Report

Linear Arithmetic with Bit-Vectors using Omega and SAT

Author(s):
Kröning, Daniel

Publication Date:
2005

Permanent Link:
https://doi.org/10.3929/ethz-a-006787534

Rights / License:
In Copyright - Non-Commercial Use Permitted
Linear Arithmetic with Bit-Vectors
using Omega and SAT

Daniel Kroening
Computer Systems Institute
ETH Zürich

Abstract. Formal analysis tools for system-level software rely on solving decision problems over bit-vector arithmetic. The common approach to decide these problems is to transform the constraints into a corresponding circuit at the net-list level, and then transform this net-list into CNF. The CNF is passed to a modern SAT-solver. The word-level structure of the original problem is lost. The Omega test is a decision procedure for a conjunction of linear constraints over integers. It is used for variable dependency analysis within compilers. In the hardware domain, a linearization of bit-vector operators has been applied successfully to data-paths. This paper proposes a variant of such an encoding to solve bit-vector arithmetic decision problems arising in software verification. A SAT solver is used for the case-splitting. We present preliminary experimental results comparing the new algorithm with the commonly used approach mentioned above.

1 Introduction

Decision procedures are the workhorse of most formal verification tools for system-level software. We will describe two examples: Bounded Model Checking and Predicate Abstraction.

In the hardware industry, formal verification is well established. Introduced in 1981, Model Checking [1, 2] is one of the most commonly used formal verification techniques in a commercial setting. However, it suffers from the state explosion problem. In case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [3].

This problem is partly addressed by a formal verification technique called Bounded Model Checking (BMC) [4]. In BMC, the transition relation for a complex design and its specification are jointly unwound to obtain a formula, which is then checked for satisfiability. This process terminates when the length of the potential counterexample exceeds its completeness threshold (i.e., is sufficiently long to ensure that no counterexample exists [5]) or when the SAT procedure exceeds its time or memory bounds. BMC has been used successfully to find subtle errors in very large industrial circuits [6, 7].

When applied to a circuit given as a net-list, BMC generates formula in propositional logic, which can be checked by using a propositional SAT procedure such as Chaff [8]. However, the higher, RT-level information about data-words is completely flattened and lost. Applying BMC to a RT-level circuit, e.g., given in Verilog, results
in a formula in bit-vector logic, containing variables with bit-vector domain and bit-vector operators such as addition or bit-wise XOR.

BMC has recently been adopted to the verification of system-level software as well. CBMC [9] unwinds sequential ANSI-C programs, flattens the resulting bit-vector logic formula, and passes the resulting propositional formula to a SAT-solver. TCBMC is a version of CBMC extended to support threaded programs [10]. Saturn [11] and F-SOFT [12] implement similar algorithms. An application of BMC to web applications is found in [13].

The disadvantage of BMC is that it is typically only applicable for refutation; the completeness threshold is too large for most practical instances. The principal method for proving properties is abstraction. Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system.

 Predicate abstraction [14, 15] is one of the most popular and widely applied methods for systematic abstraction of programs. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state system that is an abstraction of the original system with respect to a set of predicates.

The abstraction refinement process using predicate abstraction has been promoted by the success of the SLAM [16–22] project at Microsoft Research. One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the abstract-verify-refine paradigm and depend on the abstraction and refinement techniques used.

Abstract interpretation [23] is a very general framework to reason about transition systems. ASTREE implements static program analysis [24] using abstract interpretation and widening. It automatically refines abstractions of programs in order to prove the specification. However, if the proof fails, no simulation step is attempted, and thus, the algorithm may generate false alarms.

The workhorse of BMC, predicate abstraction, and abstract interpretation is the reasoning engine, which decides validity of formulae in a suitably chosen logic.

Existing Approaches Almost all program verification engines, such as symbolic model checkers and advanced static checking tools, employ automatic theorem provers for symbolic reasoning. For example, the static checkers ESCJAVA [25] and BOOOGIE [26] use the Simplify [27] theorem prover to verify user-supplied invariants.

The SLAM software model-checker uses ZAPATO [28] for symbolic simulation of C programs. The BLAST [29] and MAGIC [30] tools use Simplify for abstraction, simulation and refinement. Further decision procedures used in program verification are CVC-Lite [31], ICS [32] and Verifun [33].

As motivated in [34], these theorem provers are bad fit for program analysis, especially when applied to low-level software. They are optimized for mathematical theories, such as linear arithmetic over the integers. However, programs in languages
such as Java, C or C++ require reasoning for bounded-width bit-vector arithmetic that takes overflow into account, and bit-wise operators.

In [35], we proposed the use of propositional SAT-solvers as a reasoning engine for the verification of low-level software. The astonishing progress SAT solvers made in the past few years is the enabling technology for this approach. As in BMC, the arithmetic operators in the formula are replaced by corresponding circuits. The resulting net-list is converted into CNF and passed to a SAT solver. This allows supporting all operators as defined in the ANSI-C standard.

In [34], we report experimental results that quantify the impact of replacing ZAPATO, a decision procedure for integers, with Cogent, a decision procedure built using a SAT solver: The increased precision of Cogent improves the performance of SLAM, while the support for bit-level operators resulted in the discovery of a previously unknown bug in a Windows device driver.

However, the use of solver for linear arithmetic for program verification has a solid motivation: when transforming the arithmetic operators into circuits, the variables are split into individual bits and the word-level information is lost. Encoding an addition in propositional logic results in one XOR per bit, which are chained together through the carry bit. It is known that such XOR chains can result in very hard SAT instances. As a result, there are many programs and circuits that cannot be verified by means of a bit-level SAT solver.

**Related Work** Decision procedures for bit-vector arithmetic have been found in tools such as SVC and ICS for years. Initial work used BDDs in order to represent the arithmetic operators. SVC has been superseded by CVC, and then CVC-Lite [31].

The circuit-based translation into propositional logic is still state-of-the-art for deciding validity of formulae in a logic supporting bit-vector operators. It is implemented by Cogent and the current version of CVC-Lite. ICS is still using BDDs to reason about this logic.

The related work is mostly to be found in the hardware verification domain. Wedler et al. normalize bit-vector formulae in order to simplify the generated SAT instance in [36]. Word-level reasoning using a decision procedure such as the Omega test or the like is typically not employed.

One exception is Brinkmann and Drechsler [37], who use an encoding of linear bit-vector arithmetic into ILP in order to decide properties of circuit data-paths given at the RT-level. The Omega test is used as a decision procedure for the ILP instance. However, [37] only aims at the data-paths, and thus, does not allow a Boolean part within the original formula. This is mended by [38] using a lazy encoding with a modified DPLL search.

**Contribution** We propose a three-phase eager propositional SAT-encoding to solve bit-vector arithmetic formulae arising in software verification. 1) In the first phase, we add constraints for the word-level linear arithmetic in the formula using a variant of an encoding used in the hardware domain as described in [37]. The modified encoding enables the Omega test to take bit-vector overflow into account. 2) In the second phase, we add the constraints for the bit-wise operators, using a standard circuit encoding. 3)
In the third phase, we join the two logics by adding constraints on the bits of the word-
level variables used in phase 1). We present experimental results that show that this
new approach can solve instances that the bit-level SAT solvers cannot handle.

Outline In section 2, we provide background information about lazy and eager encod-
ings of decision problems and about the Omega test. We describe the modifications we
make to the Omega test in section 3. Experimental results are reported in section 4.

2 Background

2.1 Bit-Vector Arithmetic

The subset of bit-vector arithmetic we consider is defined by the language \( L_B \) according to the following grammar:

\[
\begin{align*}
\text{formula} & : \text{formula} \lor \text{formula} \mid \text{formula} \land \text{formula} \mid \neg \text{formula} \mid \text{atom} \\
\text{atom} & : \text{term} \ \text{rel} \ \text{term} \mid \text{Boolean-Identifier} \\
\text{rel} & : = \mid \neq \mid \leq \mid \geq \mid < \mid > \\
\text{term} & : \text{term} \ \text{op} \ \text{term} \mid \text{identifier} \mid \sim \ \text{term} \mid \text{constant} \mid \text{atom} ? \ \text{term} : \text{term} \\
\text{op} & : \oplus \mid \ominus \mid \otimes \mid \oslash \mid \ll \mid \gg \mid \& \mid | \mid ^
\end{align*}
\]

With each expression, we associate a type, which is its width in bits and whether
it is signed (two’s complement encoding) or unsigned (binary encoding). Assigning
semantics to this language is language is straight-forward, e.g., as done in [37].

We do not consider bit-extraction and concatenation operators. However, adding
these operators as part of the bit-wise operators is a simple extension. We use the
ANSI-C symbols to denote the bit-wise operators, e.g., \& denotes bit-wise AND, while
\(^\) denotes bit-wise XOR. The trinary operator \( c \? a : b \) is a case-split: the operator
evaluates to \( a \) if \( c \) holds, and to \( b \) otherwise.

We use \( \oplus \) to distinguish addition on bit-vectors with modular arithmetic from addition
on unbounded integers. Note that the relational operators \( \succ, \prec, \preceq, \succeq \), the
multiplicative operators \( \otimes, \oslash \) 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.

Following the notation in [37], we add an index to the operator and operands in
order to denote the bit-width. As an example, \( a_{[32]} \otimes_{[32]} b_{[32]} \) denotes the 32-bit mul-
tiplication of \( a \) and \( b \). Both the result and the operands are 32 bits wide, the remaining
32 bits of the result are discarded.

Example 1. As a motivating example, the following formula obviously holds on the
integers:

\[
(x - y > 0) \iff (x > y)
\]

However, if \( x \) and \( y \) are interpreted as bit-vectors, this equivalence no longer holds,
due to possible overflow on the subtraction operation.
**Definition 1.** Propositional Encoding The Propositional Encoding $\phi_{\text{enc}}$ of a bit-vector formula $\phi^B$ is obtained by replacing all atoms that are not Boolean identifiers by new fresh Boolean identifiers $e_1, \ldots, e_\nu$. The atom replaced by $e_i$ is denoted by $A(e_i)$. The set of all atoms in $\phi^B$ that are not Boolean identifiers are denoted by $A(\phi^B)$.

As an example, the propositional encoding of $\phi^B = (x = y) \land ((a \oplus b = c) \lor (x \neq y))$ is $e_1 \land (e_2 \lor \neg e_1)$, and $A(\phi^B) = \{x = y, a \oplus b = c\}$.

We denote the vector of the variables $e_1, \ldots, e_\nu$ by $e$. Furthermore, let $\psi_a(e)$ denote the atom $a$ with polarity $e$:

$$\psi_a(e) := \begin{cases} a : e \\ \neg a : \text{otherwise} \end{cases} \quad (2)$$

**Definition 2.** Linear Bit-Vector Arithmetic. A term in $L_B$ which uses only constants on the right hand side of the binary bit-wise and the multiplicative and shifting operators is called linear. We denote the linear atoms of $\phi^B$ by $A_L(\phi^B)$, and the remaining atoms (the non-linear atoms) by $A_N(\phi^B)$.

## 2.2 Encoding Decision Problems into Propositional Logic

**Lazy vs. Eager Encodings** SAT solvers have become an integral part of all modern decision procedures. There are two different ways to compute an encoding of a decision problem $\phi$ into propositional logic. In both cases, the propositional part $\phi_{\text{enc}}$ of the formula is converted into CNF first. Linear-time algorithms for computing CNF for $\phi_{\text{enc}}$ are well-known [39]. The algorithms differ in how the non-propositional part is handled.

The vector of variables $\tau : A(\phi) \rightarrow \{T,F\}$ as defined above denotes a truth assignment to the atoms in $\phi$. Let $\Psi_{A(\phi)}(\tau)$ denote the conjunction of the atoms $a_i \in A(\phi)$ where the $a_i$ are in the polarity given by $\psi(a_i)$:

$$\Psi_{A(\phi)}(\tau) := \bigwedge_{i=1}^\nu \psi_{a_i}(e_i) \quad (3)$$

An Eager Encoding considers all possible truth assignments $\tau$ before invoking the SAT solver, and computes a Boolean constraint $\phi_E(\tau)$ such that

$$\phi_E(\tau) \iff \Psi_{A(\phi)}(\tau) \quad (4)$$

The number of cases considered while building $\phi_E$ can often be dramatically reduced by exploiting the polarity information of $a$, i.e., whether $a$ appears in negated form or without negation in the negation normal form (NNF) of $\phi$. After computing $\phi_E$, $\phi_E$ is conjoined with $\phi_{\text{enc}}$, and passed to a SAT solver. A prominent example of a decision procedure implemented using an eager encoding is UCLID [40].

A Lazy Encoding means that a series of encodings $\phi^1_L, \phi^2_L$ and so on with $\phi \implies \phi^i_L$ is built. Most tools implementing a lazy encoding start off with $\phi^1_L = \phi_{\text{enc}}$. In each iteration, $\phi^i_L$ is passed to the SAT solver. If the SAT solver determines $\phi^i_L$ to be
unsatisfiable, so is \( \phi \). If the SAT solver determines \( \phi_L^i \) to be satisfiable, it also provides a satisfying assignment, and thus, an assignment \( \overline{e} \) to \( A(\phi) \).

The algorithm proceeds by checking if \( \Psi_{A(\phi)}(\overline{e}) \) is satisfiable. If so, \( \phi \) is satisfiable, and the algorithm terminates. If not so, a subset of the atoms \( A' \subseteq A(\phi) \) is determined, which is already unsatisfiable under \( \overline{e} \). The algorithm builds a blocking clause \( b \), which prohibits this truth assignment to \( A' \). The next encoding \( \phi_{L+1}^i \) is \( \phi_L^i \land b \). Since the formula becomes only stronger, the algorithm can be tightly integrated into one SAT-solver run, which preserves the learning done in prior iterations.

Among others, CVC-Lite [31] implements a lazy encoding of integer linear arithmetic. The decision problem for the conjunction \( \Psi_{A(\phi)}(\overline{e}) \) is solved using the Omega test, which is described in the next section.

### 2.3 The Omega Test

The Omega test was introduced by Pugh [41] and is intended to be used in an optimizing compiler. It is an algorithm to decide satisfiability of a conjunction of linear constraints over unbounded integer variables. Each conjunct is assumed to be either an equality of the form

\[
\sum_{i=1}^{n} a_i x_i = b
\]

or a weak inequality of the form

\[
\sum_{i=1}^{n} a_i x_i \leq b.
\]

The Omega test is a variant of the Fourier-Motzkin variable elimination algorithm. As in the case of Fourier-Motzkin, equality and inequality constraints are treated separately; any equality constraints are removed before inequalities are considered.

In order to eliminate an equality of the form of Equation 5, the algorithm first checks if there is a variable \( x_j \) with coefficient 1 or -1, i.e., \( |a_j| = 1 \). In this case, Equation 5 is solved for \( x_j \), and each occurrence of the variable \( x_j \) is substituted in all constraints.

If there is no variable with coefficient 1 or -1, the Omega test picks the variable that has the coefficient with the smallest absolute value. Without loss of generality, let \( x_n \) be that variable.

As next step, a new variable \( \sigma \) is created, and a constraint for \( \sigma \) is added such that an integer term for \( x_n \) can be computed. In the original constraint, the absolute value of the coefficient of \( \sigma \) is the same as the absolute value of the original coefficient \( a_n \), and it seems that nothing has been gained by this substitution. However, the absolute values of the other coefficients are reduced by this operation. Repeated application of the equality elimination will eventually generate a coefficient of 1 or -1 on some variable. This variable can then be eliminated directly. The details of this step can be found in [41].
Once all equalities have been eliminated, the algorithm attempts to find a solution for the remaining inequalities. As done by Fourier-Motzkin, the Omega test uses projections to decide satisfiability. A heuristic is used to pick a variable to eliminate. The constraints over this variable are then projected on the remaining constraints.

Assume that the variable to be eliminated is denoted by \( z \). As in the case of Fourier-Motzkin, all pairs of lower and upper bounds on \( z \) are considered. Variables that are not bounded both ways can be removed together with all constraints that contain them.

Even though the Omega test is concerned with constraints over integers, the first step is to check the real shadow, i.e., if there are solutions over the reals. If there is no solution for the constraint system over reals, then there is no solution over integers either, and the algorithm concludes that the system is unsatisfiable. Let \( \beta \leq bz \) be a lower bound on \( z \), and let \( az \leq \alpha \) be an upper bound on \( z \), where \( a \) and \( b \) are positive integer constants and \( \alpha \) and \( \beta \) denote the remaining partial sums. The new constraint is obtained by multiplying the lower bound by \( a \), and the upper bound by \( b \):

<table>
<thead>
<tr>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \beta \leq bz )</td>
<td>( az \leq \alpha )</td>
</tr>
<tr>
<td>( a\beta \leq abz )</td>
<td>( abz \leq b\alpha )</td>
</tr>
</tbody>
</table>

The existence of such a \( z \) implies

\[
a\beta \leq b\alpha. \tag{7}
\]

The real shadow is an over-approximating projection, as it contains more solutions than the original problem. The next step in the Omega algorithm is to compute an under-approximating projection, i.e., if it contains an integer solution, so does the original problem. This projection is called the dark shadow.

If the dark shadow is empty, we cannot conclude that the original problem has no solution as the dark shadow is an under-approximation. The Omega test searches the remaining area between the real and the dark shadow for solutions. This step may be very expensive, but rarely happens in practice. Again, we refer the reader to [41] for the details. It is important to note that the Omega test as described in [41] is much more efficient at handling equalities than at handling inequalities.

An Eager SAT-encoding for Integer Linear Arithmetic: Strichman describes an eager SAT-encoding of linear arithmetic on both real numbers and integers in [42]. The algorithm avoids enumerating the truth assignments \( \overline{\mathcal{F}} \). Instead, all atoms \( \mathcal{A}(\phi) \) are passed to the Omega test completely disregarding the Boolean structure of \( \phi \), i.e., as if they were conjoined.

However, once the Omega test finds two constraints \( C_1 \) and \( C_2 \) that contradict each other, the algorithm cannot conclude that \( \phi \) is unsatisfiable, as the Boolean structure was disregarded. Instead, the algorithm identifies the subset of the atoms \( \mathcal{A}(\phi) \) that was used to derive \( C_1 \) and \( C_2 \), and the polarities of the atoms. It then adds a constraint to \( \phi_E \) that eliminates this truth assignment.
3 The Omega Test for Bit-Vectors

3.1 Case-Splitting for Overflow

The Omega test as introduced in [41] and as implemented in CVC-Lite is for unbounded integers. For the analysis of software and hardware we aim at a decision procedure for bit-vectors. In this section, we describe how to modify the algorithm to handle linear bit-vector equations.

Let \( a \) be a linear atom. As preparation, we perform a number of transformations:

- Let \( b \gg d \) denote a bit-wise right shift in \( a \), where \( b \) is a term and \( d \) is a constant. It is replaced by \( b/2^d \). Bit-wise left shifts are handled in a similar manner. Bit-wise negation of \( b \) is replaced by \( -b + 1 \). Bit-wise AND is replaced by corresponding right-shifts and addition, bit-wise OR using bit-wise negation and bit-wise AND. We are left with addition, subtraction, multiplication, and division.

- As next step, the constraints are normalized, which removes division operators. As an example, the constraint \( a \oslash 32 = b \) becomes \( a = b \otimes 33 \). Note that the bit-width of the multiplication has to be increased in order to take overflow into account. The operands \( a \) and \( b \) are sign-extended.

After this preparation, we can assume the following form of the atoms without loss of generality:

\[
c_1 t_1 \oplus_w c_2 t_2 \text{ op } b \tag{8}
\]

where \( \text{op} \) is one of the relational operators as defined in section 2.1, \( c_1, c_2, \) and \( b \) are constants, and \( t_1 \) and \( t_2 \) are bit-vector identifiers with \( w \) bits. Bit-vector overflow may occur on the scalar multiplication and on the bit-vector addition.

As we can handle additions efficiently, scalar multiplications \( ca_{[w]} \) with a small constant \( c \) are replaced by \( c \) additions: \( 3a \) becomes \( a + a + a \). As this is ineffective for large coefficients, we use the same encoding as proposed in [37] in this case. The scalar multiplication is replaced by \( ca - 2^w \sigma \) under the following constraints:

\[
cia - 2^w \sigma \leq 2^w - 1 \land \sigma \leq c - 1 \tag{9}
\]

After this transformation, we are left with the bit-vector additions:

\[
t_1 \oplus_w t_2 \text{ op } b \tag{10}
\]

If the constraints are passed in this form to the Omega test, the potential overflow on the \( w \)-bit bit-vector addition is disregarded. Assuming \( t_1 \) and \( t_2 \) are \( w \)-bit unsigned vectors, \( t_1 \in \{0, \ldots, 2^w - 1\} \) and \( t_2 \in \{0, \ldots, 2^w - 1\} \), and thus, \( t_1 + t_2 \in \{0, \ldots, 2^{w+1} - 2\} \). We use a case-split to adjust the value of the sum in case of an overflow. We transform Equation 10 into:

\[
(t_1 + t_2 \leq 2^w - 1)?t_1 + t_2 : (t_1 + t_2 - 2^w) \text{ op } b \tag{11}
\]

The Omega test does not handle the resulting case-splits itself efficiently; but SAT solvers have been tuned for many years for this task. Thus, these case-splits should be
lifted up to the propositional level by introducing an additional propositional variable $p$:

$$p \iff (t_1 + t_2 \leq 2^w - 1) \quad (12)$$

$$p \implies (t_1 + t_2) \text{ opt } b \quad (13)$$

$$\neg p \implies (t_1 + t_2 - 2^w) \text{ opt } b \quad (14)$$

Thus, the price payed for the bit-vector semantics is two additional integer constraints for each bit-vector addition in the original problem. This is a commonly used approach to convert the trinary if operator, e.g., implemented in UCLID.

Note that the linearization scheme used in [37] and [38] introduces a fresh integer variable $\sigma$ for additions as well. A modern SAT-solver can deal very well with hundreds of thousands of propositional variables, whereas the complexity of the Omega test prohibits adding many integer variables.

Also note that the encoding in [38] transforms all equalities into two inequalities ($\leq$ and $\geq$). This can be avoided by exploiting the polarity of the equality atom $a$: if $a$ is only used without negation in the NNF, the equality can be preserved. The Omega test handles equalities in a much more efficient manner than it can handle inequalities. Exploiting this polarity information is commonly done in tools that use an eager encoding, such as UCLID. In [42], a stronger method called conjunction matrices is used for this purpose.

Formally, let $\neg A$ denote the set of atoms $a$ with $\neg a \in A$. If $a$ is an equality of the form $x = y$ with $x, y$ terms, we split the equality in two inequalities, and thus, $\{x = y\} \subseteq A \iff \{x > y, x < y\} \subseteq \neg A$.

Let $A^p_L \subseteq A_L$ denote the subset of the linear atoms that are only used in positive polarity in the NNF, and $A^n_L$ the subset of the linear atoms that are only used in negative polarity in the NNF. Let $A^*_L = A_L \setminus A^p_L \setminus A^n_L$ denote the set of atoms used in both polarities.

The final set $F$ of atoms passed to the Omega algorithm is the union of the set of atoms only used in positive polarity, plus the set of the negation of the atoms only used in negative polarity, and the set of the remaining atoms both in their original and their negated form:

$$F := A^p_L \cup \neg A^p_L \cup A^n_L \cup A^*_L$$

After applying these transformations, we pass the resulting decision problem to a procedure that computes an eager propositional encoding using the Omega test as described in [42]. Let $\phi_L$ denote these constraints.

### 3.2 Combining the Logics

The transformation described above handles linear bit-vector formulas only. The translation schemes in [37] and [38] have the same restrictions. While this is an important class, we expect even simple programs to contain at least a small number of non-linear terms. In most program analysis frameworks, operators not handled by the underlying decision procedure are usually modeled by means of uninterpreted functions.
However, in some cases, the correctness claim actually depends on arithmetic properties of such operators. We therefore propose a three-phase translation that allows to exploit the word-level information for the linear parts for better performance, while still supporting any non-linear operators if needed for the claim.

1. In the first phase, we pass $\phi_{\text{enc}} \land \phi_L$ to the SAT solver. If the instance is unsatisfiable, so is the original problem and the algorithm terminates.
2. In the second phase, we add the non-linear constraints to the problem. Let $\phi_N$ denote the corresponding constraints. We use the standard encoding using arithmetic circuits, and thus, besides shifting with variable shift distance, even multiplication of two variables and division is allowed. We pass $\phi_{\text{enc}} \land \phi_L \land \phi_N$ to the SAT solver. If the instance is unsatisfiable, so is the original problem and the algorithm terminates.
3. In the third phase, the two encodings are connected: we analyze the satisfying assignment obtained from phase two, and add constraints on the bits that differ in the encodings using bit-wise left shifts and bit-wise AND. This phase is repeated until either all inconsistencies are eliminated (and thus, a satisfying assignment is found for the original problem) or the instance becomes unsatisfiable.
Note that the formula passed to the SAT solver only gets stronger with each phase and each iteration of the third phase. Thus, using an incremental SAT procedure is beneficial. No clause removal between instances is needed.

4 Experimental Results

We initially implemented a lazy-case splitting approach similar to the one implemented in [38]. However, initial experimental results showed that most combinations of truth assignments to the overflow case-literals result in satisfying assignments. Subsequently added blocking clauses do not reduce this number, and thus, the run-time is exponential in the number of additions found in the formula.

We therefore implemented the decision procedure described above with an eager encoding. It has been integrated into CBMC. We compare the run-time of this new algorithm, called "OMEGA-BV", with the performance of an encoding of the bit-vectors into arithmetic circuits at the net-list-level, denoted by "net-list".

We experiment with bit-vector decision problems generated by CBMC \(^1\). Even when applied to small C-programs, we sometimes observe excessive run-times using the "net-list" encoding. The most trivial example we could identify exposing this behavior is shown Figure 2.

Note that these instances are rare in practice; CBMC has been applied to programs with 50,000 LOC. The program instances used for experimentation in this paper have been collected among a large number of examples for the purpose of generating hard SAT-instances to serve as a benchmark for SAT-solvers. None of the benchmarks encodes hard, known NP-complete problems or operators that are known to be hard for SAT, such as multiplication.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Result</th>
<th>Net-list</th>
<th>OMEGA-BV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Variables</td>
<td>Clauses</td>
<td>Run-time</td>
</tr>
<tr>
<td>C1</td>
<td>UNSAT</td>
<td>11700</td>
<td>39651</td>
</tr>
<tr>
<td>C2</td>
<td>SAT</td>
<td>34058</td>
<td>107338</td>
</tr>
<tr>
<td>C3</td>
<td>UNSAT</td>
<td>18970</td>
<td>63121</td>
</tr>
<tr>
<td>C4</td>
<td>UNSAT</td>
<td>46455</td>
<td>160545</td>
</tr>
<tr>
<td>C5</td>
<td>UNSAT</td>
<td>46734</td>
<td>161503</td>
</tr>
<tr>
<td>C7</td>
<td>UNSAT</td>
<td>17250</td>
<td>58990</td>
</tr>
</tbody>
</table>

Table 1. Summary of results: the columns under "Net-list" show the number of variables and clauses and the run-time in seconds on an Intel Xenon Processor with 2.8 GHz. A star denotes that the one hour timeout was exceeded.

The experiments have been performed on an Intel Xenon Processor with 2.8 GHz and using ZChaff 2003. The results are summarized in table 1. The experiments show

\(^1\) CBMC is available for experimentation by other researchers at http://www.cs.cmu.edu/~modelcheck/cbmc/
int main()
{
    int i;
    unsigned x, y;
    x = nondet_uint();
    y = x - 100;
    for (i = 0; i < 100; i++)
        y++;
    assert(x == y);
}

Fig. 2. Trivial C program: unwinding the for loop results in a bit-vector equation with 203 equalities. The run-time of the arithmetic circuit encoding and ZChaff is exponential. It is solved in linear time using the proposed algorithm.

that in cases in which SAT fails to solve the decision problem, a word-level decision procedure can be an alternative. Note that on most practical instances, SAT and the circuit encoding perform well, and the word-level decision procedure is not competitive. However, SAT solvers have been tuned for many years, while the Omega integer decision procedure is not yet geared towards large problem sizes.

As ZChaff is no longer the fastest SAT-solver available, we also experimented with Berkmin and Limmat, but obtained comparable results. The run-time of the word-level encoding algorithm is dominated by the run-time of the Omega test during the encoding. The run-time of the SAT-solver on the SAT-instances generated is negligible.

5 Conclusion

Decision procedures for bit-vector arithmetic are an important research area in hardware verification. This paper adopts results from hardware verification to the verification of low-level software programs. This paper modifies a linearization scheme for bit-vector arithmetic presented in [37]; by using propositional variables instead of additional integer variables, we push the additional complexity caused by the case-splitting required for the handling overflow case from the Omega test to the SAT-solver. As SAT-solvers can handle a much larger number of variables, this promises to improve the performance.

Other optimizations include the use of an eager encoding, which promotes learning and pruning on the arithmetic part, and of phase detection, which allows exploiting efficient algorithms for handling equalities. Preliminary experimental results show that the new algorithm can solve instances that are too hard for the commonly applied approach based on a circuit-transformation of the arithmetic operators. Also, in contrast to earlier work, we are able to combine constraints that are encoded using integer arithmetic with constraints that are encoded using arithmetic circuits.
Future Work  The experimental results are preliminary; the new algorithm is not yet well optimized. In particular, the current procedure has two bottle-necks:

1. The Omega test during the computation of $\phi_L$, if many integer variables are used in inequalities,
2. Hard operators such as multiplication in the non-linear part that is encoded as circuit.

As future work, we want to investigate the use of SAT-solvers within the Omega test itself to address the first problem, and the use of word-level decision procedures for non-linear bit-vector arithmetic.

References


31. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating
validity checker. In CAV 04: International Conference on Computer-Aided Verification,
2004.
32. Jean-Christophe Filliatre, Sam Owre, Harald Rue, and N. Shankar. ICS: Integrated can-
onizer and solver. In CAV 01: International Conference on Computer-Aided Verification,
33. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using
lazy proof explication. In CAV 03: International Conference on Computer-Aided Verifi-
34. Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving
35. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C pro-
2004.
36. Markus Wedler, Dominik Stoffel, and Wolfgang Kunz. Normalization at the arithmetic bit
37. R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear program-
constraint solver for circuits. In Design Automation Conference (DAC), pages 212–217,
2004.
40. Randal E. Bryant, Shuvendu K Lahiri, and Sanjit A. Seshia. Modeling and verifying sys-
tems using a logic of counter arithmetic with lambda expressions and uninterpreted func-
41. William Pugh. The Omega test: a fast and practical integer programming algorithm for
42. Ofer Strichman. On solving presburger and linear arithmetic with SAT. In Mark Aagaard
and John W. O’Leary, editors, Formal Methods in Computer-Aided Design (FMCAD), vol-