An Efficient Decision Procedure for a Theory of Fixed-Sized Bitvectors with Composition and Extraction*

David Cyrluk*, Oliver Möller§, Harald Rueß§

*Computer ence Lab oratory
SRI International
Menlo Park, CA 94025, USA
cyrluk@csl.sri.com

§Fakultät für Informatik
Universität Ulm
D-89069 Ulm, Germany
{moeller,ruess}@ki.informatik.uni-ulm.de

Abstract

The theory of fixed-sized bitvectors with composition and extraction has been shown to be useful in the realm of hardware verification, and in this paper we develop an efficient algorithm for deciding this theory. A proper input is an unquantified bitvector equation, say $t = u$, and our algorithm returns **true** if $t = u$ is valid in the bitvector theory, **false** if $t = u$ is unsatisfiable, and a system of solved equations otherwise. The time complexity of this solver is $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n + n^2)$, where $\mid t \mid$ is the length of the bitvector term t and n denotes the number of bits on either side of the equation. Moreover, the resulting procedure can readily be integrated into Shostak's procedure for deciding combinations of theories.

1 Introduction

Bitvectors are a fundamental datatype for many hardware verification tasks. Commonly used operations on bitvectors include sequential composition of several bitvectors and selection of one or more bits from a bitvector. Sometimes, bitvectors are simply modelled as lists of bits, but fixed-sized bitvectors model the underlying hardware more accurately. The bitvector library of the PVS system [ORS92], for example, formalizes bitvectors of length n as finite functions with domain $[0..n)$ and codomain $\{0, 1\}$.

Experience with hardware verification [SM95, Rue96] has shown that the lack of specialized decision procedures for notions related to bitvectors is the main impediment to effective automation in systems like PVS. This insight forms the starting point of this paper, and we develop an efficient decision procedure for the theory of fixed-sized bitvectors with composition and extraction. Moreover, this decision procedure can readily be incorporated into Shostak's procedure for combinations of theories [Sho84], since our algorithm fulfills the requirements for component theories as stated in [CLS96].

This paper is organized as follows. Section 2 contains background material on procedures for deciding combinations of quantifier-free theories, and in Section 3 we present the theory of fixed-sized bitvectors with composition and extraction as a many-sorted conditional equational theory. A straightforward decision procedure for this theory is developed in Section 4 and further refined in the rest of the paper. Section 5 contains a so-called *canonizer* [Sho84] for the bitvector theory, and it is shown that this canonizer fulfills the requirements as stated in [CLS96]. The main part of this paper consists of Section 6, where we develop an optimized version of a solver for the bitvector theory in Section 3, prove the requirements for a solver as given in [CLS96], and analyze its time complexity. The paper closes with some final remarks in Section 7.

*Appeared as Technical Report Nr. UIB-96-8 from the Universität Ulm, Fakultät für Informatik.

2 Deciding Combinations of Quantifier-Free Theories

This section contains background material on procedures for deciding combinations of quantifier-free theories. The key to these algorithms is the computation of the *congruence closure* of a binary relation on a finite labeled graph [Gal87]. Here, a relation is called congruent if it is both an equivalence relation and backward closed. This means informally that the congruence of two nodes follows from the congruence of all the (ordered) successor nodes.

Usually, an equivalence relation is represented by its corresponding partition, and two procedures for operating on partitions are assumed to be available: *union* and *find*. $union(u, v)$ combines the equivalence classes of u and v into a single class, and $find(u)$ returns a unique representative associated with the equivalence class of u.

Nelson and Oppen [NO80] present an algorithm for computing the congruence closure of a binary relation R on a graph using *union* and *find*. They show that if G has m edges and G has no isolated nodes, then the algorithm can be implemented to run in $\mathcal{O}(m^2)$ time; Downey, Sethi, and Tarjan [DST80] give a faster algorithm running in $\mathcal{O}(m * log(m))$ average time. Based on this algorithm for computing the congruence closure of a binary relation on a graph, Nelson and Oppen [NO79] developed a technique for combining decision procedures for individual theories to decide the combination of these theories by simply *propagating equalities* between the different procedures. The main efficiency drawback to Nelson-Oppen's approach, however, is that each theory has much of the same notion of equality resulting in duplicated effort [CLS96].

Shostak [Sho84] uses a different approach and merges simplifiers for the individual theories into a single procedure based on congruence closure. It has been mentioned in [CLS96] that, in practice, Shostak's procedure is an order of magnitude more efficient than that of Nelson and Oppen.

Shostak's procedure operates over a subclass of certain unquantified first-order theories called σ-theories. Informally, these theories have a computable canonizer function σ from terms (in the theory) to terms, such that an equation $u = v$ is valid in the theory if and only if $\sigma(u)$ is identical with $\sigma(v)$; the full set of requirements on canonizers is stated in [CLS96]. A canonizer for linear arithmetic, for example, could be constructed by transforming such expressions (using associativity, commutativity, distributivity) into the form $a_1 x_1 + \ldots + a_n x_n + c$ where each a_i is a nonzero constant and the summands are arranged in some canonical order.

To construct a decision procedure for equality in a combination of σ-theories, Shostak's method requires that the σ-theories have the additional property of *algebraic solvability*. A σ-theory is algebraically solvable if there exists a computable function *solve*, that takes an equation $s = t$ and returns either **true**, **false**, or an equivalent conjunction of equations of the form $x_i = t_i$, where the x_i's are distinct variables of t that do not occur in any of the t_i's. A solver for real linear arithmetic, for example, takes an equation of the form $a_1 x_1 + \ldots + a_n x_n + c = b_1 x_1 + \ldots + b_n x_n + d$ and returns $x_1 = ((b_2 - a_2)/(a_1 - b_1)) * x_2 + \ldots + ((b_n - a_n)/(a_1 - b_1)) * x_n + (d - c)$. Given a set of algebraically solvable σ-theories, Shostak's procedure decides the combination of theories. This method centralizes equality reasoning in the congruence closure algorithm, and the individual theories communicate through semantic canonizers and solvers. [CLS96] describe and analyze this procedure by systematically optimizing and augmenting Nelson and Oppen's procedure for computing the congruence closure of a relation on a graph.

Procedures for deciding the combination of certain quantifier-free formulas are at the core of many theorem proving systems. The Nelson-Oppen procedure is used in the Stanford Pascal Verifier [LGvH+79] and in Eves [CKM+91], while systems like EHDM [Com93] and PVS [ORS92] implement Shostak's combination procedure for theories like linear arithmetic, the theory of arrays, and inductive datatypes.

In the rest of the paper we show how the theory of bitvectors with composition and extraction can be integrated into Shostak's framework and we develop an efficient canonizer and solver for this theory.

3 A Core Theory of Bitvectors

This section develops an equational theory of fixed-sized bitvectors of length n. Note that the length n is constrained to be a positive natural number, since bitvectors of length 0 are not permitted, and the bits of a bitvector of length n are indexed, from left to right, from $n-1$ down to 0. In the following, $n, m, k, \ldots$ denote valid lengths of bitvectors. The bitvector theory contains constant bitvectors $0_{[n]}$ and $1_{[n]}$ of length n, *composition* $t \otimes u$ of bitvectors t and u, and *extraction* $t\hat{}(i,j)$, where $i, j \in \mathbb{N}$, of $i-j+1$ many bits i through j from bitvector t.

These considerations lead to a many-sorted signature (see, for example, [Gal87]) with infinitely many sort symbols $bvec_n$, $n \in \mathbb{N}^+$.

Definition 3.1 *Let Σ be the signature*

$$
\begin{aligned}
\Sigma \ ::= \ \langle & \{bvec_n \mid n \in \mathbb{N}^+\}, \\
& \{0_{[n]} \mid n \in \mathbb{N}^+\} \cup \{1_{[n]} \mid n \in \mathbb{N}^+\} \cup \\
& \{. \otimes_{n,m} . \mid n, m \in \mathbb{N}^+\} \cup \\
& \{. \hat{}_n(i,j) \mid n \in \mathbb{N}^+ \wedge i, j \in \mathbb{N} \wedge n > i \geq j \geq 0\} \\
\rangle &
\end{aligned}
$$

such that for appropriate n, i, and j:

$$
\begin{aligned}
0_{[n]} &: \ \to bvec_n \\
1_{[n]} &: \ \to bvec_n \\
. \otimes_{n,m} . &: \ bvec_n \times bvec_m \to bvec_{n+m} \\
. \hat{}_n(i,j) &: \ bvec_n \to bvec_{i-j+1}
\end{aligned}
$$

The dots to the left and to the right of function symbols indicate the use of infix notation, and extraction $\hat{}_n(i,j)$ is assumed to bind stronger than composition $\otimes_{n,m}$. In the following, $x_{[n]}, y_{[m]}, z_{[k]}, \ldots$ denote variables of sort $bvec_n$, $bvec_m$, and $bvec_k$ respectively. The set of well-formed terms is defined in the usual way and $t_{[n]}, u_{[m]}, v_{[k]}, \ldots$ denote bitvector terms of respective lengths. Subscripts are omitted whenever possible and can be inferred from the context. Moreover, $t \equiv u$ denotes syntactic equality of bitvectors t and u, and $vars(t)$ denotes the set of variables in t.

A bitvector term t is called *atomic* if it is a variable or a constant $0_{[n]}$ or $1_{[n]}$, and *simple terms* are either atomic or of the form $t_{[n]}\hat{}(i,j)$ where $t_{[n]}$ is atomic and at least one of the inequalities $i \neq n-1$, $j \neq 0$ holds. Moreover, terms of the form $t_1 \otimes t_2 \otimes \ldots \otimes t_k$ (modulo associativity), where t_i are all *simple*, are referred to as being in *composition normal form*. If, in addition, none of the neighboring simple terms denote the same constant (modulo length) and a simple term of the form $t\hat{}(i,j)$ is not followed by a simple term of the form $t\hat{}(j-1,k)$, then a term in composition normal form is called *maximally connected*.

Definition 3.2 *Let Σ be the bitvector signature defined in Definition 3.1; then: the characteristic properties of the bitvector theory with extraction and composition are given by the (conditional) Σ-equalities*

$$
\begin{array}{lrcll}
1) & (t_{[n]} \otimes u_{[m]})\hat{}(i,j) & = & u_{[m]}\hat{}(i,j) & \text{IF } m > i \geq j \geq 0 \\
2) & (t_{[n]} \otimes u_{[m]})\hat{}(i,j) & = & t_{[n]}\hat{}(i-m, j-m) & \text{IF } m+n > i \geq j \geq m \\
3) & (t_{[n]} \otimes u_{[m]})\hat{}(i,j) & = & t_{[n]}\hat{}(i-m, 0) \otimes u_{[m]}\hat{}(m-1, j) & \text{IF } m+n > i \geq m > j \\
4) & t_{[n]}\hat{}(n-1, 0) & = & t_{[n]} & \\
5) & t_{[n]}\hat{}(i,j) \otimes t_{[n]}\hat{}(j-1,k) & = & t_{[n]}\hat{}(i,k) & \\
6) & (t_{[n]} \otimes u_{[m]}) \otimes v_{[p]} & = & t_{[n]} \otimes (u_{[m]} \otimes v_{[p]}) & \\
7) & t_{[n]}\hat{}(i,j)\hat{}(k,l) & = & t_{[n]}\hat{}(k+j, l+j) &
\end{array}
$$

Note that well-formedness of terms implies that $n > i \geq j > k \geq 0$ in equation *5)* and $n > i \geq j \geq 0 \wedge i-j \geq k \geq l \geq 0$ in equation *7)* above. Semantic entailment $\models$ in the bitvector theory above is defined in the usual way.

In the bitvector library of the PVS system [ORS92], fixed-sized bitvectors of length n are interpreted as finite functions with domain $[0..n)$ and codomain $\{0,1\}$. The function symbols of the bitvector theory are

$$
\begin{array}{l}
\alpha_1(s) \ ::= \\
\quad \textsc{cases } s \textsc{ of} \qquad
\begin{array}{rcl}
t \otimes u & \rightarrow & \alpha_1(t) \otimes \alpha_1(u), \\
x_{[1]} \,\hat{}\,(0,0) & \rightarrow & x_{[1]} \ , \\
t\,\hat{}\,(i,j)\,\hat{}\,(k,l) & \rightarrow & \alpha_1(t\,\hat{}\,(k+j, l+j)), \\
(t_{[n]} \ \otimes \ u_{[m]}\ )\,\hat{}\,(i,j) & \rightarrow & \textsc{if } m > i \textsc{ then} \\
 & & \quad \alpha_1(u_{[m]} \ \hat{}\,(i,j)) \\
 & & \textsc{elseif } j \geq m \textsc{ then} \\
 & & \quad \alpha_1(t_{[n]} \ \hat{}\,(i-m, j-m)) \\
 & & \textsc{else} \\
 & & \quad \alpha_1(t_{[n]} \ \hat{}\,(i-m, 0)) \otimes \alpha_1(u_{[m]} \ \hat{}\,(n-1, j)) \\
 & & \textsc{endif} \\
\textsc{otherwise} & & s
\end{array} \\
\quad \textsc{endcases}
\end{array}
$$

Figure 1: Simplification of Bitvectors with atoms of unit length 1

interpreted as

$$
\begin{array}{rcl}
0_{[n]} & ::= & \lambda\, x : [0..n).\ 0 \\
1_{[n]} & ::= & \lambda\, x : [0..n).\ 1 \\
s \otimes t & ::= & \lambda\, x : [0..n+m).\ \textsc{if } x < m \textsc{ then } t(x) \textsc{ else } s(x-m) \\
s\,\hat{}\,(i,j) & ::= & \lambda\, x : [0..i-j+1).\ s(x+j)
\end{array}
$$

with variables $s : bvec_n$, $t : bvec_m$ and appropriate i and j. Furthermore, this library contains formalized proofs that this interpretation of fixed-sized bitvectors fulfills the equations *1)* through *7)* above; Thus, the bitvector theory in Definition 3.2 is consistent.

4 A Simple Decision Procedure

In this section we sketch a rather simple approach for deciding the bitvector theory in Section 3. Its input is a bitvector equation e, and the result satisfies the conditions for a solver given in Section 1. The algorithm proceeds roughly by

- replacing any bitvector variable $x_{[n]}$ with $x_{n-1} \otimes \ \ldots \ \otimes x_0$, where x_i are (fresh) variables of sort $bvec_1$,
- computing the composition normal form of each side,
- bitwise comparing the corresponding left-hand and right-hand sides of the equations,
- propagating the resulting equalities by processing the bitwise equalities one-by-one and building up a union-find structure,
- and finally, replacing the bitvariables with canonical representatives.

Pseudocode for this simple solver of the bitvector theory is listed in Figures 1 and 2.

We illustrate this algorithm by solving the following bitvector equation.

```
solve_1(t = u) ::=
    (t, u) ← REPLACE  EACH x ∈ vars(t, u) IN (t, u) WITH x_{k-1} ⊗ ... ⊗ x_0
              /* here, x_i are fresh bit variables */
    t_{n-1} ⊗ ... ⊗ t_0 ← α_1(t)
    u_{n-1} ⊗ ... ⊗ u_0 ← α_1(u)
    E ← {t_{n-1} = u_{n-1}, ..., t_0 = u_0}
     FOREACH t_i = u_i ∈ E DO merge(t_i, u_i) OD
    A ← ∅
     FOREACH x IN vars(t, u) DO
        A ← A ∪ {x = find(x_{k-1}) ⊗ ... ⊗ find(x_0)}
              /* with x_i the bit variables introduced above for each x */
     OD
     RETURN A


merge(t, u) ::=
     IF find(t) ≢ find(u) THEN
        union(t, u)
```

Figure 2: A Simple Algorithm for Deciding the Bitvector Theory

Example 4.1

$$x_{[3]} \otimes (y_{[4]} \otimes z_{[5]})^\wedge(5,4) \otimes 0_{[4]} = ((y_{[4]}{}^\wedge(2,1) \otimes x_{[3]})^\wedge(4,0) \otimes z_{[5]} \otimes x_{[3]})^\wedge(10,2)$$

Replacement of variables like $y_{[4]}$ with $y_3 \otimes \ldots \otimes y_0$, normalization, and bitwise comparison yields the system of equations

$$\begin{array}{rclrclrcl} x_2 & = & x_2, & x_1 & = & x_1, & x_0 & = & x_0 \\ y_0 & = & z_4, & z_4 & = & z_3, & 0_{[1]} & = & z_2 \\ 0_{[1]} & = & z_1, & 0_{[1]} & = & z_0, & 0_{[1]} & = & x_2 \end{array}$$

The next step of the algorithm processes these equations one-by-one in arbitrary order, and builds up a *union-find* structure. The first three identities are removed from this set of equations, and processing of the remaining equations yields the *canonical representatives*

$$\begin{array}{rcl} \mathit{find}(\{y_0, z_4, z_3\}) & = & z_4 \\ \mathit{find}(\{0_{[1]}, z_2, z_1, z_0, x_2\}) & = & 0_{[1]} \end{array}$$

of the resulting congruence classes. Here, the choice of canonical representatives is arbitrary as long as there are no constants in the congruence class. In these cases, the *find* algorithm is assumed to choose a constant as the canonical representative. Note also that a call to *union* fails whenever different constants are merged and solve_1 returns **false** on top level.

Finally, the newly introduced bit variables are replaced with their canonical representatives. This yields

$$\begin{array}{rcl} x_{[3]} & = & 0_{[1]} \otimes x_1 \otimes x_0 \\ y_{[4]} & = & y_3 \otimes y_2 \otimes y_1 \otimes z_4 \\ z_{[5]} & = & z_4 \otimes z_4 \otimes 0_{[1]} \otimes 0_{[1]} \otimes 0_{[1]} \end{array}$$

These terms on the variables x, y, z on the right-hand sides of the equations above can be simplified (with $a_{[2]}$ and $a_{[3]}$ fresh variables) to:

$$\begin{array}{rcl} x_{[3]} & = & 0_{[1]} \otimes a_{[2]} \\ y_{[4]} & = & a_{[3]} \otimes z_4 \\ z_{[5]} & = & z_4 \otimes z_4 \otimes 0_{[3]} \end{array}$$

This simplification step, however, is omitted in Figure 2.

Obviously, the solver in Figure 2 terminates and the worst-case computational complexity is $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n + n^2)$, with $\mid t \mid$ the length of the (larger) bitvector term t in the equation $t = u$ and n the number of bits of the bitvector t or u, since

- the canonizer α_1 is of complexity $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n)$, for each of the $\mid t \mid$ recursion steps involves – in the worst case – comparison of integers, consuming $\mathit{log}\, n$ time, and
- *merge* is called n times and its complexity is $\mathcal{O}(n)$, since the *union-find* algorithm reduces in this case to $\mathcal{O}(n)$.

In the remainder of this paper we improve this algorithm for the bitvector theory by splitting the input variables in the first step in larger chunks whenever possible.

5 Canonization

In this section we develop a semantic canonizer for the bitvector theory that fulfills the constraint for a canonizer as stated in [CLS96].

Canonization of bitvector terms is divided into two subsequent phases. The first phase normalizes a bitvector term t to an equivalent term in composition normal form (see Section 3) according to function α in Figure 3. Note that there is a non-determinism in selecting various cases in this function α.

The resulting composition normal form may still contain subterms such as $c_{[n]} \otimes c_{[m]}$ or $x_{[2]}\ \hat{}(1,1) \otimes x_{[2]}\ \hat{}(0,0)$, which can be further normalized to $c_{[n+m]}$ and $x_{[2]}$ respectively. These kinds of merging are accomplished in the second phase of canonization by the function β in Figure 3, and, consequently, a term $s = \sigma(t)$ with $\sigma(t)\ ::=\ \beta(\alpha(t))$ is a maximally connected composition normal form.

Lemma 5.1 *For all terms t, $\sigma(t)$ computes the maximally connected composition normal form*

Using this result one can prove that σ fulfills the requirements given in [CLS96] for a canonizer in Shostak's framework.

Theorem 5.2

1) An equation $t = u$ in the theory is valid if and only if $\sigma(t) \equiv \sigma(u)$.

2) If t is a term not in the theory, then $\sigma(t) \equiv t$

3) $\sigma(\sigma(t)) \equiv \sigma(t)$

4) If $\sigma(t) \equiv f(t_1, ..., t_n)$ for a term t in the theory then $\sigma(t_i) \equiv t_i$ for $1 \leq i \leq n$.

5) $vars(\sigma(t)) \subseteq vars(t)$.

$$
\begin{array}{llrcl}
\alpha(s) & ::= & \text{CASES } s \text{ OF} \quad t \otimes u & \to & \alpha(t) \otimes \alpha(u), \\
& & t_{[n]}\,\hat{}\,(n-1,0) & \to & \alpha(t_{[n]}), \\
& & t\,\hat{}\,(i,j)\,\hat{}\,(k,l) & \to & \alpha(t\,\hat{}\,(k+j, l+j)), \\
& & (t_{[n]} \otimes u_{[m]})\,\hat{}\,(i,j) & \to & \text{IF } m > i \text{ THEN } \alpha(u_{[n]}\,\hat{}\,(i,j)) \\
& & & & \text{ELSEIF } j \geq m \text{ THEN } \alpha(t_{[n]}\,\hat{}\,(i-m, j-m)) \\
& & & & \text{ELSE } \alpha(t_{[n]}\,\hat{}\,(i-m,0)) \otimes \alpha(u_{[m]}\,\hat{}\,(m-1,j)) \\
& & & & \text{ENDIF }, \\
& & \text{OTHERWISE} & & s \\
& & \text{ENDCASES} & &
\end{array}
$$

$$
\begin{array}{llrcl}
\beta(s) & ::= & \text{CASES } s \text{ OF} \quad c_{[n]} \otimes c_{[m]} \otimes u & \to & \beta(c_{[n+m]} \otimes u), \\
& & x_{[n]}\,\hat{}\,(i,j) \otimes x_{[n]}\,\hat{}\,(j-1,k) \otimes u & \to & \beta(x_{[n]}\,\hat{}\,(i,k) \otimes u), \\
& & x_{[n]}\,\hat{}\,(i,j) \otimes u & \to & x_{[n]}\,\hat{}\,(i,j) \otimes \beta(u), \\
& & \text{OTHERWISE} & & s \\
& & \text{ENDCASES} & &
\end{array}
$$

$$\sigma(t) ::= \beta(\alpha(t))$$

Figure 3: Canonizer for Bitvector Theory

Proof.

1) ($\Rightarrow$) Let $t = u$ be a valid equation and presume $\sigma(t) \not\equiv \sigma(u)$. This inequality is of the form $t_1 \otimes t_2 \otimes \ldots \otimes t_n \not\equiv u_1 \otimes u_2 \otimes \ldots \otimes u_m$ where t_i, u_j are simple terms; furthermore, let n_i and m_j respectively denote the lengths of t_i and u_j. Since $\sigma(t) \not\equiv \sigma(u)$, there is a least index i_0 such that $t_{i_0} \not\equiv u_{i_0}$.

   Case $n_{i_0} = m_{i_0}$: for $t = u$, every bit has to match. Thus, either t and u denote the same constant or extraction from the same part on the same variable; otherwise a counterexample can be constructed instantly. In any case, $t_{i_0} \neq u_{i_0}$ is a contradiction.

   Case $n_{i_0} \neq m_{i_0}$: without loss of generality let $n_{i_0} < m_{i_0}$. Then there exists a neighbor t_{i_0+1} of t_{i_0}, since t and u have the same length. t_{i_0+1} can not be combined with t_{i_0}, for $\sigma(t)$ is maximally connected. If u_{i_0} is a constant, then either t_{i_0} or t_{i_0+1} denotes something different. If u_{i_0} is a variable or an extraction term, $t_{i_0} \otimes t_{i_0+1}$ can not build up a prefix of it (for they can't be combined). This yields a contradiction.

   ($\Leftarrow$) σ respects the equations in Definition 3.2 as one can check easily.

2) In case t is not in the theory, only the OTHERWISE clause in α matches.

3) According to Lemma 5.1, $\sigma(t)$ is maximally connected. Consequently, only the OTHERWISE clause in α matches $\sigma(t)$ and none of the subterms of $\sigma(t)$ can be combined by β.

4) Again, according to Lemma 5.1, $\sigma(t)$ is maximally connected. Thus, either $\sigma(t) \equiv y$, $\sigma(t) \equiv c$, $\sigma(t) = x\,\hat{}\,(i,j)$ or $\sigma(t) \equiv t_1 \otimes \ldots \otimes t_n$, and, consequently, only the latter two cases must be considered. Since $\sigma(x) \equiv x$ we are done in the third case. The last case follows immediately from the fact that all t_i are necessarily simple terms.

$solve(t = u)$ ::=
  $t \leftarrow \sigma(t)$
  $u \leftarrow \sigma(u)$
  IF $t \equiv u$ THEN RETURN **true** ENDIF
  IF $vars(t) = \emptyset$ THEN $swap(t, u)$ ENDIF
  IF $vars(t) = \emptyset$ THEN RETURN **false** ENDIF
  $\{t_1 = u_1,\ t_2 = u_2,\ ...,\ t_m = u_m\} \leftarrow slice(\{t,\ u\})$
  $E \leftarrow \bigcup_{i=1}^{m} csolve(t_i = u_i)$
  IF **false** $\in E$ THEN RETURN **false** ENDIF
  $E_{x_1} \uplus E_{x_2} \uplus ... \uplus E_{x_p} \leftarrow E$
    /* t.i. $\{E_{x_i}\}$ is a partition of E where E_{x_i} contains all equations $x_i = ...$ */
  FOREACH $i \in \{1, ..., p\}$ DO $E_{x_i} \leftarrow slice(E_{x_i})$ OD
  FOREACH $i \in \{1, ..., p\}$ DO $lazy\_constant\_propagation(E_{x_i})$ OD
  $\{E_{x_1},\ E_{x_2},\ ...,\ E_{x_p}\} \leftarrow coarsest\_slicing(\{E_{x_1},\ E_{x_2},\ ...,\ E_{x_p}\})$
  FOREACH $i \in \{1, ..., p\}$ DO $equality\_propagation(E_{x_i})$ OD
  RETURN $\bigwedge_{i=1}^{p} \left(x_i = \beta(find(E_{x_i}))\right)$
   / * '*find*' meaning a composition of representants of each column in E_{x_i} * /

Figure 4: Solver for Bitvector Theory

5) σ does not introduce any new variables.

This completes the proof. *q.e.d.*

6 An Efficient Solver

The decision procedure in Section 4 can be improved considerably, since, in most cases, it is not necessary to reduce the problem to a bitwise comparison of t and u. In this respect, the solver *solve* in Figure 4 is much more delicate, for it does split large chunks only if needed.

Given an equation $t = u$, this solver first canonizes (see Section 5) both sides of the equation to obtain the equation $\sigma(t) = \sigma(u)$ over the maximally connected composition normal forms $\sigma(t)$ and $\sigma(u)$. The equation 4.1 of our running example

$$\underbrace{x_{[3]} \otimes (y_{[4]} \otimes z_{[5]})\hat{\ }(5,4) \otimes 0_{[4]}}_{t} = \underbrace{((y_{[4]}\hat{\ }(2,1) \otimes x_{[3]})\hat{\ }(4,0) \otimes z_{[5]} \otimes x_{[3]})\hat{\ }(10,2)}_{u}$$

canonizes to

$$\underbrace{x_{[3]} \otimes y_{[4]}\hat{\ }(0,0) \otimes z_{[5]}\hat{\ }(4,4) \otimes 0_{[4]}}_{\sigma(t)} = \underbrace{x_{[3]} \otimes z_{[5]} \otimes x_{[3]}\hat{\ }(2,2)}_{\sigma(u)}$$

The next essential step of the algorithm, called *slicing*, computes composition normal forms $t_1 \otimes \ldots \otimes t_m$ and $u_1 \otimes \ldots \otimes u_m$ of $\sigma(u)$ and $\sigma(t)$ respectively, such that each t_i and u_i are of the same length; moreover, it does so by minimizing the number m, called *granulation*, of simple terms on each side. This can be performed in $\mathcal{O}(n)$ time by introducing a boolean array and marking the places where to cut (we assume that n is small enough, so that an equality test of the occuring numbers can be performed in constant time). Slicing of the canonized equation above, for example, leads to the following equation.

$csolve(t,u)$ ::=
IF $t \equiv u$ THEN RETURN $\emptyset$ ENDIF
IF $vars(t) = \emptyset$ THEN $swap(t,u)$ ENDIF
IF $vars(t) = \emptyset$ THEN RETURN **false** ENDIF
IF $vars(u) = \emptyset$ THEN RETURN $\{var(t) = u\}$ ENDIF
IF $vars(t) = vars(u)$
THEN /* both terms are necessarily extractions */
CASE
both parts do not overlap
$fresh(a^{(1)}), fresh(a^{(2)}), fresh(a^{(3)}), fresh(b^{(1)})$
RETURN $\{var^{\dagger}(t) = a^{(1)} \otimes b^{(1)} \otimes a^{(2)} \otimes b^{(1)} \otimes a^{(3)}\}$
$\frac{1}{2} \cdot length(t) > length(overlapping\ part)$
$fresh(a^{(1)});\ fresh(a^{(2)});\ fresh(b^{(1)});\ fresh(b^{(2)})$
RETURN $\{var(t) = a^{(1)} \otimes b^{(1)} \otimes b^{(2)} \otimes b^{(1)} \otimes b^{(2)} \otimes b^{(1)} \otimes a^{(4)}\}$
$length(overlapping\ part) \geq \frac{1}{2} \cdot length(t)$
$\Gamma \leftarrow leftmost\_position(t,u) - rightmost\_position(t,u) + 1$
$\Delta \leftarrow \mid left\_position(t) - left\_position(u) \mid$
$\Lambda \leftarrow \Gamma$ MOD Δ
IF $\Lambda = 0$ THEN $fresh(a^{(1)});\ fresh(a^{(2)});\ fresh(b^{(1)} : bvec[\Delta])$
RETURN $\{var(t) = a^{(1)} \otimes b^{(1)} \otimes ... \otimes b^{(1)} a^{(2)}\}$
ELSE $fresh(a^{(1)});\ fresh(a^{(2)});$
$fresh(b^{(1)} : bvec[\Lambda]);\ fresh(b^{(2)} : bvec[\Delta - \Lambda])$
RETURN $\{var(t) = a^{(1)} \otimes b^{(1)} \otimes b^{(2)} \otimes ...$
$\otimes b^{(1)} \otimes b^{(2)} \otimes b^{(1)} \otimes a^{(2)}\}$
ENDIF
ENDCASES
ELSE $fresh(a^{(1)});\ fresh(a^{(2)});\ fresh(a^{(3)});\ fresh(a^{(4)});\ fresh(c^{(1)});$
RETURN$\{var(t) = a^{(1)} \otimes c^{(1)} \otimes a^{(2)}$, var(u) $= a^{(3)} \otimes c^{(1)} \otimes a^{(4)}\}$
ENDIF

Figure 5: Subprocedure *csolve*

$$\begin{array}{ccccccccccc} & \underbrace{x_{[3]}}_{t_1} & \otimes & \underbrace{y_{[4]}\ \hat{}(0,0)}_{t_2} & \otimes & \underbrace{z_{[5]}\ \hat{}(4,4)}_{t_3} & \otimes & \underbrace{0_{[3]}}_{t_4} & \otimes & \underbrace{0_{[1]}}_{t_5} \\ = & \underbrace{x_{[3]}}_{u_1} & \otimes & \underbrace{z_{[5]}\ \hat{}(4,4)}_{u_2} & \otimes & \underbrace{z_{[5]}\ \hat{}(3,3)}_{u_3} & \otimes & \underbrace{z_{[5]}\ \hat{}(2,0)}_{u_4} & \otimes & \underbrace{x_{[3]}\ \hat{}(2,2)}_{u_5} \end{array}$$

Obviously, this equation holds if and only if the conjunction of the equations in the set E holds.

$$\begin{aligned} E \ ::= \ \{ & x_{[3]} = x_{[3]}\ , \\ & y_{[4]}\ \hat{}(0,0) = z_{[5]}\ \hat{}(4,4), \\ & z_{[5]}\ \hat{}(4,4) = z_{[5]}\ \hat{}(3,3), \\ & 0_{[3]} = z_{[5]}\ \hat{}(2,0), \\ & 0_{[1]} = x_{[3]}\ \hat{}(2,2)\} \end{aligned}$$

Since E contains only equations over simple terms, the problem of solving an equation over arbitrary terms is reduced to solving equations over simple terms.

†if $\mid vars(t) \mid = 1$, let the function *var* map to this variable

Solving equations over simple terms: the function *csolve* in Figure 5 solves an equation $t = u$ over simple terms t and u for all variables in this equation. This process involves the introduction of fresh variables in order to describe the requirements of shared parts within a bitvector and overlapping part between different bitvectors. Consequently, we distinguish between three different kinds of fresh variables, namely variables of kind A, B, and C, where

- variables of **kind A** occur in but one place and nowhere else.
- variables of **kind B** occur in exactly one equation at least two times.
- variables of **kind C** occur in exactly two different resulting equations, once in each.

Furtheron, a, b or c denote (fresh) variables of kind A, B, and C, respectively. Solving equations over simple terms results in a set of solved equations, and one naturally distinguishes between 4 different situations; the set of solved equations:

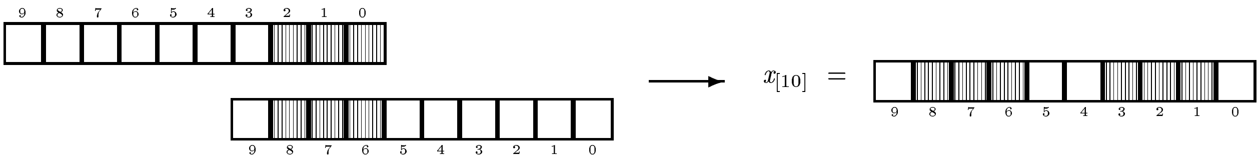
- is empty (the empty set is interpreted as **true**), if $t \equiv u$.
- contains **false** , if and only if t and u are different constants.
- contains exactly one solved equation of the form $x = s$, with $x \in \mathit{vars}(t = u)$ and s is a simple term (over constants and fresh variables of kinds A and B). This case occurs whenever x is the only variable in $t = u$.
- contains exactly two solved equations of the form $x = s_1$, $y = s_2$, with $x, y \in \mathit{vars}(t = u)$ and s_1, s_2 are terms containing fresh variables of kinds A and C.

Solving equations of the form $x\hat{}(.,.) = x\hat{}(.,.)$ is the only non-trival case, since, in these situations, one has to distinguish between the following three subcases.

In the **first** case we consider, the extracted parts on each side of the equation do not overlap. In this case, a fresh variable of kind B, is introduced. This variable occurs exactly twice in the equation, and the blank positions are denoted by variables of kind A. Altogether, the solved equation is of the form

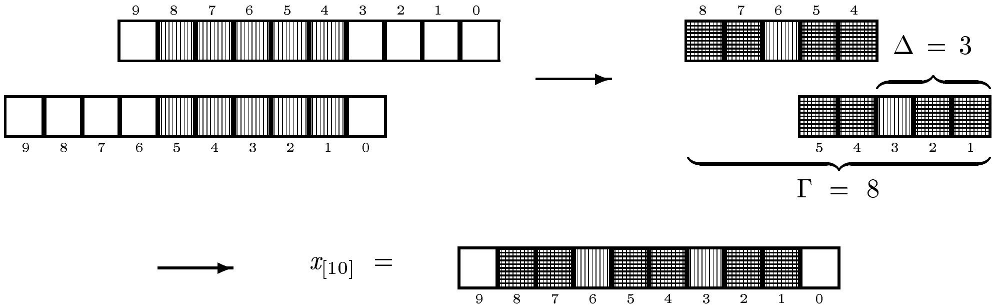
$$x = a^{(1)} \otimes b^{(1)} \otimes a^{(2)} \otimes b^{(1)} \otimes a^{(3)}$$

Note also that some or even all of the $a^{(\cdot)}$ may be omitted if they are of length 0. Consider, for example, the equation $x_{[10]} \ \hat{}\ (8,6) = x_{[10]} \ \hat{}\ (2,0)$:



Consequently, $\quad x_{[10]} \ = \ a^{(1)}{}_{[1]} \otimes b^{(1)}{}_{[3]} \otimes a^{(2)}{}_{[3]} \otimes b^{(1)}{}_{[3]}$

Second, the extracted parts on both sides of the equation overlap. Consider, for example, the equation $x_{[10]}\hat{}(8,4) = x_{[10]}\hat{}(5,1)$. As the picture below shows, we denote the *non-overlapping* part on each side with Δ and the overall length of the affected region with Γ. Apparently, the number of *overlapping* bits is 2 in our example and $\Gamma - 2\,\Delta$ in the general case. This part has to be repeated *thrice* and the positions 6 and 3 contain the same bit in the solved form.

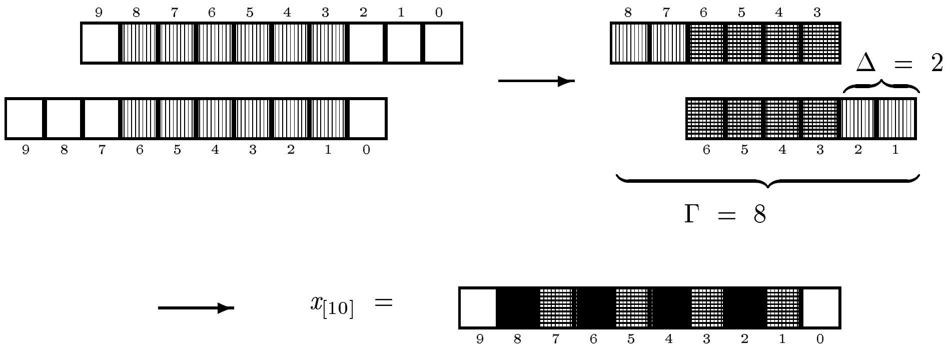


Consequently, the solved form of the example above is given by the equality

$$x_{[10]} \;=\; a^{(1)}{}_{[1]} \otimes b^{(1)}{}_{[2]} \otimes b^{(2)}{}_{[1]} \otimes b^{(1)}{}_{[2]} \otimes b^{(2)}{}_{[1]} \otimes b^{(1)}{}_{[2]} \otimes a^{(2)}{}_{[1]}$$

The length of the vector $b^{(1)}$ is determined by $\Gamma - 2\,\Delta$ and the length of $b^{(2)}$ is $3\,\Delta - \Gamma$. This is true, however, only if the overlapping part is not too big. At the extreme, if it is half as long as Γ, the length of the variable $b^{(2)}$ shrinks to zero, that means $b^{(2)}$ vanishes.

In the **third** case, we consider the slightly different situation where $\Gamma - 2\,\Delta \;>\; \frac{1}{2}(\Gamma - \Delta)$. This case is illustrated by the example $x_{[10]}\ {}^\wedge(8,3) \;=\; x_{[10]}\ {}^\wedge(6,1)$ where $\Gamma = 8$ and Δ=2:



Solving this equation involves the introduction of 2 fresh variables of kind A to pad the blank ends of the bitvectors and 2 fresh variables of kind B (and length 1) to represent the black and the grey squares above; thus a valid result is given by:

$$x_{[10]} \;=\; a^{(1)}{}_{[1]} \otimes b^{(1)}{}_{[1]} \otimes b^{(2)}{}_{[1]} \otimes b^{(1)}{}_{[1]} \otimes b^{(2)}{}_{[1]} \otimes b^{(1)}{}_{[1]} \otimes b^{(2)}{}_{[1]} \otimes b^{(1)}{}_{[1]} \otimes b^{(2)}{}_{[1]} \otimes a^{(2)}{}_{[1]}$$

Furthermore, the sequence $b^{(1)}{}_{[1]} \;\otimes\; b^{(2)}{}_{[1]}$ can be replaces by $b^{(3)}{}_{[2]} \;=\; b^{(1)}{}_{[1]} \;\otimes\; b^{(2)}{}_{[1]}$. This yields the simplified equation $x_{[10]} \;=\; a^{(1)}{}_{[1]} \otimes b^{(3)}{}_{[2]} \otimes b^{(3)}{}_{[2]} \otimes b^{(3)}{}_{[2]} \otimes b^{(3)}{}_{[2]} \otimes a^{(2)}{}_{[1]}$. One can generalize this example by introducing fresh variables of kind B of length $\Gamma \bmod \Delta$‡ and $\Delta - \Gamma \bmod \Delta$. Then the resulting equation is:

$$x_{[n]} \;=\; a^{(1)} \otimes \underbrace{b^{(1)} \otimes b^{(2)}}_{\hat{=}\Delta} \otimes \;\ldots\; \otimes b^{(1)} \otimes b^{(2)} \otimes \underbrace{b^{(1)}}_{\text{the part of } \Delta\text{, that does not ``fit'' in } \Gamma} \otimes a^{(2)}$$

Altogether, solving the equations over simple terms in our running example yields the following set of equations:

$$\begin{aligned} \mathit{csolve}(x_{[3]} \;= x_{[3]}\;) &\;=\; \varnothing \\ \mathit{csolve}(y_{[4]}\ {}^\wedge(0,0) = z_{[5]}\ (4,4)) &\;=\; \{y = a^{(1)}{}_{[3]} \otimes c^{(1)}{}_{[1]}\ , z = c^{(1)}{}_{[1]} \otimes a^{(2)}{}_{[4]}\ \} \end{aligned}$$

‡this value can be zero and the first variable may therefore - as in the given example - vanish

$$\begin{array}{rcl} csolve(z_{[5]}\ \hat{}\,(4,4) = z_{[5]}\ (3,3)) & = & \{z = b^{(1)}{}_{[1]} \otimes b^{(1)}{}_{[1]} \otimes a^{(3)}{}_{[3]}\ \} \\ csolve(0_{[3]} = z_{[5]}\ \hat{}\,(2,0)) & = & \{z = a^{(4)}{}_{[2]} \otimes 0_{[3]}\} \\ csolve(0_{[1]} = x_{[3]}\ \hat{}\,(2,2)) & = & \{x = 0_{[1]} \otimes a^{(5)}{}_{[2]}\ \} \end{array}$$

Now the corresponding equations with the *lhs* x_i are captured in sets (or, as we refer to them, *blocks*) E_{x_i}, and a slicing operation is performed on every block. In our running example, this operation yields the three blocks below.

$$E_x = \left|\ 0_{[1]}\ \middle|\ a^{(5)}{}_{[2]}\ \right| \qquad E_y = \left|\ a^{(1)}{}_{[3]}\ \middle|\ c^{(1)}{}_{[1]}\ \right| \qquad E_z = \left|\begin{array}{c|c|c} c^{(1)}{}_{[1]} & a^{(2)\prime}{}_{[1]} & a^{(2)\prime\prime}{}_{[3]} \\ b^{(1)}{}_{[1]} & b^{(1)}{}_{[1]} & a^{(3)}{}_{[3]} \\ a^{(4)\prime}{}_{[1]} & a^{(4)\prime\prime}{}_{[1]} & 0_{[3]} \end{array}\right|$$

Lazy propagation of constants: If our equation contains a constant, it is simpler in some way. If two constants "at the same place" do not match, it is unsolvable,[§] if they match, forget the second one. We can view at a construct E_{x_i} as a block with several lines and columns, each of which of the same width. Each place contains a constant or a fresh variable, which possibly has been sliced (e.g. $a^{(1)}{}_{[7]}$ might split up into $a^{(1)\prime}{}_{[3]}$, $a^{(1)\prime\prime}{}_{[2]}$, and $a^{(1)\prime\prime\prime}{}_{[2]}$). To illustrate these concepts, we introduce a new example:

$$E_{x_i} = \left|\begin{array}{c|c|c|c} \overbrace{\qquad\qquad}^{\textit{width}\ 2} & \overbrace{\qquad\qquad}^{\textit{width}\ 3} & \overbrace{\qquad\qquad}^{\textit{width}\ 2} & \overbrace{\qquad\qquad}^{\textit{width}\ 2} \\ 1 & a^{(1)\prime} & a^{(1)\prime\prime} & a^{(1)\prime\prime\prime} \\ a^{(2)\prime} & a^{(2)\prime\prime} & b^{(1)} & b^{(1)} \\ a^{(3)\prime} & a^{(3)\prime\prime} & a^{(3)\prime\prime\prime} & c^{(1)} \\ 1 & a^{(4)\prime} & a^{(4)\prime\prime} & a^{(4)\prime\prime\prime} \\ a^{(5)\prime} & a^{(5)\prime\prime} & a^{(5)\prime\prime\prime} & 0 \\ a^{(6)} & c^{(2)} & a^{(7)\prime} & a^{(7)\prime\prime} \end{array}\right|$$

In the first column, the constants match. For this, it is consistent and the variables $a^{(1)\prime}$, $a^{(2)\prime}$, $a^{(3)\prime}$, $a^{(5)\prime}$ and $a^{(6)}$ are just replaced by $1_{[2]}$ (which has no further consequences, for variables of the kind A do not occur anywhere else). The forth column evaluates to $0_{[2]}$, wich affects $b^{(1)}$, $c^{(1)}$, several variables of kind A just vanish, and $0_{[2]}$ gets propagated to column 3 (for the identity of both columns via $b^{(1)}$); this process yields the block:

$$E_{x_i} = \left|\begin{array}{c|c|c|c} \overbrace{\qquad\qquad}^{\textit{width}\ 2} & \overbrace{\qquad\qquad}^{\textit{width}\ 3} & \overbrace{\qquad\qquad}^{\textit{width}\ 2} & \overbrace{\qquad\qquad}^{\textit{width}\ 2} \\ 1 & a^{(1)\prime} & 0 & 0 \\ 1 & a^{(2)\prime\prime} & 0 & 0 \\ 1 & a^{(3)\prime\prime} & 0 & 0 \\ 1 & a^{(4)\prime} & 0 & 0 \\ 1 & a^{(5)\prime\prime} & 0 & 0 \\ 1 & c^{(2)} & 0 & 0 \end{array}\right| \qquad c^{(1)} = 0_{[2]}$$

Variable $c^{(2)}$ is not considered any more, but the equality of $c^{(1)}$ with a constant is propagated into the other block E_{x_j}, where $c^{(1)}$ occurs. If $c^{(1)}$ was but a *part* of a variable of kind C, then this split has to be performed in the block E_{x_j}, too. This is sensible at this time, since every propagated constant deletes at least one other variable and shrinks width of the blocks to a minimal size.

Coarsest Slicing: A *coarsest slicing* is a transformation of a set of equations of the form $x = t$, where t is in composition normal form, such that the cross-references between the terms in composition normal form on the right hand sides are resolved. More precisely, if a fresh variable c of kind C is split up into several parts $c', c'', \ldots$ in one equation (and possibly into parts $\hat{c}, \hat{\hat{c}}, \ldots$ in another one) these split-ups are sliced with each other, thus increasing the number of splinters of c, but computing what is the coarsest granulation

[§]given valid terms, the only case of getting **false** , as a fact

```
lazy_constant_propagation(E_{x_i}) ::=
    IF  E_{x_i} contains only one term
    THEN RETURN
    ELSE  | column_1 | column_2 | ... | column_r |← E_{x_i}
          FOR i FROM 1 TO r DO
              m ← width(column_i)
              IF 1_[m] ∈ column_i ∧ 0_[m] ∈ column_i THEN RETURN false ENDIF
              IF const_m ∈ column_i
              THEN FOR EACH fresh variable c ∈ column_i DO
                        let E_{x_j} be the other block where c occurs
                        lazy_constant_propagation(E_{x_j} WITH c ← const_m) OD
                      FOR EACH b ∈ column_i DO
                        replace b in E_{x_i} with const_m
                        repeat the propagation there (until termination) OD
              OD
          OD
    ENDIF
```

Figure 6: Subprocedure *lazy_constant_propagation*

```
equality_propagation(E_{x_i}) ::=
    IF  E_{x_i} contains less than two lines
    THEN SKIP
    ELSE  column_1 | ... | column_r ← E_{x_i}
            FOREACH  i ∈ {1, ..., r} DO
              CHOOSE γ ∈ column_i
              FOREACH  g ∈ column_i  DO
                find(g) =! find(γ)
                γ := find(γ)
              OD
             /* t.i. perform a "merge" in this column */
            OD
    ENDIF
```

Figure 7: Equality Propagation

possible at this point of the propagation.
In our example, this operation leaves the blocks untouched; given the blocks $E_v = |\ c'\ |\ c''\ |\ a\ |$ and $E_w = |\ c\ |$, however, E_w gets updated to $|\ c'\ |\ c''\ |$.

Propagation of equalities: at last one has to transform all blocks to the coarsest slicing, so all references between them can be made explicit. The principle herefore is very much the same as the naive method in section 4. However, it is applied on (hopefully) vast parts of the variables instead of tiny bits. And, it is not necessary to check the consistency with the constants, for any conflict would have already occured in the lazy constant propagation step.

Now the constants in our running example are propagated, with the only effect that ${a^{(2)}}''_{[3]}$ and ${a^{(3)}}_{[3]}$ in E_z disappear. The coarsest slicing is already achieved, for ${c^{(1)}}_{[1]}$ is not split up in E_y or E_z. This results in

$$\begin{array}{rcllllll} solve(E) & = & \{ & x_{[3]} & = & 0_{[1]} \otimes {a^{(5)}}_{[2]} \ , \\ & & & y_{[4]} & = & {a^{(1)}}_{[3]} \otimes {c^{(1)}}_{[1]} \ , \\ & & & z_{[5]} & = & {b^{(1)}}_{[1]} \otimes {b^{(1)}}_{[1]} \otimes 0_{[3]} & \} \end{array}$$

Requirements on the solver: in the following we prove that *solve* in Figure 4 is indeed a correct and complete solver for the given bitvector theory.

Theorem 6.1 *Let* $\{e_1, \dots, e_n\}$ ::= $solve(e)$*; then:* $e \Leftrightarrow (e_1 \wedge \dots \wedge e_n)$ *is valid in the bitvector theory.*

Proof Outline. The equation e is assumed to be of the form $t = u$. In the first steps, canonization and slicing on the canonical forms of t, u yields an equivalent equation of the form

$$t_1 \otimes \dots \otimes t_m = u_1 \otimes \dots \otimes u_m$$

Since t_i and u_i have equal lengths, for $i = 1, \dots, m$, this equation is equivalent to the following conjunction of equations over simple terms:

$$\bigwedge_{i=1}^{m} (t_i = u_i) \qquad (\star)$$

These equations are processed successively by the function *csolve*. It is left to the reader to check that the algorithm in Figure 5 yields a system of equations that is equivalent to $(\star)$ and where the terms on the left hand sides consist solely of basic variables; i.e. a variable in either t or u. Moreover, the right hand sides only contain constants and fresh variables of kinds A, B, or C. This set of equations is rearranged into blocks containing the right hand sides for each basic variable. Obviously, slicing on each block yields "equivalent" blocks. In this way, we consider blocks to be transformed according to the coarsest slicing. In a next step, equalities within each column are propagated. Lazy constant propagation anticipates propagation of columns containing constants. It is not difficult to check that all the transformations in the propagation step are equality-preserving. *q.e.d.*

Moreover, the bitvector decision procedure *solve* can be readily used in Shostak's framework for deciding combinations of theories, since it fulfills, besides Theorem 6.1 Shostak's requirements for individual solvers as stated in [CLS96].

Theorem 6.2 *Let* E ::= $solve(e)$*; then:*

a) $E \in \{\textbf{true}, \textbf{false}\}$ *or* $E \equiv \bigwedge_i (x_i = t_i)$.

b) If $vars(e) = \emptyset$ *then* $E \in \{\textbf{true}, \textbf{false}\}$.

c) If $E \equiv \bigwedge_i (x_i = t_i)$ *then the following holds:*

1. $x_i \in \mathrm{vars}(e)$
2. *for all* i, j: $x_i \notin \mathrm{vars}(t_j)$
3. *for all* $i \neq j$: $x_i \neq x_j$
4. *for all* i: $\sigma(t_i) = t_i$.

Proof Outline. Parts a) and b) are checked easily, so let us focus on c):

(i) *csolve* returns just terms of the form $x = t$, where $x \in vars(e)$ and $vars(t) \cap vars(e) = \emptyset$.
(ii) every *rhs* of a resulting equation (from *csolve*) is a composition over fresh variables and/or constants.
(iii) every resulting equation (from *csolve*) with the same *lhs* x is collected in E_x by the partition.

The statements c.1 and c.2 follow from (i) and (ii), for the separation between basic variables on the lhs and fresh variables on the rhs of equations is maintained throughout the algorithm. *find*(E_x) produces only one representative term t, being a composition of representatives of each column (and therefore a flat term); statement c.3 follows with (iii). Finally, application of β with this t (in the last line of *solve*) results in a maximally connected composition normal form, thus c.4 is true. *q.e.d.*

Time Complexity: finally, it is shown that the worst-case time complexity of the solver *solve* in figure 4 is $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n + n^2)$.

As shown in section 4, the naive solver has complexity $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n + n^2)$. In the worst case, the solver in Figure 4 splits up every variable into atomic bits. In this case, the complicated algorithm reduces to the naive method (modulo some simple tests) of splitting bitvector equations into equations over bits from the beginning. This split up is not too time consuming, since there are only $\mathcal{O}(n)$ steps, and propagation of equalities takes $\mathcal{O}(n^2)$ time.

On the other hand, the complexity of our solver is not *worse* than $\mathcal{O}(n^2)$. The canonizer in figure 3 works in $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n + n\ \mathit{log}\, n)$ time. The function α visits each subterm a constant number of times, since the topmost bitvector operator is eliminated in each step. The case analysis takes at most $\mathcal{O}(\mathit{log}\, n)$ time, since it involves only comparison of integers which can be coded in $\mathit{log}\, n$ bits. Thus, the complexity of α is bound by $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n)$. In phase β the algorithm makes as many steps as there are atomic extractions and each check is performed in $\mathcal{O}(\mathit{log}\, n)$ time, too. The *slicing* can be computed efficiently by transforming all terms into lists of booleans — a **true** denotes the position of a separation. Then a big OR-operation is applied onto these lists, resulting in a list that marks the places of necessary separations with **true**. This — and the separation — takes $\mathcal{O}(n)$ time. It is a bit tricky to determine the complexity of the propagation operators; but in a worst-case estimation one can say, that

- *lazy_constant_propagation* deletes variables from the terms — if they have to be propagated into other blocks, then some of the variables *there* will vanish; for this, this propagation just makes the procedure faster;
- a slicing to atomic terms possibly has to be performed; this can be done in $\mathcal{O}(n)$ time by introducing a boolean vector of length n and "mark" the positions of cuts
- the equality of terms (which have not to be split up any more) has to be propagated; this takes (as shown above) at most $\mathcal{O}(n^2)$ time

Altogether, this yields the time complexity $\mathcal{O}(\mid t \mid \cdot \mathit{log}\, n + n^2)$.

7 Final Remarks

The main achievement of this paper is the development of an efficient decision procedure for the fundamental theory of fixed-sized bitvectors with composition and extraction. To the best of our knowledge this is the first time that a specialized, efficient decision procedure for this bitvector theory has been developed. Moreover, the resulting decision procedure can readily be used in Shostak's procedure for combining decision procedures.

The bitvector decision procedures have been implemented and integrated with the decision procedures of the PVS prover. Preliminary tests suggest that the improved algorithm as described in Sections 5 and 6 outperforms the simple algorithm in Section 4, which relies on bitwise comparison, by an order of magnitude. Another interesting observation lies in the fact that the run-time of a number of examples we have dealt with is largely independent of the size of the bitvectors involved. Clearly, more experiments are needed to substantiate these claims. Most of the examples we have dealt with so far have been extracted from the verification of an industrial-strength microprocessor [SM95]. Since many of these examples involve — besides composition and extraction — additional operations on bitvectors like bitwise logical operations and shifting, we had to extend the core decision procedure accordingly. Further work includes extensions of the bitvector decision procedures to deal with arbitrary-sized bitvectors, variable extractions and with arithmetic interpretations of bitvectors.

Acknowledgements: Thanks to J. Skakkebaek for many useful comments, M.K. Srivas for supplying interesting test examples, and the second author expresses his gratitude to F.W. von Henke and J. Rushby for supporting a fruitful visit at SRI International, Menlo Park.

References

[CKM+91] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink. EVES: An Overview. volume 551 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, Noordwijkerhout, The Netherlands, October 1991.

[CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's Decision Procedure for Combination of Theories. In *Proc. of CADE'96*, Lecture Notes in Computer Science. Springer-Verlag, 1996. Accepted for Publication.

[Com93] Computer Science Laboratory, SRI International, Menlo Park, CA. *User Guide for the EHDM Specification Language and Verification System, Version 6.1*, February 1993. Three volumes.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the Common Subexpression Problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[Gal87] J.H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* J. Wiley & Sons, 1987.

[LGvH+79] D.C. Luckham, S.M. German, F.W. von Henke, R.A. Karp, P.W. Milne, D.C. Oppen, W. Polak, and W.L. Scherlis. Stanford Pascal Verifier User Manual. CSD Report STAN-CS-79-731, Stanford University, Stanford, CA, March 1979.

[NO79] G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] G. Nelson and D.C. Oppen. Fast Decision Procedures on Congruence Closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[Rue96] H. Rueß. Hierarchical Verification of Two-Dimensional High-Speed Multiplication in PVS: A Case Study. In M.K. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*, number 1166 in Lecture Notes in Computer Science. Springer-Verlag, November 1996.

[Sho84] R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SM95] M.K. Srivas and S.P. Miller. Formal Verification of the AAMP5 Microprocessor. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.