# Technical Report
# A Decision Procedure for Fixed-Width Bit-Vectors

Vijay Ganesh, Sergey Berezin and David L. Dill
Computer Science Department, Stanford University
{vganesh,berezin,dill}@stanford.edu

9th April 2005

**Abstract**

We report the design, implementation and performance of an efficient decision procedure for the equational theory of fixed-width bit-vectors. The input language supports word-level bit-vector operations (concatenation and extraction), bit-vector arithmetic operations (addition, subtraction and constant multiplication), bitwise boolean operations (conjunction, disjunction, negation, bitwise XOR, etc.), multiplexors (if-then-else operator) and predicates like comparators ("less than"). Other common functions such as right shift, sign/zero extension can be easily supported through suitable translation.

The decision procedure is implemented as part of the CVC Lite tool [BB04], a theorem prover based on combination of decision procedures in the Nelson-Oppen style. The design is novel, the decision procedure complete, and the implementation is efficient for a large class of practical examples. Our implementation also supports concrete counterexample generation.

## 1 Introduction

Efficient implementations of decision procedures for the equational theory of fixed-width bit-vectors are crucial for the formal verification of hardware and certain software systems. Also, users of formal verification tools want support for an input language which has a rich mixture of word-level and bitwise operations together with bit-vector linear arithmetic. Since many problems in verification tend to span several theories, it is also very desirable to have the decision procedure implemented as part of a combination tool such as CVC Lite [BB04], a theorem prover based on combination of decision procedures in the Nelson-Oppen style.

The theory of fixed-width bit-vectors is an equational theory over finite nonempty strings over $\{0, 1\}$, whose length is known and fixed *a priori*. The operations are *concatenation*, *extraction of bit strings*, *bitwise Boolean operations* (conjunction, disjunction, negation), *bit-vector arithmetic operations* (addition and multiplication), and *multiplexors* (if-then-else expressions). The formulas of this theory are Boolean combinations of equalities over bit-vector expressions.

The decision problem for this theory is known to be NP-hard [Möl98]. Many approaches for deciding various subsets of this theory have been proposed in the past [Möl98, CMR97, BP98, FDK98, ZKC01, KM01, BDL98, MMZ$^+$01]. They can be broadly classified into three categories.

In the first category, the input formula is translated into a SAT problem and/or linear arithmetic [FDK98, ZKC01, MMZ$^+$01]. The primary drawback of these approaches is that they destroy the original structure of the problem, and often expand the word-level operations into individual bits, resulting in an expensive search.

Approaches in the second category rely on *canonizing* bit-vector terms, thus exploiting the original structure of the input. Many approaches in the second category are based on the Shostak-style combination of decision procedures [CMR97, BP98, BDL98], which requires a complete canonizer and a solver. Alternatively, efficient canonical data structures like BDDs or BMDs can be used to represent bit-vector terms. However, for an input language as rich as ours, computing canonical form for a bit-vector term is an NP-hard problem in itself, and is impractical in most cases. Moreover, extending a theory with additional operators becomes rather difficult, since a new canonical form and a new solver algorithm need to be designed every time.

Another approach is based on automata representing bit-vector values. This has been described in connection with the MONA tool [EKM98], a decision procedure for WS1S or weak monadic second order logic with one successor [Möl98, KM01]. Given a bit-vector equation, an equivalent WS1S-formula is generated, and a corresponding *correlated automata* is constructed by MONA. If the automata accepts all strings over $\{0, 1\}$, then the original bit-vector equation is valid, otherwise it is invalid. It has been noted that this approach is infeasible for real-world examples due to the high complexity of deciding WS1S [Möl98].

In our experience, among the approaches mentioned above, translation to SAT is still the most practical and efficient, both in generality of the input language and performance. Therefore, the challenge for us is to demonstrate that our decision procedure is competitive in the majority of cases, and significantly better in some cases of interest, compared to the SAT-based method using the state-of-the-art SAT solvers such as Chaff [MMZ$^+$01].

The main contribution of this work is a collection of practical design principles and a concrete implementation of a new efficient decision procedure for a theory of fixed-width bit-vectors. Another contribution is that the decision procedure is implemented as part of CVC Lite, a Nelson-Oppen combination framework. It is important to emphasize that the decision procedure is efficient and works on a very rich set of bit-vector operations (mixture of word-level, bit-wise, and arithmetic operators).

# 2 Contributions

More specifically, the contributions of this paper are:

1. An efficient decision procedure is presented here, which is a SAT solver based algorithm with additional preprocessing steps consisting of *efficient polynomial-time normalization* and *equality rewrites.* The normalization step helps to detect equalities among terms through the word-level algebraic properties of bit-vector operators. The normalizations are further aided by propagation of bit-vector equalities derived from the input bit-vector formula. This sometimes completely solves the original problem, and in most other cases significantly simplifies the task of the SAT solver. Notice that the normalizations do not always yield a canonical form (which is NP-hard to compute for the input language we support), providing a good balance between their efficiency and the amount of reduction they achieve.

2. Another contribution is that the decision procedure has been added as a component to CVC Lite, a Nelson-Oppen combination framework. It allows us to support multiplexors (ITE terms), quantifiers over bit-vector variables, generate proofs and concrete counter-examples, and decide formulas over many theories. Being part of a Nelson-Oppen combination additionally requires the decision procedure to detect and report all equalities over certain terms (*shared constants* from the purification step) [NO79, Bar03]. In particular, it explicitly has to constrain every bit of shared bit-vector terms to be either 0 or 1. This requirement and the techniques to satisfy the same are explained in more detail in section 4.6.1.

# 3 Preliminaries

This section describes the logic of the fixed-width bit-vector theory, its signature $\Sigma$, as well as well-formed terms and formulas over $\Sigma$. The logic of the theory is many-sorted logic (MSL), and hence all symbols, terms and formulas are decorated, i.e. carry their sorts.

The theory of fixed-width bit-vectors considered here is an equational theory over finite non-empty strings of *bits* ($\{0, 1\}$) whose length is known and fixed *a priori.* The rightmost bit of a bit-vector of length $n$ is called the least significant bit (LSB) and the leftmost bit is called the most significant bit (MSB). The bits are ordered from the LSB to the MSB, with the index of the LSB being 0 and the index of the MSB being $n - 1$.

## 3.1 Signature

The signature $\Sigma = \langle \mathbf{F}, \mathbf{C}, \mathbf{S} \rangle$ of the fixed-width bit-vector theory is as follows:

**Sorts:** $\mathbf{S}$ is the set $\{\mathrm{BV}(1), \mathrm{BV}(2) \ldots\}$ of sort symbols, where BV(n) is the sort of a bit-vector of length $n \in \mathbb{N}^+$.

**Functions:** **F** denotes the following family of function symbols:

$$F = \left\{ @_{[n]},\ [i:j]_{[n]},\ +_{[n]},\ *_{[n]},\ \sim_{[n]},\ \&_{[n]},\ |_{[n]}, \mathrm{BVLT}, \mathrm{BVLE} \right\}$$

where the symbol $@_{[n]}$ stands for concatenation, $[i:j]_{[n]}$ for extraction, $+_{[n]}$ for bit-vector addition, $*_{[n]}$ for bit-vector multiplication, $\sim_{[n]}$ for bitwise negation, $\&_{[n]}$ for bitwise conjunction and $|_{[n]}$ for bitwise disjunction. The symbols $[i:j]_{[n]}$ and $\sim_{[n]}$ are unary, and $*_{[n]}$ is binary and the rest are $m$-ary for $m \geq 2$. The subscript $n > 0$ denotes the number of bits in the return sort of the function. The subscript is dropped, if it is clear from the context. The symbols BVLT and BVLE denote "bitvector less than" and "less than or equal to" predicates. We also natively support BVXOR, BVXNOR, BVNAND, BVNOR, but drop these functions from the discussion below.

**Constants:** **C** is the following set of finite non-empty strings over the alphabet $\{0,1\}$,

$$\{0, 1, 00, \ldots\}$$

where the rightmost bit is the least significant bit. For example, 1100 is a 4-bit bit-vector constant representing the positive integer 12. A $n$-bit string containing only 0 (similarly 1) is written as $0_{[n]}$ (similarly $1_{[n]}$).

## 3.2 Terms and formulas

Terms are usually denoted by $t_{[n]}, t^1_{[n]}, t^2_{[m]}, \ldots q_{[m]}, \ldots$, where the subscripts are the lengths of the bit-vector term and the superscripts provide an enumeration of the terms. Variables are denoted by $x_{[n]}, \ldots, y_{[n]}, \ldots$ and bit-vector constants are denoted by $c_{[n]}, \ldots$.

**Term:** A $\Sigma$-term $t_{[n]}$ is one of the following

$$\begin{aligned} t_{[n]} ::= \quad & c_{[n]} \;\Big|\; x_{[n]} \;\Big|\; t^1_{[q]}[i:j] \;\Big|\; t^1_{[i_1]} @ t^2_{[i_2]} @ \cdots @ t^m_{[i_m]} \;\Big|\; \sim t^1_{[n]} \\ & \Big|\; t^1_{[n]} \,\&\ldots\&\, t^m_{[n]} \;\Big|\; t^1_{[n]} \,|\ldots|\, t^m_{[n]} \\ & \Big|\; t^1_{[i_1]} +_{[n]} \cdots +_{[n]} t^m_{[i_m]} \;\Big|\; c_{[i_1]} *_{[n]} t^2_{[i_2]} \\ & \Big| t^1_{[i_1]} -_{[n]} t^2_{[i_2]} \Big| - t^1_{[n]} \end{aligned}$$

where the following conditions hold: For the concatenation term $n = i_1 + i_2 + \ldots + i_m$. For the extraction term $t^1_{[q]}[i:j]$, the length $q$ of $t^1_{[q]}$ must be greater than $n$, the number of bits in the resultant bit-vector. Also, the indices must be such that $n > i \geq j \geq 0$, given the right to left ordering of the bits, where $n = i - j + 1$.

**Atom:** A $\Sigma$-atom $a$ is of the form

$$a \quad ::= \quad t^1_{[n]} \approx t^2_{[n]} \mid \mathrm{BVLT}(t^1_{[i_1]}, t^2_{[i_2]}) \mid \mathrm{BVLE}(t^1_{[i_1]}, t^2_{[i_2]})$$

The binary symbol $\approx$ is used for the logical equality over terms, where the left hand side and right hand side bit-vector terms must be of the same length.

**Formula:** A $\Sigma$-formula $\varphi$ is one of the following:

$$\varphi ::= a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \ \mid \ \varphi_1 \vee \varphi_2.$$

Informally, an *interpretation* of a $\Sigma$-formula $\varphi$ can be defined as follows: The symbols in $\Sigma$ are interpreted in the intuitive way, and each variable is mapped to a suitable non-empty finite string of the appropriate length. A formula $\varphi$ is said to be *valid*, if it is *true* under all interpretations, and *invalid* otherwise.

# 4 The Decision Procedure

Our decision procedure is a SAT-solver based algorithm with additional preprocessing steps consisting of *efficient polynomial time normalization.* These normalizations are further aided by propagation of equalities. Such propagation of equalities is referred to as *equality rewrites* in the rest of the paper.

This work is based on two observations. First, SAT based methods for deciding bit-vector theories are still the most efficient and most general. This is due to the fact that a lot of good research has been done in tuning SAT solvers, and effective translations from bit-vector domain to the Boolean domain exist. The second observation is that there is lot of structure in bit-vector terms that can be exploited. Efficient normalization (preprocessing) of bit-vector terms over $\Sigma$, if done right, can be very helpful in detecting validities cheaply. The normalization step exploits the word-level algebraic properties of bit-vector operators. This sometimes completely solves the original problem, and in most other cases significantly simplifies the task of the SAT solver. Another key observation is that propagating equalities during the normalization step can simplify terms even further.

Computing canonical form is another way to exploit these algebraic properties. However, computing canonical forms for terms over $\Sigma$ is known to be not effective in practice [BDL98]. Moreover, an approach based on computing canonical forms cannot be extended easily. In other words, if new functions or predicates are added to $\Sigma$, then defining a new canonical form for the resultant signature maybe difficult or impossible.

Normalization, as opposed to canonization, provides a good balance between efficiency and the amount of simplification of terms.

Decision procedures in CVC Lite are required to be implemented in a *proof-rule style.* Such a style requires that the procedure be composed of a *strategy* and a set of proof rules. The proof rules are required to be in a variant of the natural deduction style of proof system. The choice of proof rules is left to the implementor. However, we recommed that the rules encode the simplest possible transformations, may not contain loops, and the soundness of the rules must be easy to check. The strategy is a function which in turn calls these transformations (proof rules) depending on the form or type or top-level operator of the formula input to the decision procedure. All the normalization transformations are implemented as proof rules, and are described below.

## 4.1 Normal Form

For the sake of clarity and the ease of implementation, the bit-vector operators from the signature $\Sigma$ are separated into two groups, with the arithmetic operators in $\Sigma_a = \{+_{[n]}, *_{[n]}, C\}$ and the remaining in $\Sigma_c = \{[i:j]_{[n]}, @_{[n]}, \sim_{[n]}$ , $\&_{[n]}, |_{[n]}, C\}$. Given normalizers for $\Sigma_a$ and $\Sigma_c$ terms, the normal form for an arbitrary bit-vector formula $\varphi$ is computed as follows. First, $\varphi$ is *purified* into $\Sigma_a$ and $\Sigma_c$-formulas $\varphi_a$ and $\varphi_c$ respectively, similar to the purification step in the Nelson-Oppen combination procedure [NO79, TH96]. Then the terms in $\varphi_a$ and $\varphi_c$ are converted into normal form by the corresponding normalizers.

### 4.1.1 Concatenation Normal Form

The concatenation normal form for $\Sigma_c$-terms is our extension of a well-known normal form to bit-wise operators. The original normal form for concatenation and extraction was first reported in [BP98].

A $\Sigma_c$-term $t_{[n]}$ is in concatenation normal form if it is constructed in the following way:

$$\begin{array}{rcl} q_{[n]} & ::= & c_{[n]} \mid x_{[n]} \mid x_{[m]}[i:j] \\ s_{[n]} & ::= & q_{[n]} \mid\sim q_{[n]} \mid (q^1_{[n]} \circ \cdots \circ q^m_{[n]}) \\ t_{[n]} & ::= & s_{[n]} \mid (s^1_{[i_1]} @ \cdots @ s^m_{[i_m]}) \end{array}$$

where $\circ \in \{\&, |\}$ and $q^1_{[n]} \prec q^2_{[n]} \prec \cdots \prec q^m_{[n]}$ for some fixed strict total ordering $\prec$ over bit-vector terms. The ordering ensures that if two bit-wise expressions can be reduced to the same expression using only commutativity, associativity, and idempotency ($q \circ q = q$), then they will be normalized to the same expression.

All the terms $q^i_{[n]}$ are required to be different, and there may not be repeated occurrences of the same variable in the term. Also, if a constant exists among $q^1_{[n]}, \ldots, q^m_{[n]}$, then it must be $q^1_{[n]}$, and $q^1_{[n]}$cannot be either of $0_{[n]}$ or $1_{[n]}$.

The adjacent terms among $s^1_{[i_1]}, \ldots, s^m_{[i_m]}$ in the concatenation must be such that they cannot be *merged*. Intuitively, two adjacent terms $s_1$ and $s_2$ in $s_1 @ s_2$ can be merged only if they are both constants, or they are merge-able extractions over the same term, i.e. $s_1 = x[k : i+1]$ and $s_2 = x[i : j]$ for some variable $x$, such that $s_1 @ s_2$ normalizes into $x[k : j]$. Otherwise $s_1$ and $s_2$ cannot be merged.

It can be shown that computing the concatenation normal form is poly-time.

### 4.1.2 Proof Rules for Concatenation Normal Form

In this section we describe the normalizer $\sigma$ for concatenation normal form. It takes a term as input and returns a term. The invariant assumed by $\sigma$ is that the immediate subexpression of the top-level operator are already in concatenation normal form.

- Base case rules:

$$\begin{array}{lcl}
\sigma(c_{[n]}) & = & c_{[n]} \\
\sigma(x_{[n]}) & = & x_{[n]}
\end{array}$$

- Rules for extraction: In the following we assume $n > i \geq j \geq 0$

$$\begin{array}{lcl}
\sigma(c_{[n]}[i:j]) & = & c'_{[i-j+1]} \\
 & & \text{where} \quad \text{c}' \, \text{is the constant} \, = \\
 & & \text{bits of c from position i down to j} \\
\sigma(t_{[n]}[n-1:0]) & = & \sigma(t_{[n]}) \\
\sigma(t_{[n]}[i:j][k:l]) & = & \sigma(t_{[n]}[k+j:l+j]) \\
 & & \text{if} \quad n > i \geq j \geq 0, \, i-j+1 > k \geq l \geq 0 \\
\sigma((t_{[n]}@u_{[m]})[i:j]) & = & \sigma(u_{[m]}[i:j]) \\
 & & \text{if} \quad m > i \geq j \geq 0 \\
\sigma((t_{[n]}@u_{[m]})[i:j]) & = & \sigma(t_{[n]}[i-m:j-m]) \\
 & & \text{if} \quad n+m > i \geq j \geq m \\
\sigma((t_{[n]}@u_{[m]})[i:j]) & = & \sigma(t_{[n]}[i-m:0])@\sigma(u_{[m]}[m-1:j]) \\
 & & \text{if} \quad i \geq m \geq j \geq 0 \\
\sigma((t^1_{[n]}\&t^2_{[n]})[i:j]) & = & \sigma(t^1_{[n]}[i:j]) \,\&\, \sigma(t^2_{[n]}[i:j]) \\
\sigma((\sim t_{[n]})[i:j]) & = & \sigma(\sim (t_{[n]}[i:j])) \\
\sigma((t^1_{[n]} \mid t^2_{[n]})[i:j]) & = & \sigma(t^1_{[n]}[i:j]) \mid \sigma(t^2_{[n]}[i:j]) \\
\sigma((t^1_{[n]} \,\hat{}\, t^2_{[n]})[i:j]) & = & (\sigma(t^1_{[n]}[i:j]) \,\hat{}\, \sigma(t^2_{[n]}[i:j]))
\end{array}$$

- Rules for bitwise NEGATION: In the following we assume $n > i \geq j \geq 0$

$$\begin{array}{lcl}
\sigma(\sim c_{[n]}) & = & c'_{[n]} \\
 & & \text{where} \quad c'_{[n]} \, \text{is the bitwise neg of} \, \text{c}_{\text{[n]}} \\
\sigma(\sim (t^1_{[n]}@t^2_{[m]})) & = & \sigma(\sim t^1_{[n]})@\sigma(\sim t^2_{[m]}) \\
\sigma(\sim (\sim t_{[n]})) & = & t_{[n]}
\end{array}$$

- Rules for bitwise AND:

$$\begin{array}{lcl}
\sigma(0bin0_{[n]} \,\&\, t_{[n]}) & = & 0bin0_{[n]} \\
\sigma(0bin1_{[n]} \,\&\, t_{[n]}) & = & t_{[n]} \\
\sigma(t_{[n]} \,\&\, t_{[n]}) & = & t_{[n]} \\
\sigma(t_{[n]} \,\&\, \sim t_{[n]}) & = & 0bin0_{[n]} \\
\sigma(0bin0_{[n-m]}0bin1_{[m]} \,\&\, t_{[n]}) & = & 0bin0_{[n-m]}@t_{[n]}[m-1:0] \\
\sigma(0bin1_{[n-m]}0bin0_{[m]} \,\&\, t_{[n]}) & = & t_{[n]}[m-1:0]@0bin0_{[n-m]} \\
\sigma(c^1_{[n]} \,\&\, c^2_{[n]}) & = & c'_{[n]} \\
 & & \text{where} \quad c'_{[n]} = \text{bitwise AND of } \mathrm{c}^1_{[\mathrm{n}]} \text{and } \mathrm{c}^2_{[\mathrm{n}]} \\
\sigma(t^2_{[n]} \,\&\, t^1_{[n]}) & = & \sigma(t^1_{[n]} \,\&\, t^2_{[n]}) \\
 & & \text{where} \quad \mathrm{t}^1_{[\mathrm{n}]} \text{ is lexicographically smaller than } \mathrm{t}^2_{[\mathrm{n}]} \\
\sigma((t^1_{[i_1]}@\ldots@t^m_{[i_m]}) \,\&\, q_{[k]}) & = & (\sigma(t^1_{[i_1]} \,\&\, q_{[k]}[k-1:k-i_1]) \quad @\ldots \\
 & & @\,\sigma(t^m_{[i_m]} \,\&\, q_{[k]}[i_m-1:0])) \\
 & & \text{where } \mathrm{k} = \mathrm{i}_1 + \ldots + \mathrm{i}_\mathrm{m} \\
 & & \text{similar rule for } \sigma(\mathrm{q}_{[\mathrm{k}]} \,\&\, (\mathrm{t}^1_{[\mathrm{i}_1]}@\ldots@\mathrm{t}^\mathrm{m}_{[\mathrm{i}_\mathrm{m}]}))
\end{array}$$

- Rules for bitwise OR:

$$\begin{array}{lcl}
\sigma(0bin0_{[n]} \mid t_{[n]}) & = & t_{[n]} \\
\sigma(0bin1_{[n]} \mid t_{[n]}) & = & 0bin1_{[n]} \\
\sigma(t_{[n]} \mid t_{[n]}) & = & t_{[n]} \\
\sigma(t_{[n]} \mid\sim t_{[n]}) & = & 0bin1_{[n]} \\
\sigma(0bin0_{[n-m]}0bin1_{[m]} \mid t_{[n]}) & = & t_{[n]}[n-1:m]@0bin1_{[m]} \\
\sigma(0bin1_{[n-m]}0bin0_{[m]} \mid t_{[n]}) & = & 0bin1_{[n-m]}@t_{[n]}[m-1:0] \\
\sigma(c^1_{[n]} \mid c^2_{[n]}) & = & c'_{[n]} \\
 & & \qquad\quad c'_{[n]} = \text{bitwise OR of } \mathrm{c}^1_{[\mathrm{n}]} \text{and } \mathrm{c}^2_{[\mathrm{n}]} \\
\sigma(t^2_{[n]} \mid t^1_{[n]}) & = & \sigma(t^1_{[n]} \mid t^2_{[n]}) \\
 & & \text{where} \quad \mathrm{t}^1_{[\mathrm{n}]} \text{ is lexicographically smaller than } \mathrm{t}^2_{[\mathrm{n}]} \\
\sigma((t^1_{[i_1]}@\ldots@t^m_{[i_m]}) \mid q_{[k]}) & = & (\sigma(t^1_{[i_1]} \mid q_{[k]}[k-1:k-i_1]) \quad @\ldots \\
 & & @\,\sigma(t^m_{[i_m]} \mid q_{[k]}[i_m-1:0])) \\
 & & \text{where } \mathrm{k} = \mathrm{i}_1 + \ldots + \mathrm{i}_\mathrm{m} \\
 & & \text{similar rule for } \sigma(\mathrm{q}_{[\mathrm{k}]} \mid (\mathrm{t}^1_{[\mathrm{i}_1]}@\ldots@\mathrm{t}^\mathrm{m}_{[\mathrm{i}_\mathrm{m}]}))
\end{array}$$

- Rules for concatenation:

$$\begin{array}{lcl}
\sigma(c^1_{[n]}@c^2_{[m]}) & = & c'_{[n+m]} \\
 & & \qquad\qquad \text{where } \mathrm{c}' \text{ is the constant } = \\
 & & \qquad\qquad \text{bits}(\mathrm{c}^1) \text{ followed by bits}(\mathrm{c}^2) \\
\sigma(t_{[n]}[i:j]@t_{[n]}[j-1:k]) & = & \sigma(t_{[n]}[i:k]) \\
 & & \qquad\qquad \text{where} \quad n > i \geq j \geq k \geq 0 \\
\sigma(t^1_{[n]}@t^2_{[m]}) & = & \text{flatten}(t^1_{[n]}@t^2_{[m]})
\end{array}$$

- Rules for the flatten function are:

$$\begin{array}{l}
\text{flatten}((t^1_{[i_1]}@\ldots@t^m_{[i_m]}) \quad @ \\
\qquad\qquad (w^1_{[j_1]}@\ldots@w^k_{[i_k]})) \quad = \quad (t^1_{[i_1]}@\ldots@t^m_{[i_m]}@w^1_{[j_1]}@\ldots@w^k_{[i_k]})
\end{array}$$

where $t_{[n]}, q_{[m]}, t^1_{[i_1]} \ldots w^1_{[j_1]}, \ldots$ are simple terms.

For all other well-formed terms not mentioned above, we have $\sigma(t) = t$.

### 4.1.3 Complexity

We show that the time complexity of $\sigma$ is polynomial in the size of the input. The size of a term is the sum of the following:

- Total number of occurences of all function symbol in the term
- Total number of occurences of all variables in the term
- $log_2(n)$ for every occurence of an integer, $n$,
- the number of bits in every bit-vector constant that occurs in the term.

Recall that an invariant adhered to by the normalizer $\sigma$ is that the sub-terms are already in normal form. Consequently, it is easy to check that the size of the outputs of the above rules are polynomial in the input size, except for the rule below:

$$\begin{array}{lcl} \sigma((t^1_{[i_1]} @ \ldots @ t^m_{[i_m]}) \circ q_{[k]}) & = & (\sigma(t^1_{[i_1]} \circ q_{[k]}[k-1 : k-i_1]) \quad @ \ldots \\ & & @\, \sigma(t^m_{[i_m]} \circ q_{[k]}[i_m - 1 : 0])) \\ & & \text{where } \mathrm{k} = \mathrm{i}_1 + \ldots + \mathrm{i}_\mathrm{m} \\ & & \text{similar rule for } \sigma(\mathrm{q}_{[\mathrm{k}]} \circ (\mathrm{t}^1_{[\mathrm{i}_1]} @ \ldots @ \mathrm{t}^\mathrm{m}_{[\mathrm{i}_\mathrm{m}]})) \end{array}$$

For the rule mentioned here, the worst-case input is of the form illustrated in figure 1. We show that the complexity is still poly time. Each row (long rectangle in the figure) is a bitvector, and the bitvectors are vertically stacked to illustrate the bitwise operations. Each solid line in a row is a concatenation point, and the dotted lines indicate the extractions that need to be carried out in the remaining bitvectors to do the bitwise operations.

It is easy to see that in the final normal form of such an input, each component of the concatenation will have bitwise operations each of which will be atmost $n$-ary (column demarcated by the dotted line represent a single component of the final concatenation). The arity of the final concatenation will be the maximum number of such columns, i.e. the largest number of concatenations in any row in the input. Consequently, the size of the final output is atmost quadratic in the size of the input.

However, care must be taken that in the process of computing this final output, no step required exponential time. Recall that the sub-terms must always be in normal form. It is easy to check that every application of the rule, applied bottom-up, is polynomial. It follows that this rule is poly-time as well.

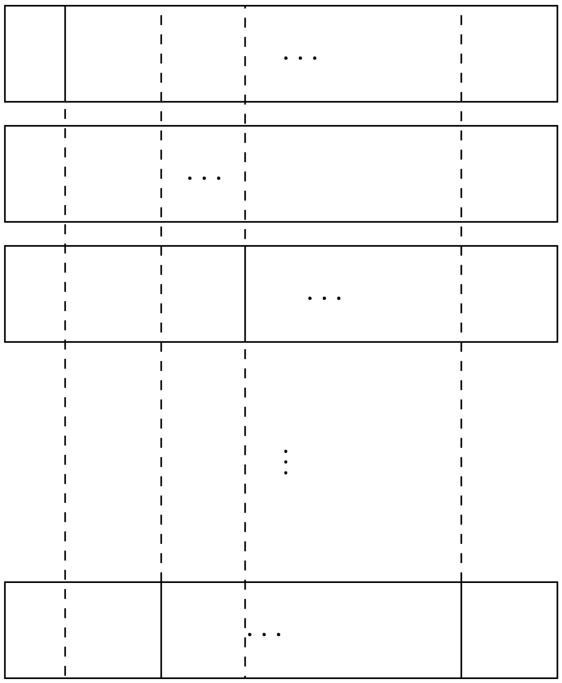

Figure 1: An illustration of $n$-ary bitwise operation over $m$-ary concatenations

### 4.1.4 Arithmetic Normal Form

Arithmetic normal form for a $\Sigma_a$-term $t_{[n]}$ is defined as follows.

$$\begin{array}{rcl} q_{[n]} & ::= & c_{[n]} \mid x_{[n]} \\ s_{[n]} & ::= & q_{[n]} \mid c_{[n]} *_{[n]} x_{[n]} \\ t_{[n]} & ::= & s_{[n]} \mid s^1_{[n]} +_{[n]} \dots +_{[n]} s^m_{[n]}. \end{array}$$

Here $\mathrm{var}(s^1_{[n]}) \prec \dots \prec \mathrm{var}(s^m_{[n]})$, where $\mathrm{var}(s)$ denotes the variable in $s$, if there is one. In other words, $\mathrm{var}(x) = x$, $\mathrm{var}(c *_{[n]} x) = x$, and $\mathrm{var}(c) = c$. Intuitively, $t_{[n]}$ is a sum of monomials with all like terms combined, and the summands ordered by their variables.

### 4.1.5 Proof rules for Arithmetic Normal Form

Following are the proof rules for reducing terms constructed out of arithmetic operators into arithmetic normal form. The invariant assumed by the normalizer $\gamma$ is that the immediate subexpression of the top-level operator are already in arithmetic normal form.

We define a function pad() as follows:

$$pad(n, t_{[i]}) \;\; = \;\; \begin{array}{cc} t_{[i]}[n-1:0] & i > n \\ t_{[i]} & i = n \\ 0bin0_{n-i}@t_{[i]} & i < n \end{array}$$

- Rules for the BVMULT operator $*_{[n]}$:

$$\begin{array}{rcl} \gamma(0bin0_{[r]} *_{[n]} t_{[s]}) & = & 0bin0_{[n]} \\ \gamma(0bin0\dots01 *_{[n]} t_{[s]}) & = & \mathrm{pad}(n, t_{[s]}) \end{array}$$

For the rules below, $a, c, c'$ denote constants, and we have $a \neq 0bin0_{[r]}$ or $a \neq 0bin0\dots01$.

$$\begin{array}{rcl} \gamma(t^1_{[i_1]} *_{[n]} t^2_{[i_2]}) & = & \gamma(\mathrm{pad}(n, t^1_{[i_1]}) *_{[n]} \mathrm{pad}(n, t^2_{[i_2]})) \\ \gamma(a_{[r]} *_{[n]} c_{[s]}) & = & c'_{[n]} \\ & & \mathrm{where\,c' \;=\; int2bv(n, bv2int(a) \cdot bv2int(c))} \\ & & \\ \gamma(t_{[n]} *_{[n]} a_{[n]}) & = & \gamma(a_{[n]} *_{[n]} t_{[n]}) \\ \gamma(a_{[n]} *_{[n]} (b_{[n]} *_{[n]} t_{[n]})) & = & \gamma(\gamma(a_{[n]} *_{[n]} b_{[n]}) *_{[n]} t_{[n]}) \\ \gamma(a_{[n]} *_{[n]} (t^1_{[n]} +_{[n]} \dots +_{[n]} t^k_{[n]})) & = & \\ & & \gamma(a_{[n]} *_{[n]} t^1_{[n]}) +_{[n]} \dots +_{[n]} \gamma(a_{[n]} *_{[n]} t^k_{[n]}) \end{array}$$

- Rules for BVPLUS:

$$\gamma(0bin0_{[r]} +_{[n]} t^1_{[n]} +_{[n]} \dots +_{[n]} t^k_{[n]}) = t^1_{[n]} +_{[n]} \dots +_{[n]} t^k_{[n]}$$

- Making terms of equal length:

$$\begin{array}{rcl}\gamma(a^1_{[k_1]} *_{[i_1]} x^1_{[j_1]} +_{[n]} \ldots +_{[n]} a^m_{[k_m]} *_{[i_m]} x^m_{[j_m]}) & = & \\ \mathrm{BVPLUS}^m_{i=1}\gamma(\mathrm{pad}(n, a^i_{[k_i]} *_{[n]} x^i_{[j_i]})) & & \end{array}$$

  - Adding constants

$$\gamma(a^1_{[n]} +_{[n]} \ldots +_{[n]} a^k_{[n]}) \quad = \quad \mathrm{int2bv(n, \Sigma^k_{i=1}bv2int(a^i_{[n]}))}$$

  - Flattening Rule:

$$\begin{array}{rcl}\gamma(t^1_{[n]} +_{[n]} \ldots +_{[n]} (q^1_{[n]} +_{[n]} \ldots +_{[n]} q^k_{[n]}) +_{[n]} \ldots +_{[n]} t^m_{[n]}) & = & \\ (t^1_{[n]} +_{[n]} \ldots q^1_{[n]} +_{[n]} \ldots q^k_{[n]} +_{[n]} \ldots +_{[n]} t^m_{[n]}) & & \end{array}$$

  - Sorting the monomials: The monomials $a_{[i]} *_{[n]} x_{[j]}$ which are subterms of a BVPLUS expression are arranged according to the ordering over terms $\prec$.
  - Combining like terms: After all the monomials are sorted w.r.t $\prec$, monomials over the same variable appear next to each other in the BVPLUS term. We combine them as follows. If a variable occurs without any coefficient, then its coefficient is set to 1. If more than one constant occurs then they are added up using the addition of constants rule.

$$\begin{array}{rcl}\gamma((a^1_{[n]} *_{[n]} x_{[n]}) +_{[n]} \ldots +_{[n]} (a^k_{[n]} *_{[n]} x_{[n]})) & = & \\ \gamma((a^1_{[n]} +_{[n]} \ldots +_{[n]} a^k_{[n]}) *_{[n]} x_{[n]}) & & \end{array}$$

- Rules for BVUMINUS

$$\begin{array}{rcl}\gamma(-0_{[n]}) & = & 0_{[n]} \\ \gamma(-t_{[n]}) & = & \gamma(-1) *_{[n]} \gamma(t_{[n]}) \\ \gamma(-c_{[n]}) & = & (\sim c_{[n]} +_{[n]} 1) \\ \gamma(-(-t_{[n]})) & = & t_{[n]} \\ \gamma(-(c_{[n]} *_{[n]} t_{[n]})) & = & \gamma((-c_{[n]}) *_{[n]} t_{[n]}) \\ \gamma(c_{[n]} *_{[n]} -t_{[n]}) & = & \gamma((-c_{[n]}) *_{[n]} t_{[n]}) \\ \gamma(-(c^1_{[n]} *_{[n]} t^1_{[n]} +_{[n]} \cdots +_{[n]} c^m_{[n]} *_{[n]} t^m_{[n]})) & = & \gamma(\gamma(-c^1_{[n]} *_{[n]} t^1_{[n]}) +_{[n]} \cdots +_{[n]} \\ & & \gamma(-c^m_{[n]} *_{[n]} t^m_{[n]}))\end{array}$$

- Extraction over BVPLUS

$$\begin{array}{rcl}\gamma((t^1_{[n]} +_{[n]} \ldots +_{[n]} t^m_{[n]})[i:j]) & = & \gamma(t^1_{[n]}[i:0] +_{[i+1]} \ldots +_{[i+1]} t^m_{[n]}[i:0])[i:j] \\ & & \text{where} \quad \mathrm{n > i+1}\end{array}$$

- Extraction over BVMULT

$$\begin{array}{rcl}\gamma((t^1_{[n]} *_{[n]} t^2_{[n]})[i:j]) & = & \gamma(t^1_{[n]}[i:0] *_{[i+1]} t^2[i:0])[i:j] \\ & & \text{where} \quad \mathrm{n > i+1}\end{array}$$

For all other terms $t$ we have $\gamma(t) = t$.

### 4.1.6 Complexity

We show that the time complexity of $\gamma$ is polynomial in the size of the input. The size of term is defined as in the section on concatenation normal form.

Notice that, the output terms of all rules have utmost as many variables, functions and constants as in the input, except the padding rules and the following rule:

$$\gamma(a_{[n]} *_{[n]} (t^1_{[n]} +_{[n]} \ldots +_{[n]} t^k_{[n]})) = \gamma(a_{[n]} *_{[n]} t^1_{[n]}) +_{[n]} \ldots +_{[n]} \gamma(a_{[n]} *_{[n]} t^k_{[n]})$$

It is easy to see that, for each rule, the blow-up of the size of the output as a function of the input size is polynomial. Also, note that each rule takes only polynomial amount of time (i.e. polynomial in the size of the input) to compute its output. It follows that the time complexity of the normalizer $\gamma$ is polynomial.

## 4.2 Partial Solver

A Shostak-style decision procedure is a composition of a canonizer and solver. A solver accepts a set of equations, and returns a set of equations in *solved form* [Bar03]. In our decision procedure, this requirement is dropped. There is a solver which solves in a lazy manner, i.e. solve for only those variables whose coefficient are already 1 or -1. In particular, for an equation of the form

$$t^1_{[n]} = t^2_{[n]}$$

we compute

$$t^1_{[n]} -_{[n]} t^2_{[n]} = 0bin0_{[n]}$$

where, $t^1_{[n]} -_{[n]} t^2_{[n]}$ is further normalized using the normalizers described above. Such a transformation generates further opportunities for normalizations. If a variable $x_{[n]}$ can be isolated into the form $x_{[n]} = t_{[n]}$, then such an isolation is performed. Furthermore, all terms containing $x_{[n]}$ are rewritten, thus eliminating it from the system.

## 4.3 Bit-blasting Rules

In this section, we describe all the proof rules and the strategy for converting a bit-vector fomula into an equivalent Boolean formula.

### 4.3.1 Proof rules for the Bit-blaster

Following is a set of proof rules to extract a single bit, as a boolean variable, from an arbitrary bit-vector term. Below, $i \in \mathbb{N}$. Side conditions are mentioned above the line. These rules have no premises, and conclusion of the rule is below the line. In the following, $t_{[n]}[i] : BV(n) \rightarrow Bool$ is a shorthand for $t_{[n]}[i:i] \approx 0bin1$. If the $i^{th}$ bit of $t_{[n]}$ is indeed $0bin1$ then the formula $t_{[n]}[i:i] \approx 0bin1$ evaluates to true, and otherwise it evaluates to false. Below, we deviate somewhat from the standard proof rule structure used in CVCL. In particular, we don't use sequents for the conclusions, and write side conditions above the line.

$$\frac{0 \leq i \leq n-1,\ c_{[n]} \in \mathcal{C},\ i\text{-th bit in c is } 0}{\vdash c[i] \Leftrightarrow false}$$

$$\frac{0 \leq i \leq n-1,\ c_{[n]} \in \mathcal{C},\ i\text{-th bit in c is } 1}{\vdash c[i] \Leftrightarrow true}$$

$$\frac{j_1 + \ldots + j_m = n \qquad 0 \leq i < j_1}{\vdash (t^m_{[j_m]} @ \ldots @ t^1_{[j_1]})[i] \Leftrightarrow (t^1_{[j_1]})[i]}$$

$$\frac{j_1 + \ldots + j_m = n \qquad j_1 + \ldots + j_{k-1} \leq i < j_1 + \ldots + j_k}{\vdash (t^m_{[j_m]} @ \ldots @ t^1_{[j_1]})[i] \Leftrightarrow (t^k_{[j_k]})[i - (j_1 + \ldots + j_{k-1})]}$$

$$\frac{0 \leq j \leq k < n,\ 0 \leq i < k - j}{\vdash (t_{[n]}[k:j])[i] \Leftrightarrow t_{[n]}[i+j]}$$

$$\frac{i_1, i_2 > 0}{\vdash (t^1_{[i_1]} +_{[n]} t^2_{[i_2]})[0] \Leftrightarrow t^1[0] \oplus t^2[0]}$$

$$\frac{0 < i \leq n-1}{\vdash (t^1_{[n]} +_{[n]} t^2_{[n]})[i] \Leftrightarrow t^1[i] \oplus t^2[i] \oplus c(t^1, t^2, i)}$$

where

- $c(t^1, t^2, 0) = t^1[0] \wedge t^2[0]$ and
- $c(t^1, t^2, i) = (t^1[i-1] \wedge t^2[i-1]) \vee (t^1[i-1] \wedge c(t^1, t^2, i-1)) \vee (t^2[i-1] \wedge c(t^1, t^2, i-1))$ for $i > 0$.

$$\frac{0 \leq i \leq n-1}{\vdash (\sim t_{[n]})[i] \Leftrightarrow \neg(t_{[n]}[i])}$$

$$\frac{0 \leq i \leq n-1}{\vdash (t^m_{[n]} \& \ldots \& t^1_{[n]})[i] \Leftrightarrow (t^m_{[n]}[i] \wedge \ldots \wedge t^1_{[n]}[i])}$$

$$\frac{0 \leq i \leq n-1}{\vdash (t^m_{[n]} \mid \ldots \mid t^1_{[n]})[i] \Leftrightarrow (t^m_{[n]}[i] \vee \ldots \vee t^1_{[n]}[i])}$$

### 4.3.2 Recursive Function to Bit-blast Terms

The recursive function $f(t_{[n]}, i) : (BV(n), \mathbb{N}) \rightarrow BOOL$ accepts two inputs, namely a bit-vector term $t_{[n]}$ and a natural number, recursively (structural recursion) applies the above bit-blasting rules, and returns a boolean formula $\varphi$ over the variables in $t_{[n]}$ such that $t_{[n]}[i] \Leftrightarrow \varphi$.

### 4.3.3 Bit-blasting Equations

Following is a proof rule whose premise is a bit-vector equation and conclusion is a boolean formula over the bits of the terms in the equation. Let the equation in the premise be $t^1 \approx t^2$, where both $t^1, t^2$ are terms of sort $BV[n]$.

$$\frac{\Gamma \vdash t^1 \approx t^2}{\Gamma \vdash \bigwedge_{i=1}^{n} t^1[i] \iff t^2[i]}$$

### 4.3.4 Bit-blasting Inequations

Following are the bit-blasting rules for bitvector comparators.

$$\frac{}{\vdash 0 <_{[1]} 1 \iff \text{true}}$$

$$\frac{}{\vdash 1 <_{[1]} 0 \iff \text{false}}$$

$$\frac{}{\vdash t_{[n]} <_{[n]} t_{[n]} \iff \text{false}}$$

$$\frac{}{\vdash c^1_{[n]} <_{[n]} c^2_{[n]} \iff \text{true}}$$

if $\text{bv2int}(\text{c}^1_{[\text{n}]}) < \text{bv2int}(\text{c}^2_{[\text{n}]})$

$$\frac{}{\vdash c^1_{[n]} <_{[n]} c^2_{[n]} \iff \text{false}}$$

if $\text{bv2int}(\text{c}^2_{[\text{n}]}) \leq \text{bv2int}(\text{c}^1_{[\text{n}]})$

$$\frac{\vdash b_1 <_{[1]} b_2}{\vdash Bool(b_1) \iff false \wedge Bool(b_2) \iff true}$$

where $b_1, b_2$ are single bit bit-vectors, and $Bool(b_i)$ is the corresponding bool value.

$$\frac{\vdash t^1 <_{[n]} t^2}{\vdash \begin{array}{cr} (t^1[n-1] <_{[1]} t^2[n-1]) & \vee \\ ((t^1[n-1] = t^2[n-1]) \wedge (t^1[n-2] <_{[1]} t^2[n-2])) & \vee \\ ((t^1[n-1] = t^2[n-1] \wedge t^1[n-2] = t^2[n-2]) \wedge (t^1[n-3] <_{[1]} t^2[n-3])) & \vee \\ \vdots & \vdots \\ ((t^1[n-1] = t^2[n-1] \wedge \cdots \wedge t^1[1] = t^2[1]) \wedge (t^1[0] <_{[1]} t^2[0])) & \end{array}}$$

Another rule which implements the rule given above in a more efficient way.

$$\frac{\vdash t^1 <_{[n]} t^2}{\vdash \begin{array}{cr} (t^1[n-1] <_{[1]} t^2[n-1]) & \vee \\ ((t^1[n-1] = t^2[n-1]) \wedge (t^1[n-2:0] <_{[n-1]} t^2[n-2:0])) & \end{array}}$$

$$\frac{}{\vdash 0 \leq_{[1]} 1 \iff \text{true}}$$

$$\frac{}{\vdash 1 \leq_{[1]} 0 \iff \text{false}}$$

$$\frac{}{\vdash t_{[n]} \leq_{[n]} t_{[n]} \iff \text{true}}$$

$$\frac{}{\vdash c^1_{[n]} \leq_{[n]} c^2_{[n]} \iff \text{true}}$$

if $\mathrm{bv2int}(\mathrm{c}^1_{[\mathrm{n}]}) \leq \mathrm{bv2int}(\mathrm{c}^2_{[\mathrm{n}]})$

$$\frac{}{\vdash c^1_{[n]} \leq_{[n]} c^2_{[n]} \iff \text{false}}$$

if $\mathrm{bv2int}(\mathrm{c}^2_{[\mathrm{n}]}) < \mathrm{bv2int}(\mathrm{c}^1_{[\mathrm{n}]})$

$$\frac{\vdash b^1 \leq_{[1]} b^2}{\vdash \mathit{Bool}(b_1) \iff \mathit{false} \vee \mathit{Bool}(b_2) \iff \mathit{true}}$$

where $b_1, b_2$ are single bit bit-vectors, and $\mathit{Bool}(b_i)$ is the corresponding bool value.

$$\frac{\vdash t^1 \leq_{[n]} t^2}{\vdash \begin{array}{cr} (t^1[n-1] <_{[1]} t^2[n-1]) & \vee \\ ((t^1[n-1] = t^2[n-1]) \wedge (t^1[n-2] <_{[1]} t^2[n-2])) & \vee \\ ((t^1[n-1] = t^2[n-1] \wedge t^1[n-2] = t^2[n-2]) \wedge (t^1[n-3] <_{[1]} t^2[n-3])) & \vee \\ \vdots & \vdots \\ ((t^1[n-1] = t^2[n-1] \wedge \cdots \wedge t^1[1] = t^2[1]) \wedge (t^1[0] <_{[1]} t^2[0])) & \vee \\ (t^1[n-1] = t^2[n-1] \wedge \cdots \wedge t^1[1] = t^2[1] \wedge t^1[0] = t^2[0]) & \end{array}}$$

Another rule which implements the rule given above in a more efficient way.

$$\frac{\vdash t^1 \leq_{[n]} t^2}{\vdash \begin{array}{c} (t^1[n-1] <_{[1]} t^2[n-1]) \qquad \vee \\ ((t^1[n-1] = t^2[n-1]) \wedge (t^1[n-2:0] \leq_{[n-1]} t^2[n-2:0])) \end{array}}$$

## 4.4 Conjunctive Normal Form

In this section we present the proof rules to convert arbitrary boolean formulas into conjunctive normal form (CNF), and the associated strategy. We denote boolean variables by $p, q, \ldots$, and $l, l_1, l_2, \ldots$ denote literals, $c_1, c_2, \ldots$ denote clauses, and arbitrary boolean formulas are denoted by $\varphi, \varphi_1, \varphi_2, \ldots$. We interchangeably use AND with $\wedge$, OR with $\vee$, IMP with $\Rightarrow$, IFF with $\iff$.

### 4.4.1 The CNF proof rules

1. BoolVar Intro Rule:

   $$\frac{\Gamma \vdash \varphi \qquad \varphi \, \text{is not a literal}}{\Gamma \vdash \exists v \, (v \, \wedge \, (v \iff \varphi))}$$

   where $v$ is a fresh boolean variable corresponding to the arbitrary boolean formula denoted by $\varphi$, and where $\varphi$ is not a literal.

2. And-CNF Rule: Let $\varphi$ denote the boolean formula AND$(\varphi_1, \ldots, \varphi_n)$, and let $v$ be the fresh boolean variable corresponding to the boolean formula denoted by $\varphi$.

   $$\frac{\Gamma \vdash v \iff \text{AND}(\varphi_1, \ldots, \varphi_n)}{\Gamma \vdash \exists v_1 \ldots v_n (\text{CNF}[v \iff \text{AND}(v_1, \ldots, v_n)] \wedge \bigwedge_{i=1}^{n} (v_i \iff \varphi_i))}$$

   where $v_1, \ldots, v_n$ are boolean variables corresponding to the boolean formulas denoted by $\varphi_1, \ldots, \varphi_n$, and CNF$[v \iff \text{AND}(v_1, \ldots, v_n)]$ is a macro denoting the cnf formula which is logically equivalent to the formula $v \iff \text{AND}(v_1, \ldots, v_n)$. The intuition behind this rule is that the immediate sub-formulas in $\varphi$ are replaced by variables, and the result is converted to CNF, and this process is repeated all the way to the level of literals. The formula denoted by CNF$[v \iff \text{AND}(v_1, \ldots, v_n)]$ is:

   $$(\neg v \, \vee \, v_1) \wedge (\neg v \, \vee \, v_2) \wedge \ldots \wedge (\neg v \, \vee \, v_n) \wedge (v \, \vee \, \neg v_1 \vee \neg v_2 \vee \ldots \vee \neg v_n)$$

3. Or-CNF Rule: Let $\varphi$ denote the boolean formula OR$(\varphi_1, \ldots, \varphi_n)$, and let $v$ be the fresh boolean variable corresponding to the boolean formula denoted by $\varphi$.

   $$\frac{\Gamma \vdash v \iff \text{OR}(\varphi_1, \ldots, \varphi_n)}{\Gamma \vdash \exists v_1 \ldots v_n (\text{CNF}[v \iff \text{OR}(v_1, \ldots, v_n)] \wedge \bigwedge_{i=1}^{n} (v_i \iff \varphi_i))}$$

   where $v_1, \ldots, v_n$ are boolean variables corresponding to the boolean formulas denoted by $\varphi_1, \ldots, \varphi_n$, and CNF$[v \iff \text{OR}(v_1, \ldots, v_n)]$ is a macro denoting the cnf formula which is logically equivalent to the formula $v \iff \mathrm{OR}(v_1, \ldots, v_n)$. The intuition behind this rule is that the immediate sub-formulas in $\varphi$ are replaced by variables, and the result is converted to CNF, and this process is repeated all the way to the level of literals. The formula denoted by $\mathrm{CNF}[v \iff \mathrm{OR}(v_1, \ldots, v_n)]$ is:

$$(v \lor \neg v_1) \land (v \lor \neg v_2) \land \ldots \land (v \lor \neg v_n) \land (\neg v \lor v_1 \lor v_2 \lor \ldots \lor v_n)$$

4. IMP-CNF Rule: Let $\varphi$ denote the boolean formula $\mathrm{IMP}(\varphi_1, \varphi_2)$ (usually written as $\varphi_1 \Rightarrow \varphi_2$) , and let $v$ be the fresh boolean variable corresponding to the boolean formula denoted by $\varphi$.

$$\frac{\Gamma \vdash v \iff \mathrm{IMP}(\varphi_1, \varphi_2)}{\Gamma \vdash \exists v_1 \ldots v_n (\mathrm{CNF}[v \iff \mathrm{IMP}(v_1, v_2)] \land \bigwedge_{i=1}^{2} (v_i \iff \varphi_i))}$$

   where $v_1, v_2$ are boolean variables corresponding to the boolean formulas denoted by $\varphi_1, \varphi_2$, and $\mathrm{CNF}[v \iff \mathrm{IMP}(v_1, v_2)]$ is a macro denoting the cnf formula which is logically equivalent to the formula $v \iff \mathrm{IMP}(v_1, v_2)$. The intuition behind this rule is that the immediate subformulas in $\varphi$ are replaced by variables, and the result is converted to CNF, and this process is repeated all the way to the level of literals. The formula denoted by $\mathrm{CNF}[v \iff \mathrm{IMP}(v_1, v_2)]$ is:

$$(\neg v \lor \neg v_1 \lor v_2) \land (v \lor v_1) \land (v \lor \neg v_2)$$

5. IFF-CNF Rule: Let $\varphi$ denote the boolean formula $\mathrm{IFF}(\varphi_1, \varphi_2)$ (usually written as $\varphi_1 \iff \varphi_2$) , and let $v$ be the fresh boolean variable corresponding to the boolean formula denoted by $\varphi$.

$$\frac{\Gamma \vdash v \iff \mathrm{IFF}(\varphi_1, \varphi_2)}{\Gamma \vdash \exists v_1 \ldots v_n (\mathrm{CNF}[v \iff \mathrm{IFF}(v_1, v_2)] \land \bigwedge_{i=1}^{2} (v_i \iff \varphi_i))}$$

   where $v_1, v_2$ are boolean variables corresponding to the boolean formulas denoted by $\varphi_1, \varphi_2$, and $\mathrm{CNF}[v \iff \mathrm{IFF}(v_1, v_2)]$ is a macro denoting the cnf formula which is logically equivalent to the formula $v \iff \mathrm{IFF}(v_1, v_2)$. The intuition behind this rule is that the immediate sub-formulas in $\varphi$ are replaced by variables, and the result is converted to CNF, and this process is repeated all the way to the level of literals. The formula denoted by $\mathrm{CNF}[v \iff \mathrm{IFF}(v_1, v_2)]$ is:

$$(\neg v \lor \neg v_1 \lor v_2) \land (\neg v \lor v_1 \lor \neg v_2) \land (v \lor v_1 \lor v_2) \land (v \lor \neg v_1 \lor \neg v_2)$$

6. ITE-CNF Rule: Let $\varphi$ denote the boolean formula $\mathrm{ITE}(\varphi_1, \varphi_2, \varphi_3)$, and let $v$ be the fresh boolean variable corresponding to the boolean formula denoted by $\varphi$.

$$\frac{\Gamma \vdash v \iff \mathrm{ITE}(\varphi_1, \varphi_2, \varphi_3)}{\Gamma \vdash \exists v_1 \ldots v_n (\mathrm{CNF}[v \iff \mathrm{ITE}(v_1, v_2, v_3)] \land \bigwedge_{i=1}^{3} (v_i \iff \varphi_i))}$$

where $v_1, v_2, v_3$ are boolean variables corresponding to the boolean formulas denoted by $\varphi_1, \varphi_2, \varphi_3$, and $\mathrm{CNF}[v \iff \mathrm{ITE}(v_1, v_2, v_3)]$ is a macro denoting the cnf formula which is logically equivalent to the formula $v \iff \mathrm{ITE}(v_1, v_2, v_3)$. The intuition behind this rule is that the immediate sub-formulas in $\varphi$ are replaced by variables, and the result is converted to CNF, and this process is repeated all the way to the level of literals. The formula denoted by $\mathrm{CNF}[v \iff \mathrm{ITE}(v_1, v_2, v_3)]$ is:

$$(\neg v \vee \neg v_1 \vee v_2) \wedge (\neg v \vee v_1 \vee v_3) \wedge (v \vee \neg v_1 \vee \neg v_2) \wedge (v \vee v_1 \vee \neg v_3)$$

An important optimization employed in the above rules is that new variables corresponding to $\varphi_i$ are not introduced if $\varphi_i$ is a literal.

### 4.4.2 The CNF strategy

Following is the strategy we employ to convert boolean formula into equivalent CNF formulas. The strategy recursively applies the appropriate proof rules given above to the appropriate boolean formula. Let the arbitrary input boolean formula be $\theta$ (Note $\theta$ is a actually a theorem).

```
Theorem CNF($\theta$) {
    1. $\varphi$ = simplify(pushNegation($\theta$)); // $\frac{\vdash \theta}{\vdash \varphi}$
    2. if $\varphi$ is a literal, return $\vdash \varphi$;
    3. Apply BoolVar Intro Rule; // $\frac{\vdash \varphi}{\vdash \exists v\,(v \wedge (v \iff \varphi))}$
    4. skolemize; //$\frac{\vdash \exists v\,(v \wedge (v \iff \varphi))}{\vdash sk(v) \wedge (sk(v) \iff \varphi)}$
    5. andElimRule_1; // $\frac{\vdash sk(v) \wedge (sk(v) \iff \varphi)}{\vdash sk(v)}$
    6. clauses.pushback($sk(v)$); //v is the new variable corresponding to $\varphi$
    7. andElimRule_2; // $\frac{\vdash sk(v) \wedge (sk(v) \iff \varphi)}{\vdash (sk(v) \iff \varphi)}$
    8. applyCNFRules($sk(v) \iff \varphi$, clauses);
    9. apply andIntroRule(clauses);
}
```

Now we present the applyCNFRules function which applies the rules given in the previous subsection. Let $OP$ denote the toplevel operator of the formula $\varphi$. $OP$ can be one of $\mathrm{AND}, \mathrm{OR}, \mathrm{IMP}, \mathrm{ITE}, \mathrm{IFF}$.

```
applyCNFRules($sk(v) \iff \varphi$, clauses) {
    1. OP-CNF Rule($sk(v) \iff \varphi$); //$\frac{\vdash sk(v) \iff \mathrm{OP}(\varphi_1,\ldots,\varphi_n)}{\vdash \exists v_1 \ldots v_n (\mathrm{CNF}[sk(v) \iff \mathrm{OP}(v_1,\ldots,v_n)] \wedge \bigwedge_{i=1}^{n} (v_i \iff \varphi_i))}$
2. skolemize; //$\frac{\vdash\exists v_1 ... v_n(\text{CNF}[v \iff \text{OP}(v_1,...,v_n)]\wedge\bigwedge_{i=1}^{n}(v_i \iff \varphi_i))}{\text{CNF}[sk(v) \iff \text{OP}(sk(v_1),...,sk(v_n))]\wedge\bigwedge_{i=1}^{n}(sk(v_i) \iff \varphi_i))}$
3. andElimRule_1; //$\frac{\text{CNF}[sk(v) \iff \text{OP}(sk(v_1),...,sk(v_n))]\wedge\bigwedge_{i=1}^{n}(sk(v_i) \iff \varphi_i))}{\text{CNF}[sk(v) \iff \text{OP}(sk(v_1),...,sk(v_n))]}$
4. clauses.pushback(CNF($sk(v) \iff op(\varphi_1, \varphi_2, \ldots, \varphi_n)$));
5. andElimRule_2; //$\frac{\text{CNF}[sk(v) \iff \text{OP}(sk(v_1),...,sk(v_n))]\wedge\bigwedge_{i=1}^{n}(sk(v_i) \iff \varphi_i))}{\bigwedge_{i=1}^{n}(sk(v_i) \iff \varphi_i)}$
6. for(i=1;i<=n;i++) {
    (a) apply andElimRule_i; //$\frac{\vdash\bigwedge_{i=1}^{n}(sk(v_i) \iff \varphi_i)}{\vdash sk(v_i) \iff \varphi_i}$
    (b) apply applyCNFRules($sk(v_i) \iff \varphi_i$,clauses);
   }
}//end of applyCNFRules
```

## 4.5 Propagation of equalities

Propagation of new facts (or equality rewrites) allows for improved performance by the normalizers. For example, suppose the input equality to the normalizer is $x\,@\,01 = 1101$. As is, this equality is already in normal form. However, now suppose that by processing a different set of equalities in the same context, the decision procedure learns that $x = y$ and $y = 10$. Then, propagating $x = y$, during the preprocessing step, generates a new equality $y\,@\,01 = 1101$. Further propagation of $y = 10$ yields $10\,@\,01 = 1101$. The normalizer now has an opportunity to normalize this equality to $1001 = 1101$, immediately detecting a contradiction.

## 4.6 Architecture: Putting it all together

The Figure 2 illustrates the architecture of the tool. Conceptually, the decision procedure is a preprocessor with a SAT solver as a back-end. The preprocessor in turn has two boxes, a normalizer for $\Sigma_c$-terms and a normalizer for $\Sigma_a$-terms, with an equalities database in a tight loop with the normalizers.

The input quantifier-free $\Sigma$-formula $\varphi$ is first equivalently purified into $\Sigma_c$-formula $\varphi^1$ and $\Sigma_a$-formula $\varphi^2$. The terms in $\varphi^1$ and $\varphi^2$ are normalized into concatenation normal form and arithmetic normal form by the respective normalizers. All equalities input by the CVC Lite core and those learnt by the decision procedure are stored in the equalities database (Figure 2). During the process of normalization, these learnt equalities are propagated (equality rewrites) thus creating new opportunities for normalization.

If there are no more opportunities for normalization and a contradiction is detected then $\varphi$ is declared unsatisfiable. Otherwise, the normalized formula $\varphi^1_{norm}$ and $\varphi^2_{norm}$ are *bit-blasted.* The process of bit-blasting essentially converts the conjunction $\varphi^1_{norm}$ and $\varphi^2_{norm}$ into a logically equivalent boolean formula.

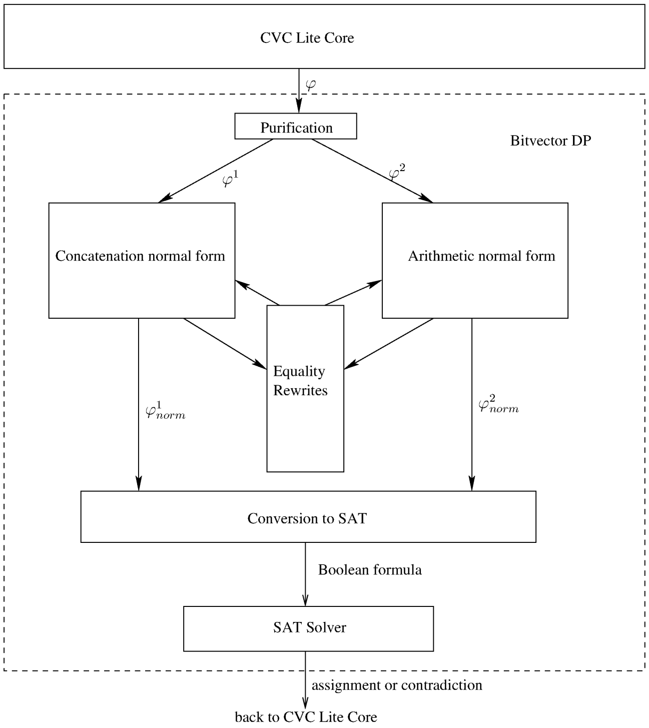

Figure 2: Architecture of the bit-vector Decision Procedure

The resultant boolean formula is fed to the SAT solver. If the SAT solver detects a contradiction, then the original formula $\varphi$ is declared unsatisfiable. Otherwise it is satisfiable.

### 4.6.1 Requirements of the Nelson-Oppen Combination

An important contribution of this effort is showing how to integrate a bit-vector decision procedure into a Nelson-Oppen style combination of decision procedures. In particular, we have implemented this decision procedure as a part of CVC Lite. Component decision procedures are required by CVC Lite to be *online, proof producing,* and propagate all equalities implied by the current *logical context* [NO79, Bar03]. This last condition is in fact imposed by the Nelson-Oppen combination theory.

Online means that a new constraint can be added to the set of existing constraints at any time during a run, and the algorithm must be able to take this into account with only incremental amount of work. Proof production is a useful feature for those users who want to check the work of CVC Lite using external proof checkers.

Also, the Nelson-Oppen combination requires that each component theory must be *stably-infinite* and the corresponding decision procedure must propagate all equalities that are implied by the current logical context, in order for the combination to be complete.

The theory of fixed-width bit-vectors presented here is in fact a *many-sorted theory* [Man96]. The fact that the theory is many-sorted is not relevant for the results in the previous sections. However, it becomes important for the purposes of Nelson-Oppen combination, as implemented in CVC Lite.[1] In the many-sorted version of the bit-vector theory, bit-vectors of length $n$ comprise a separate sort, for each $n$. Note that each sort corresponds to a finite domain, the domain of all finite non-empty strings of length $n$, over the alphabet $\{0, 1\}$. This implies that the many-sorted version of the bit-vector theory is not stably-infinite.

A $\Sigma$-theory $T$ is said to be *stably-infinite over a set of sorts*, if for every $\Sigma$-sentence $\varphi$ satisfiable in some *model* of $T$, $\varphi$ is satisfiable in a model of $T$ infinite in every sort in this set [TZ04, GBTD04].

The many-sorted theory of bit-vectors under obvious axiomatization, where each bit must be either 0 or 1, is not stably-infinite, since all models of $T$ are necessarily finite in all the sorts. Therefore, the bit-vector theory $T_{\mathrm{BV}}$ is split into two theories $T_{\mathrm{BV}} = T_1 \cup T_2$, where $T_1$ is a theory of finite strings over integers (and is, therefore, stably-infinite), and $T_2$ is an additional set of axioms constraining each integer to be either 0 or 1. The formulas of the theory $T_2$ are referred to as *type predicates.*

The actual decision procedure implemented in CVC Lite is for the theory $T_1$, and the necessary formulas from $T_2$ (the type predicates for all the terms in the original input) are being automatically added as part of the input formula.

The second Nelson-Oppen condition requires a decision procedure to propagate all equalities implied by the current logic context. The need for the propagation of all equalities is illustrated by the following example.

Let $f$ : Bitvector[1] $\to$ Bool be an uninterpreted function, and let $x, y, z$ be 1-bit bit-vector variables. Consider the mixed formula $\varphi \equiv f(x) \neq f(y) \neq f(z) \neq f(x)$. This formula implies that there exist some assignment to the variables $x, y, z$ such that $\theta \equiv x \neq y \neq z \neq x$. Since single bit variables $x, y, z$ can only take values 0 or 1, it follows that $\theta$ must be unsatisfiable. This implies that $\varphi$ is unsatisfiable.

However, the many-sorted Nelson-Oppen combination cannot detect the contradiction from $T_1$ alone. This situation can be remedied by propagating all equalities implied by the current logical context, and in particular, the type predicates from $T_2$:

$$(x = 0 \vee x = 1) \wedge (y = 0 \vee y = 1) \wedge (z = 0 \vee z = 1)$$

Processing these type predicates leads to extra cost. However, for completeness, it is sufficient to assert type predicates only for shared constants.

In the unsorted case, the stably-infiniteness condition is trivially satisfied. However, one still needs to add to the input something similar to type predicates

[1] CVC Lite implements a many-sorted version of the Nelson-Oppen combination method.

(also called as type correctness conditions, in this context), which assert that each variable has a fixed, known length. This has to be done since the variables are unsorted, and these type correctness conditions are necessary to get meaningful answers from the decision procedure. Also, the system has to somehow account for the fact that each bit is constrained to be 0 or 1. Irrespective of the method employed to constraint the bit values, it is easy to check that in the worst case something similar to type predicates will need to be asserted for shared terms in the first order case as well.

In other words, merely a transition from many-sorted to a first order combination does not imply that issues like type correctness conditions and bit constraints become unnecessary or trivial to deal with.

# 5 Experimental Results

The CVC Lite implementation of our bit-vector decision procedure has been tested on two sets of examples: a collection of industrial scale real-world verification problems[2] (figure 4), and a set of artificial examples parameterized by the width of bit-vector variables (figure 3). The industrial scale examples consist of very large bit-vector terms (64 bits or more), with hundreds of operators, and a deeply nested mixture of word-level, bitwise and arithmetic operators.

The performance of our decision procedure is compared against a SAT-based method using zChaff [MMZ$^+$01]. In figures 3 and 4, the total time (in seconds) includes translation to DIMACS format, and the time in parentheses is the actual time taken by zChaff. The real-life industrial examples (figure 4) are also compared against the SAT-based method using the CVC Lite built-in SAT solver (*CVCL SAT* column). This provides a better baseline comparison to the optimized method (*CVCL RW+SAT* column) which uses the same built-in SAT solver together with our normalizations and equality rewrites.

The experiments were run on an Intel Pentium 800 MHz processor with 384 Mb RAM under GNU/Linux 2.4 kernel. Both programs are compiled using gcc 3.3.2 with -O2 option.

## 5.1 Optimizing for Problem Domains

The decision problem for our bit-vector theory is known to be NP-hard [Möl98], and finding a decision procedure that is practically efficient in general is very unlikely. Therefore, it is important to identify specific problem domains or classes of formulas where a particular approach works well.

One of the important problem domains is program or circuit optimization, when the user is interested in verifying that an optimized version of the code fragment is functionally identical to the original code. These problems often result in formulas of the form $(t^1 = t^2) \Rightarrow \varphi$, which are ideal for our method.

[2] Due to intellectual property issues, the sources of these examples are not disclosed in this paper.

| Example | zChaff | | CVCL RW+SAT | |
|---|---|---|---|---|
| #bits | Time (zchaff) | Conflicts | Time | Conflicts |
| 16 | 1.24 (0.06) | 871 | 0.01 | 1 |
| 32 | 2.64 (0.12) | 1,657 | 0.01 | 1 |
| 64 | 6.37 (0.54) | 4,898 | 0.02 | 1 |
| 128 | 16.48 (1.52) | 9,381 | 0.04 | 1 |
| 256 | 49.81 (6.59) | 27,355 | 0.04 | 1 |
| 512 | 259.9 (117.94) | 67,459 | 0.09 | 1 |
| 1024 | — | — | 0.17 | 1 |
| 2048 | — | — | 0.33 | 1 |
| 4096 | — | — | 0.69 | 1 |
| 8192 | — | — | 1.52 | 1 |

Figure 3: Example $x = y \Rightarrow x + z = z + y$ parameterized by the number of bits in $x$, $y$, and $z$.

| Example | zChaff | | CVCL SAT | | CVCL RW+SAT | |
|---|---|---|---|---|---|---|
| | Time (zchaff) | Conflicts | Time | Conflicts | Time | Conflicts |
| Valid 1 | 4.16 (0.01) | 8 | 36.26 | 3,652 | 0.09 | 1 |
| Valid 2 | 10.22 (1.45) | 6538 | 913.51 | 40,027 | 0.4 | 1 |
| Valid 3 | 10.88 (2.24) | 8619 | 253.73 | 16,121 | 0.4 | 1 |
| Valid 4 | 9.3 (0.19) | 732 | 65.94 | 3,544 | 0.04 | 1 |
| Invalid 1 | 4.06 (0.01) | 6 | 3.76 | 9 | 1.2 | 2 |
| Invalid 2 | 7.44 (0.02) | 6 | 4.87 | 7 | 3.02 | 13 |
| Invalid 3 | 7.42 (0.01) | 9 | 5.35 | 5 | 2.64 | 3 |

Figure 4: Experimental results for industrial-scale verification problems.

In the best case, rewriting all occurrences of $t^1$ in $\varphi$ to $t^2$, and then normalizing all terms in $\varphi$ proves the entire formula valid without having to invoke the SAT solver. In figure 3 we demonstrate the scalability and effectiveness of our method in this case on a trivial example parameterized by the size of the bit-vectors.

Although the examples in figure 3 are obviously artificial, it is interesting to note that all the valid industrial examples in figure 4 also fall into the same category, and are completely solved by rewriting and normalization, despite the large formula sizes and their seeming complexity.

In many other cases, even if the formula is invalid and cannot be completely solved by these transformations, our method still shows a considerable reduction in time, and the amount of work the SAT solver has to do (see figure 4, the invalid examples).

In the worst case (for arbitrary formulas), when rewriting and normalization steps do not significantly simplify the formula, our method reduces to the direct translation to SAT. The additional overhead for these transformations is insignificant (close to linear, and relatively low in practice), and therefore, our approach is always at least as effective as the direct SAT-based method.

# 6 Conclusions

The main contribution of this work is a collection of practical design principles and a concrete implementation of a new efficient decision procedure for a theory of fixed-width bit-vectors. Another contribution is that the decision procedure is implemented as part of CVC Lite, a Nelson-Oppen combination framework. It is important to emphasize that the decision procedure is efficient and works on a very rich set of bit-vector operations (mixture of word-level, bit-wise, and arithmetic operators). The input language supports word-level bit-vector operations (concatenation and extraction), bit-vector arithmetic operations (addition and constant multiplication), bitwise boolean operations (conjunction, disjunction, negation) and multiplexors (if-then-else operator). Other common functions such as left and right shift, sign/zero extension, bit-vector subtraction, and comparators ("less than") can be easily supported through suitable translation.

The approach described here has many advantages. First, the method relies on normalization, but only to the extent that it is useful. In the worst case it falls back on SAT solvers which are known to be very effective for handling NP-complete problems. Also, present-day SAT technology seems to improve very quickly and this approach allows one to capitalize on the trend. Second, if the input language needs extension, it may be done without having to alter the design or implementation of $\Sigma_c$ or $\Sigma_a$-normalizers. It is sufficient to invent a normal form for the new functions/predicates added to $\Sigma$. In other words, the input language can be as general as needed. Third, as new rewrites are invented, they can be easily added to the preprocessing step. On the whole, this approach is flexible, allows for a very expressive and extensible input language, and is efficient.

Adding the decision procedure as a component to CVC Lite has many advantages. It allows us to support multiplexors (ITE terms), quantifiers over bit-vector variables, generate proofs and concrete counterexamples, and decide formulas over many theories.

An interesting area of future research is to explore the efficacy of using a *partial equation solver* for bit-vector arithmetic which are similar to *equation solvers* [BDL98]. Solvers take equations as input, and output equations in *solved form* [BDL98]. Partial solvers drop this last condition, and may solve only for a subset of all variables in the input.

Extending the arithmetic normal form to support non-linear terms is another area of future research. We also plan to natively support many other operations like bitwise XOR, full bit-vector multiplication, bit-vector subtraction, right shift, zero/sign extensions, and predicates like comparators. The advantage of native support is that the structure is preserved, and one can take advantage of the algebraic properties of these operators.

# References

[Bar03] C. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First -Order Theories.* PhD thesis, Stanford University, 2003.

[BB04] Clark Barret and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Computer-Aided Verification (CAV'04)*, LNCS. Springer Verlag, July 2004. `http://chicory.stanford.edu/CVCL`.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 522–527. ACM Press, 1998.

[BP98] Nikolaj Bjørner and Mark C. Pichora. Deiding fixed and non-fixed size bit-vectors. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 376–392. Springer-Verlag, 1998.

[CMR97] David Cyrluk, M. Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 60–71. Springer-Verlag, 1997.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Möller. Mona 1.x: new techniques for ws1s and ws2s. In *Computer Aided Verification, CAV '98, Proceedings*, volume 1427 of *LNCS*. Springer Verlag, 1998.

[FDK98] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. Functional vector generation for hdl models using linear programming and 3-satisfiability. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 528–533. ACM Press, 1998.

[GBTD04] V. Ganesh, S. Berezin, C. Tinelli, and D. L. Dill. Combination results for many sorted theories with overlapping signatures. Technical report, Stanford University, 2004. `http://chicory.stanford.edu/~berezin/csl2004/many-sorted-combination.ps`.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[Man96] Maria Manzano. *Extensions of First Order Logic.* Cambrige University Press, 1996.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

[Möl98] M. Oliver Möller. Solving bit-vector equations - a decision procedure for hardware verification, 1998. Diploma Thesis, available at `http://www.informatik.uni-ulm.de/ki/Bitvector/`.

[NO79] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[TH96] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop (Munich, Germany)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, March 1996.

[TZ04] Cesare Tinelli and Calogero Zarba. Combining decision procedures for theories in sorted logics. Technical Report 04-01, Department of Computer Science, The University of Iowa, February 2004.

[ZKC01] Z. Zeng, P. Kalla, and M. Ciesielski. Lpsat: a unified approach to rtl satisfiability. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 398–402. IEEE Press, 2001.