbb{Z})$-solvers.

We see our encoding-into-SMT($\mathcal{LA}(\mathbb{Z})$) approach of [4] as a first and very preliminary attempt to merge control-path-oriented and data-path-oriented approaches. In next sections we push this idea forward, within the lazy SMT($\mathcal{BV}$) framework.

4 A lazy approach to SMT($\mathcal{BV}$)

Our novel SMT($\mathcal{BV}$) solver is based on the layered lazy approach to SMT($\mathcal{T}$) (see, e.g., [5]). A *preprocessor* takes as input a representation of (the negation of) an RTL verification problem, and produces a simpler and equivalently-satisfiable SMT($\mathcal{BV}$) CNF formula φ. The search is based on the *boolean abstraction* of φ, that is a boolean formula φ^p obtained by substituting every distinct $\mathcal{BV}$-atom in φ with a fresh propositional atom. φ is also called the *refinement* of φ^p. The boolean abstraction φ^p of φ is then fed to a *modified DPLL engine*, which enumerates a complete list $\mu_1^p, ..., \mu_n^p$ of partial truth assignments which satisfy φ^p. Every time a new assignment μ_i^p is generated, the set μ_i of $\mathcal{BV}$ literals corresponding to μ_i^p is fed to a $\mathcal{BV}$*-solver*. If μ is found $\mathcal{BV}$-consistent, then φ is $\mathcal{BV}$-consistent and the whole procedure stops. Otherwise, the $\mathcal{BV}$-solver returns the subset $\eta \subseteq \mu$ which caused the inconsistency of μ (called a *theory conflict set*). The boolean abstraction η^p of η is then used by the DPLL engine to prune the future boolean search (by backjumping and learning [15]). If at the end of the boolean search none of the μ_i's is found $\mathcal{BV}$-consistent, then φ is $\mathcal{BV}$-inconsistent and the whole procedure stops.

In order to increase the efficiency of the $\mathcal{BV}$ reasoning, the $\mathcal{BV}$-solver is organized into three *layers* of increasing expressivity and complexity, s.t. the more expensive layers come into play only when strictly needed [5]. In particular, the DPLL engine invokes the $\mathcal{BV}$-solver also on assignments under construction ("*early pruning*"), which can be pruned if they are found unsatisfiable in $\mathcal{BV}$. As these checks are not necessary for the correctness and completeness of the procedure, in early-pruning calls only the cheaper layers of the $\mathcal{BV}$-solver are invoked. (We omit the description of other SMT optimizations we adopted, which can be found in [5].)
The preprocessor and the $\mathcal{BV}$-Solver are described in details in §5 and §6 respectively.

Notice that, unlike the eager approaches described in §3, our approach is *lazy*, in the sense that the encoding is performed by the $\mathcal{BV}$-solver, *on demand* and *ad hoc* for every branch in the search. Thus, only a strict subset of the $\mathcal{BV}$ atoms are assigned by DPLL and passed to the $\mathcal{BV}$-solver, corresponding to only the sub-circuits that are given an active role by the control variables assigned in the branch. This reduces the computational effort required to the $\mathcal{BV}$-solver, in particular when expensive arithmetical constructs come into play, and addresses one of the major source of inefficiency we encountered with the SMT($\mathcal{LA}(\mathbb{Z})$) approach [4]. Another advantage is that bit-vector chunks are kept as large as possible, since the splitting is carried out according to the control path currently activated; this addresses one of the drawbacks of eager encoding methods [14, 2, 4], which may result in an unnecessary slicing of bit-vector variables.

5 Preprocessing

The schema of the preprocessor is outlined in the left part of Figure 1. It consists mainly on a sequence of six processing steps.

1. Bool to word-1 encoding. The first step addresses the fact that in an RTL circuit there is not always a clear separation between the *data paths* and the *control paths*; in particular, there is no distinction between control lines and word variables of width

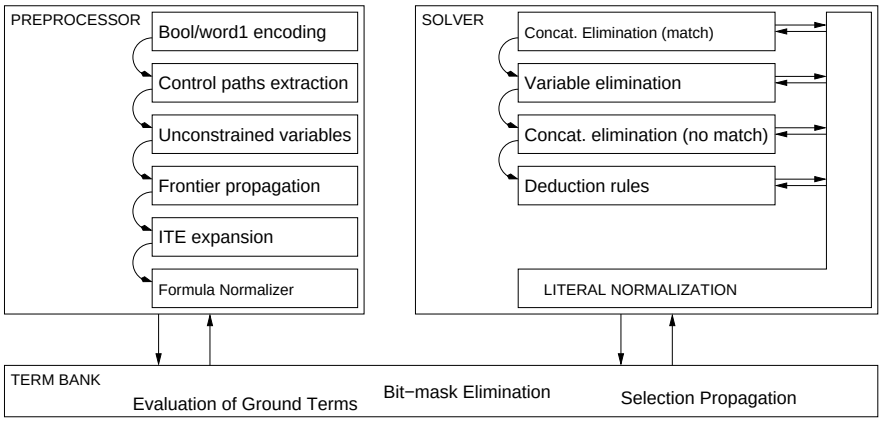

Fig. 1. The architecture of the preprocessor and of the second layer of the solver.

one. This distinction is crucial when our SMT approach is used, because the former and the latter ones must be encoded respectively as propositional *atoms* and as *terms* in the theory [4].

The encoder tags all the nodes of the circuit either as `bool` ("control") or as `word` ("data"): outputs from predicates ($=, <$) and words of width one are tagged `bool`; the result of concatenations or arithmetic operators is instead tagged as `word`. The tagging information is then propagated back and forth to the remaining parts of the circuit. (Bitwise operators with `bool` inputs are converted into boolean operators.) When tag clashes occurs (e.g., a node tagged `bool` is in input to a concatenation), they are resolved with the introduction of a one of two *tag-casting operators*: `word1( b )` translates a `bool` b into a `word` of size 1, and `bool(` $\mathbf{w}^1$ `)`, which casts a `word` $\mathbf{w}^1$ into a `bool`. The former is expanded into $\mathbf{w}^1 \wedge (b \rightarrow \mathbf{w}^1 = \mathbf{1}^1) \wedge (\neg b \rightarrow \mathbf{w}^1 = \mathbf{0}^1)$, the latter into $\mathbf{w}^1 = \mathbf{1}^1$.

2. Control-paths extraction. When `word1(.)` constructs occur in matching positions in an equality, then the equality is split into a conjunction of equalities, and the equalities between word1 variables are transformed into equivalences between booleans. For instance, $(\mathbf{t_1}^7 :: \texttt{word1}(p) :: \mathbf{t_2}^8) = (\mathbf{t_3}^7 :: \texttt{word1}(q) :: \mathbf{t_4}^8)$ is rewritten into the equivalent form $(\mathbf{t_1}^7 = \mathbf{t_3}^7) \wedge (p \leftrightarrow q) \wedge (\mathbf{t_2}^8 = \mathbf{t_4}^8)$.

3. Propagation of unconstrained variables. Industrial benchmarks often contain unconstrained variables (i.e., input variables that occur only once in the formula) used to abstract more complex subparts. An unconstrained variable may assume an arbitrary value, and hence we can rewrite the original formula ϕ into an *equisatisfiable* formula ϕ' using the following rules ($\mathbf{v}$, $\mathbf{v_1}$, and $\mathbf{v_2}$ are unconstrained variables, $\mathbf{f}$ is a fresh variable, p is a fresh propositional variable):

$$\begin{array}{lll} \mathbf{v}^n + \mathbf{t}^n \rightarrow \mathbf{f}^n & \mathbf{v_1}^n :: \mathbf{v_2}^m \rightarrow \mathbf{f}^{n+m} & \mathbf{v}^n = \mathbf{t}^n \rightarrow p \\ \mathbf{t}^n + \mathbf{v}^n \rightarrow \mathbf{f}^n & \mathbf{NOT}\ \mathbf{v}^n \rightarrow \mathbf{f}^n & \mathbf{v}^n < \mathbf{t}^n, t \not\equiv \mathbf{0}^n \rightarrow p \\ \mathbf{v}^n - \mathbf{t}^n \rightarrow \mathbf{f}^n & \mathbf{v_1}^n\ \mathbf{AND}\ \mathbf{v_2}^n \rightarrow \mathbf{f}^n & \mathbf{t}^n < \mathbf{v}^n, t \not\equiv \mathbf{2^n} - \mathbf{1}^n \rightarrow p \\ \mathbf{t}^n - \mathbf{v}^n \rightarrow \mathbf{f}^n & \mathbf{v_1}^n\ \mathbf{OR}\ \mathbf{v_2}^n \rightarrow \mathbf{f}^n & \end{array}$$

4. Frontier Propagation and Variable Inlining. When a boolean formula ϕ is asserted to a truth value, the truth value information may be propagated backward to its subformulae. For instance, if ϕ is a conjunction and ϕ is asserted to true, also its conjuncts must be true. During this process of frontier propagation it is possible to collect every equality between a `word` variable and a constant that has been marked as true and replace every occurrence of the variable with the constant. The whole process is repeated until a fixpoint is reached. (Typically a couple of iterations are enough to reach convergence.)

5. Enhanced Term-ITE expansion. Term-ITE constructs (ITE_t) are not part of the language of SMT($\mathcal{BV}$), and hence they have to be expanded. The naive approach is to introduce a fresh variable for every occurrence of a Term-ITE, and then add two implications to the original formula. For instance, $\mathbf{t_1}^n = \text{ITE}_t(q, \mathbf{t_2}^n, \mathbf{t_3}^n)$ is rewritten into $\mathbf{t_1}^n = \mathbf{f}^n \wedge (q \rightarrow \mathbf{f}^n = \mathbf{t_2}^n) \wedge (\neg q \rightarrow \mathbf{f}^n = \mathbf{t_3}^n)$. When a formula contains a considerable amount of Term-ITEs, the generation of a corresponding number of fresh variables negatively affects performance. In many applications, however, Term-ITE constructs are organized in complex clusters with the structure of a directed acyclic graph. Any maximal cluster in the formula can be transfomed into a correspondent Boolean-ITE (ITE_b) cluster by pushing the external predicate toward the leaves of the DAG. For instance, $\mathbf{t_1}^n = \text{ITE}_t(q_1, \text{ITE}_t(q_2, \mathbf{t_2}^n, \mathbf{t_3}^n), \mathbf{t_4}^n)$ can be rewritten into $\text{ITE}_b(q_1, \text{ITE}_b(q_2, \mathbf{t_1}^n = \mathbf{t_2}^n, \mathbf{t_1}^n = \mathbf{t_3}^n), \mathbf{t_1}^n = \mathbf{t_4}^n)$ saving the introduction of two fresh variables.

6. Normalization. In the language of $\mathcal{BV}$ the problem of transforming a generic bit vector expression into a canonical form is an NP-Hard problem in itself. Weaker, but effective, polynomial transformations on bit vector terms are performed, for instance elimination of concatenation with perfect match: $\mathbf{t_1}^m :: \mathbf{t_2}^n = \mathbf{t_3}^m :: \mathbf{t_4}^n$ is reduced to the conjunction of $\mathbf{t_1}^m = \mathbf{t_3}^m$ and $\mathbf{t_2}^n = \mathbf{t_4}^n$.

During the whole six-step process above, a set of cheap and "local" linear transformations are applied in order to simplify $\mathcal{BV}$ terms.

Evaluation of Ground terms. Whenever a term is composed solely of constants it is replaced by the constant of the appropriate value; similarly for boolean formulas. For example, $0100_b :: 0001_b + 00001001_b$ is evaluated into 01001010_b.

Bit-masks elimination. When a constant occurs in a binary bitwise operation, it is rewritten into concatenations of maximal sequences of 0's and 1's. For example, the constant 00011101_b is split as $000_b :: 111_b :: 0_b :: 1_b$. Then, similar splitting is applied to the other term, and then the operator is evaluated. For instance, $\mathbf{t}^8$ **AND** 00011101_b is rewritten into $000_b :: \mathbf{t}[4:2] :: 0_b :: \mathbf{t}[0:0]$.

Selection propagation. Selection operators are propagated through concatenation and bitwise operators. After this process, only selection on variables, ITE's or arithmetic operators can be left.

These transformations are implemented within the "term bank", a layer that allows for the dynamic creation of new terms, implementing perfect sharing; both the preprocessor and the solver, described in next section, rely on the term bank, and benefit from the transformations above.

6 An Incremental and Layered $\mathcal{BV}$-Solver for SMT($\mathcal{BV}$)

In this section we describe the $\mathcal{BV}$-solver, that decides the consistency of a set of bit vector literals, and in case of inconsistency, it produces a conflict set. The $\mathcal{BV}$-solver is intended to be called on-line, while the boolean search is constructing a boolean model, in order to apply early pruning. For this reason, it is implemented to be *incremental* and *backtrackable*, i.e., it is possible to add and remove constraints without restarting from scratch.

The theory solver is *layered* [5], i.e. it analyzes the problem at hand trying to detect inconsistency in layers of theories of increasing power, so that cheaper layers are called first. The *first layer* is a solver for the logic of Equality of Uninterpreted Functions ($\mathcal{EUF}$). Here all bit vector operators (functions and predicates) are treated as uninterpreted, the finiteness of the domain and codomain of variables and functions is not taken into account; all constants are however treated as distinct. The $\mathcal{EUF}$ solver [16] is incremental, backtrackable, produces conflict sets, and has the capability to deduce unassigned theory literals, which will be propagated to the boolean enumerator. In this layer, conflicts of the type $\mathbf{x} < \mathbf{y}::\mathbf{z}$, $\neg(\mathbf{x} < \mathbf{w})$, $\mathbf{w} = \mathbf{y}::\mathbf{z}$ can be detected.

The *second layer* is an incomplete solver, based on a set of inference rules for bit-vector constraints, that can be applied in an incremental and backtrackable manner. The main idea driving the design of this solver is that a complete solver is very seldom necessary. Thus, a solver based on a small number of inference rules, that can be efficiently implemented, may suffice to decide most formulas.

The *third layer* is a complete solver for conjunctions of bit vector constraints, that ultimately relies on the encoding into $\mathcal{LA}(\mathbf{Z})$ proposed in [6]. In early pruning, the first two, cheaper layers are active; the third, more expensive layer is activated only in complete calls, when a definite answer is necessary, i.e. when a satisfying boolean assignment is being analyzed.

In the rest of this section, we focus on the second layer, which is the most novel component of the solver. The architecture of the second layer is depicted in the right part of Figure 1. The control is organized into a sequence of four main stages, described below. Each of the stages transforms a set of currently active facts, by means of a syntactic inference engine, in an incremental and backtrackable manner.

Similarly to the preprocessor, the solver relies on the term bank, so that whenever a new term is created, the local simplification rules described in §5 are automatically applied. In addition, whenever a new literal is created, a set of normalization rules is used to obtain simpler literals. The rules include a subset of the normalizations applied in the preprocessor which are described in §5; in addition, negated equalities of the form $\neg(\mathbf{t}^1 = \mathbf{1}^1)$ are turned into the positive correspondent $\mathbf{t}^1 = \mathbf{0}^1$. Early termination is enforced upon detection of inconsistency: whenever a literal is reduced to false, the computation is immediately stopped. The stages are the following:

Concatenation Elimination (match). The rule for the elimination of concatenation with perfect match (see Section 5) is applied to all the literals that are amenable for reduction. We notice that the rule does not introduce any selection operator.

Variable Elimination. Whenever a fact of the form $\mathbf{v} = \mathbf{t}$ is active, and $\mathbf{v}$ does not occur in $\mathbf{t}$, then it is removed from the active facts, and every occurrence of $\mathbf{v}$ in the

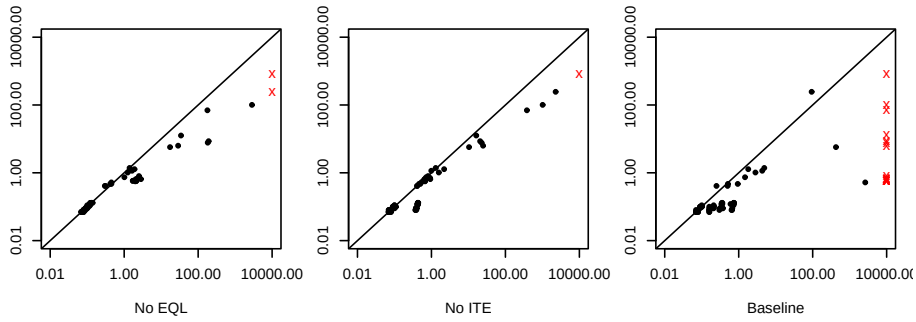

Fig. 2. Comparison between different optimizations. Execution times measured as 0 seconds have been adjusted to 0.01 seconds.

other active facts is replaced by $\mathbf{t}$. During substitution, new terms may be generated in the term bank, so that the corresponding local rules are applied, together with the literal normalization rules.

Concatenation Elimination (no match). All top-level concatenations are simplified, regardless of the match of the sizes. In particular, each active fact having the form $\mathbf{t_1}^m :: \mathbf{t_2}^n = \mathbf{t_3}^{m+n}$ is replaced by $\mathbf{t_1}^m = \mathbf{t_3}[m+n:n]$ and $\mathbf{t_2}^n = \mathbf{t_3}[n-1:0]$. We notice that at this stage concatenations may be replaced by selections, requiring thus the creation of new terms, which in turn fire local rules and literal normalization rules.

Deduction Rules. The final step is the application of the following simplification rules, until fix point is reached (here "$\mathbf{c}$" denotes a constant).

$$\frac{\mathbf{t_1} = \mathbf{t_2} \quad \mathbf{t_2} = \mathbf{t_3}}{\mathbf{t_1} = \mathbf{t_3}}\ Tr1 \qquad \frac{\mathbf{t_1} < \mathbf{t_2} \quad \mathbf{t_2} < \mathbf{t_3}}{\mathbf{t_1} < \mathbf{t_3}}\ Tr2 \qquad \frac{\mathbf{c} < \mathbf{t_1} \quad \mathbf{t_1} < \mathbf{t_2}}{\mathbf{c}+\mathbf{1} < \mathbf{t_2}}\ Tr3 \qquad \frac{A \quad \neg A}{\bot}\ Exc.$$

Some remarks are in order. First, the issue of incrementality and backtrackability poses nontrivial constraints on the implementation of the stages; in particular, variable elimination is not applied destructively, and it is, in the current implementation, the most expensive stage. It is likely that additional efficiency may be achieved by means of optimizations of the underlying data structures. Second, new rules can be plugged in with relatively little effort, possibly in a way that is dependent on the application domain; additional efficiency could be achieved by scoring their activity on the fly, with a mechanism similar to the VSIDS heuristic for SAT.

Finally, a very relevant issue is the generation of informative (i.e., small) conflict sets. Currently, if an inconsistency is detected, the leaves of the proof tree can be taken as a conflict set. However, the conflict sets generated may contain irrelevant literals, depending on the order in which inference rules are applied. Currently, we start from the assumptions of the proof and obtain a smaller conflict sets by means of deletion filtering [8]: one constraint is dropped from the conflict set, and then consistency is checked again. If the set is still inconsistent, the dropped constraint was irrelevant. This is repeated until a fix point is reached, and the remaining set of literals can be used as a conflict set. A method based on proof storing and analysis could be used to improve

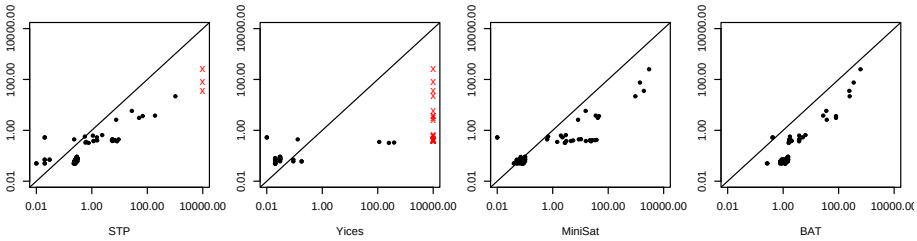

Fig. 3. Scatter plots between MathSAT and the other solvers. Execution times measured as 0 seconds have been adjusted to 0.01 seconds.

the quality of the conflict set; however, an efficient implementation may be nontrivial to achieve, and it is left as the object of future research.

7 Experimental Evaluation

The approach described in previous sections was implemented within the MathSAT system, and was experimentally evaluated on industrial verification problems. In this section we first describe the experimental set up, and report the evaluation in two parts. First we show the impact of the different optimizations. Second, we compare our solver against alternative approaches.

Experimental Set Up We evaluated our approach in a set of industrial benchmarks provided by Intel. Unfortunately, we can not disclose the benchmarks or any details on the original application domain.

For the benefit of the reader, in order to give a feeling of their structural properties we report in a technical report available at [1] for each of the benchmarks the number of: constants, words of size 1, words of size > 1, equalities, core operators, arithmetic operators, bitwise operators, ITEs, boolean connectives. We also report the size of the boolean abstraction of the SMT($\mathcal{BV}$) encoding, and the size of the bitblasted formula.

The experiments were run on an Intel Xeon 3GHz processor running Linux. For each run, the time limit was set to 1 hour and the memory limit was set to 1500 MB.

Evaluation of the Optimizations In this section we compare the impact of eq-layering and enhanced ITE expansion on the overall performance of the solver. We compare the best configuration (the one with all optimizations enabled), against configurations that have a single option disabled at a time, and against the baseline configuration, where no optimization is active. Results are shown in figure 2. The results clearly show that each of the optimizations contributes to improving the performance; we also see that without them the solver is virtually unable to solve any of the interesting instances.

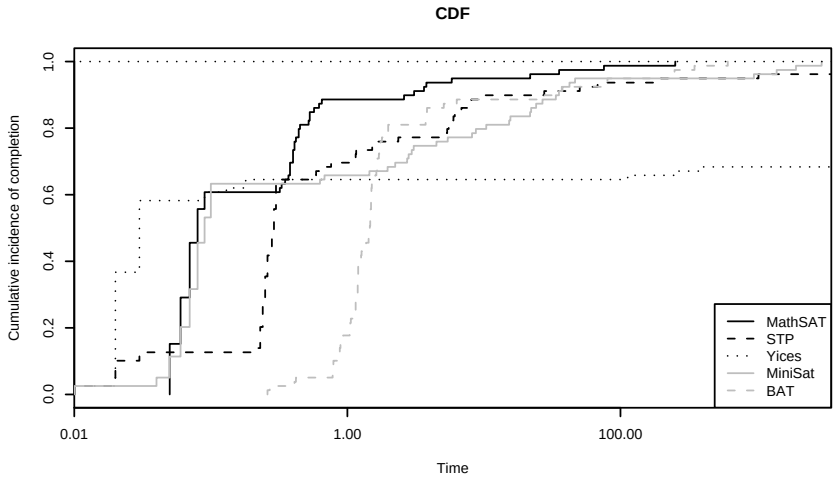

Fig. 4. Cumulative distribution functions (survival plots) comparing the solvers in log scale. Execution times measured as 0 seconds have been adjusted to 0.01 seconds.

Comparison with Other Solvers We compared our approach against the following systems.

- MiniSat 2.0, considered as the best system based on bit blasting; We are using the SatELite-like simplifying version as it performed better on these instances.
- STP, as one of the winners of the bit-vector division of the latest SMT-COMP competition; STP extends bit blasting with a normalization/preprocessing step, built on top of an incremental SAT engine. We are using a recent version provided by the authors of STP.
- Yices 1.0.3, as the other winner of the latest SMT-COMP in the bit-vector division. To the best of our knowledge, the solver for bit vector reasoning in Yices is based on bit blasting [10].
- BAT 0.2 is a recently-released system that specializes in structured, modular problems with memories. It combines a clever encoding of memory term rewriting techniques and reduction to SAT.

The problems were given in input to STP and Yices in SMT format. As for MiniSat, we generated a DIMACS file both before and after the preprocessing, and we used the one that resulted in better performance, i.e., the one before preprocessing; surprisingly, our preprocessor degraded the performance of MiniSat significantly.

Scatter plots of the execution times can be seen in figure 3. As can be seen from the figure, MathSAT clearly outperforms other solvers on the majority of instances. The notable exception is BAT which is comparable on the two hardest, although slightly slower (see Table 1 for the precise time taken).

Instance	MathSAT		STP		Yices		MiniSat		BAT	
Intel-35	3	3	72	50	>18000	>18000	42	19	81	80
Intel-37	4	3	84	68	>18000	>18000	47	17	80	80
Intel-39	4	4	133	189	>18000	>18000	35	38	28	28
Intel-76	256	76	>18000	>18000	>18000	>18000	1393	8658	344	348
Intel-77	29	22	2687	1781	>18000	>18000	948	559	245	248
Intel-78	580	252	8451	9699	>18000	>18000	2973	2199	611	611
Intel-79	62	36	>18000	9968	>18000	>18000	1929	8065	240	242

Table 1. Execution times for the hardest instances. All times are rounded to the nearest second.

From the cumulative distribution functions in figure 4 the percentiles can be read. On the easiest instances Yices is fast, but there are 25 instances it cannot solve within the time limit. MathSAT, on the other hand, is clearly superior approximately above the 60th percentile.

We now focus on the "hardest" instances, i.e. those instances where at least two systems used an execution time greater than 60 seconds. In particular, we extended the execution time limit to 5 hours. In addition, we experimented with two different translations from the source file format. (For each sample in figures 3 and 4 the translation corresponding to the best performance for each solver has been used.) The results are reported in Table 1: for each system, we report the result for both encodings. We see that MathSAT outperforms the other solvers, regardless of translation. We also notice that the translation schema may induce substantial differences in performance, in particular for STP and MiniSat: with the second translation, STP solves one more instance within the extended time limit, whereas MiniSat performs considerably worse on two instances. Yices is not able to solve any of these benchmarks within the time with either translation, while the performance of BAT is remarkably stable.

8 Conclusions and Future Work

The work described in this paper is motivated by the fact that many verification problems, especially in industrial settings, are naturally described at a level of abstraction that is higher than boolean – the additional structure is typically used to describe data paths.

We have developed a new decision procedure for Satisfiability Modulo the Theory of fixed-width bit vectors. The procedure is tailored towards hard industrial problems, and has two distinguishing features. First, it is lazy in that it invokes a solver on the theory of bit vectors *on the fly* during the search. Second, it is layered, i.e. it tries to apply incomplete but cheap forms of reasoning (e.g. equality and uninterpreted functions, term rewriting), and deal with complete solvers only when required. As a result structural information is used to significantly speed up the search, without incurring in a substantial penalty with reasoning about purely boolean parts. In an empirical evaluation performed on industrial problems, our solver outperforms state-of-the-art competitor systems by an order of magnitude on many instances.

In the future, we plan to work on the following problems. First, the current implementation can be heavily optimized; in particular the generation of conflict sets is currently an issue. Second, we plan to investigate the application of advanced theorem proving techniques. We would also like to experiment with abstraction refinement techniques, and to integrate the solver within a CEGAR loop based on the NuSMV model checker.

References

1. http://mathsat.itc.it/cav07-bitvectors/.
2. Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In *Proc. DAC '04*. ACM Press, 2004.
3. Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A Decision Procedure for Bit-Vector Arithmetic. In *Design Automation Conference*, pages 522–527, 1998.
4. M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding RTL Constructs for MathSAT: a Preliminary Report. In *Proc. PDPAR'05*, volume 144 (2) of *ENTCS*. Elsevier, 2006.
5. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P.van Rossum, S. Schulz, and R. Sebastiani. MathSAT: A Tight Integration of SAT and Mathematical Decision Procedure. *Journal of Automated Reasoning*, 35(1-3), October 2005.
6. R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. ASP-DAC 2002*, pages 741–746. IEEE, 2002.
7. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
8. John W. Chinneck and Erik W. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.
9. D. Cyrluk, O. Möller, and H. Rueß. An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In *Proc. CAV'97*, volume 1254. Springer, 1997.
10. B. Dutertre and L. de Moura. System Description: Yices 1.0. In *Proc. SMT-COMP'06*, 2006.
11. V. Ganesh, S. Berezin, and D. L. Dill. A Decision Procedure for Fixed-width Bit-vectors. Technical report, Stanford University, 2005. http://theory.stanford.edu/~vganesh/.
12. P. Johannsen and R. Drechsler. Speeding Up Verification of RTL Designs by Computing One-to-one Abstractions with Reduced Signal Widths. In *VLSI-SOC*, 2001.
13. P. Manolios, S.K. Srinivasan, and D. Vroon. Automatic Memory Reductions for RTL-Level Verification. In *Proc. ICCAD 2006*. ACM Press, 2006.
14. M. O. Möller and H. Ruess. Solving bit-vector equations. In *Proc. FMCAD'98*, 1998.
15. M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.
16. R. Nieuwenhuis and A. Oliveras. Congruence closure with integer offsets. In *Proc. LPAR'03*, volume 2850 of *LNAI*. Springer, 2003.
17. S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions. In *Proc. DAC'03*, 2003.
18. Eli Singerman. Challenges in making decision procedures applicable to industry. In *Proc. PDPAR'05*, volume 144 (2) of *ENTCS*. Elsevier, 2006.
19. Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: a unified approach to RTL satisfiability. In *Proc. DATE '01*. IEEE Press, 2001.