Solving Bit-Vector Equations

—

A Decision Procedure for Hardware Verification

Diploma Thesis at the University of Ulm

Faculty of Computer Science

submitted by:

Michael Oliver Möller

1. Supervisor: *Prof. Dr. Friedrich W. von Henke*
2. Supervisor: *Prof. Dr. Uwe Schöning*

1998

[Patchlevel: 16-April-1998]

In the Beginning

the world was void.

And then a voice came and spoke:
There shalt be Zero and One.
There shalt be a difference
between dark and light,
earth and sky
water and land.

Humanity came.
And they claimed
to discover things like
night and day
truth and beauty
mind and matter.

But, when you look close enough,
all you have are Zeroes and Ones.

Contents

Chapter 1

Introduction

The pure and simple truth
is rarely pure and never simple.
(Oscar Wilde)

Solving equations is a task as old as math itself. The concepts of generating solutions for arithmetic terms, for example, are well understood and operationalized in the sense that they can be executed on computing devices. This thesis is dedicated to the automation of solving an equational theory that is less popular but highly interesting in the realm of computer science, namely the theory of bit-vectors.

A Bit-Vector is a Bit-Vector

As the name suggests, bit-vectors are but a special case of common array-like data-structures. However, operations on bit-vectors like concatenation and extraction of contiguous parts are untypical for vectors and rather remind one of strings. Furthermore, the property of a binary alphabet not only limits the expressiveness but also offers a characteristic that can be utilized.

Of course, bit-vectors can be encoded by means of arrays, strings or natural numbers. In this thesis it is claimed, however, that these approaches water down the interesting peculiarities of bit-vector terms that are starting points for solving bit-vector equations.

Solving Equations in Hardware Verification

From an abstract point of view, solving of equations just makes the information contained more explicit. If the equation is trivial or unsatisfiable solving yields **true** or **false** respectively; otherwise, an equivalent solved system of equations is computed.

In the context of hardware verification, efficient mechanization of solving is required in order to process complex and tedious proofs. Here, formulae do seldom involve only statements about bit-vectors but are rather a conglomerate of various theories like natural numbers, lists and bit-vectors. This motivates the idea to embed solver algorithms for numerous theories in a capacious framework.

Deciding Combinations of Quantifier-Free Theories

A prominent approach for deciding the combination of theories is presented by Shostak [Sho84]. The key to his algorithm 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. By means of this technique, simplifiers for individual unquantified first-order theories are merged into a single procedure.

In addition, Shostak's method requires that each distinct theory has the property of *algebraic solvability*. A 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.

The Relevance of this Thesis

Although bit-vectors are a fundamental data structure, apparently they have not attained proper attention in the past. In particular, the characteristics of the theory of bit-vectors are widely neglected and an efficient concept for solving bit-vector equations has not yet been etablished. In industrial sized applications like the formal verification of the AAMP5 microprocessor by Miller and Srivas [MS95], a lack of efficient automatization proved to be a major bottelneck.

This thesis investigates the problem of solving bit-vector equations both from a theoretical and practical side. Complexity and decidability of bit-vector theories including concatenation, extraction, bit-wise boolean operations and arithmetic are observed. Boolean operations on are introduced by means of ordered binary decision diagrams, which also provide a conceptually clean way to encode arithmetic. Generalizations that allow the width of bit-vector variables to be unknown are discussed as well. Surprisingly, extensions that maintain the fixed size lead to theories that are releated in the way, that for all of them the word problem is NP-complete. As expected, far reaching extention of the bit-vector theory leads neccessarily to the incompleteness of any solver algorithm. A proof is obtained by reduction of the halting problem. Furthermore, a more accurate characterization of the expressiveness of solving is developed. In general, solving algorithms can be utilized to decide quantified equations, which is made explicit in the Quantification Lemma 2.9.

On the practical side, several approaches for solving bit-vector theories are explored in this thesis. The task of solving fixed-sized equations is performed by means of a translation to weak second order monadic logic, an equational transformation system and an operationalized and efficient version of the latter, supported by heuristics. These approaches are corroborated by implementations and run-time experiments. Finally, a general concept for solving bit-vector equations with variable width is given. The algorithms prsented in this thesis have all been implemented and can be obtained at the Bit-Vector Research Page in Ulm:
`http://www.informatik.uni-ulm.de/ki/Bitvector/index.html`

Overview

The thesis is organized as follows. Chapter 2 includes a formal definition for the theory of bit-vectors together with a canonizer and solver. Extensions are introduced and an inspection of the complexity increase is given. In chapter 3 decidability of expressive bit-vector theories is discussed, culminating in the proof that a solver for a rather general theory cannot exist. In chapter 4, three approaches for solving fixed-sized bit-vector equations are treated and compared according to run-time performance. An intuitive extension to variable width is presented in chapter 5.

Acknowledgements

The author would like to express his gratitude to F.W. von Henke, J. Rushby and Harald Rueß. Without their kind support and Harald's nagging questions, this work on reasoning about bit-vectors would not have taken place.

Chapter 2

Basics

The point of philosophy is to start with something so simple as not to seem worth stating, and to end with something so paradoxical that no one will believe it.

(Bertrand Russell, The Philosophy of Logical Atomism)

In this chapter an equational theory of fixed-sized bit-vectors including only concatenation and extraction operations, the so-called *core theory*, is defined. Then, a canonizer and a simple solver for the core theory is presented. Finally, extensions of the core theory to bitwise boolean operations, arithmetic and variable extraction and width are discussed.

2.1 A Core Theory of Bit-Vectors

This section develops an equational theory for the fixed-sized bit-vectors of width n. Hereby the width n is constrained to be a positive natural number, since bit-vectors of width 0 are excluded. The bits of a bit-vector of width n are indexed from left to right, starting with index 0. In the following, $n, m, p, \ldots$ denote valid widths of bit-vectors. The bit-vector theory contains constants $c_{[n]}$ of width n, *concatenation* $t \otimes u$ of bit-vectors t and u, and *extraction* $t[j:i]$, where $i, j \in I\!N$, of $i-j+1$ many bits j through i from bit-vector t.

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

Definition 2.1 Let $\Sigma_{\otimes,[1:1]}$ be the signature

$$
\begin{array}{lcll}
\Sigma_{\otimes,[1:1]} & := & \langle & \{bvec_n | n \in I\!N^+\}, \\
& & & \{c_{[n]} | n \in I\!N^+,\ 0 \leq c < 2^n\} \cup \\
& & & \{.\ \otimes_{n,m}\ . | n, m \in I\!N^+\} \cup \\
& & & \{.\ [j:i]_n | n \in I\!N^+ \wedge i, j \in I\!N \wedge 0 \leq j \leq i < n\} \quad \rangle
\end{array}
$$

such that for appropriate n, i, and j:

$$
\begin{array}{rcl}
c_{[n]} & : & \to bvec_n \\
.\ \otimes_{n,m}\ . & : & bvec_n \times bvec_m \to bvec_{n+m} \\
.\ [j:i]_n & : & bvec_n \to bvec_{i-j+1}
\end{array}
$$

⌟

The dots to the left and to the right of function symbols indicate the use of infix notation. Extraction is assumed to bind stronger than concatenation. In the following, $x_{[n]}, y_{[m]}, z_{[p]}, \ldots$ denote variables of sort $bvec_n$, $bvec_m$, and $bvec_p$ respectively. The set of well-formed bit-vector terms is defined in the usual way and $t_{[n]}, u_{[m]}, v_{[p]}, \ldots$ denote bit-vector terms of respective widths. Subscripts are omitted whenever possible and can be inferred from the context. Moreover, $t \doteq u$ denotes syntactic equality of terms t and u, and $vars(t)$ denotes the set of variables in t.

A bit-vector term t is called *atomic* if it is a variable or a constant, and *simple terms* are either atomic or of the form $x_{[n]}[j:i]$ where $x_{[n]}$ is a variable 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 *concatenation normal form*. If, in addition, none of the neighboring simple terms denote constants and a simple term of the form $x[j:i]$ is not followed by a simple term of the form $x[i+1:k]$, then a term in concatenation normal form is called *maximally connected*.

Definition 2.2 The core theory of bit-vectors with concatenation and extraction is denoted by $BV_{\otimes,[1:1]}$. Let $\Sigma_{\otimes,[1:1]}$ be the signature from Definition 2.1; then equational properties of $BV_{\otimes,[1:1]}$ are given by the (conditional) $\Sigma_{\otimes,[1:1]}$-equalities

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

⌟

Note that well-formedness of bit-vector terms implies that $0 \leq k < j \leq i < n$ in equation 5) and $0 \leq l \leq k \leq n \wedge 0 \leq l \leq k \leq i-j$ in equation 7) above. Semantic entailment $\models$ in the equational theory above is defined in the usual way.

Fixed-sized bit-vectors of width n can be interpreted as finite functions with domain $[0..n)$ and codomain $\{0,1\}$:

$$\begin{array}{rcl}
c_{[n]} & := & \lambda x : [0..n).\ (c\ DIV\ 2^x)\ MOD\ 2 \\
s_{[n]} \otimes t_{[m]} & := & \lambda x : [0..n+m).\ \text{IF } x < n \text{ THEN } s_{[n]}(x) \text{ ELSE } t_{[m]}(x-n) \\
s_{[n]}[j:i] & := & \lambda x : [0..i-j+1).\ s_{[n]}(x+j)
\end{array}$$

with appropriate i and j. Sometimes we also use the notation $x_{[n]}(i)$ in order to refer to the i^{TH} bit "0" or "1". Finally, concatenations of "1"s are abbreviated by $\mathbf{-1}_{[n]}$, which reminds of the correct notation $(2^n\text{-}1)_{[n]}$.[a]

2.2 Solving Bit-Vector Equations

Throughout this thesis, solving is understood in terms of matching the requirements of Shostak's framework of combining decision procedures, as applied in proof assistance systems like EHDM [Com93] and PVS [ORS92]. Here, two distinct functions for each equational theory have to be implemented, which are called canonizer and solver. This demand is satisfied in this section by presenting a canonizer for the core theory and also a simple though not efficient solver. Their properties are made explicit in the following.

[a]In related work, sometimes the notation $1_{[n]}$ is found. However, this could lead to confusions.

2.2.1 Canonizing Bit-Vector Terms

Informally, a *canonizer* transforms terms of some ground equational theory T into an equivalent and well-defined canonical form. In case of $BV_{\otimes,[1:1]}$, this is the maximally connected form.

Lemma 2.1:

The algorithm in Figure 2.1 yields a maximally connected form.

Proof: Sub-procedure α dissolves encapsulated structures. This results in a concatenation of simple terms and extractions like $x_{[n]}[0 : n\text{-}1]$ are simplified to $x_{[n]}$. The sub-procedure β combines all simple terms standing abreast, if they are attachable. For example, $t_{[n]}[k : j-1] \otimes t_{[n]}[j : i]$ is attached to $t_{[n]}[k : i]$. Thus, the result of $\beta \circ \alpha$ is maximally connected. □

Theorem 2.2: [Properties of the Solver]

Let $T_{\otimes,[1:1]}$ denotes the set of well-formed terms in the core theory of bit-vectors. The procedure σ displayed in Figure 2.1 fulfills the following properties:

1. An equation $t = u$ in the theory is valid iff $\sigma(t) \doteq \sigma(u)$
2. If $t \notin T_{\otimes,[1:1]}$ then $\sigma(\mathrm{t}) \doteq \mathrm{t}$
3. $\sigma(\sigma(t)) \doteq \sigma(t)$
4. If $\sigma(t) \doteq f(t_1, \ldots, t_n)$ for a term $t \in T$, then $\sigma(t_i) \doteq t_i$ for $1 \leq i \leq n$
5. $vars(\sigma(t)) \subseteq vars(t)$

Proof: By inspection and Lemma 2.1. □

Therefore, σ fulfills the requirements on canonizers as stated in [CLS96] to allow integration in Shostak's framework. It has been shown in [CMR96], that the worst-time complexity of σ is $\mathcal{O}(|s| \cdot \log n + n \log n)$.

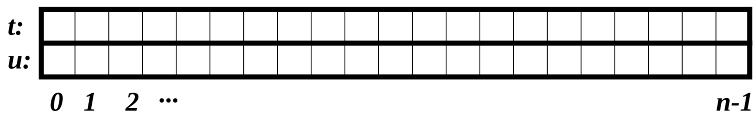
Note: In the American speech area, the separation of the technical terms *canonization* and *normalization* is sometimes a bit sloppy. In this thesis a canonical form is understood as the unique representation of a term (see 1.), whereas normalization only transforms to a well-defined form without necessarily providing this property. For example, $x_{[8]}[1 : 3] \otimes x_{[8]}[4 : 5]$ is in concatenation normal form. But the corresponding canonical term is also maximally connected, namely $x_{[8]}[1 : 5]$.

2.2.2 The Function called Solver

A solver for an algebraic theory T transforms equations in this theory into a more explicit form. In extreme cases, it returns **true** or **false** namely if the equation is valid or unsatisfiable. Otherwise, the solver returns a representation of all satisfying models, which is not neccessarily unique.

2.2.3 Solving Fixed-Sized Bit-Vector Equations

The simplest approach for solving bit-vector equations involves bit-wise splitting. A bit-vector equation $t_{[n]} = u_{[n]}$, where $t_{[n]}$ and $u_{[n]}$ are maximally connected, can be understood as a collection of equations on bits:



$$
\begin{array}{l}
\alpha(s) := \textsc{Cases}\ s\ \textsc{Of} \\
\quad \begin{array}{rcl}
t \otimes u & \rightarrow & \alpha(t) \otimes \alpha(u) \\
t_{[n]}[0 : n\text{-}1] & \rightarrow & \alpha(t_{[n]}) \\
t[j : i][l : k] & \rightarrow & \alpha(t[l + j : k + j]) \\
(t_{[n]} \otimes u_{[m]})[j : i] & \rightarrow & \textsc{If}\ i < n\ \textsc{Then}\ \alpha(t_{[n]}[j : i]) \\
 & & \textsc{Elseif}\ n \leq j\ \textsc{Then}\ \alpha(u_{[m]}[j\text{-}n : i\text{-}n]) \\
 & & \textsc{Else}\ \alpha(t_{[n]}[j : n\text{-}1]) \otimes \alpha(u_{[m]}[0 : i\text{-}n]) \\
 & & \textsc{Endif}
\end{array} \\
\quad \textsc{Endcases}
\end{array}
$$

$$
\begin{array}{l}
\beta(s) := \textsc{Cases}\ s\ \textsc{Of} \\
\quad \begin{array}{rcl}
c_{[n]} \otimes c'_{[m]} \otimes u & \rightarrow & \beta\big((c + 2^n \cdot c')_{[n+m]} \otimes u\big) \\
c_{[n]} \otimes u & \rightarrow & c_{[n]} \otimes \beta(u) \\
x_{[n]}[j : i] \otimes x_{[n]}[i + 1 : k] \otimes u & \rightarrow & \beta(\alpha(x_{[n]}[j : k]) \otimes u) \\
x_{[n]}\Big[[j : i]\Big] \otimes u & \rightarrow & x_{[n]}\Big[[j : i]\Big] \otimes \beta(u) \\
\textsc{Otherwise} & & s
\end{array} \\
\quad \textsc{Endcases}
\end{array}
$$

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

Figure 2.1: A Canonizer for Computing the Maximally Connected Form

The bit-vectors $t_{[n]}$ and $u_{[n]}$ are dissolved to their bits $t_{[n]}[0 : 0]$, $t_{[n]}[1 : 1]$ ... $u_{[n]}[n\text{-}1 : n\text{-}1]$. These bits are either constants or extractions of width one on bit-vector variables.

Each splinter of a bit-vector variable is treated like a simple boolean variable. Thus n equations in the boolean realm are obtained that can be propagated one by one. A pseudo code representation of this algorithm is given in Figure 2.2. In this form, it runs in worst-time complexity $\mathcal{O}(n^4)$, due to the Foreach - selection in line 8.

Theorem 2.3:

Let $E := s(e)$, t_i denote terms in the core theory $BV_{\otimes,[1:1]}$ and x_i represent original variables. The *solver* s in Figure 2.2 fulfils the following properties:

1. $\vdash e \quad \Leftrightarrow \quad s(e)$ is in the theory
2. $E \in \{\textbf{true}, \textbf{false}\}$ or $E = \bigwedge_i x_i = t_i$
3. If e contains no variables, then $E \in \{\textbf{true}, \textbf{false}\}$
4. If $E = \bigwedge_i x_i = t_i$, then the following holds:
   (a) $x_i \in \mathit{vars}(e)$
   (b) for all i, j : $\quad x_i \notin \mathit{vars}(t_j)$
   (c) for all i, j : $\quad x_i \neq x_j$
   (d) for all i : $\quad \sigma(t_i) = \sigma(t_i)$

Proof: By inspection.

□

$$
\begin{array}{l}
\textit{naive-solve}(t_{[n]} \stackrel{!}{=} u_{[n]}) \quad = \\
\qquad V := \textit{vars}(t_{[n]}) \cup \textit{vars}(u_{[n]}) \\
\qquad \textsc{Foreach}\ i \in \{0, \ldots, n-1\}\ \textsc{Do}\ E_i := \{\alpha(t_{[n]}[i:i]), \alpha(u_{[n]}[i:i])\}\ \textsc{Od} \\
\qquad E := \{E_1, \ldots, E_{n-1}\} \\
\qquad \textsc{While}\ \exists i, j : i \neq j : E_i \cap E_j \neq \emptyset \quad \textsc{Do} \\
\qquad\qquad E_i := E_i \cup E_j \\
\qquad\qquad E \ := E \setminus E_j \qquad \textsc{Od} \\
\qquad \textsc{Foreach}\ E_i \in E\ \textsc{Do}\ r_i := \ \textsc{If}\ c_{[1]} \in E_i \ \ \textsc{Then}\ \ c_{[1]} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{Else}\ \ v_{[m]}[i:i] \in_{CHOOSE} E_i \\
\qquad\qquad\qquad\qquad\qquad \textsc{Endif}\ \textsc{Od} \\
\qquad r(v_{[m]}[i:i] : v_{[m]} \in V) := \ \textsc{If}\ \ \exists E_j.\ \ v_{[m]}[i:i] \in E_j \ \ \textsc{Then}\ \ r_j \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{Else}\ \ v_{[m]}[i:i] \\
\qquad\qquad\qquad\qquad\qquad \textsc{Endif} \\
\qquad \textsc{Case} \quad \exists E_i : \mathbf{0}_{[1]} \in E_i \wedge \mathbf{1}_{[1]} \in E_i \ \Rightarrow \ \textsc{Return}\ \mathbf{false} \\
\qquad\qquad\qquad \forall E_i \in E : |E_i| = 1 \ \Rightarrow \ \textsc{Return}\ \mathbf{true} \\
\qquad\qquad\qquad\qquad\qquad \textsc{Else} \ \Rightarrow \ \textsc{Return} \left\{ \ v_{[m]} = \bigotimes_{i=0}^{m\text{-}1} r(v_{[m]}[i:i]) \middle| v_{[m]} \in V \ \right\} \\
\qquad \textsc{Endcase}
\end{array}
$$

Figure 2.2: A Simple Solver for the Core Theory

These properties match the requirements of Shostak's framework (cf. [CLS96]). A result in this format is referred to as being in *solved form*. The core theory has the beautiful feature of being *convex*, i.e. any most general solution can be expressed by a conjunction of equations. This property is lost while extension to variable width and therefore the notion of solving has to be extended as well. This is treated in section 2.4.5.

2.3 Extensions

In this section the core theory is extended to include operations like bit-wise boolean operations and arithmetic and also permit variable extraction and unknown width, though the latter is treated more explicitly in section 2.4. For each extension an appropriate generalization of the canonical form is discussed. First, some syntactic sugar is introduced.

2.3.1 Syntactic Sugar

The following extensions of the core theory do not enlarge the syntactic power of the core theory, for they can be completely expressed by means of concatenation and extraction.

Definition 2.3: [Syntactic Sugar]

$$
\begin{array}{rcl}
\mathit{fill}_{[n]}(b : \mathit{Bit}) & := & \Big(b \cdot (2^n\text{-}1)\Big)_{[n]} \\
\mathit{repeat}(t_{[n]}, m) & := & \textsc{If } m = 1 \ \textsc{Then} \quad t_{[n]} \\
 & & \qquad\qquad\ \ \textsc{Else} \quad t_{[n]} \otimes \mathit{repeat}(t_{[n]}, m\text{-}1) \\
 & & \textsc{Endif} \\
\mathit{ext}(t_{[n]}, l) & := & \textsc{If } n|l \quad \textsc{Then} \quad {t_{[n]}}^{l\,/\,n} \\
 & & \qquad\qquad \textsc{Else} \quad {t_{[n]}}^{l\ DIV\ n} \otimes t_{[n]}[0 : (l\ MOD\ n) - 1] \\
 & & \textsc{Endif} \\
\mathit{shift}(t_{[n]}, m) & := & \mathbf{0}_{[m]} \otimes t_{[n]} \\
\mathit{rotateleft}(t_{[n]}, m) & := & t_{[n]}[0 : n\text{-}m\text{-}1] \otimes t_{[n]}[n\text{-}m : n\text{-}1] \\
\mathit{rotateright}(t_{[n]}, m) & := & t_{[n]}[m : n\text{-}1] \otimes t_{[n]}[0 : m\text{-}1]
\end{array}
$$

⌟

The *fill* operator iterates some bit n times. If the bit is fixed, these expressions are canonized to $\mathbf{0}_{[n]}$ and $\mathbf{\text{-}1}_{[n]}$ respectively. The more flexible operation *repeat* denotes the concatenation of the same bit-vector finitely often. Like in regular expressions, this is marked by an exponent, like ${t_{[n]}}^3 = t_{[n]} \otimes t_{[n]} \otimes t_{[n]}$. A more general form of this repeat is the *extension* of a bit-vector term to a fixed length by iteration up to the desired point as defined in [BP98]. The *shift* operation adds some padding with zeroes, usually up front. A $\mathit{shift}(t_{[n]}, m)$ is equivalent to $\mathbf{0}_{[m]} \otimes t_{[n]}$. Similar are *rotations* to the left or the right that push overflowing bits back to the beginning. A $\mathit{rotateleft}(x_{[8]}, 2)$, for example, results in $x_{[n]}[0 : 5] \otimes x_{[n]}[6 : 7]$.

For these notations have an equivalent, more basic representation in the core theory, they are treated as macros. During canonization, they are unfolded to their right hand side in Definition 2.3.

2.3.2 Bit-Wise Boolean Operations

A boolean connective "$\circ$" applied on bit-vector terms is understood as the bit-wise application of this operation.

Definition 2.4

$$
s_{[n]} \circ t_{[n]} \quad := \quad \big(s_{[n]}[0:0] \ \circ \ t_{[n]}[0:0]\big) \otimes \cdots \otimes \big(s_{[n]}[n\text{-}1 : n\text{-}1] \ \circ \ t_{[n]}[n\text{-}1 : n\text{-}1]\big)
$$

⌟

Well-formedness requires that the arguments of boolean operations are bit-vector terms of equal width.

Canonization via OBDDs

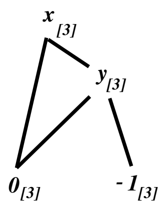
In order to obtain a canonical form of boolean bit-vector terms, the notion of *ordered binary decision diagrams* (OBDDs) [Bry92] is introduced. These structures are also known as one-time only branching programs, where the set of node variables are visited in a fixed (though arbitrary) order (cf. [vL90a, p.796ff]).

Definition 2.5 A *bit-vector OBDD* of width n is a rooted, directed and acyclic graph where the nodes are marked with bit-vector variables or extractions on bit-vector variables of width n. The only leaf nodes are $\mathbf{0}_{[n]}$ (**false**) and $\mathbf{\text{-}1}_{[n]}$ (**true**). Each non-leaf node has exactly two successors. By convention, the edges displayed on the right side in diagrams are called the *the-edges* and are followed, if the node evaluates to **true**. The left edges or *else-edges* are followed, if the evaluation yields a **false**. From root to leaves, the marks of the nodes occur according to an arbitrary but fixed order. The structure of a bit-vector OBDD is required to be maximally shared.

⌟

Obviously, bit-vector OBDDs can be utilized to represent all kinds of bit-wise boolean operations. The bit-vector term $x_{[3]} \wedge y_{[3]}$, for example, is represented as

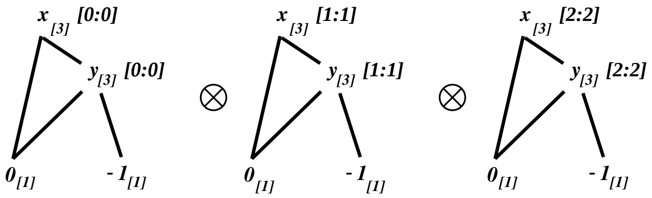
Example 2.1



$$\textsc{Ite}\ (x_{[3]}\,,\ \textsc{Ite}\ (y_{[3]}\,,\ \mathbf{\text{-}1}_{[3]}\,,\ \mathbf{0}_{[3]})\,,\ \mathbf{0}_{[3]})$$

The ITE conditional (if-then-else) is equivalent to the graph representation on the left. The intended meaning of such a bit-vector OBDD is the concatenation of n analogous binary functions, applied on the n individual bits of the bit-vector variables. Thus, the OBDD in Example 2.1 is equivalent to

Example 2.2



$$\begin{array}{l}\textsc{Ite}\ (x_{[3]}[0:0]\,,\ \textsc{Ite}\ (y_{[3]}[0:0]\,,\ \mathbf{1}_{[1]}\,,\ \mathbf{0}_{[1]})\,,\ \mathbf{0}_{[1]})\ \otimes\\ \textsc{Ite}\ (x_{[3]}[1:1]\,,\ \textsc{Ite}\ (y_{[3]}[1:1]\,,\ \mathbf{1}_{[1]}\,,\ \mathbf{0}_{[1]})\,,\ \mathbf{0}_{[1]})\ \otimes\\ \textsc{Ite}\ (x_{[3]}[2:2]\,,\ \textsc{Ite}\ (y_{[3]}[2:2]\,,\ \mathbf{1}_{[1]}\,,\ \mathbf{0}_{[1]})\,,\ \mathbf{0}_{[1]})\end{array}$$

A Canonical Form

Since both extraction and concatenation distribute over bit-vector OBDDs, the canonical form for the core theory can be extended in a straight-forward way to also include boolean operations.

Definition 2.6 [Extended Canonical Form]

A bit-vector OBDD is called *trivial* if it is of the form ITE $(t_{[n]}\,,\ \mathbf{\text{-}1}_{[n]}\,,\mathbf{0}_{[n]})$. Non-trivial bit-vector OBDDs are simple terms. A bit-vector term $u_{[n]}$ is *canonical*, if

- none of the OBDDs in $u_{[n]}$ is trivial
- $u_{[n]}$ is either a simple term or a concatenation of simple terms, such that no connected simple terms can be attached.

If no OBDDs occur, this is conform with the definition of the maximally connected form ⌟

The notion of simple terms is exteded to include also bit-vector OBDDs whose nodes are either bit-vector variables or extractions on bit-vector variables. This allows an effective representation of boolean bit-vector terms of large width while minimizing the number of introduced nodes. Trivial OBDDs ITE $(t_{[n]}\,,\ \mathbf{\text{-}1}_{[n]}\,,\mathbf{0}_{[n]})$ are canonized to $t_{[n]}$. The canonical form of $x_{[3]} \wedge y_{[3]}$ is given in example 2.1.

2.3.3 Arithmetic

In most hardware contexts, arithmetic modulo a rest class ring $\mathbb{Z}_n$ is a must. For simplicity, we restrict our attention to addition. Following the proposed concepts, other operations can be defined straight-forward. In order to express the connection between bit-vectors and natural numbers, the function pair $bv2nat(.)$ and $nat2bv_{[.]}(.)$ is introduced.

Definition 2.7 **[Arithmetic Interpretation]**

$$\begin{aligned} bv2nat(t_{[n]}) &:= \sum_{i=0}^{n-1} t_{[n]}(i) \cdot 2^i \\ nat2bv_{[n]}(c) &:= (c\ MOD\ 2^n)_{[n]} \end{aligned}$$

where $t_{[n]}$:$bvec_n$ and c:$I\!N$.

This allows to express *addition* $+_{[n]}$ as a collection of infinitely many function symbols.

Definition 2.8 **[Addition]**

$$x_{[n]} +_{[n]} y_{[n]} := nat2bv_{[n]}((bv2nat(x_{[n]}) + bv2nat(y_{[n]})\ MOD\ 2^n))$$

Boolean operations bind stronger than addition. If they are provided, *subtraction* can be introduced by means of a derived macro operation. In particular it is sufficient to define a "negative" number via the (n-1)-complement:

Definition 2.9 **[Unary Minus]**

$$-x_{[n]} := \neg x_{[n]} +_{[n]} 1_{[n]}$$

This is consistent with our declaration of $\mathbf{-1}_{[n]}$ in section 2.1, since the (n-1)-complement of -1 is a concatenation of n "1"s.

The Canonization Problem

Whenever the theory includes concatenation, extractions or boolean operations as well, the addition operation leads to canonization problems. For example, the following terms on the left hand side are equivalent to their right hand side counterpart, but there does not seem to be an intuitive criterion on which representation is to be called canonical:

Example 2.3

$$\begin{aligned} x_{[8]}+_{[8]}x_{[8]} &= 0_{[1]}\otimes x_{[8]}[0:6] \\ (x_{[8]}+_{[8]}12_{[8]})[1:7] &= x_{[8]}[1:7]+_{[7]}6_{[7]} \\ x_{[2]}+_{[2]}y_{[2]} &= (x_{[2]}\ XOR\ y_{[2]}\ XOR\ 0_{[1]}) \otimes (x_{[2]}[0:0] \wedge y_{[2]}[0:0]) \end{aligned}$$

Canonization is one of the requirements for embedding various theories into Shostak's framework of congruence closure. So either the task of canonizing has to be solved or the framework has to be enlarged to a more general one. To the best of my knowledge, neither of this has been solved yet in a way resulting in reasonable run-time (cf. also [BP98]).

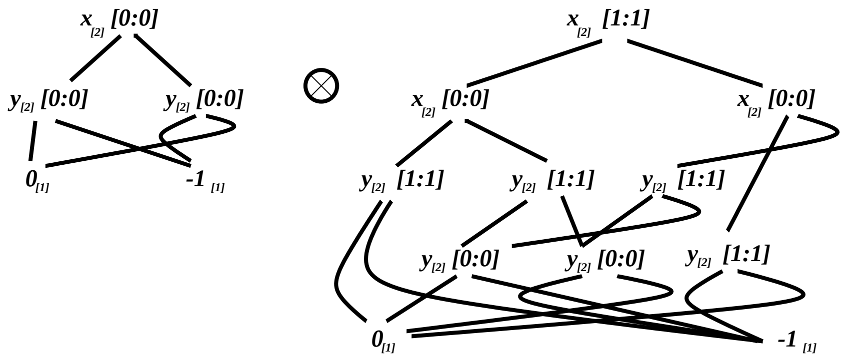
In the approach described below a canonical form is defined, thus solving the canonization problem. However, the cost—in terms of computation time—is rather high.

Arithmetic via OBDDs

Due to the finite domain, any arithmetic operation can be expressed by means of Boolean functions and consequently via OBDDs. The idea here is to split up the arguments into a concatenation of bits and compute a concatenation of OBDDs incrementally. If the operation is an addition, a ripple-carry adder describes the applied boolean functionality precisely.

Unfortunately, in many practical cases the OBDDs grow quite large as a consequence, for the number of "input bits" at a position grows with the width of the operation. For example, the term $x_{[2]}+_{[2]}y_{[2]}$ canonizes to

Example 2.4

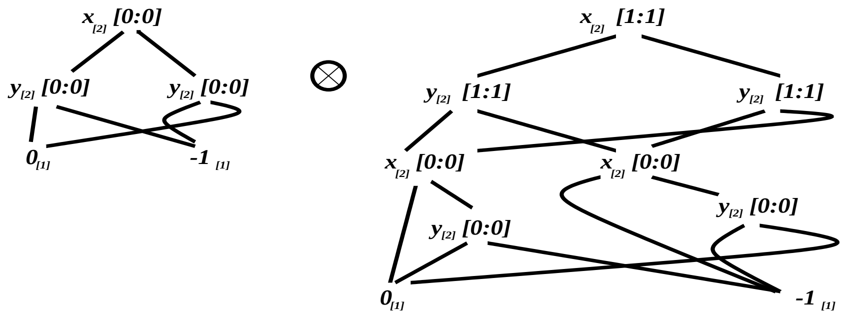


The term order applied here is $x_{[2]}[1:1] \prec x_{[2]}[0:0] \prec y_{[2]}[1:1] \prec y_{[2]}[0:0]$. As observed frequently, the size of the OBDDs is heavily dependent on the choice of this order. In fact, this order is a rather bad one. The canonization of terms $x_{[n]}+_{[n]}y_{[n]}$ uses the following resources:

n	1	2	3	4	5	6	7	8	9	10
#Nodes at [n-1]	5	11	25	55	117	243	497	1007	2029	4075
#Nodes built	7	24	61	138	295	612	1249	2526	5083	10200
time used [s][b]	0.009	0.04	0.16	0.715	3.206	14.668	63.505	285.50	1181.98	4856.48

The task of finding an optimal ordering is itself NP-complete (cf. [BW96]). The best order I found (for addition) is pushing down the least significant position. For $x_{[2]}+_{[2]}y_{[2]}$ and the order $x_{[2]}[1:1] \prec y_{[2]}[1:1] \prec x_{[2]}[0:0] \prec y_{[2]}[0:0]$. For $x_{[2]}+_{[2]}y_{[2]}$ this results in

Example 2.5



The OBDD to the right has only nine nodes (instead of eleven in Example 2.4). The advantage becomes more significant with growing width as can be deduced from the following table.

n	1	2	3	4	5	6	7	8	9	10
#Nodes at [n-1]	5	9	21	33	45	57	69	81	93	105
# Nodes built	7	23	43	63	83	103	123	143	163	183
time used [s]	0.009	0.034	0.148	0.324	0.549	0.843	1.434	1.886	2.256	2.907

[b]Measured with compiled Allegro Common Lisp 4.3, Linux on a Pentium 150MHz, 48MB of Ram.

If the operations are good natured, the construction results in small OBDDs. For example, building up a OBDD for $x_{[n]} +_{[n]} x_{[n]}$ yields only trivial nodes. Moreover, the canonical form is well defined for all expressions via the already introduced concepts. In the example 2.3 the OBDD representations of the right hand sides build the canonical form.

Note: There are classes of boolean functions, where even for the optimal ordering OBDD shows a rather bad performance. A pathological example is the hidden weight bit function (HWB):

Definition 2.10

$$\mathrm{HWB}(x_1, \dots, x_n) := x_{sum}, \text{ where } sum := x_1 + \cdots + x_n \text{ and } x_0 := 0$$

In [Bry91] and exponential lower bound in the number n of variables was proven. Refer to [BLW95] for a more explanatory proof, yielding the slightly better lower bound $2^{0.2n-1}$ for the number of OBDD nodes.

2.3.4 Variable Extraction

So far the naturals i and j in an extraction $[j : i]$ have been considered to be constants. It might be desirable not to fix the positions where extractions apply. Then, terms like $x_{[4]}[i : i+1]$ are allowed, where $i : \mathbb{N}$ is an integer variable. In general, type correctness leads to implicit constraints here. For example, the term $x_{[4]}[i : i+1]$ implies $0 \leq i \leq 2$.

Since the width of bit-vectors is still fixed, the domain these integer variables can be chosen from is finite. Apparently, the introduction of variable extraction can be interpreted as a finite case-split on the width of the contained terms. This results in a set of terms with fixed extractions that can be canonized and solved via the methods described previously. However, the task of deciding whether a bit-vector equation with variable extraction is satisfiable is NP-complete; for a proof refer to Appendix B.2.

2.3.5 Variable Width

In contrast to the extensions above, allowing variable width on bit-vector variables is a critical step. Formally, this can be achieved via introducing dependent types $bvec_n$ where n is an integer variable, $x_{[n]} : bvec_n$. This degree of freedom stipulates the solution approaches more than slightly and is discussed in detail in section 2.4.

2.3.6 A Classification of Extensions

The extensions discussed above are not isomorphic in the sense that terms from one extension can not be translated into an equivalent term in another one not being a superset in general. Thus, boolean operations, arithmetic, variable extraction and variable width generate a palette of different theories. The overall picture is sketched in Figure 2.3. The criteria used (here) to measure the "difficulty" of a theory is the complexity class of the language containing but solvable equations. More precisely, given a theory T of bit-vectors, this language is defined as

$$\mathcal{L}^T_{t=u} := \{t = u \mid t, u \in T, \text{ there exists an assignment } \alpha \text{ on } vars(t) \cup vars(u) \text{ with } \alpha \models t = u\}$$

The complexity class of the language $\mathcal{L}^T_{t=u}$ is stated in square brackets. Figure 2.3 is also anticipating some later results and refers to an *extended counting theory* that is defined in Appendix A.3 under the abbreviation $\exists BV_{[n]}$. The NP-hardness of the bit-vector theory only with composition and variable width is shown in Appendix A.2 (it is still open whether it is *in NP*) and the undecidability of the core theory with variable extraction, arithmetic, boolean operations and variable width is treated in section 3.3. The remaining proof of the NP-completeness of the topmost theory within the grey area is given here.

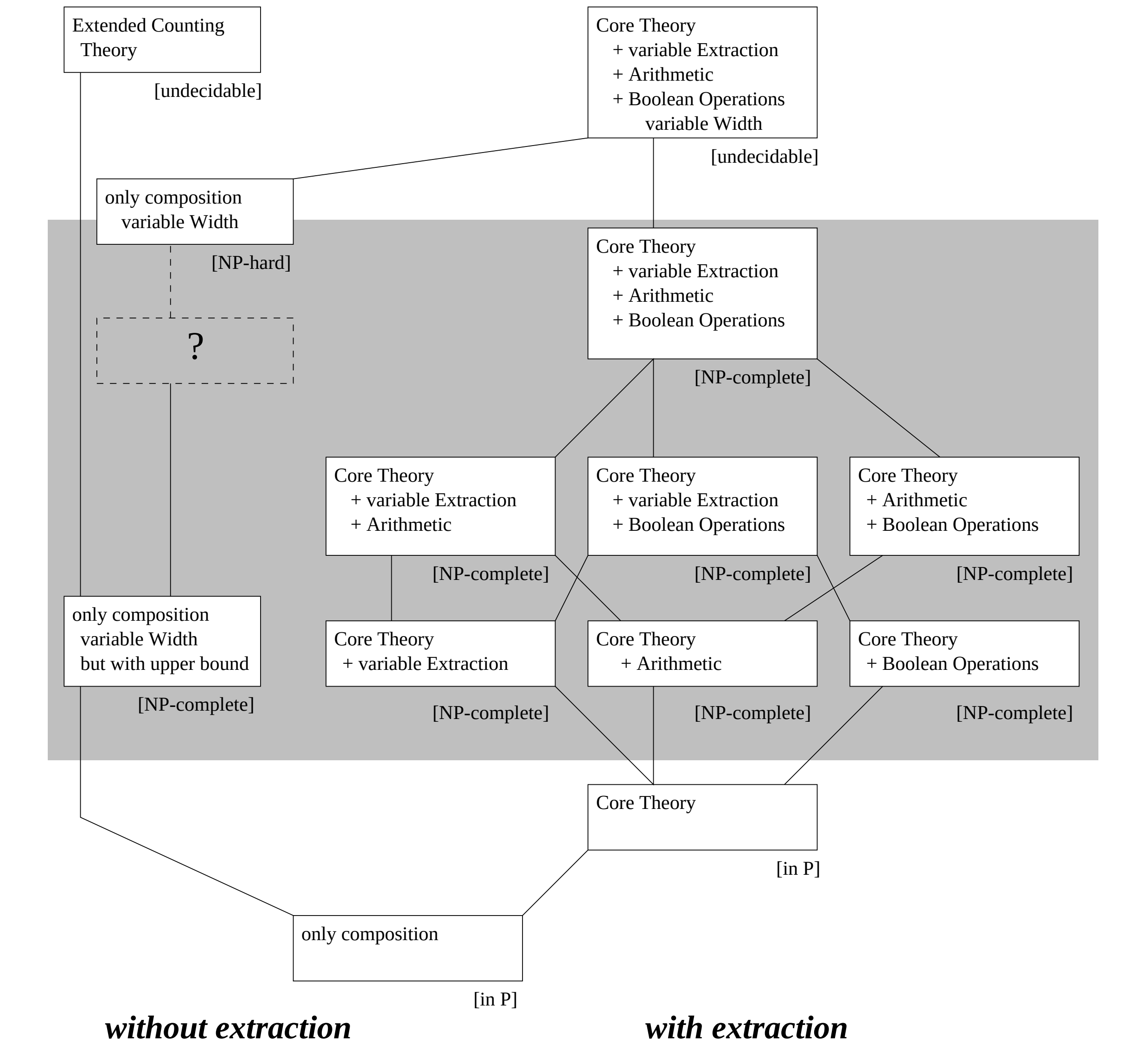


Figure 2.3: Overview on Extensions and Complexity

Theorem 2.4:

Let $BV_{\otimes,[i:j],+,\wedge}$be the theory of fixed-sized bit-vectors with variable extraction, arithmetic and boolean operations. Then the language

$$\mathcal{L}_{t=u}^{\otimes,[i:j],+,\wedge} := \{t = u | t, u \in BV_{\otimes,[i:j],+,\wedge} \text{ there exists an } \alpha \text{ on } vars(t) \cup vars(u) \text{ with } \alpha \models t = u\}$$

is NP-complete.

Proof: Since the theory is fixed-sized, any possible assignment to variables in t and u can be guessed in polynomial time (the length information is assumed to be given unary). Checking equality of fully interpreted terms is polynomial, thus $\mathcal{L}_{t=u}^{\otimes,[i:j],+,\wedge} \in NP$.
The NP-hardness follows from the NP-hardness of the sub-theory $BV_{\otimes,[i:j]}$ as shown in appendix B.2. $\square$

It is surprising, that the complexity of the resulting theories builds an area of NP-complete problems (underlayed grey in the picture) rather than a cascade.

2.4 Bit-Vectors of Unknown Width

If we do not restrict the width of bit-vector variables, the notion of a solver has to be extended. Three methods are presented and the drawbacks and advantages of each are discussed briefly. Finally, a formalism called frame solving according that follows the idea of the third method is defined.

2.4.1 The Standard Method: Everything is a Number

The simplest approach of dealing with bit-vector equation of variable width is to express everything via natural numbers. More precisely, the bit-vector terms are translated into natural number terms including the operations $\_+\_$, $\_\cdot 2^{\_}$, $\_\,DIV\,2^{\_}$ and $\_\,MOD\,2^{\_}$. Additionally, some side constraints in form of equations and inequalities arise. The transformation can be sketched as follows:

$$\begin{array}{rcll} x_{[n]} & \to & x, & x < 2^n \\ x_{[n]}\otimes y_{[m]} & \to & x_{[n]}\cdot 2^m + y_{[m]} & \\ x_{[n]}[j:i] & \to & (x_{[n]}\,MOD\,2^{j+1})\,DIV\,2^i, & 0 \leq j \leq i < n \end{array}$$

where the inequalities on the right are additional constraints. In this way a rich set of statements concerning natural numbers can be encoded. Let in the following x, y be variables we want to apply constraints on, let further a, b, c, d be auxiliary variables we are not interested in and l, m, n be position variables. Then statements of the following form can be generated alongside:

$$\begin{array}{lcll}
\text{(i)} & x \cdot 2^n = y & \text{VIA} & \begin{array}{ccc} \mathbf{0}_{[1]}\otimes a & \otimes & a\otimes x \\ a\otimes\mathbf{0}_{[1]} & \otimes & b \end{array} \stackrel{!}{=} \\[2ex]
\text{(ii)} & x \cdot 2^n = y \cdot 2^m & \text{VIA} & \begin{array}{ccccc} \mathbf{0}_{[1]}\otimes a & \otimes & \mathbf{0}_{[1]}\otimes b & \otimes & a\otimes x \\ a\otimes\mathbf{0}_{[1]} & \otimes & b\otimes\mathbf{0}_{[1]} & \otimes & b\otimes y \end{array} \stackrel{!}{=} \\[2ex]
\text{(iii)} & x_{[n]} = 2^m - 1 & \text{VIA} & \begin{array}{ccccc} \mathbf{0}_{[1]}\otimes a & \otimes & \mathbf{1}_{[1]}\otimes b_{[m]} & \otimes & x \\ a\otimes\mathbf{0}_{[1]} & \otimes & b_{[m]}\otimes\mathbf{1}_{[1]} & \otimes & b_{[m]}\otimes a \end{array} \stackrel{!}{=} \\[2ex]
\text{(iv)} & x_{[n]} = 2^m - 2^l & \text{VIA} & \begin{array}{ccccccc} \mathbf{0}_{[1]}\otimes a & \otimes & \mathbf{1}_{[1]}\otimes b_{[m-l]} & \otimes & \mathbf{0}_{[1]}\otimes c_{[l]} & \otimes & x \\ a\otimes\mathbf{0}_{[1]} & \otimes & b_{[m-l]}\otimes\mathbf{1}_{[1]} & \otimes & c_{[l]}\otimes\mathbf{0}_{[1]} & \otimes & c\otimes b\otimes a \end{array} \stackrel{!}{=} \\[2ex]
\text{(v)} & x \subseteq_{pre} y & \text{VIA} & \begin{array}{ccc} \mathbf{0}_{[1]}\otimes a & \otimes & x\otimes a \\ a\otimes\mathbf{0}_{[1]} & \otimes & y\otimes\mathbf{0}_{[1]} \end{array} \stackrel{!}{=} \\[2ex]
\text{(vi)} & x \subseteq_{sub} y & \text{VIA} & \begin{array}{ccccc} \mathbf{0}_{[1]}\otimes a & \otimes & \mathbf{0}_{[1]}\otimes b & \otimes & b\otimes x\otimes a \\ a\otimes\mathbf{0}_{[1]} & \otimes & b\otimes\mathbf{0}_{[1]} & \otimes & \mathbf{0}_{[1]}\otimes y\otimes\mathbf{0}_{[1]} \end{array} \stackrel{!}{=}
\end{array}$$

The symbols $\subseteq_{pre}$ and $\subseteq_{sub}$ denote an unstrict prefix and substring relation respectively. Note that the leftmost position is viewed as the first one.

The advantage of this approach is that the original problem is transferred into a well-known field, the natural numbers (with modulo arithmetic). On the other hand, there are also mayor drawbacks. Deciding natural numbers with non-linear constraints and modulo arithmetic is a difficult and expensive task, at least in general. Moreover, the characteristic properties of bit-vector equations become hard to exploit. The following paragraph explains that the fundamental difference between positions and values is rather watered down than lost.

There are Two Classes of Variables

The transformation to equations and inequalities over $I\!N$ does not entirely yield a untyped logic. E.G. it is clear from construction, that there are only special kinds of terms that are put up into the exponent. In fact, two categories of integer terms that can be distinguished: terms denoting the "content" or value of bit-vectors, further referred to as $\langle \mathit{Value\ Term} \rangle$ and terms denoting position or width, denoted by $\langle \mathit{Position\ Term} \rangle$. Consequently, a signature can be given as follows:

$$
\begin{array}{lrcl}
\Sigma_{I\!N} := \langle & \{ \ \langle \mathit{Value\ Term} \rangle, & \langle \mathit{Position\ Term} \rangle \ \}, & \\
& \{ \qquad c_V & : & \to \langle \mathit{Value\ Term} \rangle, \\
& \_ +_V \_ & : & \langle \mathit{Value\ Term} \rangle \times \langle \mathit{Value\ Term} \rangle \to \langle \mathit{Value\ Term} \rangle, \\
& c_P & : & \to \langle \mathit{Position\ Term} \rangle, \\
& \_ +_P \_ & : & \langle \mathit{Position\ Term} \rangle \times \langle \mathit{Position\ Term} \rangle \to \langle \mathit{Position\ Term} \rangle, \\
& \_ \cdot 2^{\_} & : & \langle \mathit{Value\ Term} \rangle \times \langle \mathit{Position\ Term} \rangle \to \langle \mathit{Value\ Term} \rangle, \\
& \_ \mathit{MOD}\ 2^{\_} & : & \langle \mathit{Value\ Term} \rangle \times \langle \mathit{Position\ Term} \rangle \to \langle \mathit{Value\ Term} \rangle, \\
& \_ \mathit{DIV}\ 2^{\_} & : & \langle \mathit{Value\ Term} \rangle \times \langle \mathit{Position\ Term} \rangle \to \langle \mathit{Value\ Term} \rangle \qquad \} \ \rangle
\end{array}
$$

The fact that this distinction can be maintained, suggests that the theory of natural numbers is too powerful to express properties of bit-vectors tightly. For example, though an operation $\_ \cdot 2^{\_}$ exists, there can not be constraints like $x \cdot 2^x = 8$.

2.4.2 One Domain: Dense Encoding

There is no conceptual reason, why length and content of a bit-vector should be distinguished. The following approach defines a one-to-one correspondence of bit-vectors of variable width to natural numbers. However, it is necessary to apply a non-standard encoding in order to achieve this.

Counting Strings

Let $\Sigma = \{0, 1\}$ be an alphabet. Then it is well-known that Σ^* can be enumerated lexicographically:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

This enumeration yields an isomorphism η from Σ^* to natural numbers:

$I\!N$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Σ^*	ε	0	1	00	01	10	11	000	001	010	011	100	101	110	111	000	001	010	011

Think of $n \in I\!N$ as generated by an infinite row

$$n = \sum_{i=0}^{\infty} c_i \cdot 2^i \tag{2.1}$$

where the coefficients $c_i \in \{0, 1, 2\}$. The set $\{c_i\}_{i=0}^{\infty}$ is *not* a unique representation of n, however a extraneous condition guarantees uniqueness:

$$\exists N : (\forall i < N : c_i \neq 0) \wedge (\forall i \geq N : c_i = 0) \tag{2.2}$$

This N is exactly the width of the intended bit-vector and the coefficients 1 and 2 stand for the bits "0" and "1" respectively. As a result of this observation, both η and η^{-1} can be computed efficiently, a simple algorithm is given as follows.

$$\begin{array}{rcl}\eta\,(x_0\cdots x_m) & = & \textsc{If}\ (m<0)\ \textsc{Then}\ \ 0 \\ & & \qquad\qquad\qquad\ \textsc{Else}\ \ 2\cdot\eta(x_1\cdots x_m)+x_0+1 \\ & & \textsc{Endif}\end{array}$$

$$\begin{array}{rcl}\eta^{-1}\,(x^D) & = & \textsc{If}\ (x^D=0)\ \textsc{Then}\ \ \varepsilon \\ & & \qquad\qquad\qquad \textsc{Else}\ \ \textsc{If}\ \mathit{even}(x^D)\ \textsc{Then}\ \ 1_{[1]}\otimes(\eta^{-1}(\frac{x^D-2}{2})) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textsc{Else}\ \ 0_{[1]}\otimes(\eta^{-1}(\frac{x^D-1}{2})) \\ & & \qquad\qquad\qquad\qquad\ \ \textsc{Endif} \\ & & \textsc{Endif}\end{array}$$

For $\{0,1\}^*$ is an intuitive description for the set of bit-vectors with finite but unrestricted size, the related natural number is a valid encoding of a bit-vector. This is called a *dense encoding*. In the following x^D denotes a dense encoding of a bit-vector $x_{[m]}$.[c] Obviously, there is a tight connection between the width of a bit-vector and its dense encoding.

Definition 2.11

$$\begin{array}{l} \mathit{width}(x^D) := \lfloor log_2\,(x^D+1)\rfloor \\ \llbracket x^D\rrbracket := 2^{\lfloor log_2\,(x^D+1)\rfloor} \end{array}$$

⌟

Dense Encoding and Bit-Vector Terms

A remarkable property of this encoding is that many bit-vector terms can be processed in a straight forward way. A concatenation, for example, results in a sum of its components, the rightmost term multiplied with an offset that is a potential of 2 (cf. equation 2.1). The coherence can be expressed as follows.

Lemma 2.5:

$$\begin{array}{lrcl} 1. & \eta(x_{[n]}\otimes y_{[m]}) & = & \eta(x_{[n]})\ +\ \llbracket\eta(x_{[n]})\rrbracket\cdot\eta(y_{[m]}) \\ 2. & \eta(\,x_{[n]}[j:i]\,) & = & \big(\eta(x_{[n]})\,\mathit{MOD}\,(2^{i+2}-1)\big)\,\mathit{DIV}\,2^j \\ 3. & \llbracket x^D\rrbracket & = & 2^{\mathit{width}(x^D)} \\ 4. & \llbracket x^D\otimes y^D\rrbracket & = & \llbracket x^D\rrbracket\cdot\llbracket y^D\rrbracket \end{array}$$

Proof: By inspection. □

The fact that dense encoding is possible demonstrates, that there are no inherent two types of integer terms and that the information of length and content can be combined to one data. On the other hand, dense encoding does not appear to be qualified as an encoding of bit-vectors for the following reasons. First, it does not avoid all type check constraints: extraction with natural numbers still leads to constraints like $i \leq \mathit{width}(x^D)$. Second, the processing of boolean operations becomes extremely difficult, for the resulting functions are quite unintuitive. Moreover, if there is composition or variable extraction contained in the bit-vector equation, the terms on natural numbers are not linear. Thus the existence of an efficient algorithm to process them is doubtful.

[c]This is just to avoid confusion; of course, x^D *is* a natural number and all operations defined in $I\!N$ are defined on x^D as usual.

2.4.3 Bit-Vectors and Naturals: A Hybrid System

The idea here is to represent unknown information (the width of a bit-vector variable) by a natural number variable. This way, a *hybrid theory* is gained containing as well an infinite number of bit-vector types $bvec_1$, $bvec_2$, ..., the type $I\!N$ and dependent types $bvec_n$, where $n : I\!N$ is a variable.
This yields a powerful type system expressive enough to represent both concatenation and extraction operations over unknown size (the positions to be extracted might as well be variables), boolean operations and even arithmetic. Thus, an infinite set of operations like $+_{[n]}$ is introduced, which allows an intuitive and well-defined notion of terms like $x_{[n]} +_{[n]} y_{[n]}$.

Obviously, this concept leads to some type correctness constraints. If e.g. a term like $x_{[n]}[2 : i]$ is encountered, the following constraints result from the well-formedness:

$$\text{(i) } i \leq n-1, \qquad \text{(ii) } 2 \leq i, \qquad \text{(iii) } 3 \leq n.$$

Now, suppose that the equation containing $x_{[n]}[2 : i]$ is satisfiable but not valid. Then these constraints should occur explicitly in the solved form, for the equation is only satisfiable with respect to them. This requires some processing of these equations. If an additional constraint $n \leq 2$ is given, the unsatisfiability of the whole equation follows.
This observation motivates the desire for a simpler way to represent variable length.

2.4.4 What is Best?

None of the approaches discussed in this section is perfect; at least one of desirable properties as intuitive encoding, easy extensibility or comfortable solving (e.g. by means of reduction to a previously solved theory) is not given. However, since typical applications include boolean operations and parts of bit-vector terms with fixed width, the small and beautiful approach of dense encoding is dropped here.

The crucial difference between the approaches in 2.4.1 and 2.4.3 is, that in the latter one a fundamental distinction between bit-vectors and natural numbers is taken into account. This requires a rather complicate processing but allows to exploit special properties of bit-vectors (like the known offset of bit-vectors with fixed width) more efficiently. Also, a composition does not necessarily lead to non-linear constraints but can be handled by means of case-splits, which will prove to be a more fruitful concept (see chapter 5).

Thus, the hybrid system is considered to be the most attractive one.

2.4.5 A More General Solver: Frame Solver

Consider a hybrid theory of bit-vectors of possibly unrestricted size and variable extraction. In order to define solving for this special theory we extend the notion of a solver and solving is defined in a more general way. This extension is motivated as follows.

Example 2.6

$$\begin{array}{ccccccc} x_{[n]} & \otimes & 0_{[1]} & \otimes & y_{[m]} & \stackrel{!}{=} \\ z_{[2]} & \otimes & 1_{[1]} & \otimes & w_{[2]} & \end{array}$$

This equation is solvable, if either $n = 1, m = 3$ or $n = 3, m = 1$ holds, but *not* if $n = 2, m = 2$. This cannot be expressed in a conjunction of equations. Thus, strictly speaking, no solver according to Theorem 2.2 can exists.

What Frames Stand for

In order to allow a kind of solving in a more general context, we have to introduce something like a disjunction of solutions. Observe that these disjunctions can only be caused by equations over integer variables. Thus, the intuitive way to describe this disjunction is to build case-splits over special integer constraints. These are called *frames* and the corresponding solving algorithm s_F is called *frame solver*.

Definition 2.12 A *frame* is a tuple (Φ, Ψ) where

- Φ is a set of bit-vector equations of the form $x_{i[n_i]} = \ldots$,
- Ψ is a set of constraints on natural numbers.

also required is that

- none of the left hand side variables in Φ is contained in a right hand side,
- Ψ contains only variables over $\mathbb{Z}$ which occur in Φ as width declarations or positions in extractions,
- Ψ is closed according to Definition 5.1 on page 66) - in particular is Ψ satisfiable.

$\lrcorner$

Definition 2.13 A *frame solver* s_F is a function mapping a set of bit-vector equations $\{E_1, \ldots, E_m\}$ to a set of frames $\{(\Phi_1, \Psi_1), \ldots, (\Phi_k, \Psi_k)\}$ where

- All variables on the left hand sides of $\Phi_\varkappa$ are contained in at least one E_μ, $\varkappa = 1, \ldots, k$; $\mu = 1, \ldots, m$
- The $\Psi_\varkappa$ are pairwise disjoint, i.e. for any two $\Psi_\varkappa, \Psi_{\varkappa'}$, $\varkappa \neq \varkappa'$, there exists no mapping from the natural variables to $I\!N$ that is a model of $\Psi_\varkappa$ and $\Psi_{\varkappa'}$
- For all interpretations $\mathcal{I}$: $\mathcal{I} \models E_1 \wedge \cdots \wedge E_m$ iff $\mathcal{I} \models (\Phi_1 \wedge \Psi_1) \vee \ldots \vee (\Phi_k \wedge \Phi_k)$

$\lrcorner$

For the equation in Example 2.6, a frame solver could yield informally:

Example 2.7

$$s_F\left(\begin{array}{c} x_{[n]}\otimes 0_{[1]}\otimes y_{[m]} \\ z_{[2]}\otimes 1_{[1]}\otimes w_{[2]} \end{array} \stackrel{!}{=} \right) = \left\{ \begin{array}{l} \left(\left\{\begin{array}{lcl} x_{[n]} & = & a_{[1]} \\ y_{[m]} & = & 1_{[1]}\otimes b_{[2]} \\ z_{[2]} & = & a_{[1]}\otimes 0_{[1]} \\ w_{[2]} & = & b_{[2]} \end{array}\right\}, \left\{\begin{array}{lcl} n & = & 1 \\ m & = & 3 \end{array}\right\}\right), \\ \\ \left(\left\{\begin{array}{lcl} x_{[n]} & = & a_{[2]}\otimes 1_{[1]} \\ y_{[m]} & = & b_{[1]} \\ z_{[2]} & = & a_{[2]} \\ w_{[2]} & = & 0_{[1]}\otimes b_{[1]} \end{array}\right\}, \left\{\begin{array}{lcl} n & = & 3 \\ m & = & 1 \end{array}\right\}\right) \end{array} \right\}$$

2.5 The Expressiveness of Solving

Solving is far more a task than just finding a solution. The fact, that a representation of all possible solutions has to be computed can be exploited in a way to construct a decision procedure for even quantified formula. This statement is made explicit in the following.

2.5.1 Verbose Solvers

First, the notion of a *verbose solving* is to be introduced for explanatory reasons. Let ϑ be an (arbitrary) ordering of variables in a theory.

Definition 2.14 A frame solver s (cf. Definition 2.13) is called *verbose* [*with respect to* ϑ], if it fulfills the conditions 1.-5. in Theorem 2.2 and the additional property:
For each result E of s, if $E = \bigwedge_i x_i = t_i$ then

6. for all $x_i \in Var$: $x_i = t_i \in E$

[7. the set E is ordered according to the left hand sides with respect to ϑ]

$\lrcorner$

In the following, a verbose solver is denoted with a hat, like $\hat{s}$.
Property 6. might result in an surplus use of fresh variables. E.G. the result $x_{[n]} = y_{[n]}$ represented as $x_{[n]} = a_{[n]} \wedge y_{[n]} = a_{[n]}$, with a fresh variable $a_{[n]}$.

Lemma 2.6:

For each [frame] solver s [with respect to ϑ] there exists a verbose solver $\hat{s}$ [with respect to ϑ].
Moreover, if s=$\mathcal{O}(f)$ then $\hat{s} = \mathcal{O}(f + |Vars| \cdot \log |Vars|)$.

Proof: Let $E := s(t = u)$. If $E \in \{\mathbf{true}, \mathbf{false}\}$, then $\hat{s}(t = u) = s(t = u)$. Otherwise, $E = \{x_1 = t_1, \ldots, x_m = t_m\}$. Let $Vars := \mathit{vars}(t) \cup \mathit{vars}(u)$. Then for each $v_i \in Vars$ occurring in a t_j, introduce a fresh variable y_i, add $v_i = y_i$ to E and replace v_i with y_i in every t_j. Finally, sort E according to ϑ in time $\mathcal{O}(n + n\,log\,n)$, where $n = |\mathrm{Vars}|$. □

2.5.2 Solvers and Quantification

Solvers are applied to unquantified equations over some logical theory in general. An immanent question is how they could deal with quantified formulas. And the answer is, surprisingly, that they already do. What serves as an example here is the $TQBF$-Problem:

Definition 2.15 Let $\Phi := Q_1 x_1.Q_2 x_2.\cdots Q_n x_n.F(x_1, \ldots, x_n)$ be a fully quantified boolean formula over variables $V := \{x_1, \ldots, x_n\}$ where $Q_i \in \{\forall, \exists\}$, $i \in \{1, \ldots, n\}$ and $F(x_1, \ldots, x_n)$ is a n-ary boolean function with connectives $\wedge$, $\vee$ and $\neg$. Then the language containing all valid expressions is denoted with

$$\mathcal{L}_{TQBF} := \{\Phi | \, \Phi \text{ is a fully quantified boolean formula}, \models \Phi\}$$

Theorem 2.7: [Pap94, p.456, Theorem 19.1]

The language $\mathcal{L}_{TQBF}$ is $PSPACE$-complete. $\lrcorner$

In particular, a variant of these language is of interest. A quantified boolean term Φ over variables $V := \{x_1, \ldots, x_n\}$ is in 3-conjunctive-normal-form ($3CNF$), if
$\Phi \doteq Q_1 x_1.\cdots Q_n x_n.(l_{11} \vee l_{12} \vee l_{13}) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$ where $l_{ij} \in V \cup \overline{V}$, $i = 1, \ldots, m,\ j = 1, 2, 3$, $Q_k \in \{\forall, \exists\},\ k = 1, \ldots, n$. The language $\mathcal{L}_{3CNF\text{-}TQBF}$ is defined as

$$\mathcal{L}_{3CNF\text{-}TQBF} := \{\Phi \big| \, \Phi \text{ is a fully quantified boolean formula in } 3CNF, \models \Phi\}.$$

Corollar 2.8:

$\mathcal{L}_{3CNF\text{-}TQBF}$ is $PSPACE$-complete.

Proof: See Appendix B.1. □

Quantification - Intuitively

In order to understand the idea of the following lemma, an intuitive approach to quantification might be of some help. As well known there are two kinds of uantors: $\forall$ and $\exists$. Think of quantification as a iterative ssignment of the quantified variables, beginning with the leftmost quantor.
The $\forall$ is inclined to express that the following statement is true *no matter what you choose* for the quantified variable. In terms of a most general solution, this means that this choice is arbitrary.
As for the $\exists$, it is only required there there *is at least one* assignment that results in a valid statement. This fact is often expressed with a functional dependency, though strictly speaking, there might be multiple such choices. This circumstance is reflected by means that, in a most general solution, one can construct a most general term with a dependency to terms previously introduced.

The statement we are focusing on, is an equation. In the context of unification, the iterative process described would be equivalent with the search for the most general unifier.

Lemma 2.9: [**Quantification Lemma**]

Let T be an algebraic theory with equation and s be a solver for T. Then s can be transformed into a decision procedure for fully quantified equational terms over T.
Moreover, if $s \in \mathcal{O}(f)$ the decision procedure works in time $\mathcal{O}(f + |s(t=u)| \cdot \log |s(t=u)|)$.

Proof: Given an quantified equation $Q_1\, x_1. \cdots Q_n\, x_n.t \stackrel{!}{=} u$, where $Q_i \in \{\forall, \exists\}$ and $V = \{x_1, \ldots, x_n\}$ is the set of variables occurring in t and u. Let $\hat{s}$ be the verbose transformation of s with respect to the ordering $x_1 \prec \cdots \prec x_n$. Then

$$\hat{s}(t \stackrel{!}{=} u) \quad = \quad \left\{ \begin{array}{ccc} x_1 & = & t_1, \\ & \vdots & \\ x_n & = & t_n \end{array} \right\}$$

A term t_i, $i = 1, \ldots, n$ is called *unrestricted* if

- t_i does not contain any constants
- t_i does not contain any fresh variables occurring in t_j with $j < i$
- t_i does not contain the same fresh variable in more than one place[d]

Since $\hat{s}$ provides a most general form of any solution of $t \stackrel{!}{=} u$, it does not apply unnecessary restrictions in the choice of any variable. Universal quantification is equivalent with the notion of "anything" and existential quantification with "something that can be expressed functionally" respectively.
Therefore let $I := \{i | Q_i = \forall\}$. Then the closed formula $Q_1\, x_1. \cdots Q_n\, x_n.t \stackrel{!}{=} u$ is valid, if and only if for every $i \in I$, t_i is unrestricted.

It is clear how to extend this proof to frame solvers. $\square$

Consider, for example, the quantified boolean formula $\Phi := \forall x.\exists y.\exists z.(x \vee y) = \neg z$. A verbose solver $\hat{s}$ with espect to $x \prec y \prec z$ for this theory yields a solution of the form

$$\hat{s}(x \vee y = \neg z) \quad = \quad \left\{ \begin{array}{ccc} x & = & a \\ y & = & b \\ z & = & \neg a \wedge \neg b \end{array} \right\}$$

Obviously, only the choice of z is dependent on former assignments whereas the choices of x and y are unrestricted. Therefore, all quantifications result in a valid formula except those where z is universally quantified. In particular, Φ is valid.

[d]The intention here is that t_i might be patterned with fresh variables for later reference, but theses patterns are not to restrict the free eligibility of values for t_i with respect to its type. E.G. $t_i = a_{[1]} \otimes a_{[1]}$ restricts t_i to the values $0_{[2]}$ or $3_{[2]}$ and prevents values $1_{[2]}$ and $2_{[2]}$.

2.5.3 Solving $BV_{\otimes,bvec_n}$ is $PSPACE$-hard

Definition 2.16 A function $g : A \to B$, $|g(a)|$ polynomial in $|a|$, is called $\mathfrak{A}$-*hard*, if the following language is $\mathfrak{A}$-hard:

$$\mathcal{L}_g := \{x \mid x \text{ is a prefix of } \langle a, b\rangle \in A \times B \text{ with } g(a) = b\}$$

In particular, g is $PSPACE$-hard, if $PSPACE \subseteq P^{\mathcal{L}_g}$, where P^A denotes the class of languages that can be decided via polynomial oracle Turing machines with oracle language A. For a formal definition see e.g. [Pap94].

⌟

As an application of the Quantification Lemma, a former result (Appendix A.2) is extended to the following statement: Let $BV_{\otimes,bvec_n}$ be the theory of bit-vectors with variable size and concatenation as the only operation.

Corollar 2.10:

Solving $BV_{\otimes,bvec_n}$ is $PSPACE$-hard.

Proof: [via Reduction of 3CNF-TQBF]
Let $\Phi := Q_1x_1.\cdots Q_nx_n.F$ be an arbitrary quantified boolean formula over $x_1,\ldots,x_n$. Then the matrix F can be encoded into a bit-vector equation $f(F)$ according to Appendix A.2. Though f introduces several variables, there is a one-to-one correspondence of each x_j to the width of a_j ($x_j = \textbf{false}$ iff $|a_j| = 1$ and $x_j = \textbf{true}$ iff $|a_j| = 2$). Note that $|f(F)|$ (and also the number of introduced variables a_j, b_j, c_j, d_j) is *linear* in $|F|$. Thus, the result of a solver enables us to check the validity of a quantified formula

$$\Phi' := Q_1a_1.\cdots Q_na_n.\exists\big(\text{all other variables in } f(F)\big).\, f(F)$$

with a polynomial overhead in $|f(F)|$ according to the Quantification Lemma 2.9. Also, the output of a solver s is necessarily polynomial in the number of original variables. Since $\Phi \Leftrightarrow \Phi'$, it holds that $\mathcal{L}_{3CNF\text{-}TQBF} \in P^{\mathcal{L}_s}$. $\mathcal{L}_{3CNF\text{-}TQBF}$ is $PSPACE$-complete, thus $PSPACE \subseteq P^{\mathcal{L}_s}$.

□

Chapter 3

On Decidability

Things are only impossible until they're not.
(Jean-Luc Picard)

Nothing is impossible for the man who doesn't have to do it himself.
(A. H. Weiler)

3.1 Where the Problems are

If we allow variable size of bit-vector variables, at first this does not seem to be a big deal. One can simply introduce dependent types $bvec_n$, where $n : \mathbb{N}$. Also variable extraction of the form $x_{[n]}[j:i]$, where i and j are variables as well, has an intuitive semantic. There are some type check constraints, sure, but in means of linear algebraic terms. Moreover, if just the size is still unknown but restricted[a], it is obvious that the whole theory remains decidable.

However, as it turns out, for *unrestricted* sizes n at least the intuitive approaches do not yield a positive result. This might be demonstrated on a rather "difficult" example.

3.1.1 Considering an Example

Example 3.1
Let $x : bvec_n;\ y, z : bvec_m,\ b, c : bvec_1$. Consider the equation

$$\begin{array}{ccccccccccccccccc}
x & \otimes & y & & y & \otimes & c & & z & \otimes & b & & z & \otimes & 1_{[1]} & \otimes & x \\
 & = & & \wedge & & = & & \wedge & & = & & \wedge & & & = & & \\
y & \otimes & x & & c & \otimes & y & & b & \otimes & z & & x & \otimes & 0_{[1]} & \otimes & y \\
 & {\scriptstyle (\alpha)} & & & & {\scriptstyle (\beta)} & & & & {\scriptstyle (\gamma)} & & & & & {\scriptstyle (\delta)} & &
\end{array}$$

Note that this conjunctive form serves but better readability, for the sizes of the upper and lower subterms are *known* to be equal. The equation has no solution, but not for trivial reasons. Follow this train of thought:

- Sub-equation (α) holds, if there is a term $a \in \{\mathbf{0},\mathbf{1}\}^+$ with $x = a^+ \quad \wedge \quad y = a^+$.
- (β) implies that y has to consist of the very same bit, concatenated arbitrarily often.
- (γ) implies that z has to consist of the very same bit, concatenated arbitrarily often.

[a]By means of that for each variable $x_{[n]}$ there exists a *fixed and known* N such that $n < N$.

- As a consequence of the first two points, x consists of the (one) same bit as y.
- In order to satisfy (δ), the positions of $1_{[1]}$ and $0_{[1]}$ have to be different.
  - If the $1_{[1]}$ is relatively left to $0_{[1]}$, then both bits $\mathbf{0}$ and $\mathbf{1}$ have to occur in x, which is impossible due to the previous argument.
  - If the $1_{[1]}$ is relatively right to $0_{[1]}$, then 0 occurs in z and 1 occurs in y. Therefore, $y = \mathbf{0}_{[m]}$ and $z = \textbf{-1}_{[m]}$. This leaves x no choice to be anything.

Following this example, bit-vectors are interpreted as natural numbers as seen in 2.4.1.
The equation in Example 3.1 would then be represented as

$$\Phi := \begin{cases} & x \cdot 2^m + y & = & y \cdot 2^n + x \\ \wedge & y \cdot 2^1 + c & = & c \cdot 2^m + y \\ \wedge & z \cdot 2^1 + b & = & b \cdot 2^m + z \\ \wedge & z \cdot 2^{n+1} + 1 \cdot 2^n + x & = & x \cdot 2^{m+1} + 0 \cdot 2^m + y \\ \wedge & x < 2^n \\ \wedge & y < 2^m \\ \wedge & z < 2^m \\ \wedge & b < 2^1 \\ \wedge & c < 2^1 \end{cases}$$

It is clear from argumentation, that $\Phi \models_{\mathbb{N}} \perp$; however, it is not obvious, if there is a calculus C strong enough to obtain $\Phi \vdash_C \perp$. Note that the equations above contain non-linear algebraic terms.
To prove unsatisfiability, one can perform a case split on b and c. If $b = 0$ and $c = 0$, then Φ narrows to

$$\Phi_{(b=0,c=0)} \quad = \quad \begin{cases} & x \cdot 2^m + y & = & y \cdot 2^n + x & & \\ \wedge & y \cdot 2 & = & y & \Rightarrow & y = 0 \\ \wedge & z \cdot 2 & = & z & \Rightarrow & z = 0 \\ \wedge & z \cdot 2^{n+1} + 1 \cdot 2^n + x & = & x \cdot 2^{m+1} + y & & \\ \wedge & x < 2^n \\ \wedge & y < 2^m \\ \wedge & z < 2^m \end{cases}$$

Here, the first line implies $x = 0$ and the fourth line yields therefore $2^n = 0$, which cannot be satisfied.
The three other cases are left to the reader.

3.2 The Theory with Variable Length and Only Concatenation

The first part of this section discusses decidability of a restricted theory of bit-vectors. This enables us to judge the "difficulty" of the problem by means of the Chomsky hierarchy. The second part presents the actual decidability result.

Definition 3.1 Let $BV_{\otimes, bvec_n}$ be the bit-vector theory with the signature

$$\begin{aligned} \langle \quad & \mathbb{N} \cup \{bvec_n \,|\, n : \mathbb{N}^+\}, \\ & \{\mathbf{0} :\rightarrow bvec_1, \quad \mathbf{1} :\rightarrow bvec_1\} \cup \{\otimes_{n,m} : bvec_n \times bvec_m \rightarrow bvec_{n+m} \,|\, n, m : \mathbb{N}^+\} \quad \rangle \end{aligned}$$

Here, dependent types of the form $bvec_n$, where n is a natural variable, are allowed. ⌟
The only predicate is equality. Since this section treats solving an equation $t_1 = t_2$ by means of recognizing languages, it is reasonable to introduce a language over $\{\mathbf{0}, \mathbf{1}, \$\}$.

$$\mathcal{L}_{(t_1=t_2)} := \{x_1\$x_2\$\cdots\$x_n\$\textit{term-string} \,|\, x_1, \ldots, x_n,\ \textit{term-string} \in \{\mathbf{0},\mathbf{1}\}^+\}$$

Each solution of our equation $t_1 = t_2$ is reflected by a word in $\mathcal{L}_{(t_1=t_2)}$ (and vice versa) as the x_i stand for instances of all occurring bit-vector variables and *term-string* is their instantiation of the pattern given in both t_1 and t_2.

3.2.1 Term Representation via Context Sensitive Grammars

The problem of solvability in $BV_{\otimes,bvec_n}$ can be expressed by means of a context sensitive grammar. More precisely, given one equation $t_1 = t_2$ in $BV_{\otimes,bvec_n}$, it is possible to effectively generate two context sensitive grammars G_1 and G_2, such that each word $w \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ represents a distinct model that satisfies $t_1 = t_2$. Since we know that context sensitive languages are closed under intersection (cf. [Sch92a]), $\mathcal{L}_{(t_1=t_2)}$ is still context sensitive.[b]

More precisely, given an equation $t_1 = t_2$, let $V := vars(t_1) \cup vars(t_2) = \{x_1, \ldots, x_n\}$. The construction yields that $L(G_i)$ only contains words over the alphabet $\{\mathbf{0}, \mathbf{1}, \$\}$ of the shape

$$x_1\$x_2\$\cdots x_n\$t_i$$

where $i \in \{1, 2\}$. The first part $x_1\$\cdots\$x_n$ (referred to as the *definition side*) guarantees an identic assignment of the variables in both productions and t_i yields a matching of both terms.

Let $count_j(x_i)$ be the number of occurrences of the variable x_i in t_j ($j \in \{1,2\}, i \in \{1, \ldots, n\}$). A *translation* of a term $t = obj_1 \otimes obj_2 \otimes \cdots \otimes obj_m$ omits the $\otimes$-operators and replaces

(i) each variable x_i with ${}^kX_i$ where $k \in \{1, \ldots, count(x_i)\}$ denotes the k^{TH} occurrence of x_i (counted from left to right),

(ii) each constant by the corresponding string over $\{\mathbf{0}, \mathbf{1}\}$.

An *indexed translation* also adds an index $l_i \in \{0, 1\}$ to every variable x_i (i.e. ${}^kX_i$ becomes ${}^kX_i^0$ or ${}^kX_i^1$) and is denoted by $translation_{(l_1,\ldots,l_n)}$.
Also, a set $Omitted_j := \{i \mid x_i \in V \wedge x_i \notin vars(t_j)\}$ is defined, for the non-occurring variables require a special treatment.

The grammar contains two kinds of non-terminal symbols. The one denoted by a ${}^kX_i^l$ explains as the k^{TH} occurrence of the bit-vector variable x_i, which eventually reduces to the letter $l \in \{0, 1\}$ at the rightmost position. The other kind, symbolic ${}^k\vec{N}_i$ is used to carry information from left to right and is referred to as a *walker*. Here, $N \in \{\mathbf{0}, \mathbf{1}\}$ is the information (one bit), addressed to the recipient ${}^kX_i^l$. The walkers vanish when reaching the rightmost occurrence ${}^{count_j(i)}X_i^l$.

Lemma 3.1:
Let $t_1 = t_2$ be an equation in $BV_{\otimes,bvec_n}$ with variables $V := \{x_1, \ldots, x_n\}$ of unknown width and let $G_j := \langle V_j, \Pi, P_j, S_j \rangle$ be the context sensitive grammar constructed as shown in Figure 3.1.[c] Then for $j \in \{1, 2\}$:

$$\mathcal{L}(G_j) \;=\; \{x_1\$\cdots\$x_n\$t_j \mid x_i \in \{\mathbf{0},\mathbf{1}\}^+,\ i = 1, \ldots, n\}$$

Proof: For sake of readability, in the following a grammar is understood as a generation system for well formed words.

(i) $\mathcal{L}(G_j) \subseteq \{x_1\$\cdots\$x_n\$t_j \mid x_i \in \{\mathbf{0},\mathbf{1}\}^+,\ i = 1, \ldots, n\}$

The production rules in (a) yields the proper shape of each resulting word. Since the right hand sides of production rules in context sensitive grammars must not be smaller (i.e. in number of symbols) than the left sides, the rightmost letter to which each variable transforms has to be chosen a priori (and is added as

[b] We can effectively construct the grammar for $\mathcal{L}_{(t_1=t_2)}$, e.g. via transforming both G_1 and G_2 to their correlated linear bound automatons, then perform a series connection of both automatons and and transform the result back to an context free grammar. However, this is purely platonic.

[c] With respect to better readability, the concatenation of grammar symbols is sometimes marked by a "$\cdot$".

$$
\begin{array}{rl}
V_j := & \{{}^{k_i}X_i^l \,|\, i=1,\ldots,n,\ k_i=0,\ldots count_j(x_i), l=1,2\} \cup \\
& \{{}^{k_j}\vec{\mathbf{0}}_i \,|\, i=1,\ldots,n,\ k_i=0,\ldots count_j(x_i)\} \cup \\
& \{{}^{k_j}\vec{\mathbf{1}}_i \,|\, i=1,\ldots,n,\ k_i=0,\ldots count_j(x_i)\} \\
\\
\Pi := & \{0,1,\$\}, \\
\\
P_j := & \begin{array}{ll}
(a) & \{S \rightarrow {}^0X_1^{l_1}\$\cdots\${}^0X_n^{l_n}\$translation_{(l_1,\ldots,l_n)}(t_j) \,|\, l_i=0,1\} \cup \\
(b) & \{{}^kX_i^0 \rightarrow \mathbf{0} \,|\, i=1,\ldots,n,\ k=0,\ldots,count(x_i)\} \cup \\
(c) & \{{}^kX_i^1 \rightarrow \mathbf{1} \,|\, i=1,\ldots,n,\ k=0,\ldots,count(x_i)\} \cup \\
(d) & \{{}^0X_i^l \rightarrow N \cdot^0 X_i^l \cdot^1 \vec{N}_i \,|\, i=1,\ldots,n,\ l=0,1,\ N=\mathbf{0},\mathbf{1}\} \cup \\
(e) & \{{}^k\vec{N}_i \cdot^k X_i^l \rightarrow N \cdot^k X_i^l \cdot^{(k+1)} \vec{N}_i \,|\, i=1,\ldots,n,\ k=1,\ldots,count(x_i),\ l=0,1,\ N=\mathbf{0},\mathbf{1}\} \cup \\
(f) & \{{}^{count(x_i)}\vec{N}_i \cdot^{count(x_i)} X_i^l \rightarrow N \cdot^{count(x_i)} X_i^l \,|\, i=1,\ldots,n,\ l=0,1,\ N=\mathbf{0},\mathbf{1}\} \cup \\
(g) & \{{}^k\vec{N}_i \cdot B \rightarrow B \cdot^k \vec{N}_i \,|\, i=1,\ldots,n,\ k=1,\ldots count(x_i),\ N=\mathbf{0},\mathbf{1},\ B \in \Pi \cup V_j \backslash \{{}^kX_i^0, {}^kX_i^1\}\} \\
(h) & \begin{array}{l} \{{}^0X_i^l \rightarrow N \cdot^0 X_i^l \,|\, i \in Omitted_j,\ l=0,1, N=\mathbf{0},\mathbf{1}\} \cup \\ \{{}^0X_i^l \rightarrow N \,|\, i \in Omitted_j,\ l=0,1,\ N=\mathbf{0},\mathbf{1}\} \end{array}
\end{array}
\end{array}
$$

Figure 3.1: Context Sensitive Grammar G_j, Representing all Instances of Term t_j

an upper right index to each variable symbol). Rules (b) and (c) provide this final reduction.
The big goal is now to allow a *simultaneous* growth of each occurrence of a variable. The problem here is to guarantee that each occurrence results in the same string. This is solved via allowing only the leftmost occurrence of a variable to spontaneously produce a new letter (see (d)). At the same time, a walker $\vec{N}$ is produced. It can jump over any symbol except the next occurrence of the same variable – there it results in a production of a proper letter and the creation of a new walker (cf. (g) and (e)). If the last occurrence of a variable is reached, the walker vanishes, see (f). If a variable x_i does not occur in the term t_j, it still has to show up in the definition side. Since the walker rules do not allow a termination in this case, special arbitrary production rules in (h) apply.

(ii) $\mathcal{L}(G_j) \supseteq \{x_1\$\cdots\$x_n\$t_j \,|\, x_i \in \{\mathbf{0},\mathbf{1}\}^+,\ i=1,\ldots,n\}$
As a short inspection reveals, the production terminates if and only if

1. Each created walker moves strictly to the right and vanishes at the last occurrence of the specific variable, thus guaranteeing that each occurrence is replaced with the *same* string over $\{\mathbf{0},\mathbf{1}\}$,

2. The variables not occurring in t_j use only production rules in (h).

□

The reader may have noticed that the construction rules $(b)-(h)$ assumes all bit-vector variables to have unknown (and unbound) width. The variables of fixed or restricted width have to be treated as an exception. Since the range of these is finite, each possible assignment can be added as a corresponding production rule in (a), leading to an exponential (but still finite) blow-up. This makes further treatment of these variables in rules (b) - (h) a surplus.

Lemma 3.2:
The Language $\mathcal{L}_{(t_1=t_2)} := \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ is isomorphic to the set of all possible solutions of the equation $t_1 = t_2$.

Proof: If a word w is in both $\mathcal{L}(G_1)$ and $\mathcal{L}(G_2)$, according to Lemma 3.1 the assignment of the variables on the definition side are the same and the constructed strings match. Thus, the assignment encoded in the

definition side is a model for the equation $t_1 = t_2$.
Vice versa, if there is a assignment A with $A \models t_1 = t_2$, the corresponding word w can be constructed in both grammars.

□

Lemma 3.1 and Lemma 3.2 lead directly to

Theorem 3.3:

The set of solutions of any equation $t_1 = t_2$ in Σ can be represented via a context sensitive language.

3.2.2 An Example

This construction is applied to the equation in Example 3.1. The terms are

$$\begin{array}{lcl} t_1 & := & x \otimes y \otimes y \otimes c_{[1]} \otimes z \otimes b_{[1]} \otimes z \otimes 1_{[1]} \otimes x \\ t_2 & := & y \otimes x \otimes c_{[1]} \otimes y \otimes b_{[1]} \otimes z \otimes x \otimes 0_{[1]} \otimes y \end{array}$$

For $b_{[1]}$ and $c_{[1]}$ have fixed size, they are not translated to variables in the grammar but result in a case split in the production rules in (a) and remain untouched by the indexed translation. There are three variables x, y, z of unknown size. With respect to better readability these are translated to X, Y, Z[d] and the walkers carry an index x, y or z[e]. Thus, the translation yields

$$\begin{array}{lcl} translation_{(l_x,l_y,l_z)}(t_1) & = & {}^1X^{l_x} \cdot {}^1Y^{l_y} \cdot {}^2Y^{l_y} \cdot c_{[1]} \cdot {}^1Z^{l_z} \cdot b_{[1]} \cdot {}^2Z^{l_z} \cdot 1 \cdot {}^2X^{l_x} \\ translation_{(l_x,l_y,l_z)}(t_2) & = & {}^1Y^{l_y} \cdot {}^1X^{l_x} \cdot c_{[1]} \cdot {}^2Y^{l_y} \cdot b_{[1]} \cdot {}^2Z^{l_z} \cdot {}^2X^{l_x} \cdot 0 \cdot {}^3Y^{l_y} \end{array}$$

A complete definition of the grammar $G_1 = \langle V_1, \{0, 1, \$\}, P_1, S_1 \rangle$ is given in Figure 3.2. G_2 is constructed in the same way. Each of both grammars contain 32 variables and 472 rules. Note that we know from construction that $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ is empty.

3.2.3 $\mathcal{L}_{(t_1=t_2)}$ is not Context Free

Theorem 3.4:

In general, the result language $\mathcal{L}_{(t_1=t_2)}$ is not context free.

Proof: We can present a quite simple equation yielding a result language $\mathcal{L}_{(t_1=t_2)}$ which is *not* context free. This is proven by application of the pumping lemma for context free languages. Let

$$\begin{array}{lclcl} t_1 & := & \mathbf{0} \otimes x \otimes y \otimes z & \otimes & x \otimes \mathbf{1} \otimes x \\ t_2 & := & x \otimes y \otimes z \otimes \mathbf{0} & \otimes & y \otimes \mathbf{1} \otimes z \end{array}$$

The width of each variable is unknown. The only purpose of the left part of both terms is to guarantee that $x, y, z \in \{\mathbf{0}\}^+$, whereas the right part yields that x, y and z are of the same size. Therefore,

$$\mathcal{L}_{(t_1=t_2)} = \{\mathbf{0}^i\$\mathbf{0}^i\$\mathbf{0}^i\$\ \mathbf{0}^i\mathbf{1}\mathbf{0}^i \mid i = 1, 2, 3, \ldots\}$$

Assuming $\mathcal{L}_{(t_1=t_2)}$ is context free, from the pumping lemma follows that there exists a n such that

$$\forall w \in \mathcal{L}_{(t_1=t_2)},\ |w| > n,\ w = abcde : \forall j : w_j := ab^j cd^j e \in \mathcal{L}_{(t_1=t_2)}$$

---

[d] Instead of X_1, X_2 and X_3 respectively.
[e] Instead of 1, 2 or 3.

$$
\begin{array}{ll}
V_1 := \{ & {}^0X^0,\ {}^1X^0,\ {}^2X^0,\ {}^0X^1,\ {}^1X^1,;{}^2X^1, \\
& {}^0Y^0,\ {}^1Y^0,\ {}^2Y^0,\ {}^0Y^1,\ {}^1Y^1,\ {}^2Y^1, \\
& {}^0Z^0,\ {}^1Z^0,\ {}^2Z^0,\ {}^0Z^1,\ {}^1Z^1,\ {}^2Z^1, \\
& {}^1\vec{0}_x,\ {}^2\vec{0}_x,\ {}^1\vec{1}_x,\ {}^2\vec{1}_x \\
& {}^1\vec{0}_y,\ {}^2\vec{0}_y,\ {}^1\vec{1}_y,\ {}^2\vec{1}_y \\
& {}^1\vec{0}_z,\ {}^2\vec{0}_z,\ {}^1\vec{1}_z,\ {}^2\vec{1}_z\ \}
\end{array}
$$

$$
\begin{array}{ll}
P_1 := \{ & S_1 \to {}^0X^0\$^0Y^0\$^0Z^0\$\,0\,\$\,0\,\$\ {}^1X^0 \cdot {}^1Y^0 \cdot {}^2Y^0 \cdot 0 \cdot^1 Z^0 \cdot 0 \cdot^2 Z^0 \cdot 1 \cdot^2 X^0 \\
& S_1 \to {}^0X^1\$^0Y^0\$^0Z^0\$\,0\,\$\,0\,\$\ {}^1X^1 \cdot {}^1Y^0 \cdot {}^2Y^0 \cdot 0 \cdot^1 Z^0 \cdot 0 \cdot^2 Z^0 \cdot 1 \cdot^2 X^1 \\
& \vdots \\
& S_1 \to {}^0X^1\$^0Y^1\$^0Z^1\$\,1\,\$\,1\,\$\ {}^1X^1 \cdot {}^1Y^1 \cdot {}^2Y^1 \cdot 1 \cdot^1 Z^1 \cdot 1 \cdot^2 Z^1 \cdot 1 \cdot^2 X^1 \\
& \\
& {}^0X^0 \to 0,\ {}^1X^0 \to 0,\ {}^2X^0 \to 0,\ {}^1X^1 \to 1,\ {}^1X^1 \to 1,\ {}^2X^1 \to 1, \\
& {}^0Y^0 \to 0,\ {}^1Y^0 \to 0,\ {}^2Y^0 \to 0,\ {}^1Y^1 \to 1,\ {}^1Y^1 \to 1,\ {}^2Y^1 \to 1, \\
& {}^0Z^0 \to 0,\ {}^1Z^0 \to 0,\ {}^2Z^0 \to 0,\ {}^1Z^1 \to 1,\ {}^1Z^1 \to 1,\ {}^2Z^1 \to 1, \\
& \\
& {}^0X^0 \to 0 \cdot^0 X^0 \cdot^1 \vec{0}_x,\ {}^0X^0 \to 1 \cdot^0 X^0 \cdot^1 \vec{1}_x,\ {}^0X^1 \to 0 \cdot^0 X^1 \cdot^1 \vec{0}_x,\ {}^0X^1 \to 1 \cdot^0 X^1 \cdot^1 \vec{1}_x, \\
& {}^0Y^0 \to 0 \cdot^0 Y^0 \cdot^1 \vec{0}_y,\ {}^0Y^0 \to 1 \cdot^0 Y^0 \cdot^1 \vec{1}_y,\ {}^0Y^1 \to 0 \cdot^0 Y^1 \cdot^1 \vec{0}_y,\ {}^0Y^1 \to 1 \cdot^0 Y^1 \cdot^1 \vec{1}_y, \\
& {}^0Z^0 \to 0 \cdot^0 Z^0 \cdot^1 \vec{0}_z,\ {}^0Z^0 \to 1 \cdot^0 Z^0 \cdot^1 \vec{1}_z,\ {}^0Z^1 \to 0 \cdot^0 Z^1 \cdot^1 \vec{0}_z,\ {}^0Z^1 \to 1 \cdot^0 Z^1 \cdot^1 \vec{1}_z, \\
& \\
& {}^1\vec{0}_x \cdot^1 X^0 \to 0 \cdot^1 X^0 \cdot^2 \vec{0}_x,\ {}^1\vec{1}_x \cdot^1 X^0 \to 1 \cdot^1 X^0 \cdot^2 \vec{1}_x,\ {}^1\vec{0}_x \cdot^1 X^1 \to 0 \cdot^1 X^1 \cdot^2 \vec{0}_x,\ {}^1\vec{1}_x \cdot^1 X^1 \to 1 \cdot^1 X^1 \cdot^2 \vec{1}_x, \\
& {}^1\vec{0}_y \cdot^1 Y^0 \to 0 \cdot^1 Y^0 \cdot^2 \vec{0}_y,\ {}^1\vec{1}_y \cdot^1 Y^0 \to 1 \cdot^1 Y^0 \cdot^2 \vec{1}_y,\ {}^1\vec{0}_y \cdot^1 Y^1 \to 0 \cdot^1 Y^1 \cdot^2 \vec{0}_y,\ {}^1\vec{1}_y \cdot^1 Y^1 \to 1 \cdot^1 Y^1 \cdot^2 \vec{1}_y, \\
& {}^1\vec{0}_z \cdot^1 Z^0 \to 0 \cdot^1 Z^0 \cdot^2 \vec{0}_z,\ {}^1\vec{1}_z \cdot^1 Z^0 \to 1 \cdot^1 Z^0 \cdot^2 \vec{1}_z,\ {}^1\vec{0}_z \cdot^1 Z^1 \to 0 \cdot^1 Z^1 \cdot^2 \vec{0}_z,\ {}^1\vec{1}_z \cdot^1 Z^1 \to 1 \cdot^1 Z^1 \cdot^2 \vec{1}_z, \\
& \\
& {}^2\vec{0}_x \cdot^2 X^0 \to 0 \cdot^2 X^0,\ {}^2\vec{1}_x \cdot^2 X^0 \to 1 \cdot^2 X^0,\ {}^2\vec{0}_x \cdot^2 X^1 \to 0 \cdot^2 X^1,\ {}^2\vec{1}_x \cdot^2 X^1 \to 1 \cdot^2 X^1, \\
& {}^2\vec{0}_y \cdot^2 Y^0 \to 0 \cdot^2 Y^0,\ {}^2\vec{1}_y \cdot^2 Y^0 \to 1 \cdot^2 Y^0,\ {}^2\vec{0}_y \cdot^2 Y^1 \to 0 \cdot^2 Y^1,\ {}^2\vec{1}_y \cdot^2 Y^1 \to 1 \cdot^2 Y^1, \\
& {}^2\vec{0}_z \cdot^2 Z^0 \to 0 \cdot^2 Z^0,\ {}^2\vec{1}_z \cdot^2 Z^0 \to 1 \cdot^2 Z^0,\ {}^2\vec{0}_z \cdot^2 Z^1 \to 0 \cdot^2 Z^1,\ {}^2\vec{1}_z \cdot^2 Z^1 \to 1 \cdot^2 Z^1 \quad \} \cup \\
\{ & {}^1\vec{0}_x \cdot B \to B \cdot^1 \vec{0}_x \mid B \neq {}^1X^0 \wedge B \neq {}^1X^1 \quad \} \cup \{ {}^1\vec{1}_x \cdot B \to B \cdot^1 \vec{1}_x \mid B \neq {}^1X^0 \wedge B \neq {}^1X^1 \quad \} \cup \\
\{ & {}^2\vec{0}_x \cdot B \to B \cdot^2 \vec{0}_x \mid B \neq {}^2X^0 \wedge B \neq {}^2X^2 \quad \} \cup \{ {}^2\vec{1}_x \cdot B \to B \cdot^2 \vec{1}_x \mid B \neq {}^2X^0 \wedge B \neq {}^2X^2 \quad \} \cup \\
\{ & {}^1\vec{0}_y \cdot B \to B \cdot^1 \vec{0}_y \mid B \neq {}^1Y^0 \wedge B \neq {}^1Y^1 \quad \} \cup \{ {}^1\vec{1}_y \cdot B \to B \cdot^1 \vec{1}_y \mid B \neq {}^1Y^0 \wedge B \neq {}^1Y^1 \quad \} \cup \\
\{ & {}^2\vec{0}_y \cdot B \to B \cdot^2 \vec{0}_y \mid B \neq {}^2Y^0 \wedge B \neq {}^2Y^2 \quad \} \cup \{ {}^2\vec{1}_y \cdot B \to B \cdot^2 \vec{1}_y \mid B \neq {}^2Y^0 \wedge B \neq {}^2Y^2 \quad \} \cup \\
\{ & {}^1\vec{0}_z \cdot B \to B \cdot^1 \vec{0}_z \mid B \neq {}^1Z^0 \wedge B \neq {}^1Z^1 \quad \} \cup \{ {}^1\vec{1}_z \cdot B \to B \cdot^1 \vec{1}_z \mid B \neq {}^1Z^0 \wedge B \neq {}^1Z^1 \quad \} \cup \\
\{ & {}^2\vec{0}_z \cdot B \to B \cdot^2 \vec{0}_z \mid B \neq {}^2Z^0 \wedge B \neq {}^2Z^2 \quad \} \cup \{ {}^2\vec{1}_z \cdot B \to B \cdot^2 \vec{1}_z \mid B \neq {}^2Z^0 \wedge B \neq {}^2Z^2 \quad \}
\end{array}
$$

Figure 3.2: Variables and Rules of G_1

For any n there exists a word w in $\mathcal{L}_{(t_1=t_2)}$ with $|w| > n$; however, alone w_2 is not in $\mathcal{L}_{(t_1=t_2)}$, since this would imply

Case 1: Either b or d contain a \$ symbol - then w_2 contains more than three \$s and is therefore not in $\mathcal{L}_{(t_1=t_2)}$.

Case 2: Neither b nor d contain a \$. If one of them is on the definition side, the condition $|x| = |y| = |z|$ cannot be maintained and else the translation term is not sound with the instances of the variables; in any case, $w_2 \notin \mathcal{L}_{(t_1=t_2)}$.

Therefore, the pumping lemma cannot hold in general and as a consequence $\mathcal{L}_{(t_1=t_2)}$ cannot be context free.

$\square$

Note: This negative result is proven by exploiting the special structure of words in the result language $\mathcal{L}_{(t_1=t_2)}$. However, it is clear how to enhance it on general representations of any solution language.

3.2.4 An Interpretation of this Result

By means of the construction presented, the satisfiability problem can be reduced to the emptiness problem of languages of type 1 according to the well-known Chomsky hierarchy. Unfortunately, the emptiness problem is undecidable in general for languages of type 1 (cf. [Sch92a, p. 83]). Thus no general algorithm can be presented that could be used as a solver.
On the other hand, the reduction is not an isomorphism, thus this result does not imply the general impossibility of solving $BV_{\otimes,bvec_n}$. The author has to admit this is quite unfulfilling – however, it might give a hint to the "difficulty" of solving even these restricted bit-vector equations.

3.2.5 $\mathcal{L}_{(t_1=t_2)}$ is Decidable

As a matter of fact, the problem defined in section 3.2 occurres in a number of mathematical fields in a different shape—sufficiently disguised to prevent a unique notion of it. Mathematicians of western countries refer to it as Löb's problem by means of solving word equations over a free monoid (investigated by Lentin and Schützenberger in 1969). In eastern countries it was referred to as Markov's problem and in the field of automated deduction as the string unification problem. This diversion may display both fundamentelness and non-triviallity of this specific problem. For a more elaborate overview confer to [GHR93], unification theory, chapter 5.

The positive result that $\mathcal{L}_{(t_1=t_2)}$ is decidable, derives from the decidability result for arbitrary single equations in a free monoid G.S. Makanin presented in 1977. For a description see [Mak92]. More interesting in our context is the result of Jaffar [Jaf90], where an algorithm for construction of all models is given. The actual complexity class of $\mathcal{L}_{(t_1=t_2)}$ is unknown, but known to be NP-hard [Benanav *et al.* 1985]. Independently, this fact was shown in 1996 by the author, confer to Appendix A.2. The actual complexity of Makanin's algorithm is a lot worse: It is nondeteministic double exponential in the exponent of periodicity of a minimal solution of the equation [Jaffar, 1990].

3.3 An Unsolvable Problem

The theory of bit-vectors with variable width together with concatenation, boolean operations and variable extraction is not solvable by means that a complete frame solver cannot exist. This is shown by a transformation of the halting problem on empty tape to a quantified bit-vector term. The existence of a solver would—together with the Quantification Lemma 2.9—yield a decision procedure for this undecidable problem.

3.3.1 Turing Machines (cf. [Sch92a])

A *deterministic Turing machine*[f] M is a 6-tuple

$$M = (S, \Sigma, \delta, s_0, \Box, s_T)$$

where

- S is a finite set of states
- Σ is the alphabet
- $\delta : S \times \Sigma \to S \times \Sigma \times \{L, R, N\}$ is the transition function

[f]The applied restrictions are without loss of generality.

- $s_0 \in S$ is the initial state
- $\square \in \Sigma$ is the blank symbol
- s_T is the (only) accepting terminal state

A *configuration* is a tuple (s, p, q) where $s \in S$ and $p \in \Sigma^*$, $q \in \Sigma^+$. Intuitively, at this point of time the Turing machine is in the internal state s and the read/write-head is right of p and on the first symbol of q. The tape is considered infinite to both sides, i.e. left from p and right from q there are but blanks.

The binary relation $\vdash$ ("computes to") is defined as follows:

$$(s, a_1 \cdots a_l, b_1 \cdots b_m) \vdash \begin{cases} (s', a_1 \cdots a_l, db_2 \cdots b_m) & \text{if} \quad \delta(s, b_1) = (s', d, N) \\ (s', a_1 \cdots a_{l-1}, a_l db_2 \cdots b_m) & \text{if} \quad \delta(s, b_1) = (s', d, L) \\ (s', a_1 \cdots a_l d, b_2 \cdots b_m) & \text{if} \quad \delta(s, b_1) = (s', d, R) \end{cases}$$

Together with two special cases

$$\begin{array}{lcll} (s, \varepsilon, b_1 \cdots b_m) & \vdash & (s', \varepsilon, \square d b_1 \cdots b_m) & \text{if} \quad \delta(s, b_1) = (s', d, L) \\ (s, a_1 \cdots a_l, b_1) & \vdash & (s', a_1 \cdots a_l d, \square) & \text{if} \quad \delta(s, b_1) = (s', d, R) \end{array}$$

Intuitively, if the available tape is exceeded, some blanks are supplied.

An *accepting computation on x* is a finite string $k_0\$k_1\$ \cdots \$k_n$ where each k_i is a configuration, $k_0 = (s_0, \varepsilon, x)$, $k_n = (s_T, \varepsilon, \square)$ and $\forall i \in \{0, \ldots, n-1\}$: $k_i \vdash k_{i+1}$.

Theorem 3.5:

Given a Turing machine M. It is undecidable whether there exists an accepting computation on the string $\square$.

Confer for example to [Sch92a, p. 121].

3.3.2 Encodings of Computations

An accepting computation can be encoded via bit-vector terms with unrestricted size, variable extraction and boolean operations. In particular, given a Turing machine M, one can compute a bit-vector equation $\Xi(M)$ with:

$$M \text{ stops on empty tape} \qquad \textit{iff} \qquad \Xi(M) \text{ is satisfiable}$$

Encoding the Alphabet Σ

Let Σ be an arbitrary alphabet with $\square \in \Sigma$. Without loss of generality there exist three special characters "$\langle$", "$\rangle$", "$\#$" $\notin \Sigma$. Let $\Sigma' := \Sigma \cup \{\rangle, \#\}$, $\quad \Sigma'' := \Sigma \cup \{\langle, \rangle, \#\}$. Since we also want to encode states with 0/1-strings and a encoding of static size is desired, we define

$$r \quad := \quad max\Big(\lceil \log_2(|\Sigma'|) \rceil + 1, \quad \lceil \log_2(|S|) \rceil + 1\Big)$$

Then each letter of Σ' can be encoded via a String in $\{0,1\}^{r-1}$. Define a mapping $\varphi : \Sigma'' \cup S \to \{0,1\}^r$ where

$$\begin{array}{rlccc} & \varphi: & \Sigma' & \to & 0 \cdot \{0,1\}^{r-1} \\ \text{with} & \varphi: & \# & \mapsto & 0^r \\ \text{and} & \varphi: & \langle & \mapsto & 1^r \\ \text{and} & \varphi: & S & \to & 0 \cdot \{0,1\}^{r-1} \end{array}$$

φ is chosen injective with respect as well to Σ'' as to S, though not to $\Sigma'' \cup S$. The extension of φ to $(\Sigma \cup S)^*$ is denoted with $\hat{\varphi}$.

It is crucial for the following argumentation that the encoding of "$\langle$" is the only one beginning with a "1".

Encoding State Transitions

There has to be a method to encode the "correctness" of a computation step by means of "matching" configurations k_i and $k_i + 1$. There following difficulties are encountered:

- it has to be possible for characters to "swap" to the other side of the read/write-head
- sometimes it is necessary to "pump" some blanks in or - if we encountered the end of the tape - delete some blanks.

Informally, the relevant point of any computation step can be coded to bit-vectors like

$$s_{[r]}, \quad [\cdots a_{[r]}], \quad [b_{[r]} c_{[r]} \cdots] \quad \vdash \quad s'_{[r]}, \quad [\cdots a'_{[0|r|2r]}], \quad [d_{[r]} c'_{[0|r|2r]} \cdots]$$

where $d_{[r]}$ is the new character under the read/write-head and $a'_{[0|r|2r]}$,$c'_{[0|r|2r]}$ are the end of the preceding tape (respectively the beginning of the succeeding tape). The notation $[0|r|2r]$ is meant to indicate that either of these three sizes is possible. Note that the actual "correct" value is dependent on δ.

Instead of $a'_{[0|r|2r]}$ consider the string $a'' := a'_{[0|r|2r]} \otimes 1 \otimes 0^{2r}$. Though the size of a'' remains unknown, all relevant information is contained in the leftmost $2r$ bits. Moreover, since the "padding" on the right side is terminated with a "1", it is possible to reconstruct content *and* size of $a'_{[0|r|2r]}$ from the lower $2r + 1$ bits. This trick is used to perform the following definition:

$$valid_\delta\Big(s_{[r]}, a_{[r]}, b_{[r]}, c_{[r]},\ s'_{[r]}, (a'_{[0|r|2r]} \otimes 1 \otimes 0^{2r})[0:2r], d_{[r]}, (c'_{[0|r|2r]} \otimes 1 \otimes 0^{2r})[0:2r]\Big) := \begin{cases} 1_{[1]} & \text{if} \quad (s, \cdots a, bc \cdots) \vdash (s', \cdots a', dc' \cdots) \\ 0_{[1]} & \text{else} \end{cases}$$

Strictly speaking, the input to $valid_\delta$ is a combination of $r + r + r + r + r + 2r + 1 + r + 2r + 1 = 10r + 2$ bit-vectors of width 1. It is clear that, given δ, a corresponding boolean function resulting in a bit-vector of width 1 can be constructed.

Encoding Computations

The whole encoding idea works as follows:

- Guess a string encoding an accepting computation,
- Verify that each step of the computation is correct.

Here we have got the problem of separators—informally, they cannot be encoded reliable. More precisely, one cannot encode an arbitrary string *not* containing the encoding of a special character. This problem is avoided by introducing a "step counter". This is a string over $\{\#\}^+$, increasing with every step by one "#". The k^{TH} computation step $(s, \cdots a, bc \cdots) \vdash (s', \cdots a', dc' \cdots)$ is represented as

$$\langle s_{[r]} \#^k \rangle p_{[?\cdot r]} a_{[r]} \#^k b_{[r]} c_{[r]} q_{[?\cdot r]} \langle s'_{[r]} \#^{k+1} \rangle p_{[?]} a'_{[0|r|2r]} \#^{k+1} d_{[r]} c'_{[0|r|2r]} q_{[?\cdot r]}$$

The term $\#^k$ is not a valid bit-vector term itself, but can be replaced by one. Since φ maps $\#$ to 0^r, this can be matched by a bit-vector term $k_{[?\cdot r]}$ consisting entirely out of **0**s. This property, though, can be enforced by adding the simple equation $k_{[?\cdot r]} \otimes 0_{[1]} \overset{!}{=} 0_{[1]} \otimes k_{[?\cdot r]}$. Thus, the term above is really represented as

$$\langle s_{[r]} k_{[?\cdot r]} \rangle p_{[?\cdot r]} a_{[r]} k_{[?\cdot r]} b_{[r]} c_{[r]} q_{[?\cdot r]} \langle s'_{[r]} k_{[?\cdot r]} 0^r \rangle p_{[?]} a'_{[0|r|2r]} k_{[?\cdot r]} 0^r d_{[r]} c'_{[0|r|2r]} q_{[?\cdot r]}$$

Dealing with Large Alphabets

The aspired formula is be roughly something like

$$\exists w_{[n\cdot r]} \forall i > const \exists j : w_{[n\cdot r]}[j:i] = \langle \text{k}^{\text{TH}} \text{ state} \ldots \langle (\text{k+1})^{\text{TH}} \text{ state} \ldots$$

However, this can not be achieved straight-forwars.
The i^{TH} character of $w_{[n\cdot r]}$ is required to be a "⟨"—this cannot be true for all i. The first idea is to introduce some padding in front of "⟨k$^{\text{TH}}$ state..." to allow the matching to slip to the next "⟨". But this is not a solution. Since the computation is arbitrary long, the encoded tape cannot be restricted. Therefore, the length of the padding cannot be restricted in general as well. This means, that the matching could slip to *any* next "⟨", thus destroying the correctness condition, for "nonsense steps" could exist that this formula would not encounter as failures.
The second thought is to avoid certain bit-patterns in the padding, thus introducing something like separators. As stated above, this does not seem to be possible in general for alphabets larger than $|\Sigma| = 2$.
However, the following idea works: If the first character is *not* a "⟨", the whole matrix evaluates to a trivial equation at once. Let $t_{[m]} \stackrel{!}{=} u_{[m]}$ be the original matrix equation. Then this idea is realized by the following detour:

- Instead of $t_{[m]} \stackrel{!}{=} u_{[m]}$, equivalently state $\neg(t_{[m]} \equiv u_{[m]}) \stackrel{!}{=} 0_{[m]}$.
- Due to the special encoding of "⟨", the extraction $[j : i]$ matches an "⟨", iff $w_{[n\cdot r]}[i : i] = 1_{[1]}$.
- Let $v_{[m]}$ be another bit-vector variable
- $v_{[m]}$ can be "forced" to consist purely of the same bit like $w_{[n\cdot r]}[i : i]$ via $v_{[m]} \otimes w_{[n\cdot r]}[i : i] \stackrel{!}{=} w_{[n\cdot r]}[i : i] \otimes v_{[m]}$
- Thus, $\neg\Big((t_{[m]} \equiv u_{[m]}) \vee v_{[m]}\Big) \stackrel{!}{=} 0_{[m]}$ is satisfied either if
  - $t_{[m]}$ and $u_{[m]}$ can be made equivalent,
  - *or* the term $w_{[n\cdot r]}[j : i]$ does not start with "⟨".

We are now ready to take a look at the complete formula.

The Reduction Formula

Let M be a Turing machine and $valid_\delta$ the corresponding boolean function. Then a bit-vector equation $w_{[n\cdot r]}$ encoding an accepting computation is encoded in the following quantified formula $\Xi_Q(M)$:

$$
\begin{array}{l}
\exists n \in I\!N, n > 1. \exists l_w \in I\!N. \exists w : bvec_{l_w\cdot r}. \exists l_{w'} \in I\!N. \exists w' : bvec_{l_{w'}\cdot r}.\\
\quad \forall i \in I\!N. \exists j > i. \exists v : bvec_{(j-i+1)\cdot r}. \exists l_k \in I\!N. \exists k : bvec_{l_k\cdot r}.\\
\quad \exists l_p \in I\!N. \exists p : bvec_{l_p\cdot r}. \exists l_q \in I\!N. \exists q : bvec_{l_q\cdot}. \exists a : bvec_r. \exists b : bvec_r. \exists c : bvec_r.\\
\quad \exists l_a \in I\!N_0, l_a < 2. \exists a' : bvec_{l_a\cdot r}. \exists d : bvec_r. \exists l_c \in I\!N, l_c < 2. \exists c' : bvec_{l_c\cdot r}.\\
\qquad\qquad v \otimes w[i\cdot r : i\cdot r] \stackrel{!}{=} w[i\cdot r : i\cdot r] \otimes v\\
\qquad \wedge \quad valid_\delta\big(s, a, b, c, s', (a' 1_{[1]} 0_{[2r]})[0 : 2r], d, (c' 1_{[1]} 0_{[2r]})[0 : 2r]\big) \stackrel{!}{=} 1_{[1]}\\
\qquad \wedge \quad \neg\Big(\big(\varphi(\langle) s k \varphi(\rangle) p a k b c q \varphi(\langle) s' k 0_{[r]} \varphi(\rangle) p a' k 0_{[r]} d c' q \equiv w[i\cdot r : j\cdot r]\big) \vee v\Big) \stackrel{!}{=} 0_{[(j-i+1)\cdot r]}\\
\qquad \wedge \quad \hat{\varphi}\Big(\langle s_0 \# \rangle \Box \# \Box\Box\Big) \otimes w' \otimes \hat{\varphi}\Big(\langle s_T \#^n \rangle \Box \#^n \Box\Box\Big) \stackrel{!}{=} w
\end{array}
$$

The same formula without quantification is referred to as $\Xi(M)$.
With respect to better readability the matrix is written as a conjunction. However, it can easily be verified that this conjunction can be written equivalently as *one* equation, where

$$\bigotimes(\text{left hand sides}) \stackrel{!}{=} \bigotimes(\text{right hand sides}).$$

Lemma 3.6: [Turing Simulation]
An accepting computation of M exists iff $\Xi_Q(M)$ is valid.

Proof: $\Rightarrow$: Assume $K := k_0\$k_1\$\cdots\$k_n$ is an accepting computation. Then K can be transformed into a bit-vector $w_{[l_w \cdot r]} := \xi(k_0)\otimes\xi(k_1)\otimes\cdots\otimes\xi(k_n)$, where

$$\xi(s_i, p_i, q_i) = \hat{\varphi}\Big(\langle\, s_i\#^i\rangle p_i\#^i q_i\Big)$$

Then, for each step an extraction $w_{[l_w \cdot r]}[i:j]$ exists. Thus, $\Xi_Q(M)$ proves to be valid with respect to the quantification and according to the Quantification Lemma 2.9.

$\Leftarrow$: If $\Xi_Q(M)$ proves to be valid, a bit-vector w exists that can be transformed via φ^{-1} into an accepting computation.

$\square$

3.3.3 The Non-Existence Theorem

The preceding observations lead to the main theorem of this section:

Theorem 3.7: [Non-Existence Theorem]

There is no complete frame solving algorithm for the theory of bit-vectors with variable width, variable extractions and boolean operations.

Proof: Assume such an algorithm s exists. Then, given an arbitrary Turing machine M, $s(\Xi(M))$ yields either **false**, **true** or a set of frames representing a most general solution. In the first case, no accepting computation exists. In the later two cases the application of the quantification lemma (2.9) allows to check the validity of the quantified formula $\Xi_Q(M)$. According to Lemma 3.6, a computation exists if and only if $\Xi_Q(M)$ is valid. Thus the halting problem of M on the empty tape can be decided. This yields the contradiction.

$\square$

3.4 Semaphore

We have seen that the solution of bit-vector equations is a task less trivial than it is desired. There seems to be a natural difference between solving in theories with fixed or unknown size. In particular, no complete classic solver can exist for the later one (cf. Example 2.6).
This motivates the further proceeding: The next chapter is dedicated to the discussion of solving fixed size. Some approaches are presented and compared with respect to their efficiency. One of these is taken over to the last chapter, discussing unknown size, and adopted to the requirements there.

Chapter 4

Solving Fixed-Sized Bit-Vector Equations

By three methods we may learn wisdom:
First, by reflection, which is noblest;
Second, by imitation, which is easiest;
and third by experience, which is the bitterest.
(Confucius, 551-479 b.C.)

In this chapter, three distinct approaches for solving fixed-sized bit-vector equations are explored and compared regarding expressiveness, efficiency and possible extension to more general theories.

4.1 Solving Bit-Vector Equations via Monadic Logic

Weak monadic second order logic of one successor is used to express bit-vector equations. This language is a decidable fragment of second order logic (cf. [Büc62]). By means of the MONA Tool [BK95, HJJ$^+$96] the formulae are transformed to finite automata that can be utilized to construct a solver for the theory of fixed-sized bit-vectors with concatenation, extraction, boolean operations and arithmetic. Various run-time experiments are performed.

4.1.1 In the Domain of WS1S

$WS1S$ stands for weak second order logic with one successor.

Definition 4.1 [The Language WS1S]
The language $WS1S$ consists of three syntactic categories, each of which contains variables, constants and existential and universal quantification:

- *booleans* [0^{TH} order] including the usual boolean connectives,
- *positions* [1^{ST} order] interpreted as natural numbers from 0 up to the upper bound \$,
- *sets* [2^{ND} order] of positions, namely subsets of $\{0, \ldots, \$\}$.

Formulae in $WS1S$ are defined in the usual way. Of special interest is the operation $p + n$ which is defined as a function on positions or sets, given the second argument n is a fixed integer number. The semantics of '+' is an increment respectively a family of increments of n steps to the right, applied on the first argument. ⌟

```
pred at_least_two(var0 a, b, c) = (a & b) | (a & c) | (b & c);
pred mod_two(var0 a, b, c, d) = (a <=> b <=> c <=> d);

pred add2(var2 A, B, Result) =
 ex2 C: ((all1 p : (mod_two (p in A, p in B, p in C, p in Result)) &
                   ( (p+1 > 0) =>
                    ((p+1 in C ) <=>
                     at_least_two(p in A, p in B, p in C)))) &
         (0 notin C));
```

Figure 4.1: A Simple Ripple-Carry Adder in $WS1S$

Note that the language $WS1S$ is expressive enough to encode Presburger arithmetic (cf. [Pre29], [Bar93, Rabin: Decidable Theories]). If each set is understood as a collection of bits (starting at position 0 as the least significant bit), this results in an intuitive encoding of $I\!N$. Addition with two summands is encoded by means of a ternary predicate, simulating a ripple-carry adder (cf. Figure 4.1), extension to more summands is straight-forward.

4.1.2 Encoding Fixed-Sized Bit-Vector Equations in $WS1S$

Given a bit-vector equation $t = u$, an equivalent $WS1S$-formula is generated, which is then translated to a correlated automaton. Finally, this automaton is used to generate a most general solution for $t = u$.

Definition 4.2 Let $pad : \bigcup_{i=1}^{\$} bvec_i \rightarrow \{WS1S\text{–}sets\}$ be defined as

$$pad(c_{[n]}) \quad := \quad \left\{ i \,\middle|\, 0 \leq i < n \;\wedge\; c_{[n]}[i:i] = \mathbf{1}_{[1]} \right\}$$

A bit-vector equation $t = u$ and a $WS1S$-formula φ are called *equivalent*, if

- The only free variables in φ are second order.
- There exists an isomorphism ψ between $V_{t=u} := vars(t) \cup vars(u)$ and $V_\varphi := vars(\varphi)$.
- for all $w_1 : bvec_{m_1}, \ldots, w_n : bvec_{m_n}$:
  $x_1 = w_1 \wedge \ldots \wedge x_n = w_n \models t = u \quad \text{iff} \quad \psi(x_1) = pad(w_1) \wedge \ldots \wedge \psi(x_n) = pad(w_n) \models \varphi.$

$\lrcorner$

Lemma 4.1:

For each bit-vector equation $t = u$, there exists an equivalent $WS1S$-formula φ.

Proof: [**by Construction**]
Let $t = u$ be a bit-vector equation with variables $V_{t=u} = \{x_{1[m_1]}, \ldots, x_{n[m_n]}\}$. Roughly, the construction defines a conjunction Φ of $WS1S$ formulas called axioms, a conjunction Ψ of $WS1S$ conditions and a second order equation $T = U$ that is a translation of $t = u$ to $WS1S$. A sketch of the construction of φ is given as follows:

- For each occurring length of subterms introduce a second order variable $Filter_i$ and add to Φ:
  `all1 p: ((p <` i`) => p in` $Filter_i$`) & ((`i `<= p) => p notin` $Filter_i$`)`
- For each $x_{i[m_i]}$ introduce a second order Variable VAR_x_i and add the following condition to Ψ:
  `all1 p: (`m_i `< p) => p notin` VAR_x_i

- For each constant $c_{[n]}$ introduce a second order Variable $CONST\_n\_value$, where all information about $c_{[n]}$ is added to Φ. For $c_{[n]} = 1_{[2]}$ this would be:
  `(all1 p: ((2 <= p) => (p notin` $CONST\_2\_1$`)))`
  `& (0 in` $CONST\_2\_1$`) & (1 notin` $CONST\_2\_1$`)`
- Replace concatenations $t_{1[l_1]} \otimes t_{2[l_2]}$ by a position shift of the right argument, followed by a union, like `(t1) union (t2 +` l_1`)`
- Replace extractions by intersections with the proper $Filter_i$, eventually preceded by a position shift. E.G. $x_{[8]}[5:2]$ transforms to
  `(`VAR_x `- 2) inter` $Filter_4$
- Addition is simulated by means of a relation, i.e. an addition $t_{1[l]} +_{[l]} t_{2[l]}$ is represented as a constraint `add2`($t_{1[l]}$,$t_{2[l]}$,$Result$) in Ψ and the sum in the $WS1S$-term is replaced by ($Result$ `inter` $Filter_l$). For additions with more than two arguments, the corresponding predicates `add3, add4, ...` are introduced.
- Boolean operations are first mapped into equivalent boolean formulae with the operator set $\neg, \wedge, \vee$. These can be represented straightforward by the $WS1S$ operations complement, intersection and union respectively.

The actual $WS1S$ formula to process is defined as

$$\varphi \ := \ \texttt{ex2}\, Filter_i \ \texttt{ex2}\, CONST\_i\_j : \quad (\Phi \wedge (\Psi \Rightarrow (T = U)))$$

Φ contains properties of filters and constants, thus guaranteeing the only correct interpretation is choosen by the existential quantification. Ψ basically states that no variable does contain **true**-bits beyond it's width. The equivalence to $t = u$ follows by inspection. □

To give an example, the simple unsatisfiable formula $\mathbf{1}_{[1]} \otimes x_{[5]} = x_{[5]} \otimes \mathbf{0}_{[1]}$ is transformed via the construction in Lemma 4.1 to the equivalent $WS1S$ formula in Figure 4.2.

4.1.3 From Bit-Vector Equations to Finite Automata

It is well understood, that $WS1S$ is strongly related to regular expressions (cf. [vL90b, p.137]). This relation can be expressed by means of finite automata:

Definition 4.3 Let φ be a $WS1S$-formula with free second order variables $V := \{v_1, \dots, v_n\}$ and upper bound \$ of the position. Then a finite n-tape automaton is called the *correlated automaton* A_φ, if

$$\forall w_1, \dots, w_n \subseteq \{0, 1, \dots, \$\} : \qquad A_\varphi(w_1, \dots, w_n) \text{ accepts} \quad \text{iff} \quad \{v_1 = w_1 \wedge \dots \wedge v_n = w_n\} \models \varphi$$

┘

Roughly, for each $\varphi \in WS1S$, the MONA Tool computes the correlated automaton A_φ. If the formula φ was unsatisfiable or valid, A_φ accepts $\emptyset$ or Σ^* respectively.

Lemma 4.2: [MONA **Construction**]

For each $WS1S$-formula φ, the correlated automaton can be effectively constructed.

Proof: Confer to [HJJ+96]. □

Theorem 4.3:

For each bit-vector equation $t = u$ there exists a finite automaton $A_{t=u}$ such that

1. $A_{t=u}$ accepts Σ^* if $t = u$ is a tautology,
2. $A_{t=u}$ accepts $\emptyset$ if $t = u$ is unsatisfiable,
3. $A_{t=u}$ accepts $\{(w_1, \dots, w_n) \mid t[x_{i[m_i]} \,/\, w_i] \ = \ u[x_{i[m_i]} \,/\, w_i]\}$ else.

Proof: By Lemma 4.1 and Lemma 4.2. □

```
var2 VAR_X_5;

ex2 CONST_1_1,CONST_1_0,Filter_5: (
 ((all1 p: (((p < 5) => (p in Filter_5)) &
           ((5 <= p) => (p notin Filter_5)))) &
  (all1 p: ((1 <= p) => (p notin CONST_1_0))) &
  (0 notin CONST_1_0) &
  (all1 p: ((1 <= p) => (p notin CONST_1_1))) &
  (0 in CONST_1_1) ) &
  (( ((all1 p: ((5 <= p) => (p notin VAR_X_5))) ))
   =>  ( (CONST_1_1 union ((VAR_X_5 inter Filter_5) + 1))
      =  ((VAR_X_5 inter Filter_5) union (CONST_1_0 + 5)) )) )
```

Figure 4.2: $WS1S$ Representation of the Unsatisfiable Equation $1_{[1]} \otimes x_{[5]} = x_{[5]} \otimes 0_{[1]}$

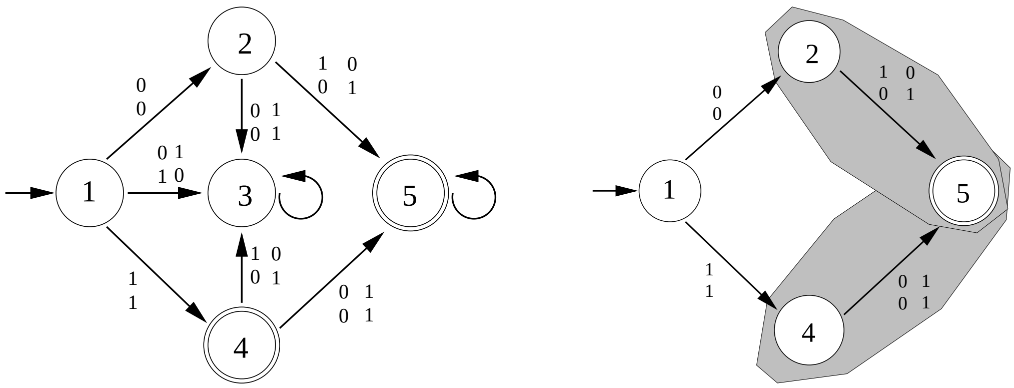
4.1.4 Constructing Solutions from Automata

Though an automaton represents an expressive encoding of a most general solution, it is desirable to compute a solved form according to section 2.2.3. This can be performed generatively by means of introducing OBDDs (see Definition 2.5). In the following a straight-forward algorithm is presented, that can be refined to yield a shorter representation of the solution.

Lemma 4.4:

For each finite automaton A, a corresponding solved form can be constructed.

Example 4.1 [Constructing the Solved Form]
Consider the bit-vector equation $x_{[2]} +_{[2]} y_{[2]} \stackrel{!}{=} 2_{[2]}$. After translation to the equivalent $WS1S$ formula, MONA returns a finite automaton, which is reduced to a form omitting non-successful branches:



Automaton as Constructed by MONA — Reduced Automaton

The upper symbol on each transition edge refers to variable $x_{[2]}$ and the lower symbol to $y_{[2]}$. Accepting nodes in a depth falling short the length of the smallest bit-vector variable are transformed to non-accepting nodes. An accepting node can only occur in the depth matching the width of the longest variable, deeper nodes and edges are deleted.
The construction works bit per bit. The first bit of $x_{[2]}$ is unrestricted, for either choice can lead to the accepting node. Thus, $x_{[2]}[0:0]$ is set to a fresh variable δ. Then, $y_{[2]}[0:0]$ is necessarily δ as well. This completes the computation of the leftmost bits, the old root (1) is deleted, leaving the two automata highlighted grey.
Starting at either of the new roots (2) or (4), the choice of $x_{[2]}[1:1]$ is arbitrary and therefore set to δ'.

In the lower path (4)-(5), where δ evaluates to **true**, $y_{[2]}[1:1]$ has to be chosen opposite to δ', whereas in the upper path (2)-(5), it has to be identical with δ'. This is expressed by means of the OBDD $\textsc{Ite}\,(\delta\,,\,\textsc{Ite}\,(\delta'\,,\,\mathbf{1}\,,\,\mathbf{0})\,,\,\textsc{Ite}\,(\delta'\,,\,\mathbf{0}\,,\,\mathbf{1}))$. The final result presents as

$$\begin{array}{lcl} x_{[2]} & = & \delta \;\otimes\; \delta' \\ y_{[2]} & = & \delta \;\otimes\; \textsc{Ite}\,(\delta\,,\,\textsc{Ite}\,(\delta'\,,\,\mathbf{1}\,,\,\mathbf{0})\,,\,\textsc{Ite}\,(\delta'\,,\,\mathbf{0}\,,\,\mathbf{1})). \end{array}$$

Proof of Lemma 4.4: [by Construction]
First, the reduced automaton is constructed, which is the initial element of a set Λ of reduced automata. The algorithm proceeds bit by bit, i.e. it traces breadth-first through every automaton in Λ in parallel, introducing fresh variables (if necessary) and checking dependencies respectively.
Each step might—while deleting the root nodes—split the automata to several new rooted automata that are stored in Λ. With each step, the representation of each concerned original variable grows by one bit, that is either a constant, a fresh variable or an OBDD. For each original variable $x_{i[m_i]}$, step j can be sketched as follows:

- Build an OBDD reflecting the dependencies of the next choice for $x_{i[m_i]}[j:j]$. It might be necessary to introduce a fresh variable as well to represent ambiguities in a distinct branch of the OBDD (i.e. if there is no functional dependency of the j^{TH} bit of $x_{i[m_i]}$).
- This OBDD might simplify to fresh variables or to a constant.
- Append this OBDD to the so-far description of $x_{i[m_i]}$, unless the width of $x_{i[m_i]}$ was already exceeded.

Continue until every automaton in Λ is reduced to the accepting node. The obtained description of each original variable is a complete and correct representation of a solution, but not necessarily the simplest one.

□

Optimization of the Algorithm

There are two points where an optimization of the sketched algorithm can be applied. The major drawback is the split-up to every single bit. This can be avoided by introducing the notion of *hyper-edges*, marked with strings instead of characters. While clustering edges to hyper-edges, which are allowed to perform several transitions at once, a smaller but equivalent automaton is obtained.

A second point is that the algorithm implicitly assumes that every variable is dependent on every other. Practically, this is rarely the case. It might be possible to split the original automaton into a set of smaller automata, each only processing a disjoint set of original variables. The optimal partition can be found by an (possibly exhaustive) independency check, like "If $y[i:i]$ is chosen arbitrarily, does this affect $x[j:j]$ in any branch?" for all x, y, i, j.

4.1.5 A Short Glimpse at the Complexity

The complete transformation of a bit-vector equation to a $WS1S$ formula leads to a moderate blow-up in size. Transformations of concatenations, extractions and even additions are performed in linear time, and yield just a linear overhead of introduced *Filter* and *Result* variables. The transformation of boolean operations to equivalent expressions using only operators $\neg, \wedge, \vee$ is possible with expense $\mathcal{O}(2^n)$, if n is the number of arguments in the original expression with any 1-ary and 2-ary connectives allowed. Not too bad. We cannot expect to obtain a solver polynomial for all inputs anyway, at least if we are fond of the conjecture $P \neq NP$. Really time-consuming is the construction of the correlated automaton—though $WS1S$ is decidable, the complexity of deciding a formula in general is staggering. As demonstrated in the following, realistic sized examples push the limits of this approach.

4.1.6 Run-Time Experiments

A grid of (hopefully) representative examples was processed by means of MONA, incrementing the usage of operators as well as the width of the terms. Either results in a noticeable slowdown—surprisingly, the capacity of MONA is soon exceeded. In particular, the usage of position shifts resulted in run-time errors[a], or—even worse—in a crash of the MONA program (as in example C1) at low width. The author does not have a satisfying explanation for this behavior, since the operation itself is expected just to shift sets of internal variables by an fixed offset.

The experiments used an Allegro Common Lisp 4.3 translation algorithm presented in Appendix C.1 and were executed on a 143MHz Sparc Ultra 1 Workstation. Measured was the run-time of the MONA program (Version 1.1), the translation time to $WS1S$ formulae was neglected. The examples are grouped into three categories A, B and C, each containing six plots. The first three plots are terms in the core theory together with boolean operations, whereas the ones on the right-hand side allow addition as well.

A. Checking Tautology

The equations processed in this category are tautologies. The task of detecting this is usually performed via canonization, but as noted in section 2.3.2, canonization of boolean operations can be an expensive operation itself. Thus, the MONA tool had to check the equivalence of n-bit functions here. As observed in A3, a concatenation (expressed by means of position shifts) is a very consumptive operation, whereas addition (A4-A6) is processed in reasonable time.

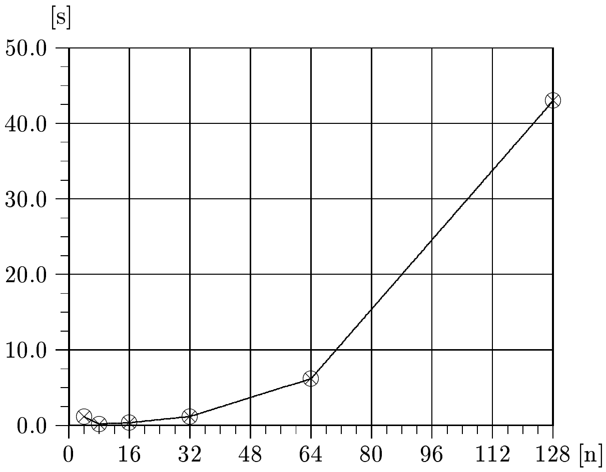
B. Checking Unsatisfiability

For unsatisfiable equations, MONA eventually results in an automaton accepting $\emptyset$. The way to obtain this automaton might be difficult, though. Again, concatenation (B1) yields a bad performance, whereas the examples involving logic an addition take at most 65 seconds.

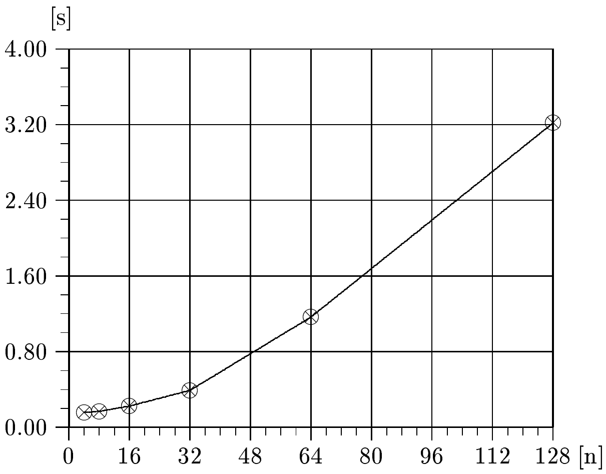
C. Satisfiable Equations

The equations in this category are satisfiable, but not valid. The time measured does *not* include the back-translation of the automaton to a solved form, but the gained automata encode the most general solution. Position shifts are once more a problem, caused by an extraction in case C1. It might be surprising, that the behavior in examples C4-C6 was roughly the same as e.g in B4-B6, though the resulting automata are far more complicated here.

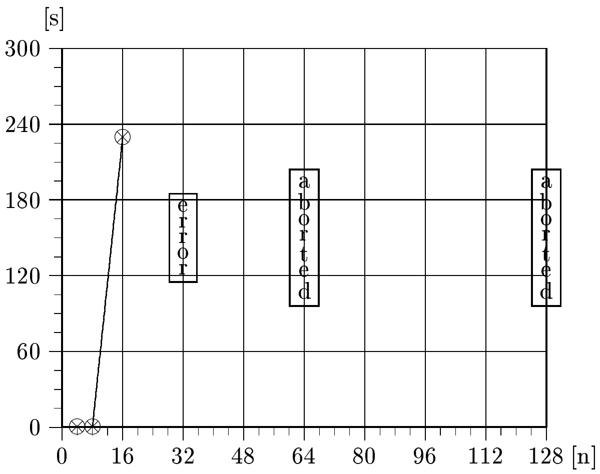
[a]In examples A3, B1, C2 and C3 MONA returned at width 16 the error message "`Memory management library: error: mem_get_block: allocation failed`".



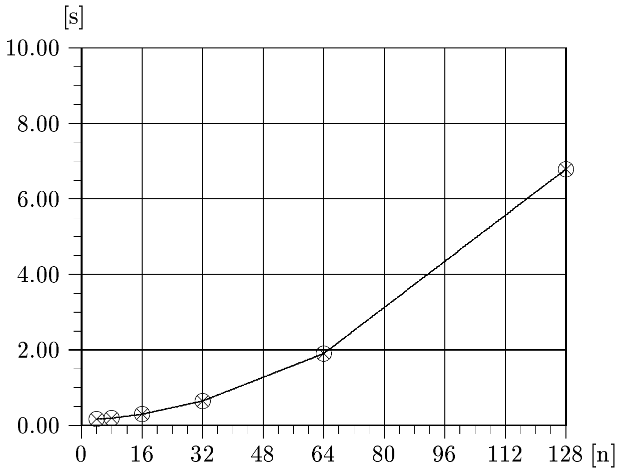
A1: $c_{[n]} \stackrel{!}{=} c_{[n]}$



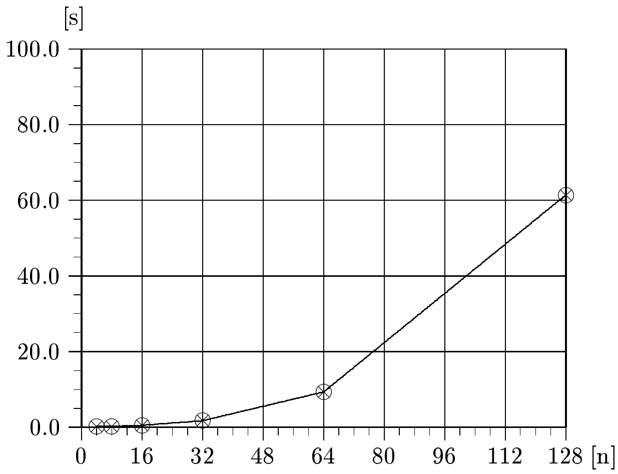
A2: $x_{[n]} \stackrel{!}{=} x_{[n]}$



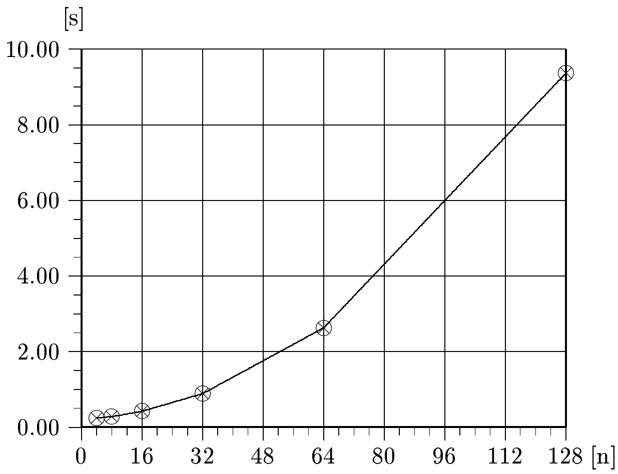
A3: $x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]} \stackrel{!}{=} x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]}$



A4: $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} y_{[n]} +_{[n]} x_{[n]}$

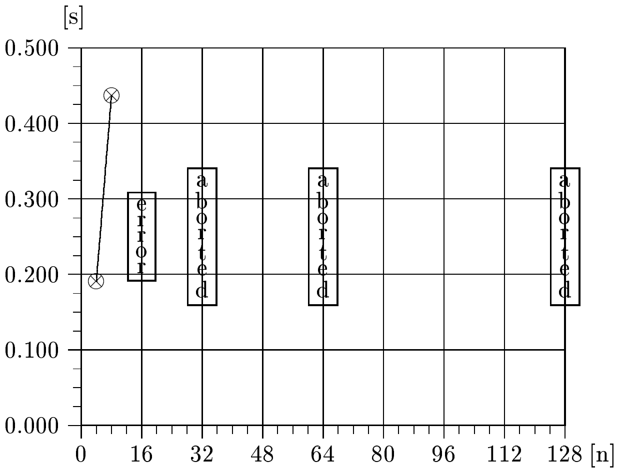


A5: $x_{[n]} +_{[n]} 0_{[n]} \stackrel{!}{=} x_{[n]}$

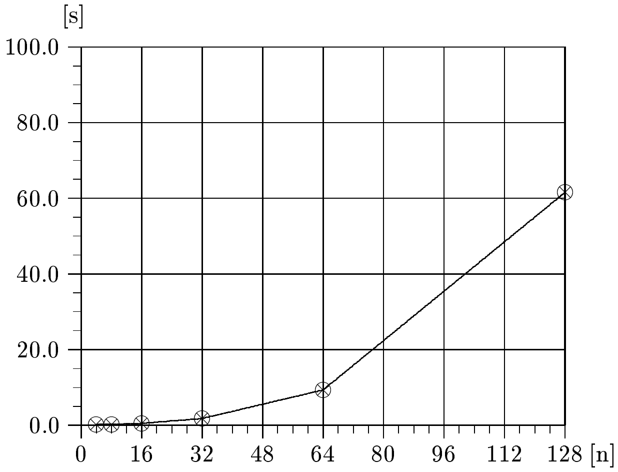


A6: $x_{[n]} +_{[n]} x_{[n]} \stackrel{!}{=} 0_{[1]} \otimes x_{[n]}[0:n\text{-}1]$

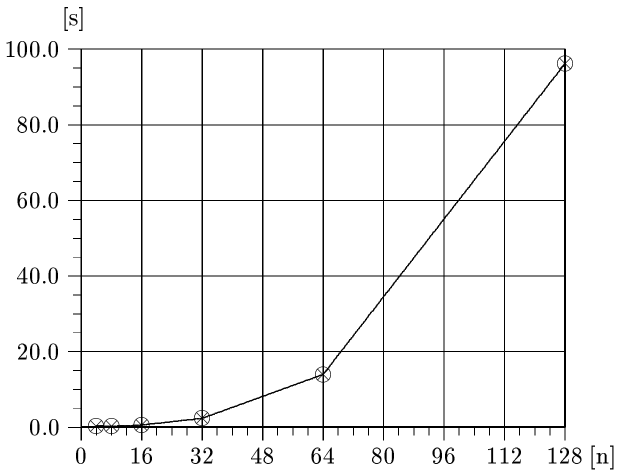
Table 4.1: Checking Tautologies via MONA



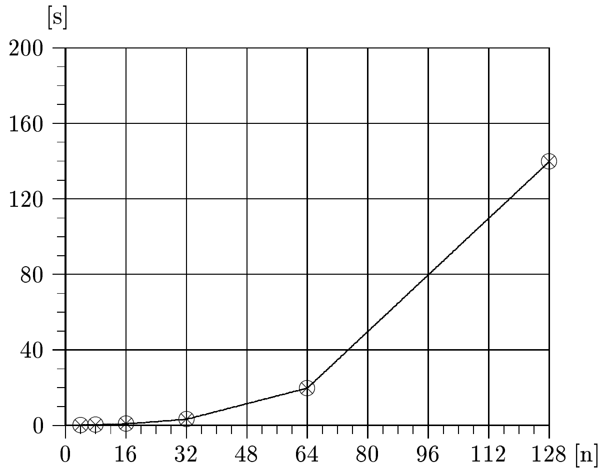
B1: $x_{[\frac{n}{2}]} \otimes x_{[\frac{n}{2}]} \stackrel{!}{=} 1_{[n]}$



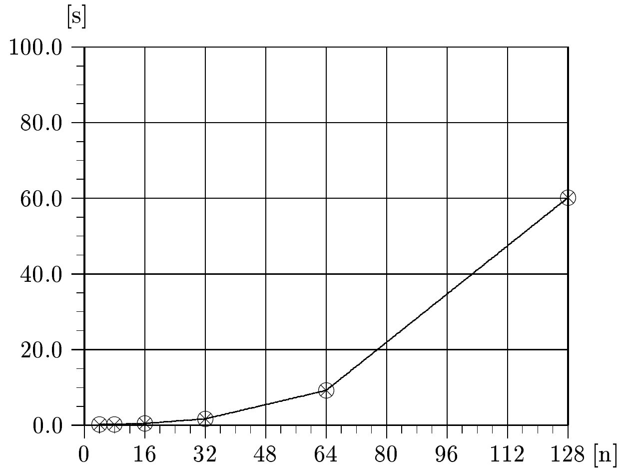
B2: $x_{[n]} \wedge \neg x_{[n]} \stackrel{!}{=} 1_{[n]}$



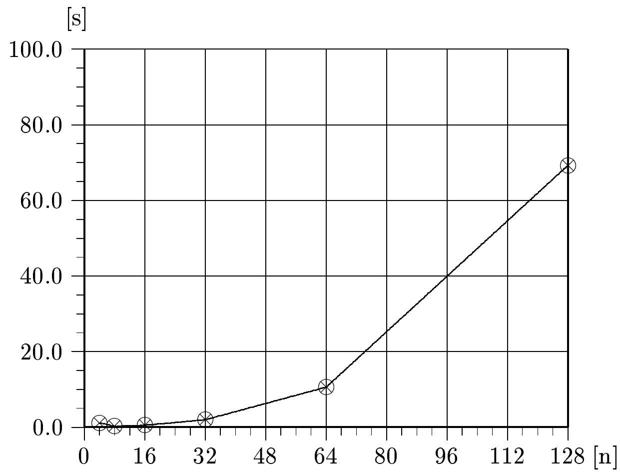
B3: $x_{[n]} \vee 1_{[n]} \stackrel{!}{=} 0_{[n]}$



B4: $c_{[n]} +_{[n]} c'_{[n]} \stackrel{!}{=} 0_{[n]}$

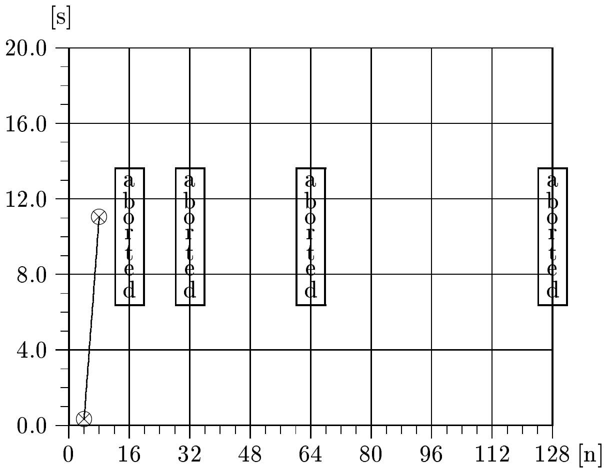


B5: $x_{[n]} +_{[n]} 1_{[n]} \stackrel{!}{=} x_{[n]}$

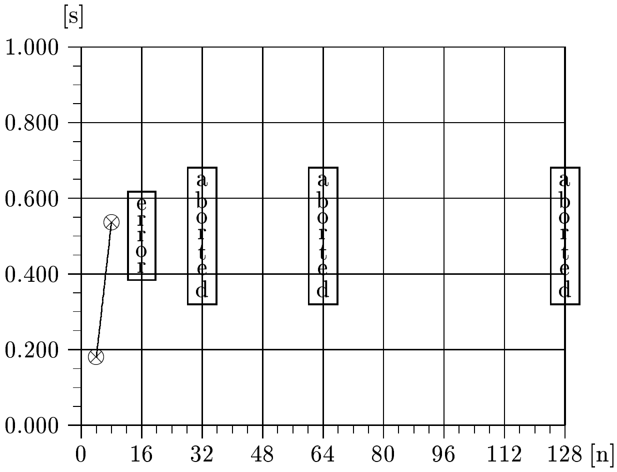


B6: $x_{[n]} \wedge (\mathbf{0} \otimes x_{[n]}[0:n\text{-}2]) \stackrel{!}{=} (\mathbf{1} \otimes \mathbf{0})^{\frac{n}{2}}$

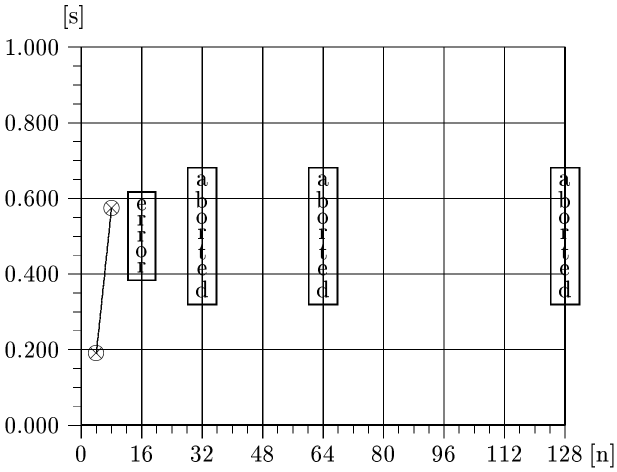
Table 4.2: Checking Unsatisfiability via MONA



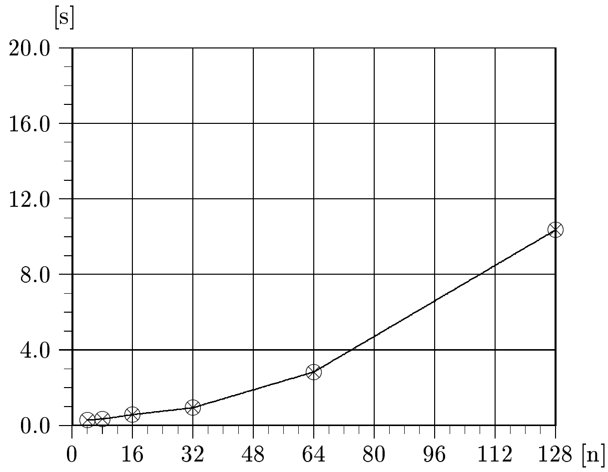
C1: $x_{[2n]}[n:2n\text{-}1] \stackrel{!}{=} x_{[2n]}[0:n\text{-}1]$



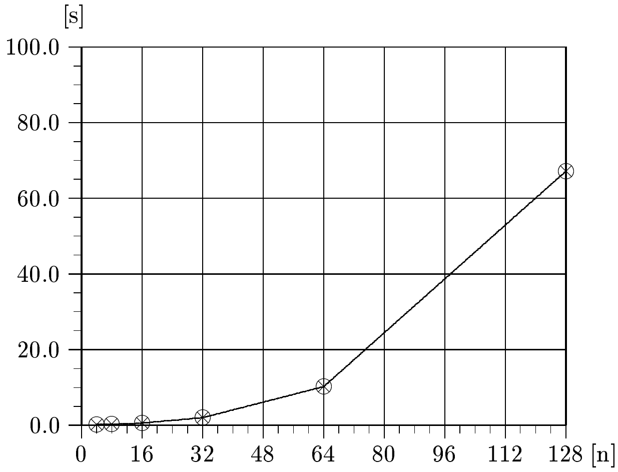
C2: $x_{[\frac{n}{2}]}\otimes y_{[\frac{n}{2}]} \stackrel{!}{=} y_{[\frac{n}{2}]}\otimes z_{[\frac{n}{2}]}$



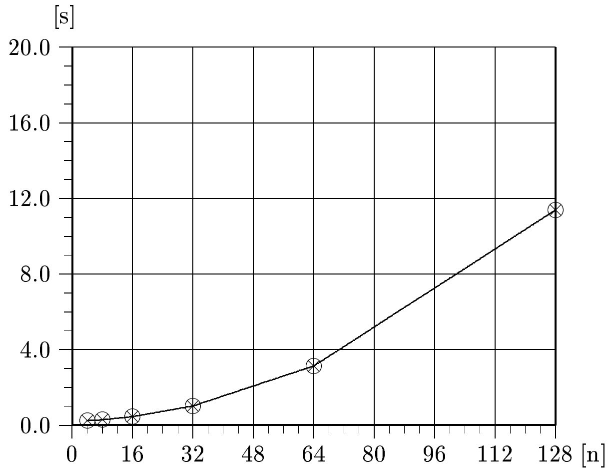
C3: $0_{[2]}\otimes x_{[n\text{-}2]} \stackrel{!}{=} x_{[n\text{-}2]}\otimes 0_{[2]}$



C4: $x_{[n]} \wedge y_{[n]} \stackrel{!}{=} z_{[n]}\, XOR\, (\neg x_{[n]})$



C5: $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} 0_{[n]}$



C6: $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} x_{[n]}\, XOR\, y_{[n]}$

Table 4.3: Satisfiable Equations via MONA

4.1.7 Extension to Larger Theories?

The $WS1S$ representation of bit-vector equations works for fixed size. Why can $x_{[n]}$ not be encoded in a similar way, if n is unknown? Of course, *one* parameter n can be allowed, if t and u contain but one parameterized variable at the rightmost position. Then, not $WS1S$ but $S1S$ formulae are constructed. However, this attempt covers only a small family of equations and the usability is therefore questionable.

The crucial point is, that in order to encode a concatenation, one has to perform a *shift* on the bit positions of some variable. These shifts are $WS1S$, iff they are *constant.* While trying to perform *variable* shifts, a serious problem is encountered. In order to represent terms like $Set + var$, it is vital to introduce a semantic of a shift by a natural number. The only way to encode this, is by means of a second order variable. If second order variables are used to encode as well the values as the width of a bit-vector, there is an ambivalence. It has to be stated which is which. At least if the proposed encoding is followed, there seems to be no way to cross this obstacle. According to the author's estimation, these difficulties are invariant under modification of the encoding.

How about using a stronger version of monadic logic, namely $WS2S$? Here we have a kind of tree logic, i.e. monadic theory with two successor functions. Though this is one of the most powerful theories known to be still decidable (cf. [BGG97]), the task of encoding bit-vector theory into it failed for the following reasons:

- We have to represent bit-vector variables as objects of arbitrary size. This can only be performed by means of exploiting the tree depth.
- In order to express a concatenation, it has to be possible to "stick" one bit-vector to the end of the other one. Thus, "the end" has to be known or marked.
- We can describe distinct positions in a tree, but the term structure allows no *shift* of a uncertain amount of positions.

These are just arguments and not a proof that the proposed task is mutually impossible.

4.1.8 Semaphore

Monadic second order logic provides a system powerful enough to encode and solve bit-vector equations. Concerning boolean connectives or simple arithmetic it shows a reasonable response time, but position shifts seem to add enormously to this complexity. This indicates that $WS1S$ does not adapt closely to the peculiarities of the theory of bit-vectors, where concatenations and extractions are used exhaustively. In addition, it remains unclear how to extend the notion in a way to express variable width of bit-vectors in general. This motivates the search for other approaches, adapting closer to the characteristics of bit-vector terms.

4.2 Solving via an Equational Transformation System

Solving is, from a certain point of view, extrapolation of information. Thus, while processing an equation, nothing is added actually, but brought into a more convenient form. This motivates to develop a solver just by means of transforming information, until a fix point—the solved form—is reached.
In this section, a simple equational transformation system is presented, that can be utilized as a solver for the core theory. It builds a basis for extension to boolean operations, arithmetic or variable width.

4.2.1 Equational Transformation Systems

Definition 4.4: [Equational Transformation System]
A *equational transformation rule* R is a formula

$$\{p_1 = q_1,\ \cdots,\ p_n = q_n\},\ pred(p_1, q_1, \ldots, p_n, q_n) \ \rightarrowtail\ \{l_1 = r_1,\ \cdots,\ l_m = r_m\}.$$

where *pred* is a 2n-ary predicate. If *pred* is omitted, it is supposed to be the constant **true**. The set $\{p_1 = q_1,\ \cdots,\ p_n = q_n\}$ is referred to as $lhs(R)$ and $\{l_1 = r_1,\ \cdots,\ l_m = r_m\}$ is denoted by $rhs(R)$.
A *matching* (M, τ) with respect to a equational transformation rule R is a set M of bit-vector equations together with an substitution τ, such that $\tau(M) \doteq lhs(R)\ \wedge\ pred(M)$.
A *equational transformation system* or *ETS* is a set $\Re$ of equational transformation rules. It operates on a set Υ of bit-vector equations. In this context it is understood that for a matching (M, τ) with respect to $R \in \Re$, Υ updates to

$$\Upsilon \quad \leftarrow \quad \Big(\Upsilon \setminus \tau\big(lhs(R)\big)\Big)\ \cup\ \tau\big(rhs(R)\big).$$

A set Υ is called *terminal* with respect to $\Re$, if there exists no matching in Υ with respect to a $R \in \Re$. ⌟

4.2.2 A Simple Strategy: Reduced Chopper

Bit-vector terms that are either variables, extractions on variables or constants are called *chunks*. The CTRS presented in this section orientates on the

Concept of the Largest Chunks:
Treat only the chunks with the highest possible width as distinct objects.

Let $x_{[n]}$ and $y_{[m]}$ denote bit-vector variables, $p_{[n]}$ and $q_{[m]}$ represent chunks and $s_{[n]}$, $t_{[k]}$ and $u_{[l]}$ stand for general bit-vector terms. The index of terms denotes their overall width. In the core theory these are in fact fixed numbers. Constants are represented as defined in section 2.1.
The underlying data structure is a set Υ of bit-vector equations. Some equational transformation rules just transform one equation to a set of other equations, some require two equations to perform a match. The equational transformation system presented in Figure 4.3 is called *reduced chopper* and is abbreviated by $\mathcal{C}_{\Re}$. In order to explain how it works, it is necessary to introduce some special notions.

Definition 4.5 A *coarsest slicing rule* or short *CS-rule* is a bit-vector equation of the form $p_{[n]} = s_{[n]}$, where $p_{[n]} \not\doteq s_{[n]}$ but $p_{[n]} \doteq \alpha(s_{[n]})$.
An *initial CS-rule set* with respect to $t = u$ or short *init-CS*$(t = u)$ is defined as

$$\textit{init-CS}(t = u) \ :=\ \big\{p_{[n]} = s_{[n]}\ \big|\ p_{[n]} = s_{[n]} \text{ is a CS-rule that can be built with chunks in } t \text{ and } u\ \big\}$$

For example, if $x_{[4]}[0:1]$ occurs in $t = u$, *init-CS*$(t = u)$ contains the CS-rule $x_{[4]} = x_{[4]}[0:1] \otimes x_{[4]}[2:3]$.
Let $t = u$ be a bit-vector equation on the variables $x_{1[m_1]}$, ..., $x_{n[m_n]}$. A *solved set* $\Upsilon_\perp$ with respect to $t = u$ is a set of equations on $x_{1[m_1]}$, ..., $x_{n[m_n]}$ where the following holds:

1. $vars\big(\Upsilon_\perp\big)\ \subseteq\ vars(t)\ \cup\ vars(u)$,
2. $\Upsilon_\perp\ \models\ t = u$ and $t = u\ \models\ \Upsilon_\perp$,
3. for each $x_{i[m_i]} \in vars(\Upsilon_\perp)$, there exists an equation $x_{i[m_i]} = t_{[m_i]}\ \in\ \Upsilon_\perp$, which is not a CS-rule,
4. $\Upsilon_\perp$ is terminal with respect to $t = u$.

⌟

A solved set is not a valid solution according to section 2.2.3, since the same variable might occur on both sides of an equation. However, a solved form can be easily obtained.
Lemma 4.5:
Given a solved set $\Upsilon_\perp$, a solved form $\Upsilon'_\perp$ according to 2.2.3 can be constructed, such that

$$\Upsilon_\perp\ \models\ \Upsilon'_\perp \qquad \text{and} \qquad \Upsilon'_\perp\ \models\ \Upsilon_\perp$$

$$
\begin{array}{llrll}
(1) & p_{[n]} \otimes t = q_{[n]} \otimes u & & \rightarrowtail & \left\{ \begin{array}{l} p_{[n]} = q_{[n]} \\ t = u \end{array} \right\} \\
(1') & p_{[n]} \otimes t = q_{[m]} \otimes u, & n < m & \rightarrowtail & \left\{ \begin{array}{l} p_{[n]} = \alpha(q_{[m]}[0 : n-1]) \\ \alpha(q_{[m]}[n : m-1]) \otimes u = t \\ q_{[m]} = \alpha(q_{[m]}[0 : n-1]) \otimes \alpha(q_{[m]}[n : m-1]) \end{array} \right\} \\
(1'') & p_{[n]} \otimes t = q_{[m]} & & \rightarrowtail & \left\{ \begin{array}{l} p_{[n]} = \alpha(q_{[m]}[0 : n-1]) \\ \alpha(q_{[m]}[n : m-1]) = t \\ q_{[m]} = \alpha(q_{[m]}[0 : n-1]) \otimes \alpha(q_{[m]}[n : m-1]) \end{array} \right\} \\
(1''') & q_{[m]} = p_{[n]} \otimes t & & \rightarrowtail & \left\{ \begin{array}{l} p_{[n]} = \alpha(q_{[m]}[0 : n-1]) \\ \alpha(q_{[m]}[n : m-1]) = t \\ q_{[m]} = \alpha(q_{[m]}[0 : n-1]) \otimes \alpha(q_{[m]}[n : m-1]) \end{array} \right\} \\
(2) & c_{[n]} = c'_{[n]} \ , & c \neq c' & \rightarrowtail & \textbf{FAIL} \\
(3) & t = t & & \rightarrowtail & \{\} \\
(4) & \left\{ \begin{array}{l} p = t \\ q = u \end{array} \right\}, & q \subseteq t & \rightarrowtail & \left\{ \begin{array}{l} p = t[q/u] \\ q = u \end{array} \right\} \\
(5) & \left\{ \begin{array}{l} p = q \\ q = r \end{array} \right\} & & \rightarrowtail & \left\{ \begin{array}{l} p = r \\ q = r \end{array} \right\} \\
(6) & \left\{ \begin{array}{l} p = q \\ q = p \end{array} \right\} & & \rightarrowtail & \{ \ p = q \ \} \\
(7) & \left\{ \begin{array}{l} p = t \\ p = u \end{array} \right\} & & \rightarrowtail & \left\{ \begin{array}{l} p = t \\ u = t \end{array} \right\} \\
(8) & c = t, & t \neq const & \rightarrowtail & \{ \ t = c \ \}
\end{array}
$$

Figure 4.3: Reduced Chopper $\mathcal{C}_{\mathfrak{R}}$ – A ETS for Solving Bit-Vector Equations

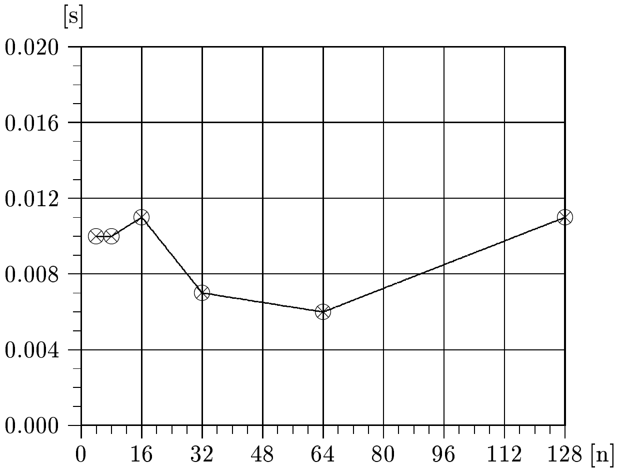
Proof: Replace all occurrences of extractions on variables on the right hand sides of the terms $x_{i[m_i]} = t_{[m_i]}$ in $\Upsilon_\perp$ by fresh variables to obtain $\Upsilon'_\perp$. □

Theorem 4.6:
Let $t = u$ be a bit-vector equation $BV_{\otimes,[1:1]}$. Started on $\{t = u\} \cup \mathit{init\text{-}CS}(t = u)$, the equational transformation system $\mathcal{C}_{\mathfrak{R}}$ always terminates with a solved set.

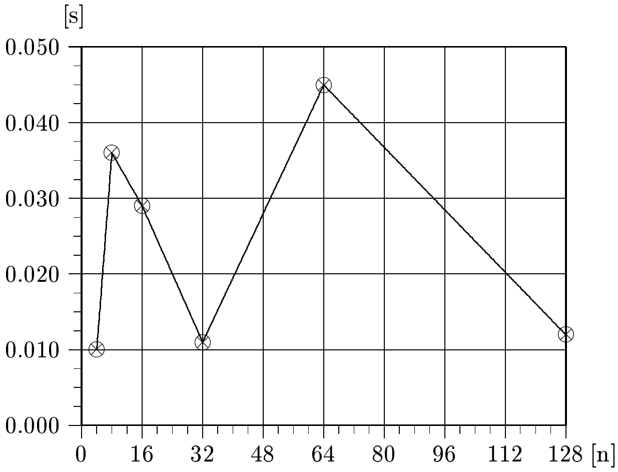
Proof Sketch:

- By inspection, the equational transformation rules (1)-(8) are equivalence-preserving, in the sense that the replacement of the matchings with the right hand sides does neither introduce new nor omit existing information. Thus, given termination, the result is a solved set.
- Termination follows from the observations
  (a) The rules (1)-(1′′′) actually decrease the width of the involved terms; thus, they can be applied only a finite number of times.
  (b) Rules (2), (3) and (6) yield a smaller set Υ.
  (c) Rules (4), (5), (7) and (8) do not enlarge Υ. Together with (6) they build up a kind of union-find structure, which is loop-free and thus terminates.

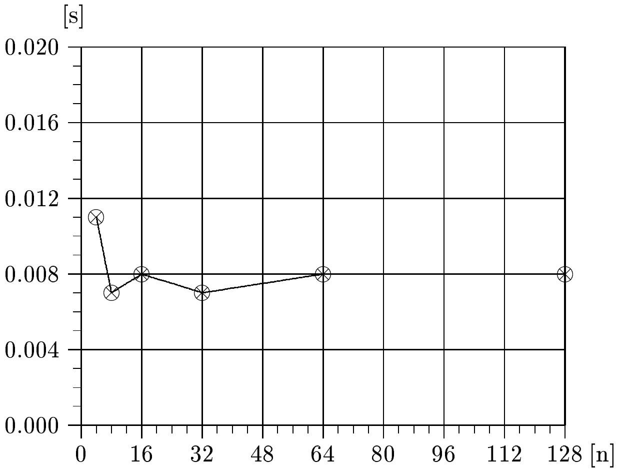
□



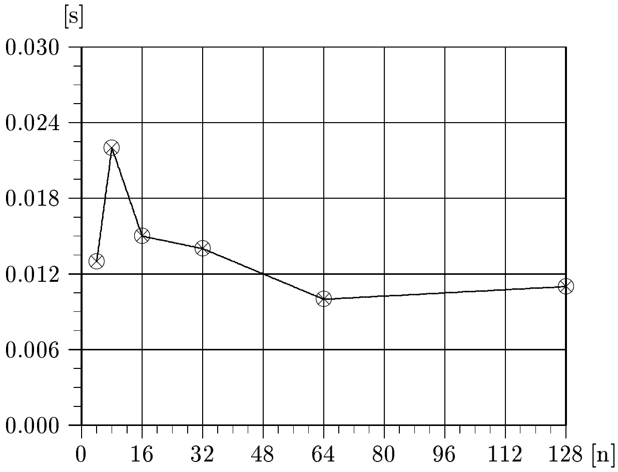
A1: $c_{[n]} \stackrel{!}{=} c_{[n]}$



A3: $x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]} \stackrel{!}{=} x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]}$



A2: $x_{[n]} \stackrel{!}{=} x_{[n]}$



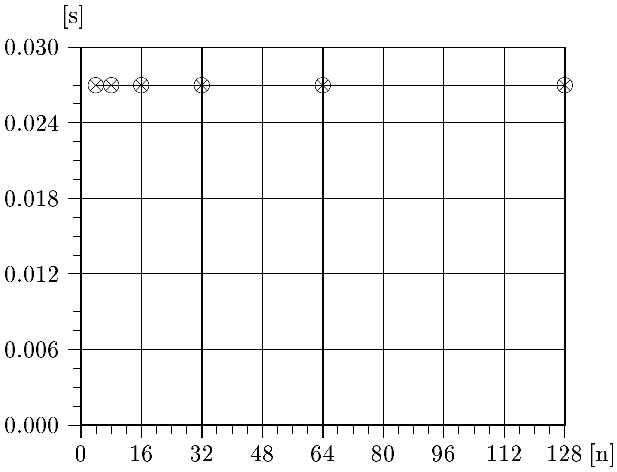
B1: $x_{[\frac{n}{2}]} \otimes x_{[\frac{n}{2}]} \stackrel{!}{=} 1_{[n]}$

Table 4.4: Checking Tautology and Unsatisfiability via Reduced Chopper

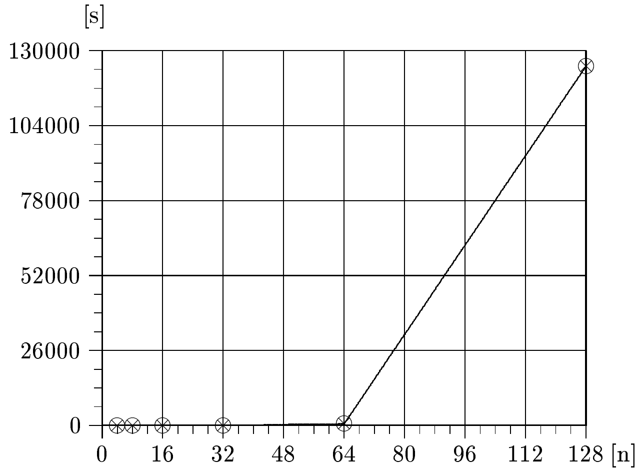
4.2.3 Run-Time Experiments with $\mathcal{C}_{\Re}$

$\mathcal{C}_{\Re}$ is implemented in Common Lisp by means of a match-and-rewrite strategy on a set of equations. The algorithm applies the rules (1)-(8) in an randomized order, thus the run-time performance is rather a hint than an accurate characterization. The measured time—again on a Sparc Ultra 1 Workstation—includes the computation of *init-CS*$(t = u)$. Since boolean operations or arithmetic are not included, some of the experiments in section 4.1.6 can not be processed. Tautologies and unsatisfiable equations are displayed in Table 4.4 and satisfiable but not valid formulae in Table 4.5.

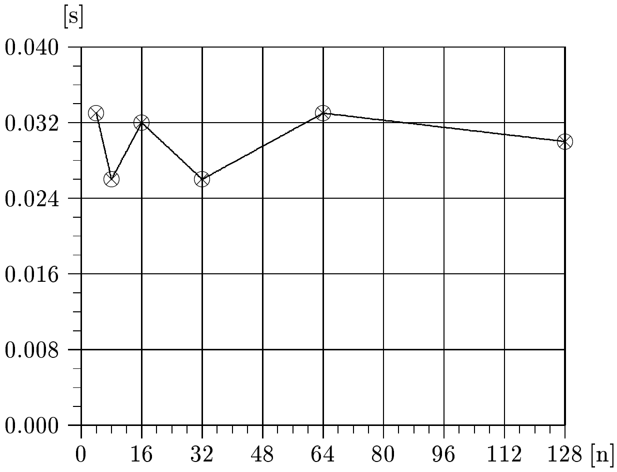
$\mathcal{C}_{\Re}$ shows a good performance on simple equations that involve concatenations like B1. It deals only with the largest chunks possible, so run-time increases not necessarily with term width (as demonstrated in examples B1, C1 and C2). This yields much faster results than the procession via $WS1S$ (cf. pages 44ff). On the other hand, it is rather expensive to split up a wide variable to numerous bits, as seen in C3 and C3b. This is explained by the growing number of CS-rules.



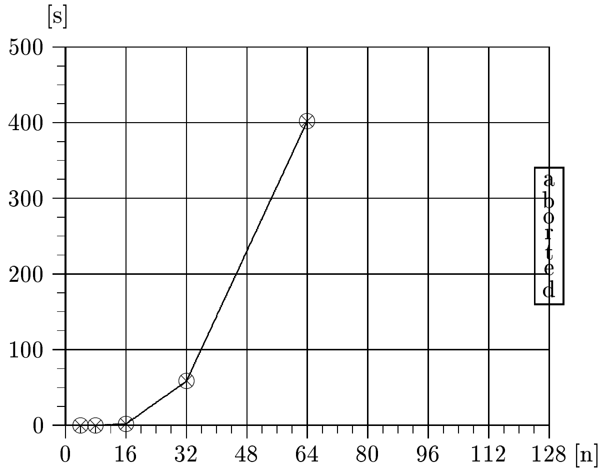
C1: $x_{[2n]}[n : 2n\text{-}1] \stackrel{!}{=} x_{[2n]}[0 : n\text{-}1]$



C3: $0_{[2]} \otimes x_{[n\text{-}2]} \stackrel{!}{=} x_{[n\text{-}2]} \otimes 0_{[2]}$



C2: $x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]} \stackrel{!}{=} y_{[\frac{n}{2}]} \otimes z_{[\frac{n}{2}]}$



C3b: $0_{[2]} \otimes x_{[n\text{-}2]} \stackrel{!}{=} x_{[n\text{-}2]} \otimes 0_{[2]}$

Table 4.5: Satisfiable Equations via reduced chopper

4.2.4 Semaphore

The reduced chopper algorithm is understood as the basis of a real efficient solver. The following drawbacks are to mention:

- Nondeterminism - the guessing of the next step is a time-consuming feature.
- Explicitness of slicing information - the storage of all occurring CS-rules in Υ can lead to an exponential blow-up in the number of equations, for there are $2^{n\text{-}1}\text{-}n$ possible CS-rules concerning the term $x_{[n]}[0 : 0] \otimes \cdots \otimes x_{[n]}[n\text{-}1 : n\text{-}1]$.
- Lack of boolean operations, arithmetic and variable width.

It is not reasonable to expect a solving concept to be as well most general and efficient. An argument in favor of this is the Non-Existence Theorem 3.7. Thus, the extension forks in two ways. First, Construct an efficient solving algorithm for fixed size on the basis of the reduced chopper; this should allow boolean operations and arithmetic. This is done in the following section. Second, Expand the equational transformation system in a way that allows processing of bit-vector terms with variable width, thus constructing a frame solver. This is what the next chapter is about.

4.3 The Operationalization: Fixed Solver

The concept of the largest chunks—as applied in the previous section—is extended to an efficient and deterministic representation of the reduced chopper algorithm. Simultaneously, boolean operations and arithmetics are added. The resulting algorithm is referred to as *fixed solver*.

In order to operationalize the reduced chopper, it is necessary to introduce a new paradigm: the distinction between left-hand side and right-hand side of an equation. Roughly, the left-hand side is reserved to original variables, and on the right-hand side there are constants and fresh variables. Fresh variables—in the following denoted with $a_{[n]}$, $b_{[n]}$, $d_{[n]}$ and $e_{[n]}$—are introduced in order to express equality of chunks of original bit-vector variables. Consider, for example,

$$\begin{array}{rcl} x_{[8]} & = & a_{[4]} \otimes \mathbf{0}_{[4]} \\ y_{[16]} & = & b_{[12]} \otimes a_{[4]} \end{array}$$

These equations express that $x_{[4]}$ starts with the same four bits (denoted as $y_{[16]}$ ends with. The original variables $x_{[8]}$ and $y_{[16]}$ are on the left hand side and the fresh variables on the right hand side. This proves to be a very useful concept.

4.3.1 The Algorithm in an Overview

The complete algorithm can be separated in seven subsequent phases, as sketched in Figure 4.4. Boolean operations and arithmetic is introduced by means of OBDDs and the canonical form is defined according to Definition 2.6. The details are explained in sections 2.3.2 and 2.3.3. The input to the algorithm is an bit-vector equation $t \stackrel{!}{=} u$ with fixed size, concatenation, fixed extraction, boolean operation and arithmetic allowed. The output is a solved form according to section 2.2.3. Since canonization is—strictly speaking—not part of the solver, it is listed as Phase 0. Note that tautologies are detected right after canonization.

4.3.2 Phase 1: Slicing

The two canonical terms t' and u' are either simple terms (i.e. constants, variables, extractions or bit-vector OBDDs) or concatenations of simple terms. They are *sliced* to possibly smaller simple terms, where the width of all the chunks in t'' and u'' match by pairs:

$$\begin{array}{rcl} t' & \doteq & t'_{1[l_1]} \otimes t'_{2[l_2]} \\ u' & \doteq & u'_{1[k_1]} \otimes u'_{2[k_2]} \otimes u'_{3[k_3]} \end{array} \quad \overset{\text{slicing}}{\rightarrowtail} \quad \begin{array}{rcl} t'' & \doteq & t_{1[m_1]} \otimes t_{2[m_2]} \otimes t_{3[m_3]} \otimes t_{4[m_4]} \\ u'' & \doteq & u_{1[m_1]} \otimes u_{2[m_2]} \otimes u_{3[m_3]} \otimes u_{4[m_4]} \end{array}$$

Also, all constants $c_{[n]}$ have to be split up to concatenations of terms $\mathbf{0}_{[m]}$ and $\mathbf{-1}_{[m]}$. This is necessary in order to apply transformation to OBDD leaf nodes whenever required. Since all width information is fixed, this step can be performed deterministically. The resulting set of possibly smaller equations is processed one by one via the procedure *chunk-solve*.

4.3.3 Phase 2: Chunk-Solve

The input to this sub-procedure is an equation $t_{i[m_i]} \stackrel{!}{=} u_{i[m_i]}$ on simple terms. *Chunk-solve* yields a set of equations of the form

$$\langle \textit{original variable} \rangle \quad = \quad \langle \textit{term over constants and fresh variables} \rangle$$

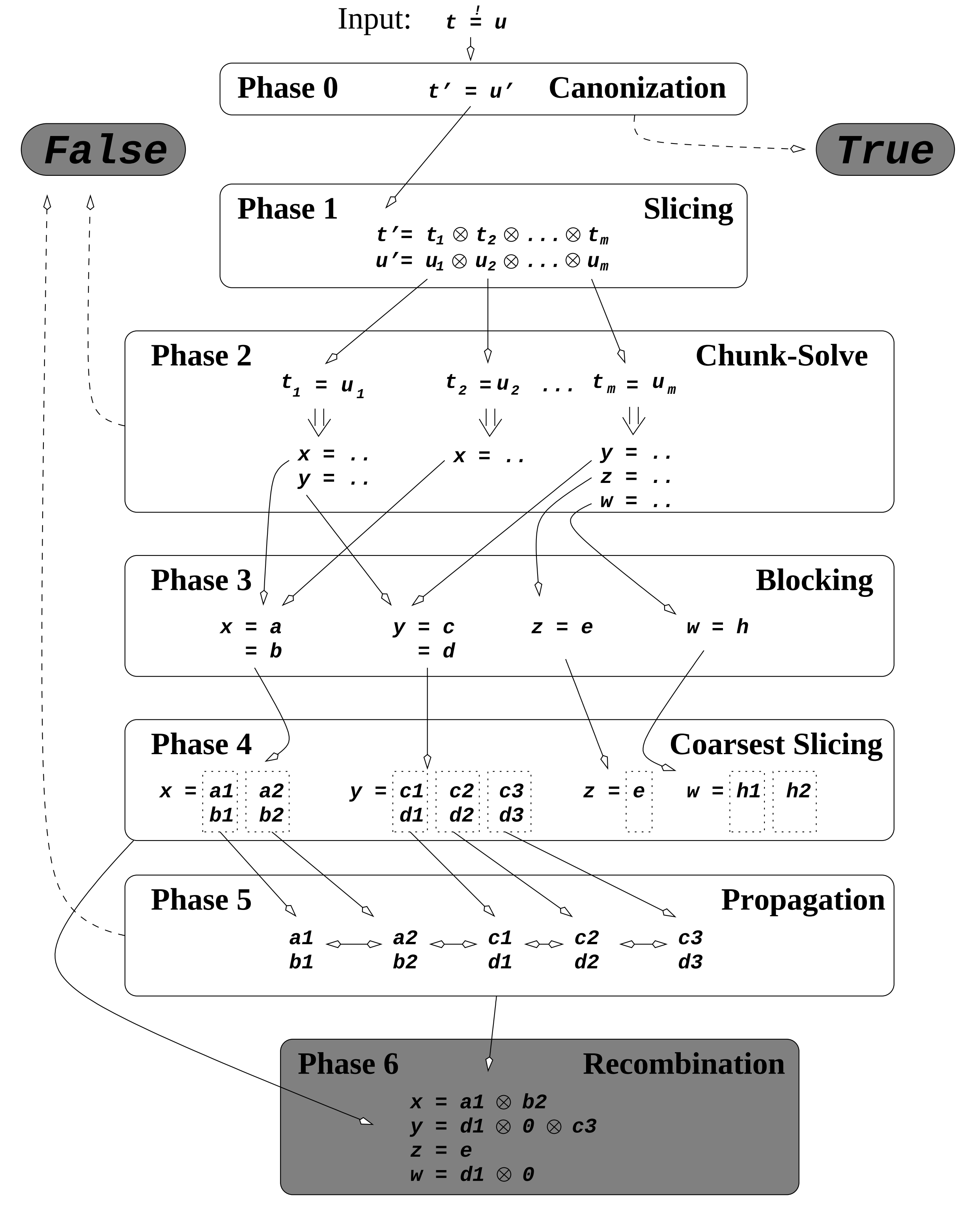


Figure 4.4: The Fixed Solver in an Overview

Recursive Procedure OBDD-solve
Input S: OBDD
LET S = ITE $(P\,,\,B_P\,,\,B_{\overline{P}})$ IN
  LET δ be a fresh variable
  $EQ_P\ :=\ P = $ ITE $(B_P\,,$ ITE $(B_{\overline{P}}\,,\,\delta\,,\,\mathbf{true})\,,\,\mathbf{false})$
  $Cstrnt\ :=$ ITE $(B_P\,,\mathbf{true}\,,B_{\overline{P}})$
    $/\star\ \ B_P \vee B_{\overline{P}}\ \ \star/$
  $EQ\text{-}set\ :=$ **OBDD-solve**$(Cstrnt)$

RETURN $\{$*replace-original-vars-on-right-hand-side-via-EQ-set*$(EQ_P)\}\ \cup$ *EQ-set*

Figure 4.5: The Recursive Sub-Procedure *OBDD-solve*

Basically, there are four cases (a)-(d):

(a) At least one term $t_{i[m_i]}$ is a constant

If the second term $u_{i[m_i]}$ is identic to $t_{i[m_i]}$, the empty set is returned. If it is a different constant, the solver aborts with **false**; otherwise, *chunk-solve* yields the set $\{u_{i[m_i]} = t_{i[m_i]}\}$.

(b) The set *vars*$(t_{i[m_i]}) \cap$ *vars*$(u_{i[m_i]})$ is empty

If no OBDDs are involved, $t_{i[m_i]}$ and $u_{i[m_i]}$ are just two chunks that are meant to be equal. In the reduced chopper algorithm, this fact is represented by this very equation, but following the paradigm of left-hand side and right-hand side, it is expressed by means of introducing a fresh variable $a_{[m_i]}$ that is put on the right-hand-side at the appropriate position. E.G. *chunk-solve*$(x_{[2]} = y_{[4]}[0:1])$ results in the set

$$\left\{\begin{array}{lcl} x_{[2]} & = & a_{[2]}, \\ y_{[4]} & = & a_{[2]} \otimes b_{[2]} \end{array}\right\}.$$

Here, the fresh variable $b_{[2]}$ is just a place holder to pad parts of $y_{[4]}$ that are not effected by the equation.

(c) Both terms are extractions on the same variable: $x_{[n]}[j:i] \stackrel{!}{=} x_{[n]}[l:k]$

Without loss of generality, let $j \leq l$. Then, three different cases are possible:

$$\begin{array}{lrcl} (1) & j = l \wedge k = l & \Rightarrow & \emptyset \\ (2) & i < l & \Rightarrow & \{\ x_{[n]} = b_{[j\text{-}1]} \otimes a_{[i\text{-}j+1]} \otimes d_{[l\text{-}i\text{-}1]} \otimes a_{[i\text{-}j+1]} \otimes e_{[n\text{-}k\text{-}1]}\ \} \\ (3) & i \geq l & \Rightarrow & \{\ x_{[n]} = b_{[j\text{-}1]} \otimes \mathrm{ext}(a_{[l\text{-}j]}, k\text{-}j+1) \otimes d_{[n\text{-}k\text{-}1]}\ \} \end{array}$$

In (2) and (3), the variables b, d and e are paddings that are omitted if their length evaluates to 0. In general, *chunk-solve* offers here a shortcut of the iterative application of the reduced chopper rules (1)-(1‴). For a detailed explanation of this case split see [CMR96], where a predecessor of the fixed solver algorithm was published.

(d) One term is an OBDD

In this case, the other term is lifted to an (possibly trivial) OBDD and the boolean connective *equivalence* "$\equiv$" is applied. Instead of solving $OBDD_1 \stackrel{!}{=} OBDD_2$, the equation $\big(OBDD_1 \equiv OBDD_2\big) \stackrel{!}{=} \mathbf{true}$ is processed. As stated in [CMR97], this can be performed via the procedure *OBDD-solve* presented in Figure 4.5. The procedure yields a set of equations, where original variables are on the left hand side and fresh variables are nodes in an OBDD on the right hand side.
To give an example, the equation $x_{[4]} \vee y_{[4]} \stackrel{!}{=} \textbf{-1}_{[4]}$ is processed as follows:

$$\textit{OBDD-solve}(\text{ITE}\ (x_{[4]}\,,\,\textbf{-1}_{[4]}\,,\,\text{ITE}\ (y_{[4]}\,,\,\textbf{-1}_{[4]}\,,\,\mathbf{0}_{[4]}))) = \left\{\begin{array}{lcl} x_{[4]} & = & a_{[4]}, \\ y_{[4]} & = & \text{ITE}\ (a_{[4]}\,,\,\text{ITE}\ (b_{[4]}\,,\,\textbf{-1}_{[4]}\,,\,\mathbf{0}_{[4]})\,,\,\textbf{-1}_{[4]}) \end{array}\right\}$$

4.3.4 Phase 3: Blocking

The numerous equations returned by *chunk-solve* are collected according to the original variables. The set of bit-vector terms on the right hand side is referred to as a *block*. The equality of these terms is propagated later on, similar to the reduced chopper rule (7) in Figure 4.3.

4.3.5 Phase 4: Coarsest Slicing

On each block a slicing according to its contents is performed. Thus, the block splits up into a set of *columns*, each consisting of a set of simple terms that are required to be equal. In order to propagate equality, referential transparency is desired. This means, if information about a chunk a is processed, only places are affected where chunk a occurs. At the current state there might exist terms like $a_{[4]}$ and $a_{[4]}[0:1]$. During this phase, all occurrences of $a_{[4]}$ are then replaced by the term $a_{[4]}[0:1]\otimes a_{[4]}[2:3]$. Possibly, this leads to further split-ups of the columns an so on. This iterative process terminates at latest, when each column is of width one. The finally reached split-up of fresh variables into the coarsest possible chunks reached is referred to as the *coarsest slicing*.

It is a justified question why this step is applied here and not much earlier. For example, if it would have been applied *during* the slicing, all occurring chunks were already known. Roughly, this is the approach followed by Bjørner and Pichora [BP98], where a normal form of an equation is computed a priori by means of functions *cut* and *dice*, thus anticipating a kind of coarsest slicing.

An argumentative consideration leads to the design decision drawn here. Due to the typical application in formal verification, most equations processed via decision procedures are either tautologies or unsatisfiable. The strategy is to detect these cases as soon as possible.

Tautologies are detected after canonization. But unsatisfiability could be detected at first after the chunk-solving. Consider for example the equation

$$\begin{array}{ccc} \mathbf{1}_{[1]} & \otimes & \mathbf{0}_{[1]}\otimes x_{[n]} & \stackrel{!}{=} \\ \mathbf{0}_{[1]} & \otimes & x_{[n]}\otimes\mathbf{0}_{[1]} & \end{array}$$

A coarsest slicing eventually splits up $x_{[n]}$ to chunks of width one. But it is obvious that this equation cannot be satisfied because of the non-matchable leftmost constants. This is detected by *chunk-solve*, thus a split-up of $x_{[n]}$ can be avoided.

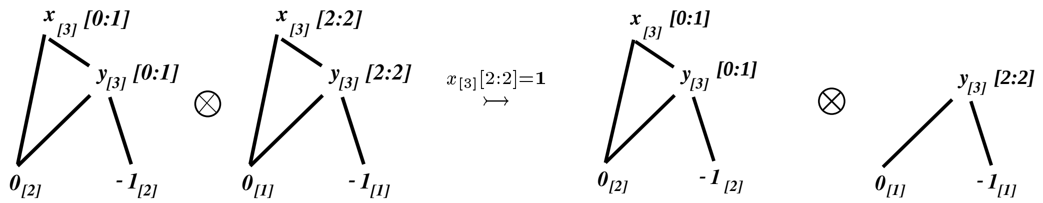
4.3.6 Phase 5: Propagation

In this step the equality within each column is propagated. The applied method is to build up a union-find structure (for example see [Sho78, GHR93]) in a way that there is a procedure pair *union* and *find*, where

$$\begin{array}{rcl} \mathit{find}(\,.\,) & : & \text{maps each occurring chunk to a unique representative} \\ \mathit{union}(\,.\,,\,.\,) & : & \text{merges the representatives of both arguments} \end{array}$$

In the beginning of the propagation, *find* maps each chunk to itself. When a union of two different constants is attempted, the algorithm aborts with a **false** like in the reduced chopper rule (2). If no OBDDs are involved in a union, the unique representative is set to the constant (if any) or one of the representatives of both arguments. If a chunk was updated to another representative, it has to vanish completely. It is crucial that this replacement operation affects every occurrence, especially in OBDD nodes. If additional information concerning a OBDD node applies, the structure of the OBDD changes. Consider for example the OBDD $x_{[3]} \wedge y_{[3]}$, where a $\mathit{union}(x_{[3]}[2:2], \mathbf{1}_{[1]})$ is called. The OBDD structure specializes as follows:

Example 4.2



$\text{ITE}\ (x_{[3]}\ , \mathbf{0}_{[3]}\ , \text{ITE}\ (y_{[3]}\ , \mathbf{0}_{[3]}\ ,\ \ \mathbf{-1}_{[3]}))$

$\text{ITE}\ (x_{[3]}[1:2]\ , \text{ITE}\ (y_{[3]}[1:2]\ , \mathbf{-1}_{[2]}\ , \mathbf{0}_{[2]})\ , \mathbf{0}_{[2]}) \quad \otimes$
$\text{ITE}\ (y_{[3]}[0:0]\ , \mathbf{1}_{[1]}\ , \mathbf{0}_{[1]})$

It is guaranteed by the coarsest slicing that the split-up as seen on the left side was already performed.

Applying *union* on Bit-Vector OBDDs

If at least one argument of *union* is an OBDD, the operation is slightly more complex. Intuitively, it has to be made explicit that the equation $argument_1 \overset{!}{=} argument_2$ evaluates to **true** in any case. Both arguments can contain arbitrary many chunks. This task can be performed as follows:

- Build $OBDD := (argument_1 \ \equiv \ argument_2)$,
- if $OBDD = \textbf{false}$, abort the algorithm and return **false**; else
- Compute *eq-set* := *OBDD-solve*($OBDD$),
- Replace any occurrence of a chunk on the left-hand side in *eq-set* by the corresponding right-hand side.

4.3.7 Phase 6: Recombination

If the algorithm did not abort with **true** or **false** previously, a conjunction of equations of the following form is generated:

$$\langle \textit{original variable} \rangle \quad = \quad \langle \textit{term over constants and fresh variables} \rangle$$

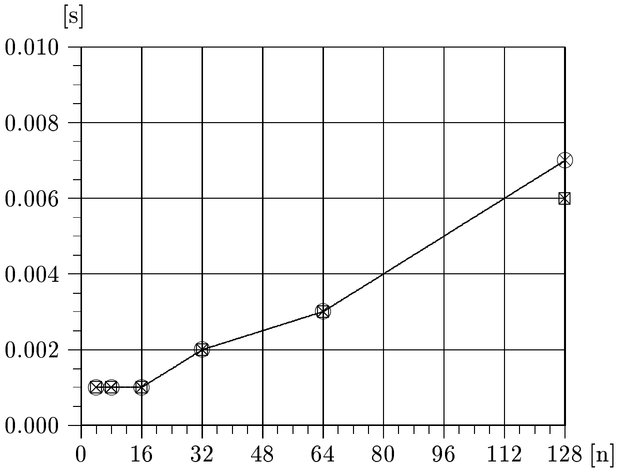
The right hand side is constructed as a concatenation of the columns obtained in phase 4. Since the equality in each column was propagated in phase 5, any element of the column is mapped via *find* to a unique representative that is either a constant, a fresh variable, an extraction on a fresh variable or an OBDD.

If any extractions on fresh variables occur, they were generated in phase 4 during the computation of the coarsest slicing. If a clean output with no redundant extractions is desired, all of them can be easily replaced by fresh variables of appropriate width. In any case, the set of equations put out is a solved form according to section 2.2.3.

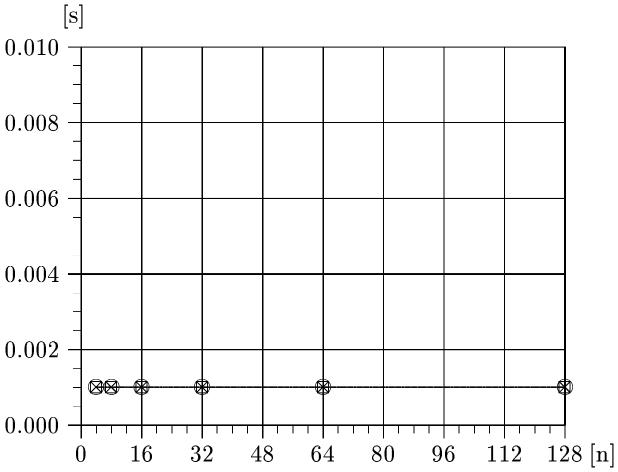
4.3.8 Run Time Experiments

The fixed solver described above was implemented in Allegro Common Lisp. A source code is included in Appendix C.2. This implementation has been applied to all the examples described in section 4.1.6. The run-time measured on a Sparc Ultra 1 contains both canonization and solving. The canonization time is marked by a "⊠" and the overall computation time by "⊗".

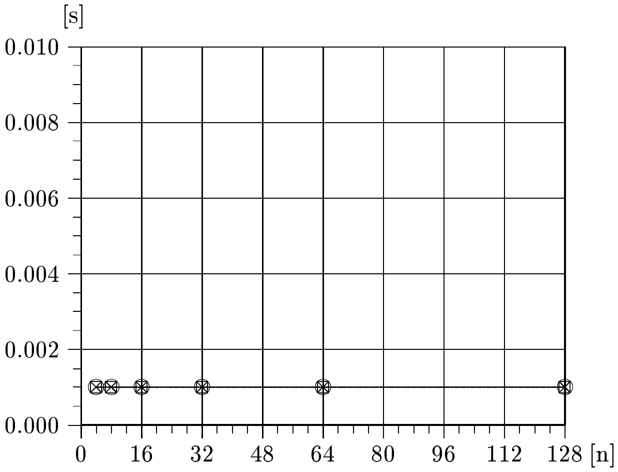
To put it in a nutshell, the fixed solver shows a good performance with concatenations and extractions and a moderately good behavior, if complex boolean operations or arithmetic is involved. It adapts better than the reduced chopper to cases where iterated slicing has to be applied, as demonstrated in example C3. If complicated OBDD information is processed, the algorithm soon reaches it's limits as seen in C5 and C6. This is explained by the expensive propagation of equality, if OBDDs are involved excessively. The computation on width 8 was aborted after more than 20 hours. As described in the next section, the performance in these cases can be improved notably by means of introducing heuristics.



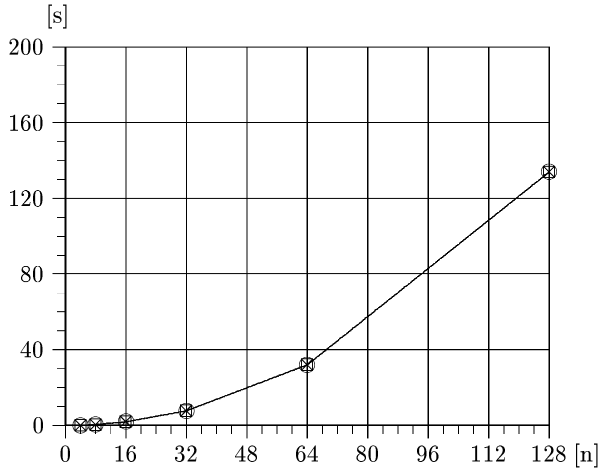
A1: $c_{[n]} \stackrel{!}{=} c_{[n]}$



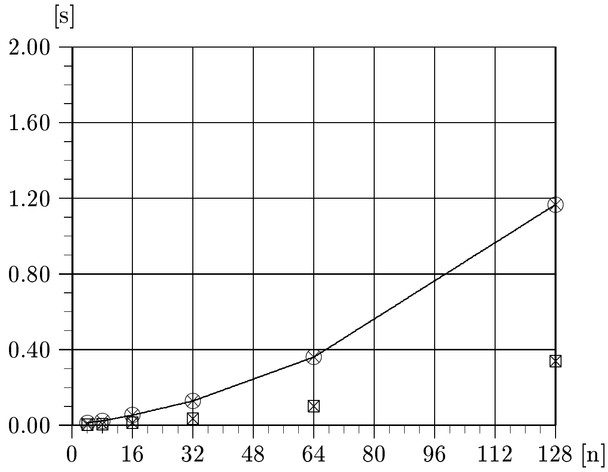
A2: $x_{[n]} \stackrel{!}{=} x_{[n]}$



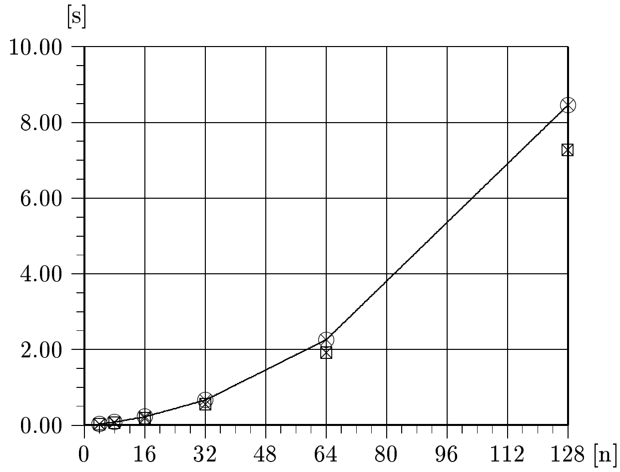
A3: $x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]} \stackrel{!}{=} x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]}$



A4: $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} y_{[n]} +_{[n]} x_{[n]}$

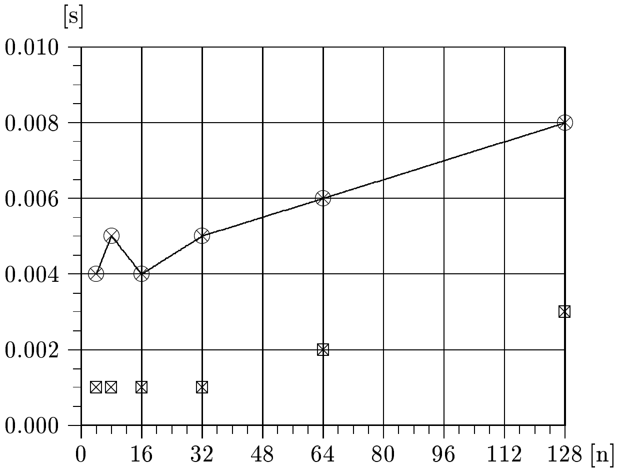


A5: $x_{[n]} +_{[n]} 0_{[n]} \stackrel{!}{=} x_{[n]}$

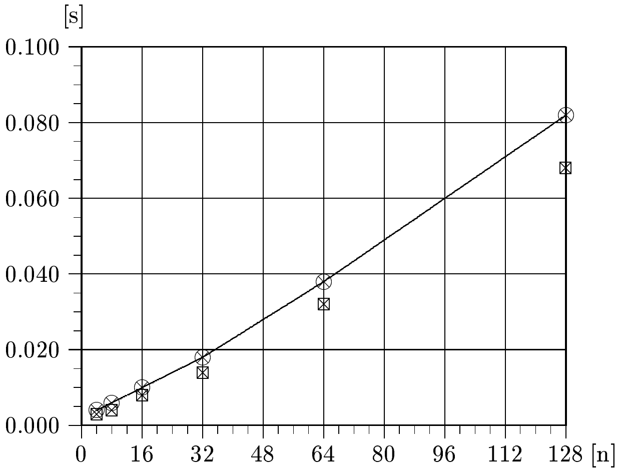


A6: $x_{[n]} +_{[n]} x_{[n]} \stackrel{!}{=} 0_{[1]} \otimes x_{[n]}[0 : n\text{-}1]$

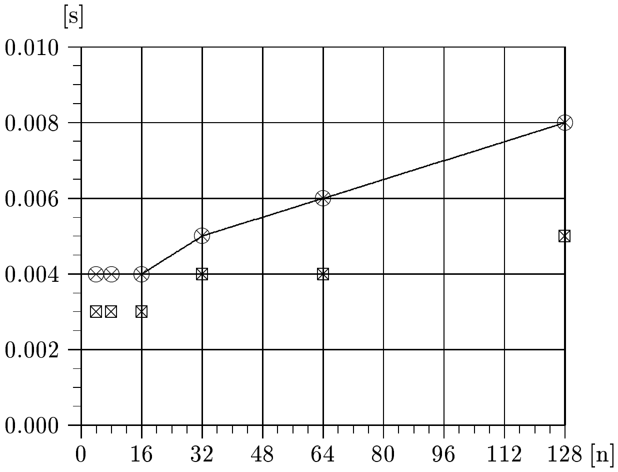
Table 4.6: Checking Tautologies via Fixed Solver



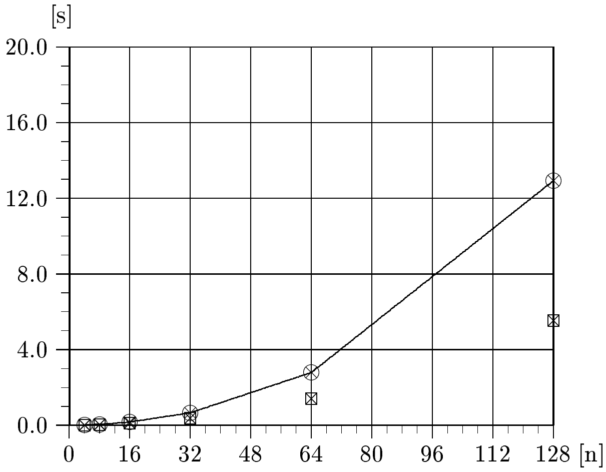
B1: $x_{[\frac{n}{2}]} \otimes x_{[\frac{n}{2}]} \stackrel{!}{=} 1_{[n]}$



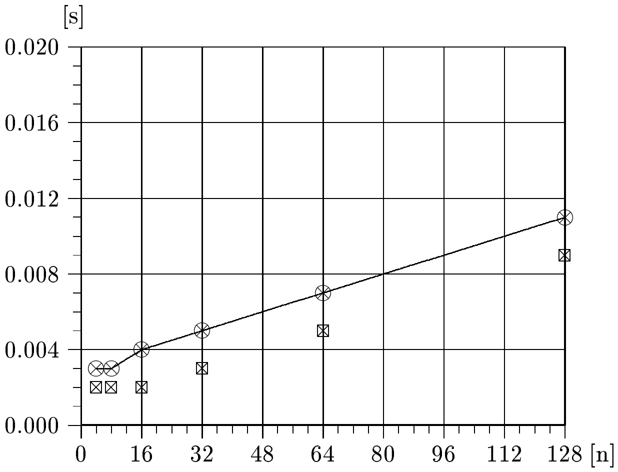
B4: $c_{[n]} +_{[n]} c'_{[n]} \stackrel{!}{=} 0_{[n]}$



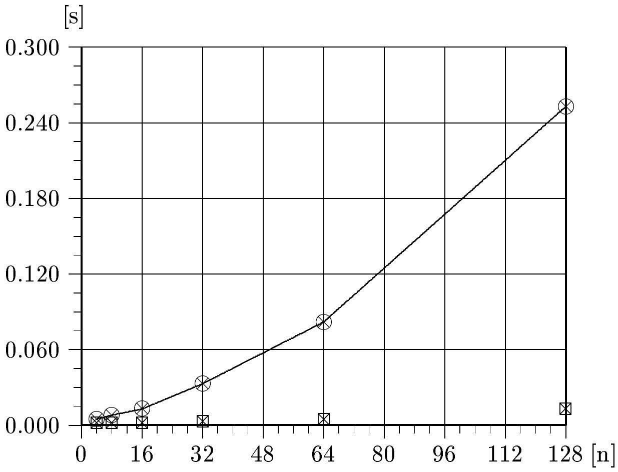
B2: $x_{[n]} \wedge \neg x_{[n]} \stackrel{!}{=} 1_{[n]}$



B5: $x_{[n]} +_{[n]} 1_{[n]} \stackrel{!}{=} x_{[n]}$

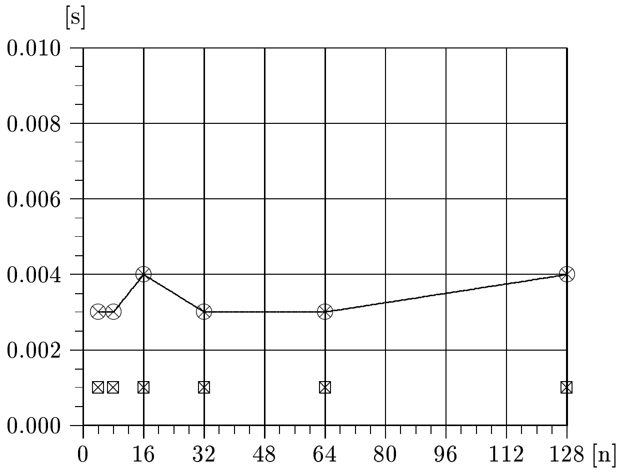


B3: $x_{[n]} \vee 1_{[n]} \stackrel{!}{=} 0_{[n]}$

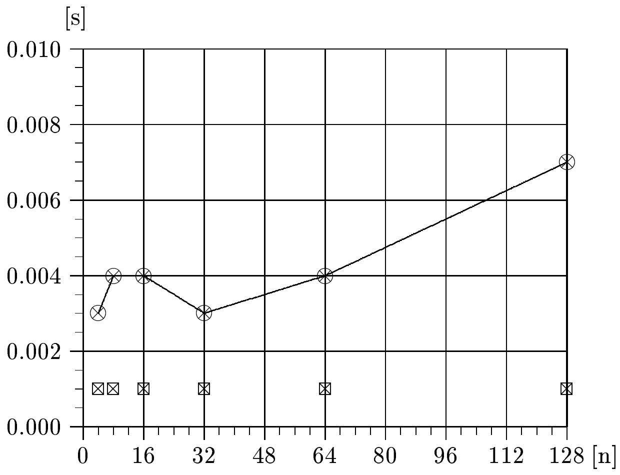


B6: $x_{[n]} \wedge (\mathbf{0} \otimes x_{[n]}[0:n\text{-}2]) \stackrel{!}{=} (\mathbf{1} \otimes \mathbf{0})^{\frac{n}{2}}$

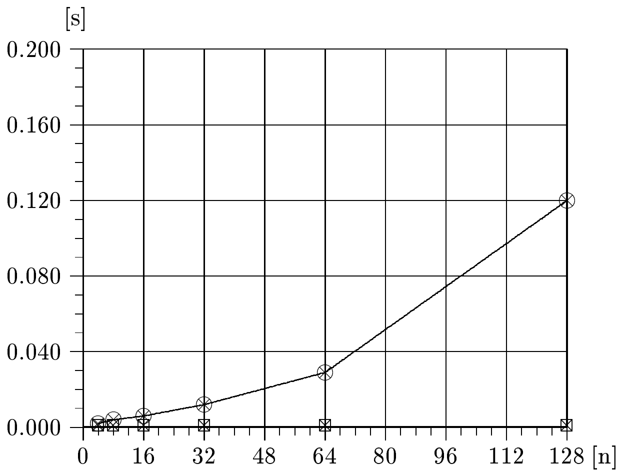
Table 4.7: Checking Unsatisfiability via Fixed Solver



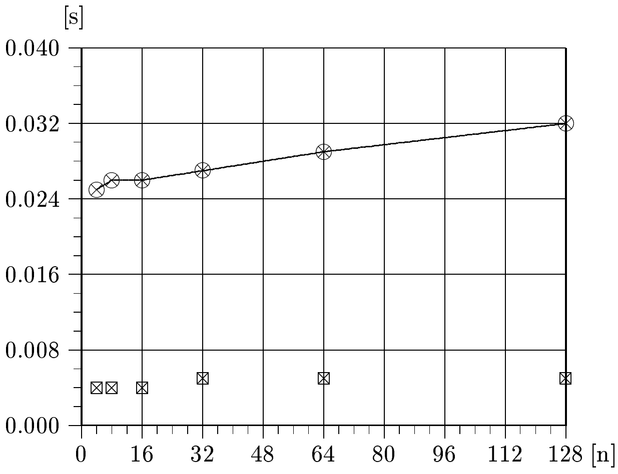
C1: $x_{[2n]}[n:2n\text{-}1] \stackrel{!}{=} x_{[2n]}[0:n\text{-}1]$



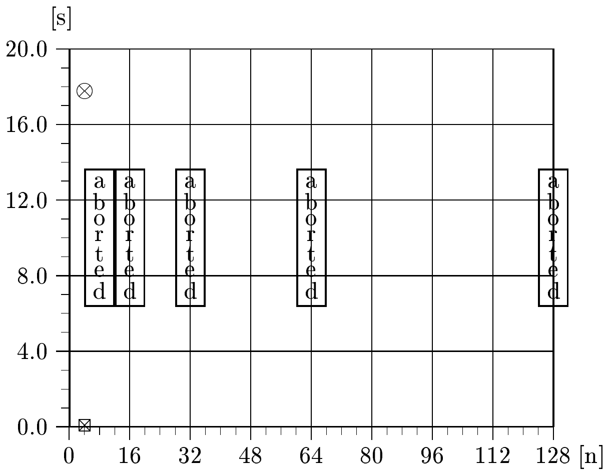
C2: $x_{[\frac{n}{2}]} \otimes y_{[\frac{n}{2}]} \stackrel{!}{=} y_{[\frac{n}{2}]} \otimes z_{[\frac{n}{2}]}$



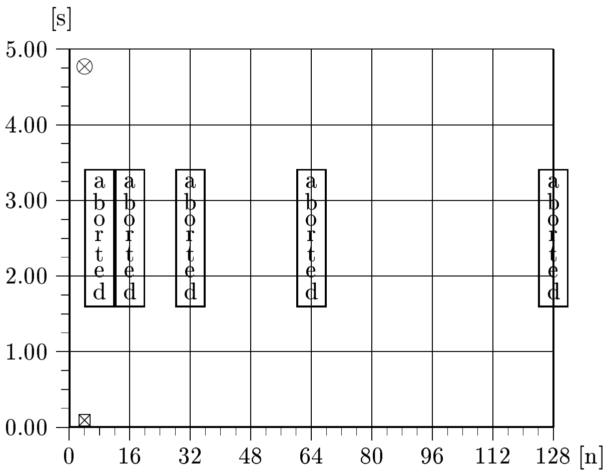
C3: $0_{[2]} \otimes x_{[n\text{-}2]} \stackrel{!}{=} x_{[n\text{-}2]} \otimes 0_{[2]}$



C4: $x_{[n]} \wedge y_{[n]} \stackrel{!}{=} z_{[n]}\, XOR\, (\neg x_{[n]})$



C5: $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} 0_{[n]}$



C6: $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} x_{[n]}\, XOR\, y_{[n]}$

Table 4.8: Satisfiable Equations via Fixed Solver

4.4 Introducing Heuristics for the Fixed Solver

The propagation of equality with OBDDs is an expensive task, for each *merge* operation might result in a large number of replacements. This is the case in the examples C5 and C6 (page 59). In this section, heuristics to optimize the propagation for the fixed solver are discussed. Sometimes it is advantageous to process informations contained in OBDDs earlier. Since valid equations are detected right after canonization, the advantages applies only to unsatisfiable and non-valid formulae. The idea of the heuristic is explained using pigeon hole formulae.

4.4.1 The Pigeon Hole Principle

This classic scenario explains as follows. Consider a number n of pigeon holes and $n+1$ pigeons. Now, it is not feeding time and therefor every pigeon is supposed to be in a hole. Since there are not enough holes for all the pigeons, at least one hole is occupied by more than one pigeon. This can be modeled in boolean logic, e.g. via introducing $(n+1)\cdot n$ propositional variables m_{ij}, $i=1,\ldots,n+1$, $j=1,\ldots,n$, which are interpreted as

$$m_{ij} = \mathbf{true} \qquad \text{iff} \qquad \text{Pigeon } \#i \text{ is in Hole } \#j, \qquad i=1,\ldots,n+1, \quad j=1,\ldots,n$$

On this basis, three boolean formulae are defined, each expressing one part of the scenario:

$$\begin{array}{lcl} \Phi_1 & : & \text{Every pigeon is in at least one hole.} \\ \Phi_2 & : & \text{In no hole there is more than one pigeon.} \\ \Phi_3 & : & \text{No pigeon is in more than one hole.} \end{array}$$

If the number of holes $n=2$, these formulae present as

$$\begin{array}{lcl} \Phi_1 & = & (m_{11} \vee m_{12}) \wedge (m_{21} \vee m_{22}) \wedge (m_{31} \vee m_{32}) \\ \Phi_2 & = & (m_{11} \rightarrow \neg m_{21} \wedge \neg m_{31}) \wedge (m_{21} \rightarrow \neg m_{11} \wedge \neg m_{31}) \wedge (m_{31} \rightarrow \neg m_{11} \wedge \neg m_{21}) \wedge \\ & & (m_{12} \rightarrow \neg m_{22} \wedge \neg m_{32}) \wedge (m_{22} \rightarrow \neg m_{12} \wedge \neg m_{32}) \wedge (m_{32} \rightarrow \neg m_{12} \wedge \neg m_{22}) \\ \Phi_3 & = & \neg(m_{11} \wedge m_{12}) \wedge \neg(m_{21} \wedge m_{22}) \wedge \neg(m_{31} \wedge m_{32}) \end{array}$$

It is obvious that the formula $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$ cannot be satisfied. Moreover, formula Φ_3 can be neglected, since even if a pigeon manages to be in more than one hole at once, $\Phi_1 \wedge \Phi_2$ is necessarily false.
In spite of the clear intuition, this example is a challenge for mechanical proof systems. In 1985 Haken showed, that there is an exponential lower bound in n for the number of steps any proof of unsatisfiability needs, when using resolution as decision procedure (cf. [Hak85]).

4.4.2 Expressing Pigeon Hole in the Bit-Vector Theory

For each boolean variable m_{ij}, a corresponding bit-vector $m_{ij[1]}$ is introduced. Then the formulae Φ_1, Φ_2 and Φ_3 can be built as OBDDs by means of applying the boolean connectives on bit-vector terms of width one. In the following, Φ_1, Φ_2 and Φ_3 are assumed to be bit-vector OBDDs. Now there are several slightly different ways to express the pigeon hole principle:

$$\begin{array}{rrcl} (I) & \Phi_1 \otimes \Phi_2 \otimes \Phi_3 & \stackrel{!}{=} & \mathbf{1}_{[1]} \otimes \mathbf{1}_{[1]} \otimes \mathbf{1}_{[1]} \\ (II) & \Phi_1 \otimes \Phi_2 & \stackrel{!}{=} & \mathbf{1}_{[1]} \otimes \mathbf{1}_{[1]} \\ (III) & \Phi_1 \wedge \Phi_2 \wedge \Phi_3 & \stackrel{!}{=} & \mathbf{1}_{[1]} \\ (IV) & \Phi_2 \wedge \Phi_2 & \stackrel{!}{=} & \mathbf{1}_{[1]} \end{array}$$

In any case, the solver is expected to return **false**. The run-time of the fixed solver to show unsatisfiability depends heavily on the kind of formalization, as shown in Table 4.9. The y-axis is scaled in a logarithmic manner, the bottom line "0" is to read as "below one second".

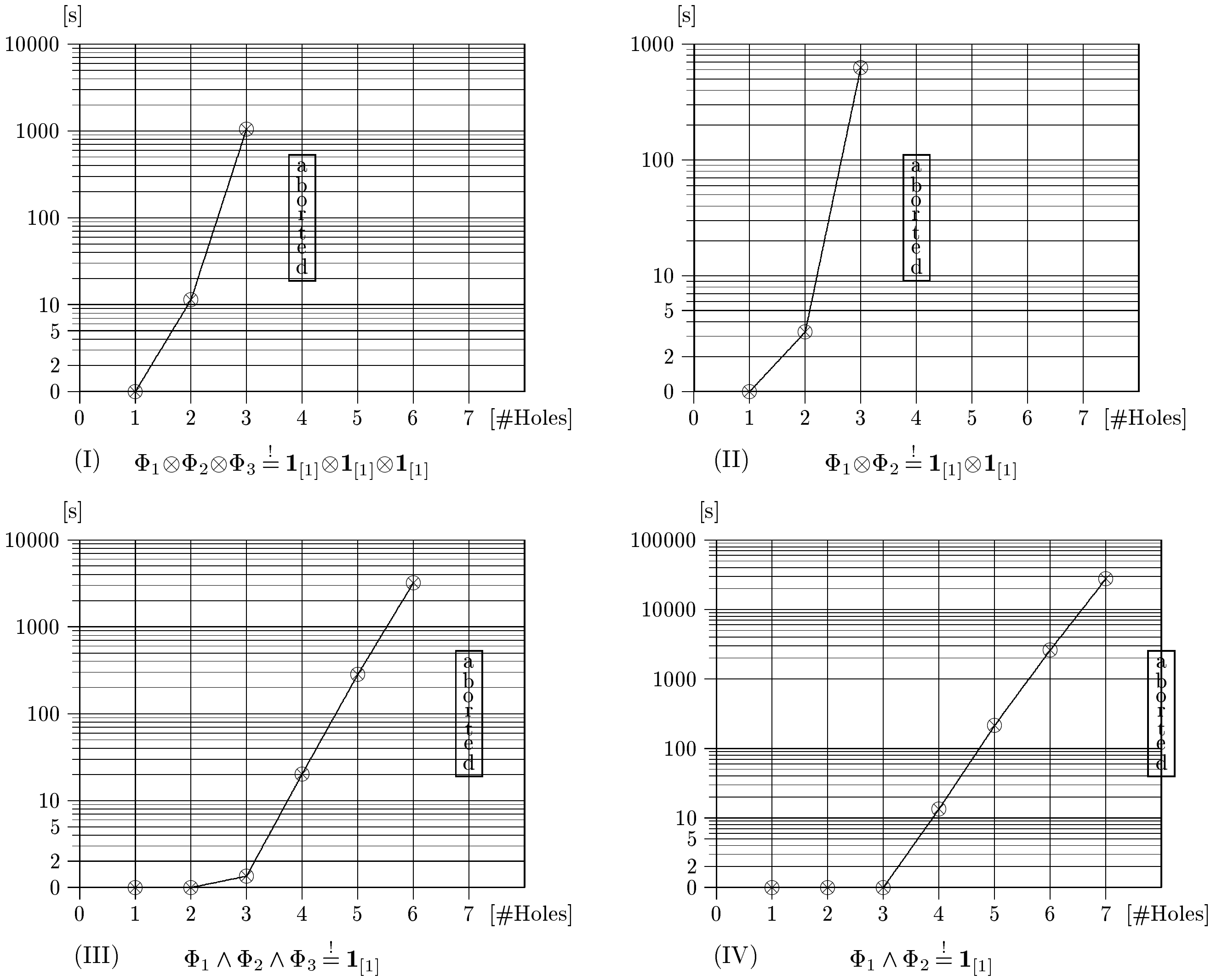


Table 4.9: The Pigeon Hole Principle via the Fixed Solver [without Heuristic]

The performance of the fixed solver gets better from (I) to (IV) with a noticeable gap between (I)/(II) and (III)/(IV). This is due to the fact that since the formulae Φ_1 and Φ_2 are both satisfiable, in (I) and (II) the unsatisfiability of the overall equation is detected no sooner than in phase 5. In examples (III) and (IV), the OBDD alone canonizes to **false**, thus the algorithm aborts already after the chunk-solving in phase 2. The information required to show unsatisfiability is the same in all cases. It is desirable to bring it in an advantageous form.

4.4.3 The Idea: OBDD Melting

While processing example (I), at the beginning of phase 2 there are three calls to *chunk-solve*: $\Phi_1 \stackrel{!}{=} \mathbf{1}_{[1]}$, $\Phi_2 \stackrel{!}{=} \mathbf{1}_{[1]}$ and $\Phi_3 \stackrel{!}{=} \mathbf{1}_{[1]}$. Each call yields a set of equations for all occurring original variables $m_{ij_{[1]}}$, where the right hand sides are OBDDs over fresh variables. This is not efficient, for the equality of three terms on the right hand side for each $m_{ij_{[1]}}$ has to be propagated. It seems to be more reasonable first to *melt* the

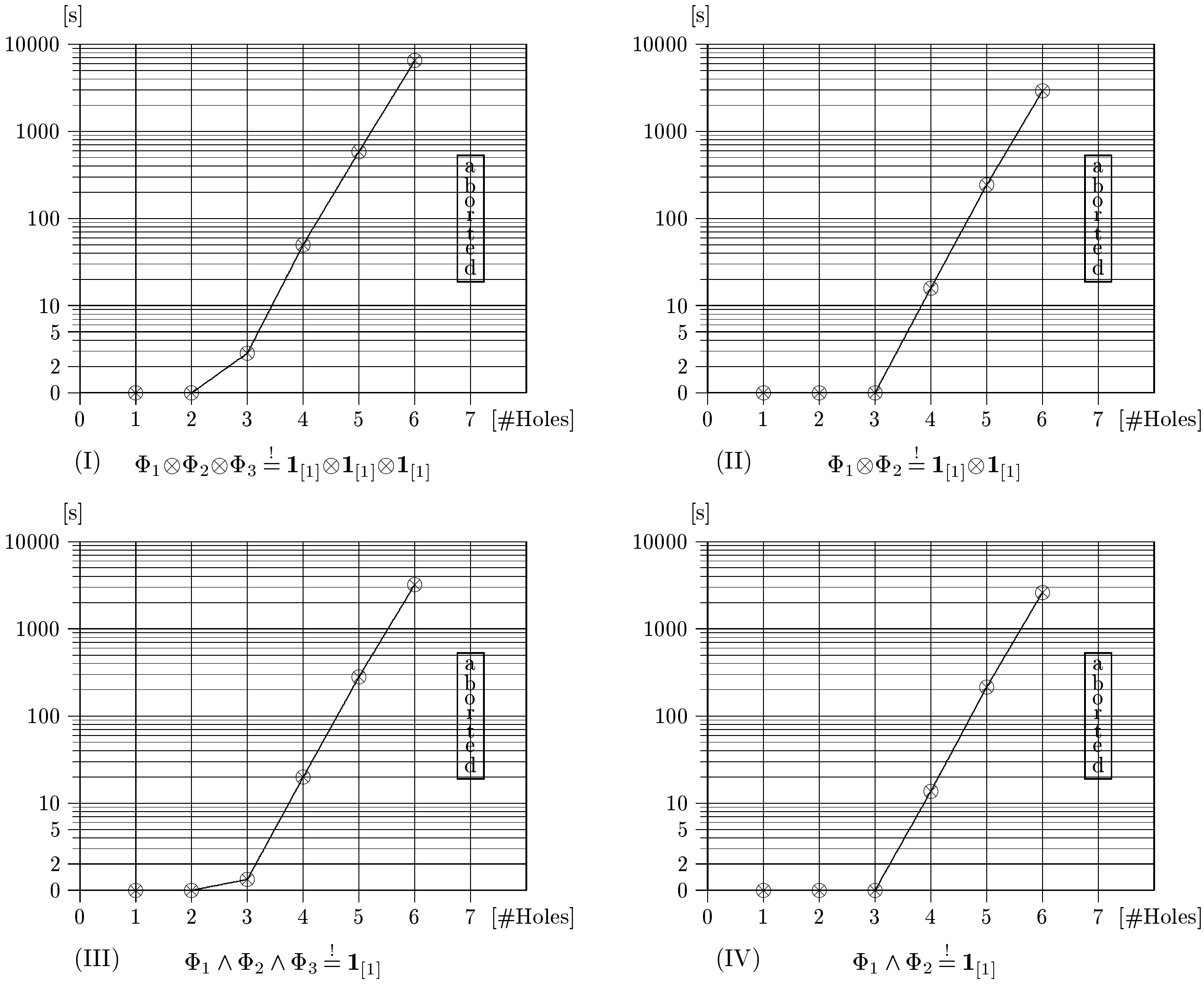


Table 4.10: The Pigeon Hole Principle via the Fixed Solver [with Heuristic 1]

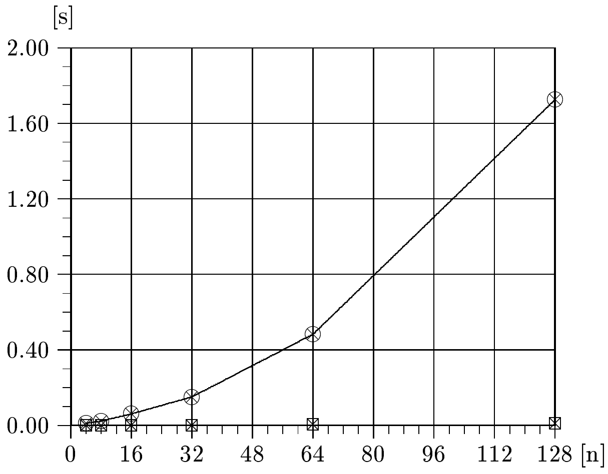
OBDDs by means of processing a conjunction instead of separate formulae:

$$\Phi_1 \overset{!}{=} \mathbf{1}_{[1]},\ \Phi_2 \overset{!}{=} \mathbf{1}_{[1]},\ \Phi_3 \overset{!}{=} \mathbf{1}_{[1]} \qquad \overset{melt}{\longrightarrow} \qquad \big((\Phi_1 \equiv \mathbf{1}_{[1]}) \wedge (\Phi_2 \equiv \mathbf{1}_{[1]}) \wedge (\Phi_3 \equiv \mathbf{1}_{[1]})\big) \overset{!}{=} \mathbf{1}_{[1]}$$

Now *chunk-solve* is called only once and returns **false**. For satisfiable equations, there would be but one right hand side for each variable instead of three. This leads to

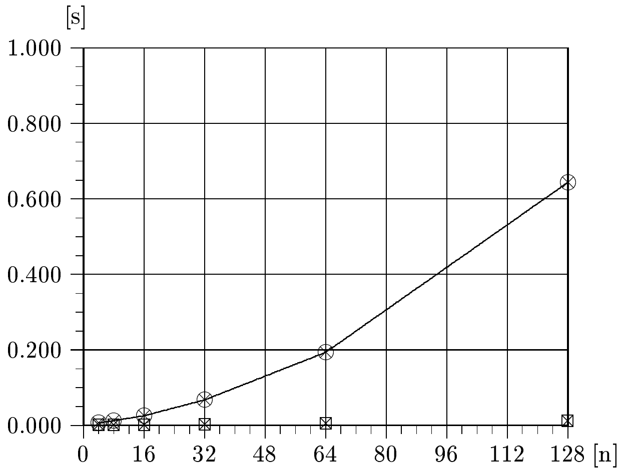
Heuristic 1: Before phase 2, melt any OBDD equations that have at least one node mark in common.

In case of (I) and (II), all three respectively two OBDDs are melted. This leads to a far better performance as displayed in Table 4.10.



B6: $x_{[n]} \wedge (\mathbf{0} \otimes x_{[n]}[0:n\text{-}2]) \stackrel{!}{=} (\mathbf{1} \otimes \mathbf{0})^{\frac{n}{2}}$

[Heuristic 1]



B6: $x_{[n]} \wedge (\mathbf{0} \otimes x_{[n]}[0:n\text{-}2]) \stackrel{!}{=} (\mathbf{1} \otimes \mathbf{0})^{\frac{n}{2}}$

[Heuristic 2]

Table 4.11: Example B6, Processed With Two Different Heuristics

4.4.4 Refinement of the Heuristic

It is a mistake to assume that Heuristic 1 is favorable in any case. Consider the previously processed example B6: $x_{[n]} \wedge (\mathbf{0}_{[1]} \otimes x_{[n]}[0:n\text{-}2]) \stackrel{!}{=} (\mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]})^{\frac{n}{2}}$ (cf. Table 4.7). This equation is unsatisfiable due to the least significant position. The call to *chunk-solve* with $x_{[n]}[0:0] \wedge \mathbf{0}_{[1]} \stackrel{!}{=} \mathbf{1}_{[1]}$ results in **false**. If Heuristic 1 is applied, all n OBDD nodes are melted, before this **false** is detected. As seen in Table 4.11, this yields a noticeable slowdown (the fixed solver without heuristics computed the longest example within 0.25 seconds).

This example suggests to restrict the melting of OBDDs to cases where the occurrence of node variables overlaps to a stronger degree:

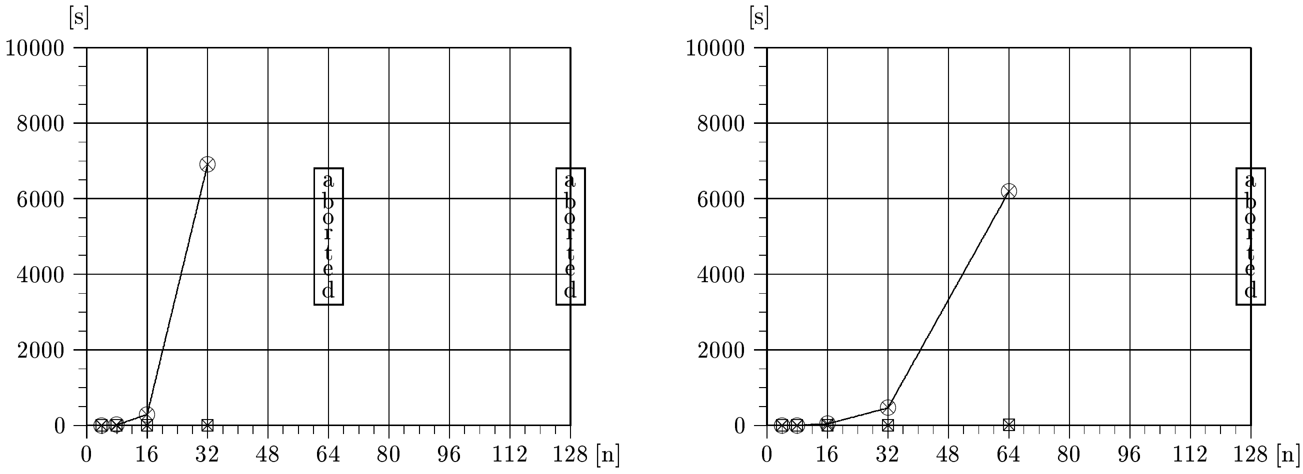
Heuristic 2: Before phase 2, melt any OBDD equations that have all node marks in common.

In example B6 this leads to a better performance, as observed in Table 4.11. It is slower than the computation without heuristics, because any two of the n OBDD equations have to be checked regarding to the melting condition of Heuristic 2.

Neither of the two extreme heuristics show an overall good performance. In particular, the examples C5 and C6 are not processed more efficiently. And, in both cases, there are examples where the fixed solver performs noticeable worse than without melting at all. It is reasonable to expect a good heuristic for melting somewhere in the middle. The best one I found is the following:

Heuristic 3: Before phase 2, melt any two OBDD equations $OBDD_1$ and $OBDD_2$, if
$$|\mathit{vars}(OBDD_1)| + |\mathit{vars}(OBDD_2)| \ \leq\ 2.4 \cdot |\mathit{vars}(OBDD_1) \cap \mathit{vars}(OBDD_2)|$$

In Heuristic 2, the factor would be 2 instead of 2.4 and in Heuristic 1 it would be ∞, where $\infty \cdot 0 := 0$. It was narrowed down while using factors 4, 2.5, 2.4, 2.3 and 2.2.
With high probability, for *any* factor there exist examples where the behavior is sub-optimal. However, 2.4 resulted for the examples C5 and C6 in a not overwhelming but noticeable speed-up, as displayed in Table 4.12. At the highest width processed, Heuristic 3 melted 32 equations to 3 OBDDs in example C5 (respectively 64 equations to 4 OBDDs in example C6).



C5: $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} 0_{[n]}$ **C6:** $x_{[n]} +_{[n]} y_{[n]} \stackrel{!}{=} x_{[n]}\, XOR\, y_{[n]}$

Table 4.12: Satisfiable but not Valid Examples [with Heuristic 3]

A Short Summary on Heuristics

The usage of heuristics together with the fixed solver seems to be crucial in order to avoid a break-down in many pathological cases. Although it is not likely that an overall optimal heuristic for melting exists, the processed examples suggest to apply rather a mediocre one than none at all. The price might be a slow-down in some simple examples, but this is not a high price if the alternative is a gambling whether the system will ever respond again.

There might be other places where it is reasonable to apply heuristics. The order of *chunk-solving* and propagation are promising points, for both offer the chance to detect unsatisfiability early. However, these considerations were not followed in this thesis.

4.5 Looking Back at Fixed Size

Though any of the followed approaches has its justification, there was always a pathologic example exceeding a reasonable respond time. Though most of these unwanted behaviors could be patched somehow, there does not seem to be a *conceptually clean way* to gain satisfying efficiency. Having spent quite some time in consideration of alternatives in the past two year, the author feels inclined to summarize this in the following estimation:

> *It is not likely, that there is a simple concept that provides an efficient solver for the theory of bit-vectors, even for fixed size. In order to match the needs of industrially sized applications, sophisticated strategies and detection of special cases are required.*

Chapter 5

Beyond Fixed Size

The only way to discover the limits of the possible is to go beyond them into the impossible.
(Arthur C. Clarke, Technology and the Future)

5.1 A Solver for Variable Width: Split-Chop

This section defines an extension of the reduced chopper algorithm from section 4.2.2 for solving bit-vector equations of variable width. In order to cope with the non-convex properties, a concept of context splits in connection with reasoning about integer terms is defined.

5.1.1 Reasoning about Integers

The Definition 2.13 of the frame solver implies that reasoning on integer constraints is required. The following example shows that it is not sufficient to express equalities (like $width(t) = width(u)$) and inequalities (like integer $1 \leq l$).

Example 5.1

$$\begin{array}{ccc} x_{[l]} & \otimes & \mathbf{1}_{[1]}\otimes\mathbf{0}_{[1]} \stackrel{!}{=} \\ \mathbf{1}_{[1]}\otimes\mathbf{0}_{[1]} & \otimes & x_{[l]} \end{array}$$

Here, l is an integer variable. It is easy to see that this equation is satisfiable, if and only if l is even. This can be expressed by the constraint $\frac{1}{2} \cdot l \in \mathbb{Z}$.

Definition 5.1 [**Closed Sets of Integer Constraints**]

Let $L := \{l_1, \ldots, l_k\}$ denote the set of integer variables. The set of integer numbers is denoted by $\mathbb{Z}$. An *integer term* χ is a term that can be constructed from L, rational numbers, addition, multiplication by a rational number, operations *DIV*, *MOD* and the application of the operations *max* and *min* on an arbitrary number of integer terms. Also, there exist symbols -∞ and ∞ which are defined as -$\infty := max\,\emptyset$, $\infty := min\,\emptyset$. A *set* Ψ *of integer constraints* is a quadruple $(\mathfrak{Leq}, \mathfrak{Bet}, \mathfrak{Map}, \mathfrak{Int})$, where

- $\mathfrak{Leq}$ is a set of inequalities on integer terms,
- $\mathfrak{Bet}$ is a collection of inequalities of the form $\chi_i^- \leq l_i \leq \chi_i^+$,
- $\mathfrak{Map}$ is a set of tuples $(l_i \mapsto \chi_i)$; none of the l_i in $\mathfrak{Map}$ occurs in $\mathfrak{Leq}$ or $\mathfrak{Int}$,
- $\mathfrak{Int}$ is a collection of integer terms; it is understood that each term is required to evaluate to an integer.

Ψ is called *closed*, if

- No redundant information is contained in $\mathfrak{Leq}$, $\mathfrak{Bet}$ or $\mathfrak{Int}$; e.g. the inequality $1 \le l$ is subsumed by the inequality $2 \le l$ and therefore $1 \le l$ should be omitted,
- For every variable $l_i \in L$, there exists exactly one inequality $\chi_i^- \le l_i \le \chi_i^+ \in \mathfrak{Bet}$,
- Ψ is satisfiable, i.e. there exists an interpretation $\mathcal{I} : L \to \mathbb{Z}$ such that $\mathcal{I} \models \forall (l_i \mapsto \chi_i) \in \mathfrak{Map} : l_i = \chi_i$ and $\mathcal{I} \models \mathfrak{Leq} \wedge \mathfrak{Bet} \wedge \bigwedge_{\chi \in \mathfrak{Int}} \chi \in \mathbb{Z}$.

$\lrcorner$

The shape of Ψ is motivated by a recursive function *int-close* computing an equivalent but closed set for a given Ψ. The idea is similar to the Fourier-Motzkin elimination method (cf. [Sch86, TH]).

int-close$(\mathfrak{Leq}^*, \mathfrak{Int})$
  IF $vars(\mathfrak{Leq}^*) = \emptyset$
  THEN IF $\mathfrak{Leq}^* \models \bot$ THEN RETURN **false**
      ELSE RETURN $(\emptyset, \emptyset, \emptyset, \emptyset)$
    ENDIF
  ELSE LET $l \in vars(\mathfrak{Leq}^*)$
    IN $Neg := \{\chi \mid \text{there is an } ineq \in \mathfrak{Leq}^* \text{ with } ineq \Leftrightarrow \chi \le l\}$
      $Pos := \{\chi \mid \text{there is an } ineq \in \mathfrak{Leq}^* \text{ with } ineq \Leftrightarrow l \le \chi\}$
      $(\mathfrak{Leq}, \mathfrak{Bet}, \mathfrak{Map}, \mathfrak{Int}) := \textit{int-close}\left(\begin{array}{l}(\mathfrak{Leq}^* \setminus \{ineq \mid l \in ineq\}) \cup \\ \{neg \le pos \mid neg \in Neg,\ pos \in Pos\}\end{array}, \mathfrak{Int}\right)$
      $\mathfrak{Int} \leftarrow \mathfrak{Int} \cup \{l\}$
      $\mathfrak{Bet} \leftarrow \mathfrak{Bet} \cup \{max(Neg) \le l \le min(Pos)\}$
      WHILE $\exists \beta \in \mathfrak{Bet} : (\mathfrak{Leq} \wedge \beta \wedge \bigwedge_{\chi \in \mathfrak{Int}} \chi \in \mathbb{Z}) \Rightarrow l = \chi_l$ DO
        REPLACE l IN $\mathfrak{Leq}$, $\mathfrak{Int}$ BY χ_l
        $\mathfrak{Map} \leftarrow \mathfrak{Map} \cup \{(l \mapsto \chi_l)\}$
      OD
      IF $\mathfrak{Leq} \wedge \mathfrak{Bet} \wedge \bigwedge_{\chi \in \mathfrak{Int}} \chi \in \mathbb{Z} \models \bot$ THEN RETURN **false**
        ELSE RETURN $(\mathfrak{Leq}, \mathfrak{Bet}, \mathfrak{Map}, \mathfrak{Int})$
      ENDIF
  ENDIF

5.1.2 Splitting Context: The Solver Split-Chop

The solver for variable width presented now—in the following referred to as *split-chop* or short $\mathcal{S}_{\Re}$—is described via a straight-forward generalization of an equational transformation system (cf. Definition 4.4). With each application of an equational transformation rule, not only the set of bit-vector equations Υ but also the integer constraints Ψ are modified. A pair (Υ, Ψ) that is not terminal (with respect to $\mathcal{S}_{\Re}$) is referred to as a *context*. If one of the rules (1), (1)* matches an equation in Υ, the respective context is split to several cases. Each case yields a new context that is processed by the split-chop algorithm.
A split-chop computation starting with $t_1 = t_2$ is sketched as follows.

$$
\begin{array}{llll}
 & & \textsc{Cases} & \\
 & & n < m & \rightarrowtail \left\{ \begin{array}{l} p_{[n]} = \alpha(q_{[m]}[0:n-1]) \\ \alpha(q_{[m]}[n:m-1]) \otimes u = t \\ q_{[m]} = q_{[m]}[0:n-1] \otimes q_{[m]}[n:m-1] \end{array} \right\} \\
(1) & p_{[n]} \otimes t = q_{[m]} \otimes u, & n = m & \rightarrowtail \left\{ \begin{array}{l} p_{[n]} = q_{[m]} \\ t = u \end{array} \right\} \\
 & & n > m & \rightarrowtail \left\{ \begin{array}{l} q_{[m]} = \alpha(p_{[n]}[0:m-1]) \\ \alpha(p_{[n]}[m:n-1]) \otimes t = u \\ p_{[n]} = p_{[n]}[0:m-1] \otimes p_{[n]}[m:n-1] \end{array} \right\} \\
 & & \textsc{Endcases} & \\
(1)^* & x_{[n]}[i:j] = x_{[n]}[k:l], & \begin{array}{l} j\text{-}i = l\text{-}k, \\ i < k \end{array} & \rightarrowtail \left[\begin{array}{l} \left(\begin{array}{l} \left\{ x_{[n]}[i:l] = {a_{[k\text{-}i]}}^{\frac{l-i+1}{k-i}} \right\}, \\ \frac{1}{k\text{-}i} \cdot (l-i+1) \in \mathfrak{Int} \end{array} \right) \\ \left(\begin{array}{l} \left\{ \begin{array}{l} x_{[n]}[i:l] = a_{[(l\text{-}i+1) MOD (k\text{-}i)]} \otimes \\ (b_{[(i\text{-}l\text{-}1) MOD (k\text{-}i)]} \otimes a_{[(l\text{-}i+1) MOD (k\text{-}i)]})^{(l\text{-}i+1) DIV (k\text{-}i)} \end{array} \right\}, \\ \big((l-i+1)\ MOD\ (k-i)\big) > 0 \end{array} \right) \end{array} \right] \\
(2) & c_{[n]} = {c'}_{[n]} \quad , & c \neq c' & \rightarrowtail \textbf{FAIL} \\
(3) & t = t & & \rightarrowtail \{\} \\
(4) & \left\{ \begin{array}{l} p = t \\ q = u \end{array} \right\}, & q \subseteq t & \rightarrowtail \left\{ \begin{array}{l} p = t[q/u] \\ q = u \end{array} \right\} \\
(5) & \left\{ \begin{array}{l} p = q \\ q = r \end{array} \right\} & & \rightarrowtail \left\{ \begin{array}{l} p = a \\ q = a \\ r = a \end{array} \right\} \\
(6) & \left\{ \begin{array}{l} p = q \\ q = p \end{array} \right\} & & \rightarrowtail \left\{ \begin{array}{l} p = a \\ q = a \end{array} \right\} \\
(7) & \left\{ \begin{array}{l} p = t \\ p = u \end{array} \right\} & & \rightarrowtail \left\{ \begin{array}{l} p = t \\ u = t \end{array} \right\} \\
(8) & c = t, & t \neq const & \rightarrowtail \{\ t = c\ \}
\end{array}
$$

Figure 5.1: The Equational Transformation System $\mathcal{S}_{\mathfrak{R}}$ for Variable Width

1. Define a set $\Upsilon := \{t_1 = t_2\} \cup$ *init-CS*$(t_1 = t_2)$ according to Definition 4.5.
2. Compute a set Ψ of integer constraints based on the term structure of $t_1 = t_2$.
3. Compute the closure of Ψ via *int-close*; if it fails, abort this context with **false**.
4. Start the equational transformation system $\mathcal{S}_{\mathfrak{R}}$ in Figure 5.1 on the context (Υ, Ψ).
5. If no rule is applicable, the computation in this branch terminates with the frame (Υ, Ψ) – see Definition 2.13.
6. If rule (1) or rule $(1)^*$ encounters ambiguities due to the variable term width, perform a case split and generate pairs $(\Upsilon_1, \Psi_1), \ldots, (\Upsilon_k, \Psi_k)$.
7. With each of the pairs $(\Upsilon_\varkappa, \Psi_\varkappa)$, $\varkappa = 1, \ldots, k$, continue the computation at point 3.

Example 5.2 [cf. Example 2.6]
Consider the equation

$$x_{[n]} \otimes 0_{[1]} \otimes y_{[m]} \stackrel{!}{=} z_{[2]} \otimes 1_{[1]} \otimes w_{[2]}$$

where n and m are integer variables. Set $\Psi := (\{n+1+m \leq 5,\ 5 \leq n+1+m\}, \emptyset, \emptyset, \emptyset)$. The procedure *int-close* results in $\Psi = (\emptyset, \{1 \leq n \leq 3\}, \{(m \mapsto 4-n)\}, \{n, 4-n\})$. Next, the equational transformation rule (1) tries to match the leftmost chunks $x_{[n]}$ and $z_{[2]}$. This leads to the case-split

(a) $n < 2 \quad \overset{\text{int-close}}{\rightsquigarrow} \quad n \mapsto 1; m \mapsto 3$

(b) $n = 2 \quad \overset{\text{int-close}}{\rightsquigarrow} \quad n \mapsto 2; m \mapsto 2 \quad \overset{\mathcal{S}_{\mathfrak{R}}}{\rightsquigarrow} \quad$ **false**

(c) $n > 2 \quad \overset{\text{int-close}}{\rightsquigarrow} \quad n \mapsto 3; m \mapsto 1$

The split-chop algorithm (cf. Table 5.1) in combination with *int-close* terminates with the two frames

$$\left(\left\{\begin{array}{lcl} x_{[n]} & = & a_{[1]}, \\ y_{[m]} & = & 1_{[1]} \otimes b_{[2]}, \\ z_{[2]} & = & a_{[1]} \otimes 0_{[1]}, \\ w_{[2]} & = & b_{[2]} \end{array}\right\}, \left(\emptyset, \left\{\begin{array}{lcccl} 1 & \leq & n & \leq & 4, \\ 1 & \leq & m & \leq & 4 \end{array}\right\}, \left\{\begin{array}{lcl} n & \mapsto & 1, \\ m & \mapsto & 3 \end{array}\right\}, \left\{\begin{array}{l} 1, \\ 3 \end{array}\right\}\right)\right),$$

$$\left(\left\{\begin{array}{lcl} x_{[n]} & = & a_{[2]} \otimes 1_{[1]}, \\ y_{[m]} & = & b_{[1]}, \\ z_{[2]} & = & a_{[2]}, \\ w_{[2]} & = & 0_{[1]} \otimes b_{[1]} \end{array}\right\}, \left(\emptyset, \left\{\begin{array}{lcccl} 1 & \leq & n & \leq & 4, \\ 1 & \leq & m & \leq & 4 \end{array}\right\}, \left\{\begin{array}{lcl} n & \mapsto & 3, \\ m & \mapsto & 1 \end{array}\right\}, \left\{\begin{array}{l} 3, \\ 1 \end{array}\right\}\right)\right)$$

Thus, all possible solutions are covered.

5.1.3 The Context Split Rule

In order to explain the context split applied in rule $(1)^*$, Example 5.1 is processed via the split-chop algorithm. The application of the equational transformation rules in Figure 5.1 works as follows:

- Start with context $(\Upsilon, \Psi) := \big(\{x_{[l]} \otimes \mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]} = \mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]} \otimes x_{[l]}\}, (\{1 \leq l\}, \emptyset, \emptyset, \emptyset)\big)$.
- *int-close* yields $\Psi \leftarrow (\emptyset, \{1 \leq l < \infty\}, \emptyset, \{l\})$
- Rule (1) matches with leftmost chunks $x_{[l]}$ and $\mathbf{1}_{[1]}$. Since l is variable, the context (Υ, Ψ) is split to integer constraints $\Psi_1 := \Psi \cup \{l < 1\}$, $\Psi_2 := \Psi \cup \{l = 1\}$ and $\Psi_3 := \Psi \cup \{l > 1\}$.
- Context (Υ, Ψ_1) yields **false** immediately, since $1 \leq l$ is violated. The context (Υ, Ψ_2) yields **false** after a few rule applications, for $x_{[1]} \otimes \mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]} \stackrel{!}{=} \mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]} \otimes x_{[1]}$ triggers rule (2).
- In context (Υ, Ψ_3), the equations $\{x_{[l]}[0:0] = \mathbf{1}_{[1]},\ x_{[l]}[1:l-1] \otimes \mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]} = \mathbf{0}_{[1]} \otimes x_{[l]}\}$ are added to Υ. In an attempt to match the second equation with rule (1), Ψ_{31}, Ψ_{32} and Ψ_{33} are generated with additional constraints $l < 2$, $l = 2$ and $l > 2$ respectively.
- Ψ_{31} yields **false**, Ψ_{32} terminates with the frame $\big(\{x_{[2]} = \mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]}\}, (\emptyset, \{2 \leq l \leq 2\}, \{l \mapsto 2\}, \{2\})\big)$.
- In context (Υ_3, Ψ_{33}), the rule $(1)^*$ matches equation $x_{[l]}[2:l-1] = x_{[l]}[0:l-3]$. There is a case split, according to whether $l \ MOD\ 2 = 0$. Contexts $(\Upsilon_{33}, \Psi_{331})$ and $(\Upsilon_{33}, \Psi_{332})$ are generated, where the terms $\frac{1}{2} \cdot l$ respectively $l \ MOD\ 2 = 1$ are added to Ψ_{331} respectively Ψ_{332}.

- To Υ_{331}, the equation $x_{[l]} = (a_{[2]})^{\frac{l}{2}}$ is added, thus terminating in a frame $\big(x_{[l]} = (\mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]})^{\frac{l}{2}}, (\emptyset, \{3 \leq l < \infty\}, \emptyset, \{\frac{1}{2} \cdot l\})\big)$.
- To Υ_{332}, the equation $x_{[l]} = b_{[1]} \otimes (d_{[1]} \otimes b_{[1]})^{\frac{l-1}{2}}$ is added. Together with $x_{[l]}[0:0] = \mathbf{1}_{[1]}$ and $x_{[l]}[l-1:l-1] = \mathbf{0}_{[1]}$, this context yields a **false**.

Thus the split-chop algorithm terminates with the set of non-**false** frames

$$\Big\{\big(\{x_{[2]} = \mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]}\}, (\emptyset, \{2 \leq l \leq 2\}, \{l \mapsto 2\}, \{2\})\big), \quad \big(x_{[l]} = (\mathbf{1}_{[1]} \otimes \mathbf{0}_{[1]})^{\frac{l}{2}}, (\emptyset, \{3 \leq l < \infty\}, \emptyset, \{\tfrac{1}{2} \cdot l\})\big)\Big\}$$

A short inspection confirms that all possible solutions are represented.

5.1.4 Experiments

The split-chop algorithm has been implemented in Allegro Common Lisp. However, by the end of this diploma thesis it did not reach a state of high trustworthiness. At the world wide web site `http://www.informatik.uni-ulm.de/ki/Bitvector/` a prototype version can be obtained.

5.2 Semaphore

The last approach explained an intuitive way towards solving large sub-sections of the bit-vector theory with variable width. Since the split-chop algorithm does not allow boolean operations or bit-vector arithmetic, the usability in practical applications is limited. Moreover, an extension of split-chop to boolean operations is necessarily incomplete (cf. Non-Existence Theorem 3.7).

It is to be mentioned that in the same time but independently, Bjørner and Pichora [BP98] presented an algorithm that allows to solve several special cases of bit-vector equations with variable width. More precisely, a parametrized term like $x_{[aN+b]}$ can be processed symbolically. Though a close comparison with the approach presented here is still outstanding, the split-chop algorithm allows a greater degree of freedom. For example, variable extractions can be processed by means of a simple case-split on the term structure in the beginning.

Thus, the concept presented here is prototypic. General methods for processing boolean operations and arithmetic for variable width are still left to be desired. In the present form it is to expect that for many pathologic examples the split-chop algorithm runs—regrettably like the author of this diploma thesis—out of time.

Chapter 6

Conclusion

All human progress involves, as it first condition,
the willingness of the pioneer to make a fool of himself.
(Bernard Shaw)

This diploma thesis contains a number of insights on solving bit-vector equations both from a theoretical and practical point of view. That might be grouped as follows.

Understanding the Complexity of Bit-Vectors

Though the "NP-completeness" of some equational bit-vector languages is not surprising, the correlation in the extension of theories as displayed in Figure 2.3 is better understood now. Moreover, the Quantification Lemma 2.9 explains that solving is far more a task than this chart might suggest. In most non-trivial extensions, a call to a solver can be used to decide $PSPACE$-hard problems (cf. section 2.5.3).

As expected, sufficient enrichment of the bit-vector theory leads to unsolvable problems. With the Non-Existence Theorem 3.7, at first a proof succeeded. Though bit-vectors are a simple data structure, they should have lost their image as "trivial" by now.

Exploration of Solving Approaches

We have presented three approaches for solving fixed-sized bit-vector terms in chapter 4. Their actual implementation on the one hand confirms the idea an on the other hand reveals weak performance in several pathologic cases. The fixed solver together with heuristics patches most of these, thus elaborating the extensive usage of OBDDs. A concept for solving non-fixed size bit-vector equations was given via the split-chop algorithm in section 5.1.

The vexing point is that in *any* approach simple examples were found that lead to horrendous response time. This suggests that solving bit-vector theories with a rich set of operators can not be performed by a simple concept, but rather by means of a sophisticated strategy and heuristics. If these observations are not fundamentally misleading, this is bad news. First, because the development of an efficient solver requires a lot of tedious work and second, since the trustworthiness of a complicated solving algorithm tends to be low. In the context of hardware verification, a doubtful mechanization is less than desirable.

Further Work

Due to the huge variety and surprising depth, not all of the topics centering around the problem of solving bit-vector equations could be discussed exhaustively. In particular, the following tasks seem to offer promising aspects for further investigation.

First, research the inherent complexity of solving more throughoutly. The characterization via the complexity class of the language of satisfiable equations seems to be misleading, for the severity of solving is not captured accurately. Thus a more expressive notion of complexity is desired. Moreover, the detection of decidable fragments is far from being complete. It is conjectured that the theory of bit-vectors with variable width and variable extractions is decidable via a combination of Makanin's algorithms and an exhaustive but finite case-split on the term structure.

Second, develop an efficient "master solving algorithm" for fixed size. This could follow the ideas of the fixed solver algorithm, but moreover use $WS1S$ encoding or other whenever it is considered to be advantageous. Without doubt, a lot of experiments are needed to draw reasonable decisions here. In addition, it is crucial to be specific in the conceptual details in order to enhance trustworthiness.

Third, the split-chop algorithm from chapter 5 should be implemented into a non-convex framework (STeP [BBC+] might be an interesting choice). Use reasoning about integers contained there instead of implementing a separated method for computing the closure on integer constraints. Then, add boolean operations and arithmetic.

The author is not optimistic, that these topics will be covered in short a time, let alone by *his* further work. After roughly two years of thinking about bit-vectors, he needs a break.

Appendix A

Former Results at the SRI

The results displayed here originate from the author's work at the SRI in autumn 1996. They were yet unpublished and appear here in the "original" form, excuse the typos. Only the notation was updated to avoid confusion.

A.1 An NP-complete Problem

The Problem BVEVE-Solvability

Consider the theory of bitvectors with fixed (finite nonzero) size and the operations $.\otimes.$ (composition) and $.[j:i]$ (extraction). i and j are allowed to be cardinal variables. Also, there exist constants of arbitrary but fixed length n, further denoted as $\mathbf{0}_{[n]}$ and $\mathbf{-1}_{[n]}$.
Let t_1 and t_2 be terms over such bitvectors. The problem to decide whether the there exists a sollution to the equation $t_1 = t_2$ is called BVEVE-Solvability (BitVector Equation with Variables in Extraction Solvability).

Claim

BVEVE-Solvability is NP-complete.

Proof:

a) BVEVE-Solvability $\in$ NP

One can guess a polynomial-sized sollution (t.i. polynomial in the term-length and the maximum length of the used bitvector variables) and check if the equation holds.

b) BVVE-SAT is NP-hard (Reduction 3-CNF-SAT $\leq_m$ BVEVE-Solvability)

Let $F = (L_{11} \vee L_{12} \vee L_{13}) \wedge ... \wedge (L_{m1} \vee L_{m2} \vee L_{m3})$ be a boolean formula in 3-CNF over variables $x_i \in Var = \{x_1, ..., x_n\}$. L_{ij} are literals in $Var \cup \overline{Var}$, $(i \in \{1, ..., m\},\ j \in \{1, 2, 3\})$.

For each $x_i \in Var$ introduce a bitvector variable a_i of length 3 and integer-variables r_i, s_i. Additionally, we need m bitvector variables b_j of length 3 and and integer variables t_j for padding.

$$\begin{array}{llll} \text{Let} & \varphi_i & := & (a_i[r_i:2])\otimes(a_i[0:s_i]) \\ & g(x_i) & := & a_i[0:s_i] \\ & g(\overline{x_i}) & := & a_i[r_i:2] \\ & G_j & := & g(L_{j1})\otimes g(L_{j2})\otimes g(L_{j3})\otimes(b_j[0:t_j]) \end{array} \quad \begin{array}{l} \\ \}\forall i \in \{1, ..., n\} \\ \\ \forall j \in \{1, ..., m\} \end{array}$$

Now define the reduction f as follows:

$$f(F) := \left\{ \begin{array}{ccccccc} a_1 \otimes ... \otimes a_n & \otimes & b_1 \otimes ... \otimes b_m & \otimes & \varphi_1 \otimes ... \otimes \varphi_n & \otimes & \mathbf{1}_{[1]} \otimes G_1 \otimes \mathbf{1}_{[1]} \otimes ... \otimes \mathbf{1}_{[1]} \otimes G_m \\ \mathbf{0}_{[3]} \otimes ... \otimes \mathbf{0}_{[3]} & \otimes & \mathbf{0}_{[3]} \otimes ... \otimes \mathbf{0}_{[3]} & \otimes & \mathbf{0}_{[3]} \otimes ... \otimes \mathbf{0}_{[3]} & \otimes & \mathbf{1}_{[1]} \otimes \mathbf{0}_{[7]} \otimes \mathbf{1}_{[1]} \otimes ... \otimes \mathbf{1}_{[1]} \otimes \mathbf{0}_{[7]} \end{array} \right. \stackrel{!}{=}$$

The equation $f(F)$ is solvable, if (and only if) all variables a_i and b_i equal 0-valued bitvectors; that means also, that each φ_i has to be of the length 3 to match the $\begin{array}{c}\mathbf{1}_{[1]}\\ \mathbf{1}_{[1]}\end{array}$-position. This can only be achieved if $r_i = s_i + 1$ for each i, thus $r_i \in \{1, 2\}$. Consider $x_i = true$ equivalent to $r_i = 1$ (and $x_i = false$ equivalent to $r_i = 2$). In order to match the lower term, each G_j has to be of the length 7. The length of $b_j[0 : t_j]$ is in $\{1, 2, 3\}$, thus at least *one* of the $g(L_{ij})$ in each clause has to be of the length two. This is equivalent to the notion that one of the literals L_{ij} evalueates to $true$.

Thus,

F is satisfiable $\Leftrightarrow$ there exists a mapping $\alpha : Var \rightarrow \{true, false\}$ with $\alpha(F) = true \Leftrightarrow$ there exists a mapping $\beta : \{r_i, s_i, t_j | i = 1, .., n; j = 1, .., m\} \rightarrow \mathcal{N}$ such that $f(F)$ is solvable.

$\mathcal{Q.E.D}$

A.2 An NP-hard Problem

The Problem $BV_{\otimes,bvec_n}$-Solvability

Consider the theory of bitvectors with variable size and the operation $.\otimes.$ (composition). A bitvector variable is denoted to be of the type $bvec_n$, where n is either a nonzero cardinal or a variable. The theory contains constants of arbitrary but fixed length m, further denoted as $\mathbf{0}_{[m]}$ and $\mathbf{-1}_{[m]}$.
Let t_1 and t_2 be terms in this theory. The problem to decide whether the there exists a sollution of the equation $t_1 = t_2$ is called $BV_{\otimes,bvec_n}$-Solvability.

Claim

$BV_{\otimes,bvec_n}$-Solvability is NP-hard.

Proof: (Reduction 3-CNF-SAT $\leq_m$ $BV_{\otimes,bvec_n}$-Solvability)

Let $F = (L_{11} \vee L_{12} \vee L_{13}) \wedge ... \wedge (L_{m1} \vee L_{m2} \vee L_{m3})$ be a boolean formula in 3-CNF over variables $x_i \in Var = \{x_1, ..., x_n\}$. L_{ij} are literals in $Var \cup \overline{Var}$, $i \in \{1, ..., m\}$, $j \in \{1, 2, 3\}$.
For each $x_i \in Var$ introduce two bitvector variables $a^{(i)} : bvec_{[p_i]}$ and $b^{(i)} : bvec_{[q_i]}$ of unknown size. Also we need m pairs of variables $c^{(j)} : bvec_{[r_j]}$, $d^{(j)} : bvec[s_j]$ for padding.

$$
\begin{array}{llll}
\text{Let} & \varphi_i & := & \mathbf{1}_{[1]}\otimes a^{(i)}\otimes b^{(i)} & \forall i \in \{1, ..., n\} \\
& \psi_j & := & \mathbf{1}_{[1]}\otimes c^{(j)}\otimes d^{(j)} & \forall j \in \{1, ..., m\} \\
& g(x_i) & := & a^{(i)} & \left.\right\} \forall i \in \{1, ..., n\} \\
& g(\overline{x_i}) & := & b^{(i)} & \\
& G_j & := & g(L_{j1})\otimes g(L_{j2})\otimes g(L_{j3})\otimes c^{(j)} & \forall j \in \{1, ..., m\} \\
& \Psi & := & a^{(p_1)}\otimes..\otimes a^{(p_n)}\otimes b^{(q_1)}\otimes..\otimes b^{(q_n)}\otimes c^{(r_1)}\otimes..\otimes c^{(r_m)}\otimes d^{(s_1)}\otimes..\otimes d^{(s_m)} &
\end{array}
$$

Now define the reduction f as follows:

$$
f(F) := \left\{ \begin{array}{lllllllll}
\mathbf{0}_{[1]}\otimes\Psi & \otimes & \varphi_1\otimes...\otimes\varphi_n & \otimes & \psi_1\otimes...\otimes\psi_m & \otimes\mathbf{1}_{[1]}\otimes & G_1\otimes\mathbf{1}_{[1]}\otimes...\otimes G_m & \stackrel{!}{=} \\
\Psi\otimes\mathbf{0}_{[1]} & \otimes & \mathbf{1}_{[1]}\otimes\mathbf{0}_{[3]}...\mathbf{1}_{[1]}\otimes\mathbf{0}_{[3]} & \otimes & \mathbf{1}_{[1]}\otimes\mathbf{0}_{[4]}...\mathbf{1}_{[1]}\otimes\mathbf{0}_{[4]} & \otimes\mathbf{1}_{[1]}\otimes & \mathbf{0}_{[7]}\otimes\mathbf{1}_{[1]}\otimes...\otimes\mathbf{0}_{[7]} &
\end{array} \right.
$$

This equation is solvable, if (and only if) Ψ equals a composition of 0-valued bitvectors, because the $\mathbf{0}_{[1]}$ is propagated to every cell of Ψ.
Therefore each sum $p_i + q_i$ must equal 3 and each sum $r_j + s_j$ must equal 4. One bitvector of the pair $a^{(i)}$, $b^{(i)}$ has the length 2, the other one is of the length 1. Consider $x_i = true$ equivalent to $||a^{(i)}|| = 2$ (and $x_i = false$ equivalent to $||a^{(i)}|| = 1$). For there are no zero length bitvectors allowed, each $||c^{(j)}||$ can be chosen between 1 and 3. Every G_j has to reach the length 7, this is equivalent to the notion that in each disjunction of the 3-CNF-formula F at least one literal evaluates to $true$.

Thus,
F is satisfiable $\Leftrightarrow$ there exists a mapping $\alpha : Var \rightarrow \{true, false\}$ with $\alpha(F) = true \Leftrightarrow$ there exists a mapping $\beta : \{p_i, q_i, r_j, s_j | i = 1, .., n; j = 1, .., m\} \rightarrow \mathcal{N}$ such that $f(F)$ is solvable.

$\mathcal{Q.E.D}$

Remark:

The question if $BV_{\otimes,bvec_n}$-Solvability $\stackrel{?}{\in} NP$ is still open. There seems to be no hint that for every solvable equation exists a polynomial sized (in the input, that is) sollution.

A.3 An Unsolvable Problem

The Problem $\exists BV_{[n]}$-Solvability

Consider the theory of bitvectors with nonzero size and the operations .⊗. (composition) and .[j:i] (extraction). i and j are allowed to be cardinal variables. Also, there exist constants of arbitrary but fixed length c, further denoted as $0_{[c]}$ and $1_{[c]}$.
Additional, there exists one special cardinal variable n, on which the terms can be dependent on, t.i. the cardinal variables can be indexed from 1 through to n and we allow an operation $\bigotimes_{i=1}^{n}$, which is a composition of n arguments. (Note that in the following $\bigotimes_{i=1}^{m}$ is not really a new operation, for m will be a *fixed* number.)
Let t_n and u_n be bitvector-terms dependent on n. The problem to decide whether there exists a sollution to the equation $\exists n : t_n = u_n$ is called $\exists BV_{[n]}$-Solvability.

Claim

$\exists BV_{[n]}$-Solvability is undecidable.

Proof:

(Reduction Post's correspondence problem $\leq \exists BV_{[n]}$-Solvability)

Let $P = \{(a_1, b_1), ..., (a_m, b_m)\}$, $m \geq 1$ be an instance of Post's correspondence problem, $a_i, b_i \in \Sigma^+$. Basically, we will construct two $\bigotimes_{i=1}^{n}$-compositions which match if (and only if) there is a corresponding sequence $i_1, ..., i_n$ that solves the correspondence problem (t.i. $a_{i_1} a_{i_2} \cdots a_{i_n} = b_{i_1} b_{i_2} \cdots b_{i_n}$).
Let $\sigma\, \Sigma \rightarrow \Sigma' = \{0,1\}^{\nu}$ be a mapping from the basic alphabet into an artifical one, where $\nu := max(3, \lceil log_2 \|\Sigma\| \rceil)$. $\sigma^* : \Sigma^* \rightarrow {\Sigma'}^*$ is defined according to this. Let $\omega := max\{|\sigma^*(a_i)| : i = 1,..,m\} \cup \{|\sigma^*(b_i)| : i = 1,..,m\}$.

Let $\mu := \lceil log_2 m \rceil$, the binary length needed to store the information, which pair (a_i, b_i) is chosen. Thus let $\tau : \{1, \ldots, m\} \rightarrow \{0_{[1]}, 1_{[1]}\}^{\mu}$ be the binary encoding of these numbers, represented in compositions of bitvector constants.
We introduce the following abbreviations:

$$\begin{array}{lcl}
A_0 & := & \bigotimes_{i=1}^{m} \Big(1_{[1]} \otimes 0_{[|\sigma^*(a_i)|-2]} \otimes 1_{[1]} \quad \otimes \quad 1_{[\mu]} \Big) \\
A_{\text{CHOOSE}} & := & \bigotimes_{i=1}^{m} \Big(\sigma^*(a_i) \quad \otimes \quad \tau(i) \Big) \\
B_0 & := & \bigotimes_{i=1}^{m} \Big(1_{[1]} \otimes 0_{[|\sigma^*(b_i)|-2]} \otimes 1_{[1]} \quad \otimes \quad 1_{[\mu]} \Big) \\
B_{\text{CHOOSE}} & := & \bigotimes_{i=1}^{m} \Big(\sigma^*(b_i) \quad \otimes \quad \tau(i) \Big) \\
\Omega & := & \bigotimes_{i=1}^{m} 0_{[max(|\sigma^*(a_i)|, |\sigma^*(b_i)|)+\mu]}
\end{array}$$

We also need some (indexed) integer variables to express which pair is chosen. l_i and r_i will be the marker for the first argument (a), l'_i and r'_i accordingly to the second one. Also, we introduce some 'puffers', namely c_i, d_i, e_i and c'_i, d'_i, e'_i to avoid arithmetic operations. It will yield that for every $i \in \{1, .., n\}$:

$$\left.\begin{array}{lll}
c_i = l_i - r_i & d_i = r_i - 1 & e_i = r_i - \mu \\
c'_i = l'_i - r'_i & d'_i = r'_i - 1 & e'_i = r'_i - \mu
\end{array}\right\} (*)$$

To make sure, that l_i, r_i always extract a valid pair-component out of A_{CHOOSE}(respectively l'_i, r'_i from B_{CHOOSE}), we define:

$$\begin{array}{lllllll}
\Phi_n & := & \bigotimes_{i=1}^{n} & A_0[r_i : l_i] & \Phi'_n & := & \bigotimes_{i=1}^{n} B_0[r'_i : l'_i] \\
\Psi_n & := & \bigotimes_{i=1}^{n} & \left(1_{[1]}\otimes 0_{[\omega]}[0 : c_i]\otimes 1_{[1]}\right) & \Psi'_n & := & \bigotimes_{i=1}^{n} \left(1_{[1]}\otimes 0_{[\omega]}[2 : c'_i]\otimes 1_{[1]}\right)
\end{array}$$

To enforce the 'right order' of the segments, we will match

$$\Delta_n^A := \bigotimes_{i=1}^{n} A_{\text{CHOOSE}}[e_i : d_i] \qquad \text{and} \qquad \Delta_n^B := \bigotimes_{i=1}^{n} B_{\text{CHOOSE}}[e'_i : d'_i]$$

To make shure that $(*)$ yields, we introduce

$$\begin{array}{lllllllll}
\Gamma_n & := & \bigotimes_{i=1}^{n} \Big(\Omega[r_i : l_i] & \otimes 1_{[1]}\otimes & \Omega(r_i, d_i) & \otimes 1_{[1]}\otimes & \Omega[e_i : d_i] & \otimes 1_{[1]}\Big) \\
\Lambda_n & := & \bigotimes_{i=1}^{n} \Big(\Omega[0 : c_i] & \otimes 1_{[1]}\otimes & 0_{[2]} & \otimes 1_{[1]}\otimes & 0_{[\nu]} & \otimes 1_{[1]}\Big) \\
\Gamma'_n & := & \bigotimes_{i=1}^{n} \Big(\Omega[r'_i : l'_i] & \otimes 1_{[1]}\otimes & \Omega(r'_i, d'_i) & \otimes 1_{[1]}\otimes & \Omega[e'_i : d'_i] & \otimes 1_{[1]}\Big) \\
\Lambda'_n & := & \bigotimes_{i=1}^{n} \Big(\Omega[0 : c'_i] & \otimes 1_{[1]}\otimes & 0_{[2]} & \otimes 1_{[1]}\otimes & 0_{[\nu]} & \otimes 1_{[1]}\Big)
\end{array}$$

Now define the reduction f as follows:

$$f(P) := \left\{ \begin{array}{ccccccccccc}
\Gamma_n & \otimes & \Gamma'_n & \otimes & \Phi_n & \otimes & \Phi'_n & \otimes & \Delta_n^A & \otimes & \bigotimes_{i=1}^{n} A_{\text{CHOOSE}}[r_i : l_i] \\
\Lambda_n & \otimes & \Lambda'_n & \otimes & \Psi_n & \otimes & \Psi'_n & \otimes & \Delta_n^B & \otimes & \bigotimes_{i=1}^{n} B_{\text{CHOOSE}}[r'_i : l'_i]
\end{array} \right. \overset{!}{=}$$

It is obvious, that P has a sollution, if (and only if) $f(P)$ is solvable.

$\mathcal{Q.E.D}$

Appendix B

Complexity Theory

B.1 3CNF-TQBF is $PSPACE$-complete

This fact is probably not new and even less surprising.
Nevertheless a proof is included here, for the common literature does not seem to refer to this detail. The idea of this proof was taken from the lecture "Algorithmen und Kalküle", taught by Prof. Dr. Uwe Schöning in the Summer Term 1996 at the University of Ulm and originally showed that $3CNF$-SAT is NP-complete.

Let $\Phi := Q_1 x_1 . \cdots Q_n x_n . (l_{11} \vee l_{12} \vee l_{13}) \wedge \ldots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$ be a fully quantified boolean formula in 3-conjunctive-normal-form ($3CNF$), $l_{ij} \in V \cup \overline{V}$, $V := \{x_1, \ldots, x_n\}$, $Q_k \in \{\forall, \exists\}$, $i = 1, ..., m$, $j = 1, 2, 3$, $k = 1, \ldots, n$. Then the language $3CNF$-$TQBF$ is defined as

$$3CNF\text{-}TQBF := \{\Phi | \models \Phi\}.$$

Claim:

$$3CNF\text{-}TQBF \text{ is } PSPACE\text{-complete.}$$

Proof: (Reduction $TQBF \leq_m 3CNF$-$TQBF$)

Let $\Phi := Q_1 x_1 . \cdots Q_n x_n . F$ be a fully quantified boolean formula with arbitrary matrix F over variables $V := \{x_1, \ldots, x_n\}$ and connectives $\wedge$, $\vee$ and $\neg$. F can be transformed in polynomial time into an equivalent formula $\hat{F}$ in Negation Normal Form (NNF) where negations only occur directly at variables via

$$\begin{array}{rcl} \neg(\alpha \wedge \beta) & \rightarrow & \neg\alpha \vee \neg\beta \\ \neg(\alpha \vee \beta) & \rightarrow & \neg\alpha \wedge \neg\beta \\ \neg\neg\alpha & \rightarrow & \alpha \end{array}$$

A transformation of $\hat{F}$ into a formula $\tilde{F}$ in $3CNF$ is defined as follows:
Let $T_{\hat{F}}$ be the tree representation of $\hat{F}$ where each node is marked with a $\wedge$ or an $\vee$ and each leaf is in $V \cup \overline{V}$. Let $\circ_{TOP} \in \{\wedge, \vee\}$ be the root node mark. For each non-leaf introduce a fresh boolean variable y_i, $i = 1, \ldots, k$. $T_{\hat{F}}$ is transformed into

$$\begin{array}{rccl} \tilde{F} := & y_1 & \wedge & [y_1 \Leftrightarrow (y_2 \circ_{TOP} y_3)] \\ & & \wedge & [y_2 \Leftrightarrow (y_4 \circ y_5)] \\ & & \vdots & \\ & & \wedge & [y_k \Leftrightarrow l_{k1} \circ l_{k2}] \end{array} \qquad (\star)$$

The arguments of $\circ \in \{\wedge, \vee\}$ are the left and right subtree of the corresponding node. The transformation terminates at leaf level, $l_{k1}, l_{k2} \in V \cup \overline{V}$.
Each of the expressions [..] is equivalent to three CNF clauses:

$$\begin{array}{rcl}
[y_i \Leftrightarrow (\alpha \circ \beta)] & \equiv & (y_i \Rightarrow \alpha \circ \beta) \wedge (\alpha \circ \beta \Rightarrow y_i) \\
& \equiv & (\overline{y_i} \vee (\alpha \circ \beta)) \wedge (\neg(\alpha \circ \beta) \vee y_i) \\
& \equiv & ((\overline{y_i} \vee \alpha) \circ (\overline{y_i} \vee \beta)) \wedge ((y_i \vee \overline{\alpha}) \bar{\circ} (y_i \vee \overline{\beta})) \\
& \equiv & \textsc{If } \circ = \wedge \quad \textsc{Then} \quad (\overline{y_i} \vee \alpha) \wedge (\overline{y_i} \vee \beta) \wedge (y_i \vee \overline{\alpha} \vee \overline{\beta}) \\
& & \qquad\qquad\quad \textsc{Else} \quad (\overline{y_i} \vee \alpha \vee \beta) \wedge (y_i \vee \overline{\alpha}) \wedge (y_i \vee \overline{\beta}) \\
& & \textsc{Endif}
\end{array}$$

Let $\phi : V \to \{\mathbf{true}, \mathbf{false}\}$ and $\psi : \{y_i, \ldots y_k\} \to \{\mathbf{true}, \mathbf{false}\}$ be assignments of variables. Then

$$\begin{array}{rcl}
\phi \models \hat{F} \quad \text{iff} & & \phi, \psi \models \tilde{F} \\
& \wedge & \psi \text{ assigns each value of } y_i \text{ according to } \phi
\end{array}$$

In particular, for each ϕ with $\phi \models \hat{F}$ there *exists* a mapping ψ resulting in a model of $\tilde{F}$, namely the one assigning $y_i \mapsto eval(\text{Node}_i)$. Vice versa, for a $\phi' \not\models \hat{F}$ there is no ψ with $\phi', \psi \models \tilde{F}$, for $(\star)$ cannot be satisfied. This leads to the statement

$$Q_1 x_1. \cdots Q_n x_n.F \quad \equiv \quad Q_1 x_1. \cdots Q_n x_n. \exists y_1. \cdots \exists y_k. \tilde{F}$$

Thus, each quantified boolean formula can be transformed in polynomial time into an equivalent quantified boolean formula in $3CNF$. This yields the reduction.

$\square$

B.2 $BV_{\otimes,[i:j]}$-Solvability is NP-complete

This fact was also recorded in autum 1996 at the SRI. The original (and less elegant) version of the proof is displayed in Appendix A.1.

Let $BV_{\otimes,[i:j]}$ be the theory of fixed-sized bit-vectors with composition and variable extraction. Assume the equation $t = u$ in this theory contains the variables $V_{t=u} := vars(t) \cup vars(u) = \{v_1, \ldots, v_k\}$. Define $\mathcal{L}_{t=u}$-SAT as

$$\mathcal{L}_{t=u}\text{-}SAT := \{t = u | \text{There exists an assignment } \alpha \text{ of } V_{t=u} \text{ with } \alpha \models t = u\}$$

Claim:

$$\mathcal{L}_{t=u}\text{-}SAT \text{ is } NP\text{-complete.}$$

Proof: (Reduction from $3CNF$-SAT)

(1.) $\mathcal{L}_{t=u}$-$SAT \in NP$

Given an equation $t = u$. Then nondeterministically guess aa assignment $\alpha : v_1 \mapsto a_1 \wedge \cdots \wedge v_k \mapsto a_k$ of all variables $v_i \in V_{t=u}$, $i = 1, \ldots, k$. Since there are upper bounds for each integer variable, $|\alpha|$ is polynomial in $|t = u|$ (assume the length information is given unary). It is also a polynomial task to check whether $\alpha \models t = u$.

(2.) $\mathcal{L}_{t=u}$-SAT is NP-hard.

Let $\Phi := (l_{11} \vee l_{12} \vee 13) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$ be a boolean formula in $3CNF$ over variables $V = \{x_1, \ldots, x_n\}$, $l_{jp} \in V \cup \overline{V}$, $j = 1, \ldots, m$, $p = 1, 2, 3$.
For each x_i introduce an integer variable m_i $(i = 1, \ldots, n)$ and for each clause an integer variable c_j, $(j = 1, \ldots, m)$. m_i is designed to be 0 if x_i is assigned to **false** and 1 if x_i is set **true**.
Define

$$\begin{array}{rcl} \phi(x_i) & := & \left(0_{[1]} \otimes 1_{[1]}\right)[m_i : m_i] \\ \phi(\overline{x_i}) & := & \left(1_{[1]} \otimes 0_{[1]}\right)[m_i : m_i] \end{array}$$

Translate Φ [in polynomial time] into the bit-vector equation

$$f(\Phi) := \left\{ \begin{array}{ccccc} \big(\phi(l_{11}) \otimes \phi(l_{12}) \otimes \phi(l_{13})\big)[c_1 : c_1] & \otimes & \cdots & \otimes & \big(\phi(l_{m1}) \otimes \phi(l_{m2}) \otimes \phi(l_{m3})\big)[c_m : c_m] \\ 1_{[1]} & \otimes & \cdots & \otimes & 1_{[1]} \end{array} \stackrel{!}{=} \right\}$$

The equation $f(\Phi)$ has a solution iff every term $\phi(l_{j1}) \otimes \phi(l_{j2}) \otimes \phi(l_{j3})$, $j = 1, \ldots, m$ contains at least one $1_{[1]}$. This is the case, iff at least one of the $\phi(l_{jp}) = 1_{[1]}$ which is equivalent to $l_{jp} = \textbf{true}$. Thus, Φ is satisfiable iff there exists a solution for $f(\Phi)$ and $\Phi \in 3CNF$-SAT iff $f(\Phi) \in \mathcal{L}_{t=u}$-$SAT$, completing the reduction. $\square$

Appendix C

Source Codes

All files were implemented in Allegro Common Lisp 4.3. It is heavily recommended to use an Allegro Lisp dialect in order to process them without further problems. The source-codes can be obtained at `http://www.informatik.uni-ulm.de/ki/Bitvector/`

C.1 Solve via Mona

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Diploma Thesis:
;;; "Solving Bit-Vector Equations
;;;  - A Decision Procedure for Hardware Verifikation"
;;;
;;; M. Oliver M"oller
;;; University of Ulm
;;; Faculty for Computer Science (Informatik)
;;; AI Department (Abteilung fuer k"unstliche Intelligenz)
;;; Supervising Professor:  F. von Henke
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; File:  check_via_mona.cl
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; HOW TO USE:  [Allegro Common Lisp Version]
;; ~~~~~~~~~~~
;;  (1) Install Mona Version 1.1
;;      [ to optain eg. at  http://www.brics.dk/~mona/index.html ]
;;  (2) Fetch also the files  bvec_structures.cl  and  bvec_arith.cl
;;      [ to optain eg. at
;;        http://www.informatik.uni-ulm.de/ki/Bitvector/  ]
;;  (3) Create a local directory, where Mona can store the
;;      input-and output files (e.g. mona/)
;;  (4) Modify the constants
;;        *home-directory*  *mona-local-directory-path*  *mona-call-exe*
;;      according to your local settings
;;  (5) Load common lisp files in this order
;;        bvec_structure.cl
;;        bvec_arith.cl
;;        check_via_mona.cl
;;      [compilation recommended]
;;  (6) Start >> Solver << with
;;        (solve-via-mona '(bv-equal <term-1> <term-2>))
;;      <term-1> and <term-2> are assumed to be canonized(!)
;;      An example for term structure is given in section * Examples *
;;      NOTES:
;;       (a) The notation is NOT identic with the one in the Diploma Thesis;
;;           Due to an (obsolete) design decision, the least significant
;;           bit is at the rightmost position and concatenations are the
;;           other way round as a consequence;
;;           eg. ( x{4} o y{4} ) ^ (1,0) = y{4} ^ (1,0)
;;       (b) The finite automaton is not put out as a default;
;;           if the directive  "-w" is added to the call to the mona-program
;;           (cf. function call-mona-program ), the automaton is put to file
;;             mona_output_file.txt
;;      The function  solve-via-mona  returns one of the symbols
;;        TAUTOLOGY         iff the equation is a tautology
;;        UNSATISFIABLE     iff the equation is unsatisfiable
;;        COUNTEREXAMPLE    if there is a model,
;;                          but the formula is not a tautology
;;        ERROR             if an error occurred
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; General Remarks:    This implementation realizes
;; 'Solving' fixed bv equations via a transformation to S1S
;; And creating an output which can be processed with MONA 1.1
;; Strictly speaking, this is not an implementation of a solver
;; but a test for TAUTOLOGY or UNSATISFIABILITY;
;; This can be used to generate a solving algorithm, by means of
;; Repeatedly replace bits of variables with constants;
;; [See Diploma Thesis for detailed description]
;;
;; The big revenue of this approach is, that it allows to check
;; FIXED SIZE BITVECTOR EQUATIONS WITH LINEAR ARITHMETIC
;;  AND LOGIC
;; for validity (which is the most common usage of decision procedures)
;; The big drawback is the sometimes horrendous run-time
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package user)

;;; ***************************************
;;;  Constants to Customize
;;; ***************************************

(defconstant *home-directory* "/home/hiwi/moeller/")

(defconstant *mona-local-directory-path*
   (CLOS::string-append *home-directory* "da/etc/mona/"))

(defconstant   *mona-call-exe*   "/usr/local/share/ai-systems/mona/mona-
1.1/mona")

;;; *********************************************************

(defconstant *mona-input-file*
   (CLOS::string-append *mona-local-directory-path*
                        "mona_input_file.mona"))
(defconstant *mona-output-file*
   (CLOS::string-append *mona-local-directory-path*
                        "mona_output_file.txt"))
(defconstant *mona-result-file*
   (CLOS::string-append *mona-local-directory-path*
                        "mona_result_file.txt"))
(defconstant *full-adder-file*
   (CLOS::string-append *mona-local-directory-path*
                        "fulladder.mona"))

(defconstant *allow-complement* nil)

(defconstant *negate* nil)
;;; *negate* = nil -> If the equation is a tautology,
;;;                       Mona will reply 'valid'
;;;                 Else   Mona will present a counterexample
;;;                       [Here, the automaton is a encoding
;;;                        of the SOLUTION]
;;; -----------------
;;; *negate* = t   -> If the equation is unsatisfiable
;;;                       Mona will reply 'valid'
;;;                 Else   Mona will present a counterexample
;;; -----------------


(defun solve-via-mona (bv-eq &optional (trigger-mona-on nil)
                                       (trigger-mona-off nil))
  ;; Returns  'tautology      iff the equation is a tautology
  ;;          'unsatisfiable  iff the equation is unsatisfiable
  ;;          'counterexample if there is a model,
  ;;                          but the formula is not a tautology
  ;;          'errir          if the mona-output-file is not as expected
  (ifassert (and (consp bv-eq)
                 (eq (car bv-eq) 'bv-equal)))
  (ifassert (is-fixed-arith-bool-bv? (cadr bv-eq)))
  (ifassert (is-fixed-arith-bool-bv? (cadr bv-eq)))
  (let* ((term1 (rewrite-boolean-logic-AON (cadr bv-eq)))
         (term2 (rewrite-boolean-logic-AON (caddr bv-eq)))
         (width (arith-bool-bv-length term1)))
    ;; !!! Assume canonized terms !!!
    (labels ((get-sub-structures (recog?)
               (remove-duplicates
                (append (all-recognized-in-arith-bool-bv-term term1 recog?)
                        (all-recognized-in-arith-bool-bv-term term2 recog?))
                :TEST #'equal)))
```

```
    (let* ((vars (sort
               (get-sub-structures #'(lambda (x) (rec-bv-var? x)))
               #'(lambda (x y)
                   (string< (bv-var-name x)
                            (bv-var-name y)))))
           ;;; (var-names (mapcar #'bv-var-name vars))
           (var-alen (mapcar #'(lambda (x)
                                 '(,(bv-var-name x) . ,(bv-var-length x)))
                             vars))
           (var-len (loop for pair in var-alen collect (cdr pair)))
           (var-fnames (mapcar #'(lambda (x)
                                   (CLOS::string-append
                                    "VAR_"
                                    (princ-to-string (bv-var-name x))
                                    "_"
                                    (princ-to-string (bv-var-length x))))
                               vars))
           ;;; (var-assoc (pairlis var-len var-fnames))
           (constants (get-sub-structures
                       #'(lambda (x) (rec-bv-const? x))))
           (additions (get-sub-structures
                       #'(lambda (x) (rec-bv-addition? x))))
           (negations (get-sub-structures
                       #'(lambda (x) (rec-bv-negation? x))))
;;! Compl does not work in Mona (early version)
           (complements (get-sub-structures
                         #'(lambda (x) (rec-bv-negation? x))))
           (complement-fnames
            (loop for i from 1 to (length complements) collect
                  (CLOS::string-append "Compl_"
                                       (princ-to-string i))))
           (complement-alist
            (if *allow-complement*
                nil
              (pairlis complements complement-fnames)))
;;!
           (filter-lengths
            (remove-duplicates
             (append var-len
                     (mapcar #'(lambda (x) (bv-addition-modulo x))
                             additions)
                     (mapcar #'(lambda (x) (arith-bool-bv-length x))
                             negations)
                     (mapcar #'bv-extraction-length
                             (get-sub-structures #'(lambda (x) (rec-bv-extraction?
x)))))))
           (filter-fnames
            (loop for e in filter-lengths collect
                  (CLOS::string-append "Filter_"
                                       (princ-to-string e))))
           (filter-assoc (pairlis filter-lengths filter-fnames))
           (addition-fnames
            (loop for i from 1 to (length additions) collect
                  (CLOS::string-append "RES_"
                                       (princ-to-string i))))
           (constant-fnames
            (loop for e in constants collect
                  (CLOS::string-append "CONST_"
                                       (princ-to-string (bv-const-length e))
                                       "_"
                                       (princ-to-string (bv-const-value e)))))
           (addition-alist (pairlis additions addition-fnames))
           (constant-alist (pairlis constants constant-fnames))
           (replace-alist (append (mapcar #'(lambda (x)
                                              (cons (car x)
                                                    (CLOS::string-append
                                                     "("
                                                     (cdr x)
                                                     " inter "
                                                     (dassoc (bv-addition-modulo (car
x)) filter-assoc)
                                                      ")")))
                                          addition-alist)
                                  (mapcar #'(lambda (x)
                                              (cons (car x)
                                                    (CLOS::string-append
                                                     "("
                                                     (cdr x)
                                                     " inter "
                                                      (dassoc (bv-var-length (car x))
filter-assoc)
                                                      ")")))
                                          (pairlis vars var-fnames))
                                  constant-alist
                                  (mapcar #'(lambda (x)
                                              (cons (car x)
                                                    (CLOS::string-append
                                                     "("
                                                     (cdr x)
                                                     " inter "
                                                    (dassoc (arith-bool-bv-length (car
x)) filter-assoc)
                                                      ")")))
                                          complement-alist)))

           ;;; --- Create Axioms ---
           (filter-axioms
            (loop for e in filter-assoc collect
                  (let ((len (car e))
                        (name (cdr e)))
                    '(all1 "p" (and (=> (< "p" ,len)
                                        (in "p" ,name))
                                    (=> (<= ,len "p")
                                        (notin "p" ,name)))))))
           (constant-axioms
            (loop for e in constant-alist append
                  (let* ((con (car e))
                         (len (bv-const-length con))
                         (val (bv-const-value con))
                         (bits (reverse (nat2bools val len)))
                         (name (cdr e)))
                    '((all1 "p" (=> (<= ,len "p")
                                    (notin "p" ,name)))
                      ,@(loop for i from 0 to (1- len) collect
                              (if (nth i bits)
                                  '(in ,i ,name)
                                '(notin ,i ,name)))))))
           (addition-axioms-strings
            (loop for e in addition-alist collect
                  (let* ((ad     (car e))
                         ;; (modulo (cadr ad))
                         (args   (cddr ad))
                         (len    (length args))
                         (name   (cdr e)))
                    (eval '(CLOS::string-append
                            "add" ,(princ-to-string len) "("
                            ,@(loop for e in args append
                                    '(,(translate-form-to-string-with-replace e replace-alist)
","))
                            ,name ")")))))
           ;!;
           (complement-axioms-strings
            (loop for e in complement-alist collect
                  (CLOS::string-append
                   "Complement( "
                   (translate-form-to-string-with-replace (car (bv-bool-args (car
e))) replace-alist)
                   " , "
                   (cdr e)
                   " )")))
           )
      (labels ((check-via-mona (negate width trigger-on trigger-off)
                 ;; Ouptut the created Terms
                 (new-mona-file)
                 (output-var2-and-addition-defs
                  var-fnames
                  (apply #'max
                         (cons 0
                               (mapcar #'(lambda (x)
                                           (length (bv-addition-args x)))
                                       additions)))
                  width)
                 (output-main-implication
                  (append (mapcar #'(lambda (x) (translate-form-to-string-
with-replace x replace-alist))
                                  (append filter-axioms
                                          constant-axioms))
                          addition-axioms-strings
                          complement-axioms-strings ;!;
                          )
                  (translate-form-to-string-with-replace term1 replace-alist)
                  (translate-form-to-string-with-replace term2 replace-alist)
                  (append constant-fnames
                          addition-fnames
                          filter-fnames
                          complement-fnames ;!;
                          )
                  negate)
                 (ifuncall trigger-on)
                 (call-mona-program)
                 (ifuncall trigger-off)
                 (scan-mona-output)))
        ;;; -- THE ONLY ONE CALL TO MONA --
        (check-via-mona nil width trigger-mona-on trigger-mona-off))))))


(defun translate-form-to-string-with-replace (form alist)
  (flet ((make-bracket-string (op list)
           (eval '(CLOS::string-append
                   "("
                   ,@(loop for e in (butlast list) collect
                           (CLOS::string-append
                            (translate-form-to-string-with-replace e alist)
                            " " op " "))
                   ,(translate-form-to-string-with-replace (car (last list)) alist)
                   ")")))
         (make-intermediate-string (op arg1 arg2)
           (CLOS::string-append
            "("
            (translate-form-to-string-with-replace arg1 alist)
            " " op " "
            (translate-form-to-string-with-replace arg2 alist)
            ")"))
         (make-quantified-string (quan var body)
           (CLOS::string-append
            "("
            quan
            " "
            var
            ": "
            (translate-form-to-string-with-replace body alist)
            ")")))
```

```
    (cond
     ((assoc form alist :TEST #'equal)
      (cdr (assoc form alist :TEST #'equal)))
     ((stringp form) form)
     ((integerp form) (princ-to-string form))
     ((consp form)
      (case (car form)
        (<=> (make-bracket-string ”<=>” (cdr form)))
        (=   (make-bracket-string ”=” (cdr form)))
        (and (make-bracket-string ”&” (cdr form)))
        (or  (make-bracket-string ”—” (cdr form)))
        (<=  (make-bracket-string ”<=” (cdr form)))
        (=>  (make-bracket-string ”=>” (cdr form)))
        (<   (make-bracket-string ”<” (cdr form)))
        (>   (make-bracket-string ”>” (cdr form)))
        (not (CLOS::string-append
              ”~”
              (translate-form-to-string-with-replace (cadr form) alist)
              ))
            (in   (make-intermediate-string ”in” (princ-to-string (cadr form))
(caddr form)))
        (notin (make-intermediate-string ”notin” (princ-to-string (cadr form))
(caddr form)))
        (all0 (make-quantified-string ”all0” (cadr form) (caddr form)))
        (all1 (make-quantified-string ”all1” (cadr form) (caddr form)))
        (all2 (make-quantified-string ”all2” (cadr form) (caddr form)))
        (exist0 (make-quantified-string ”exist0” (cadr form) (caddr form)))
        (exist1 (make-quantified-string ”exist1” (cadr form) (caddr form)))
        (exist2 (make-quantified-string ”exist2” (cadr form) (caddr form)))
        (union (make-bracket-string ”union” (cdr form)))
        (inter (make-bracket-string ”inter” (cdr form)))
        (compl (CLOS::string-append
                ”(compl ”
                (translate-form-to-string-with-replace (cadr form) alist)
                ”)”))
          (+ (translate-offset-to-string-with-replace (cadr form) (caddr form)
alist))
           (- (translate-offset-to-string-with-replace (cadr form) (-  0 (caddr
form)) alist))
        (bv-compose (translate-composition-to-string-with-replace form alist))
        (bv-extract (translate-extraction-to-string-with-replace form alist))
        ;; -- boolean --
        (bv-and (translate-form-to-string-with-replace
                 '(inter ,@(bv-bool-args form)) alist))
        (bv-or (translate-form-to-string-with-replace
                '(union ,@(bv-bool-args form)) alist))
        (bv-not (translate-form-to-string-with-replace
                 '(inter (compl ,@(bv-bool-args form))
                         ,(CLOS::string-append
                           ”Filter_”
                           (princ-to-string (arith-bool-bv-length form))))
                  alist))
             (t  (error-misc ”translate-form-to-string-with-replace”  form  ”not
caught.”)))))))


(defun translate-composition-to-string-with-replace (cmp alist)
  (let* ((args (reverse (bv-composition-content cmp)))
         (offset 0)
         (offsets (loop for e in args collect
                        (let ((old offset))
                          (incf offset (arith-bool-bv-length e))
                          old))))
    (translate-form-to-string-with-replace
     '(union ,@(loop for e in (pairlis args offsets) collect
                     '(+ ,(translate-form-to-string-with-replace (car e) alist)
                         ,(cdr e))))
     alist)))

(defun translate-offset-to-string-with-replace (term offset alist)
  (let ((str (translate-form-to-string-with-replace term alist)))
    (cond
     ((= offset 0)
      str)
     ((< offset 0)
      (CLOS::string-append ”(” str ” - ” (princ-to-string (-  0 offset)) ”)”))
     (t
      (CLOS::string-append ”(” str ” + ” (princ-to-string offset) ”)”)))))

(defun translate-extraction-to-string-with-replace (extr alist)
  (let ((len (bv-extraction-length extr))
        (bv  (bv-extraction-bv extr))
        (off (bv-extraction-right extr)))
    (translate-form-to-string-with-replace
     '(inter (- ,bv ,off)
             ,(CLOS::string-append ”Filter_” (princ-to-string len)))
     alist)))


;;; ****************************************
;;;  AUXillary Functions
;;; ****************************************


(defun create-filter-names (n)
  (loop for i from 1 to n collect
        (CLOS::string-append ”Filter”
                             (princ-to-string i))))

(defun dassoc (a alist)
  (cdr (assoc a alist :TEST 'equal)))

(defun rewrite-boolean-logic-AON (term)
  ;; Replace boolean operations: XOR
  ;; By AND OR NOT
  (cond
   ((rec-bv-var? term)
    term)
   ((rec-bv-const? term)
    term)
   ((rec-bv-extraction? term)
    (make-bv-extraction
     (rewrite-boolean-logic-AON (bv-extraction-bv term))
     (bv-extraction-left term)
     (bv-extraction-right term)))
   ((rec-bv-composition? term)
    (make-bv-composition-from-list
     (mapcar #'rewrite-boolean-logic-AON
             (bv-composition-content term))))
   ((rec-bv-addition? term)
    (make-bv-addition-from-list
     (bv-addition-modulo term)
        (mapcar #'rewrite-boolean-logic-AON
                (bv-addition-args term))))
   ((rec-bv-bool? term)
    (case (bv-recognizer term)
      (bv-and term)
      (bv-or  term)
      (bv-not term)
      (bv-xor (let ((x (rewrite-boolean-logic-AON
                        (car (bv-bool-args term))))
                    (y (rewrite-boolean-logic-AON
                        (cadr (bv-bool-args term)))))
                (make-bv-or (make-bv-and (make-bv-not x) y)
                            (make-bv-and x (make-bv-not y)))))
      (t (error-misc ”rewrite-boolean-logic-AON [bool]” term ”not caught.”))))
   (t (error-misc ”rewrite-boolean-logic-AON” term ”not caught.”))))


;;; ************************************************
;;;  File Handling
;;; ************************************************

(defun new-mona-file ()
  (with-open-file (stream *mona-input-file*
                   :direction :output
                   :if-exists :supersede
                   :if-does-not-exist :create)
     (format stream ”## Check Fixed Sized Bit Vector Equations via Mona
1.1~%”)
    (format stream ”## part of Diploma Thesis~%”)
    (format stream ”## M. Oliver Moeller 1997~%”)
    (format stream ”linear;~%”)
    ))

(defun output-var2-and-addition-defs (names add-upto width)
  (with-open-file (stream *mona-input-file*
                   :direction :output
                   :if-does-not-exist :error
                   :if-exists :append)
    (loop for n in names do
          (format stream ”var2 ~a;~%” n))
    (if (> add-upto 1)
        (progn
          (with-open-file (include *full-adder-file*
                           :direction :input
                           :if-does-not-exist :error)
            (loop for i from 1 to (file-length include) do
                  (princ (read-char include) stream)))
          (create-addition-predicates add-upto stream)))
    (unless *allow-complement*
      (format stream ”~%pred Complement(var2 X,Y) = all1 p : (p < ~d) =>
( (p in X) <=> (p notin Y));~%~%” width))))

(defun  output-main-implication  (axiom-stringlist  str1  str2  existential-
quantified negate)
  (with-open-file (stream *mona-input-file*
                   :direction :output
                   :if-does-not-exist :error
                   :if-exists :append)
    (flet ((conjunction-output-if (list)
             (if (null list)
                 (format stream ” true ”)
               (progn
                 (format stream ” (”)
                 (loop for a in (butlast list) do
                       (format stream ”~a & ~%  ” a))
                 (format stream ”~a )” (car (last list)))))))
      (format stream ”~%## Check the following ~a Implication:~%~a(”
              (if negate ”(negated)” ””)
              (if (consp existential-quantified)
                  (eval
                   '(CLOS::string-append
                     ,(if negate ”~(” ”(”)
                     ”ex2 ”
                     ,@(loop for e in (butlast existential-quantified) append
                             '(,e ”,”))
                      ,(car (last existential-quantified))
                      ”: ”))
                ””))
      (conjunction-output-if axiom-stringlist)
```

```
    (format stream   " &~%   ( ~a~%  =  ~a ) )  );~%" str1 str2))))


(defun call-mona-program ()
  (shell
    (CLOS::string-append
          *mona-call-exe*
          " -c -u "
          *mona-input-file*
          " 2> "
          *mona-output-file*
          " 1> "
          *mona-output-file*
          )))

(defun scan-mona-output ()
  (let ((res 'error))
    (flet ((testfor (string)
             (shell (CLOS::string-append
                     "egrep "
                     " '" string "' "
                     *mona-output-file* " > "
                     *mona-result-file*))
             (with-open-file (stream *mona-result-file*
                              :direction :input
                              :if-does-not-exist :error)
               (if (> (file-length stream)
                      0)
                   t
                 nil))))
      (cond
       ((testfor "Formula is valid")
        (setf res 'tautology))
       ((testfor "Counter-example")
        (setf res 'counterexample))
       ((testfor "Formula is unsatisfiable")
        (setf res 'unsatisfiable)))
      res)))

;;; ******************************
;;;  Creating Formulae
;;; ******************************

(defun create-addition-predicates (n stream)
  ;; from binary to n-ary terms
  (flet ((add-pred (i)
           (let ((vars (loop for j from 1 to i collect
                             (CLOS::string-append "S"
                                                  (princ-to-string j)))))
             (eval
              '(CLOS::string-append
                "pred add" ,(princ-to-string i) "(var2 "
                ,@(loop for e in vars append
                        '(,e ","))
                "Result) = ~%   ex2 Z: add"
                ,(princ-to-string (1- i))
                "("
                ,@(loop for e in (butlast vars) append
                        '(,e ","))
                "Z) & ~%           add2(Z,"
                ,(car (last vars))
                ",Result);~%~%")))))
    (loop for i from 3 to n do
          (format stream (add-pred i)))))

;;; ************
;;; * Examples *
;;; ************

(defun ex-1 () ;;; TAUTOLOGY
  (solve-via-mona
   '(bv-equal
     (bv-addition 4
      (bv-compose
       (bv-extract (bv-var x 4) (tupcons 2 0))
       (bv-const 0 1))
      (bv-const 1 4)
      (bv-const 0 4)
      (bv-const 0 4))
     (bv-compose (bv-extract (bv-var x 4) (tupcons 2 0))
      (bv-const 1 1)))))

(defun ex-2 () ;;; UNSATISFIABLE
  (solve-via-mona
   '(bv-equal
     (bv-compose (bv-var x 5) (bv-const 1 1))
     (bv-compose (bv-const 0 1) (bv-var x 5)))))

(defun ex-3 () ;;; COUNTEREXAMPLE
  (solve-via-mona
   '(bv-equal
     (bv-addition 3 (bv-var x 3) (bv-var y 3))
     (bv-const 3 3))))
```

C.2 Fixed Solver

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Diploma Thesis:
;;; "Solving Bit-Vector Equations
;;;  - A Decision Procedure for Hardware Verifikation"
;;;
;;; M. Oliver M"oller
;;; University of Ulm
;;; Faculty for Computer Science (Informatik)
;;; AI Department (Abteilung fuer kuenstliche Intelligenz)
;;; Supervising Professor:  F. von Henke
;; ------------------------------------------------------------
;; -              SOLVER FOR BITVECTOR-THEORY
;; -              ~~~~~~~~~~~~~~~~~~~~~~~~~~~
;; - WITH  +fixed size
;; -       +fixed extraction
;; -       +composition
;; -       +boolean operations
;; -       +artithmetic [via OBDDs]
;; -----------------------------------------------------------
;; -      Oliver M"oller (moeller@ki.informatik.uni-ulm.de)
;; -----------------------------------------------------------
;; - File:     bvec_fixed_solver
;; - Purpose:  Provides canonizer   fixed-sigma
;; -           and solver           fixed-bv-solve
;; - Requires: bvec_structure.cl
;; -           bvec_bdd_solve.cl
;; -           bvec_arith.cl
;; -           bvec_slicing.cl
;; -----------------------------------------------------------
;; - USAGE:
;; - Load and compile the required files in the listed order;
;; - Call (fixed-bv-solve (BV-EQUAL <term1> <term2>))
;; - [for term structure refer to bvec_structure.cl]
;; - The function returns list, that contains
;; -  (a) the symbol 'TRUE  if the equation is a tautology
;; -  (b) the symbol 'FALSE if the equation is unsatisfiable
;; -  (c) else, a number of lists
;; -       (BV-EQUAL <original variable>  <bv-term>)
;; -----------------------------------------------------------
;; NOTE:
;;  The term notation is NOT identic with the one in the
;;   Diploma Thesis!
;;  Due to an (obsolete) design decision, the least significant
;;  bit is at the rightmost position and concatenations are the
;;  other way round as a consequence;
;;  eg. ( x{4} o y{4} ) ^ (1,0) = y{4} ^ (1,0)
;;
;; Allegro Common Lisp Version
;; BEGUN: 1/9/1998
;;
;; +++++++++++++++++++++++++
;; + General Settings
;; +++++++++++++++++++++++++

(in-package user)

;;; observing

(defconstant *obs-fs* nil)

(defmacro obs-fs (arg)
  (if *obs-fs* arg))

(defvar *heuristic-node-prior* 0)
(defvar *heuristic-node-post* 0)

;;; ** Loading **

(defun l1 () (load "/home/hiwi/moeller/da/lisp/bvec_fixed_solver.cl"))

(defun ll ()
  (load "/home/hiwi/moeller/da/lisp/bvec_structure.cl")
  (load "/home/hiwi/moeller/da/lisp/bvec_bdd_solve.cl")
  (load "/home/hiwi/moeller/da/lisp/bvec_arith.cl")
  (load "/home/hiwi/moeller/da/lisp/bvec_slicing.cl")
  (load "/home/hiwi/moeller/da/lisp/bvec_fixed_solver.cl")
  )

(defun cc ()
  (compile-file "/home/hiwi/moeller/da/lisp/bvec_structure.cl")
  (compile-file "/home/hiwi/moeller/da/lisp/bvec_bdd_solve.cl")
  (compile-file "/home/hiwi/moeller/da/lisp/bvec_arith.cl")
  (compile-file "/home/hiwi/moeller/da/lisp/bvec_slicing.cl")
  (compile-file "/home/hiwi/moeller/da/lisp/bvec_fixed_solver.cl")
  ;-
  (load "/home/hiwi/moeller/da/lisp/bvec_structure.fasl")
  (load "/home/hiwi/moeller/da/lisp/bvec_bdd_solve.fasl")
  (load "/home/hiwi/moeller/da/lisp/bvec_arith.fasl")
  (load "/home/hiwi/moeller/da/lisp/bvec_slicing.fasl")
  (load "/home/hiwi/moeller/da/lisp/bvec_fixed_solver.fasl"))



;;;; ------------------------------------------------------------
;;;; ------------------------------------------------------------
;;;; THE CANONIZER ---------------------------------------------
;;;; ------------------------------------------------------------
;;;; ------------------------------------------------------------
```

```
(defun fixed-sigma (bv)
  ;; (ifassert (is-fixed-bv? bv))
  (fixed-beta (fixed-alpha bv)))

(defun fixed-alpha (bv)
  (cond
   ((or (rec-bv-var? bv)
        (rec-bv-const? bv))
    bv)
   ((rec-bv-composition? bv)
    (make-bv-composition
     (fixed-alpha (bv-decompose-left bv))
     (fixed-alpha (bv-decompose-right bv))))
   ((rec-bv-extraction? bv)
    (fixed-alpha-extraction bv))
   ((rec-bv-bool-apply? bv)
    (fixed-gamma bv))
   ((node-p bv)
    (lift-bdd-if bv))
   ((rec-bv-addition? bv)
    (fixed-delta bv))
   (t (error-misc "fixed-alpha" bv "not caught."))))

(defun fixed-alpha-extraction (bv)
  (let* ((arg (fixed-alpha (bv-extraction-bv bv)))
         (left (bv-extraction-left bv))
         (right (bv-extraction-right bv))
         (len (bv-length arg))
         (times (1+ (- left right))))
    (cond
     ((eq len times) arg)
     ((rec-bv-extraction? arg)
      (fixed-alpha-extraction
       (make-bv-extraction
        (bv-extraction-bv arg)
        (+ left (bv-extraction-right arg))
        (+ right (bv-extraction-right arg)))))
     ((rec-bv-composition? arg)
      (let* ((bv-left (bv-decompose-left arg))
             (bv-right (bv-decompose-right arg))
             (lr (bv-length bv-right)))
        (cond
         ((< left lr)
          (fixed-alpha-extraction
           (make-bv-extraction bv-right left right)))
         ((>= right lr)
          (fixed-alpha (make-bv-extraction bv-left (- left lr) (- right lr))))
         (t (make-bv-composition
             (fixed-alpha-extraction
              (make-bv-extraction bv-left (- left lr) 0))
             (fixed-alpha-extraction
              (make-bv-extraction bv-right (- lr 1) right)))))))
     ((rec-bv-const? arg)
      (extract-bv-const (bv-const-value arg) left right))
     ((rec-bv-var? arg) (make-bv-extraction arg left right))
     ((rec-bv-bool-apply? arg)
      (fixed-gamma (make-bv-bool-apply (car arg)
                                       (mapcar #'(lambda (x) (fixed-alpha
(make-bv-extraction x left right)))
                                    (cdr arg)))))

     ((node-p arg)
      (fixed-bdd-extraction arg left right))
     ;;--!-- tbc
     (t (error-misc "fixed-alpha-extraction" bv "not caught.")))))


(defun fixed-bdd-extraction (arg left right)
  ;; Extract in all nodes...
  (let ((content (fixed-alpha
                  (make-bv-extraction (node-variable arg) left right)))
        (n (1+ (- left right)))
        (then (node-then arg))
        (else (node-else arg)))
    (if (leaf-node? arg)
        (cond
         ((true-node? arg) (make-true-node n))
         ((false-node? arg) (make-false-node n))
         (t (error-misc "fixed-bdd-extraction" arg "illegal leaf node.")))
      (make-unique-node
       :VARIABLE content
       :ELSE (fixed-bdd-extraction else left right)
       :THEN (fixed-bdd-extraction then left right)))))


(defun fixed-beta (bv)
  (if (rec-bv-composition? bv)
      (let* ((list (bv-composition-content bv))
             (actual (car list))
             (next (cadr list))
             (rest (cddr list))
             (res nil))
        (flet ((matches ()
                 (or (and (rec-bv-const? actual)
                          (rec-bv-const? next))
                     (and (rec-bv-extraction? actual)
                          (rec-bv-extraction? next)
                          (equal (bv-extraction-bv actual)
                                 (bv-extraction-bv next))
                          (= (1+ (bv-extraction-left next))
```

```
                             (bv-extraction-right actual)))
                     (and (node-p actual)
                          (node-p next)
                          (nodes-match? actual next))))
               (melt ()
                 (setf
                     actual (cond
                             ((rec-bv-const? actual)
                              (append-bv-const actual next))
                             ((rec-bv-extraction? actual)
                              (fixed-alpha (make-bv-extraction
                                            (bv-extraction-bv actual)
                                            (bv-extraction-left actual)
                                            (bv-extraction-right next))))
                             ((node-p actual)
                              (attach-nodes actual next))
                             (t (error "[local]melt" actual "not caught.")))
                     next   (car rest)
                     rest   (cdr rest))))
          (loop while next do
                (loop while (and next (matches)) do (melt))
                (push actual res)
                (setf actual next
                      next   (car rest)
                      rest   (cdr rest)))
          (if actual (push actual res))
          (make-bv-composition-from-list (reverse res))))
    bv))

(defun fixed-gamma (bv)
  ;; Turns boolan expressions into BDDs
  (cond
   ((node-p bv) bv)
   ((rec-bv-bool-apply? bv)
    (let* ((op   (car bv))
           (args (mapcar #'(lambda (x)
                             (fixed-alpha
                              (flatten-bv-constants x)))
                         (cdr bv)))
           (slicing (overlay-vector-list
                     (mapcar #'bv-term-to-slicing args)))
           (arg-lists (ziplis (mapcar #'(lambda (x)
                                          (mapcar #'lift-to-bdd
                                                  (slice-bv-term x slicing)))
                                      args))))
      ;;(princ (format nil "ARGS    : ~a~%ARG-LISTS:~a~%" args arg-lists))
      (make-bv-composition-from-list
       (mapcar #'(lambda (x) (lift-bdd-if
                              (bdd-apply-n op x (bv-length (car x)))))
               arg-lists))
      ))
   (t (error-misc "fixed-gamma" bv "not caught."))))

(defun fixed-delta (bv)
  ;; canonize an bv-addition (!)
  (let* ((len (bv-addition-modulo bv))
         (args (bv-addition-args bv))
         (nargs (length args)))
    (case nargs
     (0 (make-bv-const 0 len))
     (1 (fixed-alpha (car args)))
     (2 (let ((slicing (make-fullslice len)))
          (init-leaf-nodes 1)
          (make-bv-composition-from-list
           (riple-carry-add (pairlis
                             (mapcar #'lift-to-bdd (slice-bv-term (first args)
slicing))
                           (mapcar #'lift-to-bdd (slice-bv-term (second args)
slicing)))
                         *false-node*))))
     (t (fixed-delta
         (make-bv-addition-from-list
          len
          (cons (fixed-delta
                 (make-bv-addition len (car args) (cadr args)))
                (cddr args))))))))

;;; &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
;;;          T H E         S O L V E R
;;; &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

(defun fixed-bv-solve (eq &key (stream nil)
                              (heuristic-level 0)
                              (call-after-canon nil))
 ;;; Heuristics:
 ;;;  0:  No melting of BDDs in advance of solving
 ;;;  1:  Melting on connect= ONE node matches (factor oo)
 ;;;  3:  Melting on connect= factor 4
 ;;;  5:  Melting on connect= factor 2.5
 ;;;  7:  Melting on connect= factor 2
 ;;;  9:  Melting on connect= factor 2.4
  (ifassert (is-fixed-arith-bv-equation? eq))
  (ifassert (eq (bv-length (second eq)) (bv-length (third eq))))
  (setf *heuristic-node-prior* 0
        *heuristic-node-post*  0)
  (let ((term1 (flatten-bv-constants (fixed-sigma (second eq))))
        (term2 (flatten-bv-constants (fixed-sigma (third eq)))))
    (ifassert (eq (bv-length term1) (bv-length term2)))
    (ifuncall call-after-canon)
    (obs-fs (princ (format nil "Term1: ~a~%" term1) stream))
    (obs-fs (princ (format nil "Term2: ~a~%" term2) stream))
```

```
    (fail-closure :SET nil)

    (if (equalp term1 term2)
        '(TRUE)
      (let* (
            (dummy (obs-fs (princ (format nil "##################
Starting Csolve ################~%") stream)))
            (heuristically-combined nil)
            (not-trivially-true nil)
            (slice (overlay-vector-list
                    (list (bv-term-to-slicing term1)
                          (bv-term-to-slicing term2))))
            (eqs (remove-duplicates
              (pairlis
               (slice-bv-term term1 slice)
               (slice-bv-term term2 slice))
              :TEST #'same-pair?))
            (bool-eqs (if (= 1 (logand 1 heuristic-level))
                          (remove-if-not #'(lambda (x) (or (node-p (car x))
                                                           (node-p (cdr x))))
                                         eqs)))
            (eqs (if (= 1 (logand 1 heuristic-level))
                     (remove-if #'(lambda (x) (or (node-p (car x))
                                                  (node-p (cdr x))))
                                eqs)
                   eqs)) ;; non-bool-eqs
            (vars (remove-duplicates
                   (append (loop for vs in (mapcar #'car eqs) append
                                 (all-vars-in-bv-term vs))
                           (loop for vs in (mapcar #'cdr eqs) append
                                 (all-vars-in-bv-term vs))
                           (loop for vs in (mapcar #'car bool-eqs) append
                                 (all-vars-in-bv-term vs))
                           (loop for vs in (mapcar #'cdr bool-eqs) append
                                 (all-vars-in-bv-term vs)))
                   :TEST #'equal))
            (nvars (length vars))
            (blocks (make-array (length vars) :initial-element nil))
            (slices (loop for v in vars collect
                          (make-nullslice (bv-var-length v))))
            (changes t)
            (final nil))
        (declare (ignore dummy))
        (labels ((fresh-var-call (var)
                   (nconc vars   (list var))
                   (nconc slices (list (make-nullslice (bv-var-length var)))))
                 (var-pos (var) (position var vars :TEST #'equal))
            (var-slice (var)
              (obs-fs (princ (format nil "-- (var-slice ~a) --> " var) stream))
              (let ((res
                     (cond
                      ((rec-bv-var? var) (nth (var-pos var) slices))
                      ((rec-bv-const? var)
                       (make-nullslice (bv-const-length var)))
                      ((rec-bv-extraction? var)
                       (extract-slice (nth (var-pos (bv-extraction-bv var)) slices)
                                      (bv-extraction-left  var)
                                      (bv-extraction-right var)))
                      ((and (node-p var)
                            (leaf-node? var))
                       (make-nullslice (bv-length var)))
                      ((node-p var)
                       (overlay-vector-list
                        (mapcar #'var-slice (vars-of-node var))))
                      (t (error-misc "(local) var-slice" var "not caught.")))))
                (obs-fs (princ (format nil "~a~%" res) stream))
                res))
            (entry-slice-var (extr)
              (obs-fs (princ (format nil "-- (entry-slice-var ~a) --> ~a~%" extr
"?") stream))
              (cond
               ((rec-bv-const? extr))
               ((rec-bv-var? extr))
               ((node-p extr)
                (mapcar #'entry-slice-var (all-bdd-nodes extr)))
               ((rec-bv-extraction? extr)
                (let ((var   (bv-extraction-bv extr))
                      (left  (bv-extraction-left extr)))
                  ;; only there is the new cut:
                  (setf (aref (var-slice var) left) t)))
               (t (error-misc "[local]entry-slice-var" extr "not caught."))))
            ;;;(has-failed? () (not possibly-solvable))
            ;;;(fail () (setf possibly-solvable nil))
            )
        (obs-fs (princ (format nil "Original Vars: ~a~%" vars) stream))
         (loop for extr in (append (all-recognized-in-arith-bool-bv-term term1
#'(lambda (x) (rec-bv-extraction? x)))
                                   (all-recognized-in-arith-bool-bv-term term2
#'(lambda (x) (rec-bv-extraction? x)))) do
              (entry-slice-var extr))
        (loop for e in eqs do
              (obs-fs (princ (format nil "csolve[~a = ~a]~%" (car e) (cdr e))
stream))
              (loop for b in (fixed-csolve (car e) (cdr e)
                                           :FRESH-VAR-CALL #'fresh-var-call
                                           :FAIL #'fail-closure) do
                    (obs-fs (princ (format nil " => ~a = ~a~%" (car b) (cdr b))
stream))
                    (setf not-trivially-true t)
                    (push (cdr b)
                          (aref blocks (var-pos (car b)))))
              (if (has-failed?) (return)))
        (setf heuristically-combined (heuristically-combine bool-eqs
                                        :STREAM stream
                                        :HEURISTIC-LEVEL heuristic-level))
        (if (some #'false-node? heuristically-combined)
            (fail-closure)
          (loop for e in (heuristically-combine bool-eqs
                                        :STREAM stream
                                        :HEURISTIC-LEVEL heuristic-level)
                do
                (obs-fs (princ (format nil "csolve[~a = TRUE]~%" e) stream))
                (loop for b in (fixed-csolve-bdd e
                                        :FRESH-VAR-CALL #'fresh-var-call
                                        :FAIL #'fail-closure) do
                      (obs-fs (princ (format nil " => ~a = ~a~%" (car b) (cdr b))
stream))
                      (setf not-trivially-true t)
                      (push (cdr b)
                            (aref blocks (var-pos (car b)))))
                (if (has-failed?) (return))))
          (obs-fs (princ (format nil "Vars    : ~a~%Slices : ~a~%" vars slices)
stream))
        ;;; -----------------------------
        (if (has-failed?)  (return-from fixed-bv-solve '(FALSE)))
        (unless not-trivially-true (return-from fixed-bv-solve '(TRUE)))
        ;;; -----------------------------
        ;;;  Coarsest Slicing
        ;;; -----------------------------
          (obs-fs (princ (format nil "################## Starting
Coarsest Slicing ################~%") stream))
        (loop while changes do
         (setf changes nil)
         (loop for i from 0 to (1- nvars) do
               (let* ((sl     (nth i slices))
                      (sl-old (copy-slicing sl)))
                 (loop for term in (aref blocks i) do
                       (let ((slice-list (mapcar #'var-slice
                                                 (reverse (bv-flat-term-content term)))))
                         (or-of-slices!  sl     slice-list)
                         (and-of-slices! sl-old slice-list)))
                 (unless (equalp sl sl-old)
                   (setf
                       changes t
                       (aref blocks i)
                       (mapcar #'(lambda (x)
                                   (make-bv-composition-from-list
                                         (slice-bv-term x sl :ENTRY-SLICE-VAR
#'entry-slice-var)))
                               (aref blocks i)))))))
        ;;; -----------------------------
        ;;; Propagation of Equality
        ;;; -----------------------------
          (obs-fs (princ (format nil "#################### Starting
Propagation ######################~%") stream))
         (obs-fs (princ (format nil "Vars    : ~a~%Slices : ~a~%Blocks : ~a~%"
vars slices blocks) stream))
        (setf final
          (mapcar #'fixed-beta
             (process-fixed-propagation
              (loop for i from 0 to (1- nvars) collect
                    (let ((j 0)
                          (columns (make-list (length (slicing-to-chop-list (nth i
slices))) :initial-element nil)))
                      (loop for line in (aref blocks i) do
                            (setf j 0)
                            (loop for term in (bv-flat-term-content line) do
                                  (pushnew term (nth j columns)
                                           :TEST #'equalp)
                                  (incf j)))
                      columns))
              :FAIL    #'fail-closure
              :HAS-FAILED? #'has-failed?
              :STREAM stream)))
        ;;;-------------------
        ;;; END
        ;;;-------------------
        (obs-fs (loop for i from 0 to (1- nvars) do
                      (princ (format nil "~a  =  ~a~%" (nth i vars) (nth i final))
stream)))
        ;;;-------------------
        (if (has-failed?)
            '(FALSE)
          (loop for i from 0 to (1- nvars) collect
                (make-bv-equation
                 (nth i vars)
                 (nth i final))))
      )))))


;;; ******************************
;;;  Failing Closure
;;; ******************************

(defun fail-closure (&key (set t) (quest nil))
  (defvar *internal-fail-var*)
  (cond
   (quest *internal-fail-var*)
   (t (setf *internal-fail-var* set))))

(defun has-failed? ()
  (fail-closure :QUEST t))
```

```
;;; ---

(defun pair-starts-with-1? (x)
  (eq (car x) 1))

;;; ========================================

(defun fixed-csolve (t1 t2 &key (fresh-var-call nil)
                                (fail #'break))
  ;;; Chunk-Solve
  ;;;  where t1 and t2 are flat terms
  ;;; when creating fresh variables, fresh-var-call is called with
  ;;;  the fresh var as an argument
  (cond
   ((equal t1 t2) nil)
   ((and (rec-bv-const? t1)
         (rec-bv-const? t2))
    (ifuncall fail) nil)
   ((or (node-p t1)
        (node-p t2))
    (fixed-csolve-bdd (bdd-apply-n 'BV-EQUIV
                                   '(,(lift-to-bdd t1) ,(lift-to-bdd t2))
                                   (bv-length t1))
                      :FRESH-VAR-CALL fresh-var-call
                      :FAIL fail))
   ((rec-bv-const? t1)
    (list (cons (fixed-bv-content t2)
            (pad-fixed-bv-if t1 t2 :FRESH-VAR-CALL fresh-var-call))))
   ((rec-bv-const? t2)
    (list (cons (fixed-bv-content t1)
            (pad-fixed-bv-if t2 t1 :FRESH-VAR-CALL fresh-var-call))))
   ;;; Two non-constants!
   ((and (rec-bv-var-or-var-extract? t1)
         (rec-bv-var-or-var-extract? t2))
    (fixed-csolve-var-var t1 t2 :FRESH-VAR-CALL fresh-var-call))

   (t (error-misc "fixed-csolve" (list t1 t2) "not caught."))))

(defun fixed-csolve-var-var (t1 t2 &key (fresh-var-call nil))
  (let ((v1 (fixed-bv-content t1))
        (v2 (fixed-bv-content t2)))
    (if (equal v1 v2)
        ;;; csolve-same-var [extraction, neccessarily]
        (let* ((len (bv-var-length v1))
               (l1 (bv-extraction-left t1))
               (r1 (bv-extraction-right t1))
               (l2 (bv-extraction-left t2))
               (r2 (bv-extraction-right t2))
               (hl (max l1 l2))
               (ll (min l1 l2))
               (hr (max r1 r2))
               (lr (min r1 r2)))
          (if (< ll hr)
              (let ((common (make-fresh-bv-var (1+ (- hl hr))
                                  :FRESH-VAR-CALL fresh-var-call)))
                (list
                 (cons
                  v1 (make-bv-composition-from-list
                      (list (make-fresh-bv-var (- len hl 1)
                                  :FRESH-VAR-CALL fresh-var-call)
                            common
                            (make-fresh-bv-var (- hr ll 1)
                                  :FRESH-VAR-CALL fresh-var-call)
                            common
                            (make-fresh-bv-var lr
                                  :FRESH-VAR-CALL fresh-var-call))))))
            (let* ((delta (- hr lr))
                   (overlap (1+ (- ll hr)))
                   (gamma (+ overlap delta delta))
                   (k-times (DIV gamma delta))
                   (lambda (MOD gamma delta))
                   (fresh1 (make-fresh-bv-var lambda
                                  :FRESH-VAR-CALL fresh-var-call))
                   (fresh2 (make-fresh-bv-var (- delta lambda)
                                  :FRESH-VAR-CALL fresh-var-call)))
              (list
               (cons v1
                     (fixed-alpha
                      (pad-fixed-bv-if
                       (make-bv-composition-from-list
                        (cons fresh1 (loop for i from 1 to k-times append
                                        (list fresh2 fresh1))))
                       (make-bv-extraction v1 hl lr)
                       :FRESH-VAR-CALL fresh-var-call)))))))

        ;; -- different vars --
          (let ((fresh (make-fresh-bv-var (bv-length t1) :FRESH-VAR-CALL
fresh-var-call)))
           (list
          (cons v1 (pad-fixed-bv-if fresh t1 :FRESH-VAR-CALL fresh-var-call))
              (cons v2 (pad-fixed-bv-if fresh t2 :FRESH-VAR-CALL fresh-var-
call)))))))

(defun fixed-csolve-bdd (node &key (fresh-var-call nil)
                                   (fail nil))
  (let* ((len  (bv-length node)))
    (init-leaf-nodes len)
    (loop for res in (bdd-solve node
                               :FRESH-VAR-CALL fresh-var-call
                               :FAIL fail) collect
          (cons
           (fixed-bv-content (car res))
             (pad-fixed-bv-if (cdr res) (car res) :FRESH-VAR-CALL fresh-var-
call)))))


(defun heuristically-combine (bool-eq-list &key (stream nil)
                                                (heuristic-level 1))
  ;;; Heuristic-Level:    1 -> weakly connected
  ;;;                     3 -> only strongly connected
  ;;;                     5 -> only heavy connected
  (setf *heuristic-node-prior* (length bool-eq-list))
  (unless (null bool-eq-list)
    (let ((phi (loop for b in bool-eq-list collect
                     (let ((node (bdd-apply-n 'BV-EQUIV
                                        '(,(lift-to-bdd (car b))
                                          ,(lift-to-bdd (cdr b)))
                                        (bv-length (car b)))))
                       (cons node (all-bdd-nodes node)))))
          (actual nil)
          (res nil)
          (changes nil))
      (flet ((connected? (a b)
               (case heuristic-level
                 (1 (not (null (intersection (cdr a) (cdr b) :TEST #'equal))))
                 (3 (<= (+ (length (cdr a)) (length (cdr b)))
                        (* 4 (length (intersection (cdr a) (cdr b) :TEST #'equal)))))
                 (5 (<= (+ (length (cdr a)) (length (cdr b)))
                              (* 2.5 (length (intersection (cdr a) (cdr b) :TEST
#'equal)))))
                 (7 (=  (+ (length (cdr a)) (length (cdr b)))
                        (* 2 (length (intersection (cdr a) (cdr b) :TEST #'equal)))))
                 (9 (<= (+ (length (cdr a)) (length (cdr b)))
                              (* 2.4 (length (intersection (cdr a) (cdr b) :TEST
#'equal))))) ;; Testbed
                    (t (error-misc "heuristically combine" (list heuristic-level)
"unknown heuristic")))))
        (loop while phi do
              (setf actual (car phi)
                    phi    (cdr phi)
                    changes t)
              (loop while changes do
                    (setf changes nil)
                    (loop for new in phi do
                          (if (connected? actual new)
                            (let ((new-node (bdd-apply-n 'BV-AND '(,(car actual)
,(car new)) (bv-length (car actual)))))
                              (setf changes t
                                actual  (cons new-node (all-bdd-nodes new-node))
                                   phi (delete new phi :TEST #'equalp))
                              (return)))))
              (push (car actual) res))
            (obs-fs (princ (format nil "Heuristically combine resulted in ~d
Nodes:~%~a~%" (length res) res) stream))
        (setf *heuristic-node-post* (length res))
        res))))


;;; ==============================

(defun process-fixed-propagation (llist &key (fail #'break)
                                             (has-failed? #'break)
                                             (stream nil))
  ;; Propagate equality within a list of lists of equivalence classes
  ;;  [each list representing one original variable]
  ;; builds assoc-list representing the union-find-structure
  ;; Returns a list of _terms_
  (let* ((alist nil)
         (all-contents (loop for e in llist append
                                          (apply #'append e)))
         (flats (remove-if #'is-bv-const?
                          (loop for cont in all-contents
                             append
                               (if (node-p cont)
                                   (all-bdd-nodes cont)
                                 '(,cont)))))
         (flat-times (mapcar #'(lambda (x) (count x flats :TEST #'equal))
                             flats))
         (flat-zip (pairlis flat-times flats))
         (one-timers
          (mapcar #'cdr
                 (remove-if-not #'pair-starts-with-1? flat-zip)))
         (bdds (remove-if-not #'node-p
                             (remove-duplicates all-contents :TEST #'equal)))
         (n-bdds (length bdds))
         (num-to-bdd (pairlis (count-up n-bdds) bdds))
         (rep-llist (loop for var in llist collect
                          (loop for column in var collect
                                (loop for entry in column collect
                                      (if (node-p entry)
                                          (find-via-rassoc entry num-to-bdd :TEST
#'eq)
                                        entry)))))
         (chunks (remove-duplicates flats :TEST #'equal))
         (chunk-to-indices
          (loop for c in chunks collect
                (cons c
                      (loop for b in bdds append
                            (if (member c (all-bdd-nodes b) :TEST #'equal)
                                (list (car (rassoc b num-to-bdd)))))))))
```

```
        (classes
         (loop for varcl in rep-llist append
               (loop for cl in varcl collect
                     (set-difference cl one-timers :TEST #'equal)))))
    (labels ((replace-in-indices (chunk bdd)
               (if bdds
                   (loop for i in (cdr (assoc chunk chunk-to-indices :TEST
#'equal)) do
                    (let* ((old-bdd (cdr (assoc i num-to-bdd)))
                           (old-chks (all-bdd-nodes old-bdd))
                            (new-bdd (bdd-compose old-bdd (lift-to-bdd bdd)
chunk))
                           (new-chks (all-bdd-nodes new-bdd))
                           (obsolete (set-difference old-chks new-chks
                                                     :TEST #'equal))
                           (newcomers (set-difference new-chks old-chks)))
                        (obs-fs (princ (format nil "Replacing [~d] ~a~%via ~a
<- ~a~%by ~a~%" i old-bdd chunk bdd new-bdd) stream))
                      (setf num-to-bdd
                        (cons (cons i new-bdd)
                              (delete i num-to-bdd :TEST #'(lambda (x y) (eq
x (car y))))))
;;;    The following, strangely, does not work
;;;                 (setf (cdr (assoc i num-to-bdd)) new-bdd)
                     (loop for c in obsolete do
                          (let ((find (assoc c chunk-to-indices :TEST #'equal)))
                            (if find
                                (setf (nth (position find chunk-to-indices :TEST
#'equal)
                                                   chunk-to-indices)
                                  (delete i find)))))
                           (obs-fs (princ (format nil "chunk-to-indices after
deleting:~%~a~%" chunk-to-indices) stream))
                    (loop for c in newcomers do
                          (insert-index c i))))))
             ;;;-------------------
             (insert-index (c ind)
               (let ((find (assoc c chunk-to-indices :TEST #'equal)))
                 (if find
                     (setf (nth (position c chunk-to-indices
                                          :TEST #'(lambda (x y) (equal x (car y))))
                                chunk-to-indices)
                       (nconc find '(,ind)))
                   (push (list c ind) chunk-to-indices))
                   (obs-fs (princ (format nil "New index for ~a: ~a~%Chunk-to-
indices: ~a~%" c ind chunk-to-indices) stream))))
             ;;; ---------------
             (fs-merge (a b)
               (let ((af (find-via-assoc a alist))
                     (bf (find-via-assoc b alist)))
                 (obs-fs (princ (format nil "-- merging: ~a~%              ~a~%" af
bf) stream))
                 (cond
                  ((equal af bf))
                  ((and (rec-bv-const? af)
                        (rec-bv-const? bf))
                   (ifuncall fail)
                   (return-from process-fixed-propagation nil))
                ;;; --- here check for booleans:
                  ((or (numberp af)
                       (numberp bf))
                   (let* ((bdd1 (lift-to-bdd (find-via-assoc af num-to-bdd)))
                          (bdd2 (lift-to-bdd (find-via-assoc bf num-to-bdd)))
                          (n (bv-length bdd1))
                          (node (bdd-apply-n 'BV-EQUIV '(,bdd1 ,bdd2) n)))
                     (loop for eq in (bdd-solve node :FAIL fail) do
                           (obs-fs (princ (format nil "..merge ~a => ~a~%" (car
eq) (cdr eq)) stream))
                         (if (ifuncall has-failed?)
                             (return-from process-fixed-propagation nil))
                         (if (node-p (cdr eq))
                             (progn
                               (loop for c in (all-bdd-nodes (cdr eq)) do
                                     (insert-index c n-bdds))
                               (replace-in-indices (car eq) (cdr eq))
                               (push (cons n-bdds (cdr eq)) num-to-bdd)
                               (push (cons (car eq) n-bdds) alist)
                               (incf n-bdds))
                           (progn
                             (replace-in-indices (car eq) (cdr eq))
                             (push eq alist))))))
                  ((rec-bv-const? af)
                   (setf alist (acons bf af alist))
                   (replace-in-indices bf af))
                  (t (setf alist (acons af bf alist))
                     (replace-in-indices af bf))))))
      (obs-fs (progn
                 (princ (format nil "BDD-Embeddings:~%~a~%" num-to-bdd)
stream)
             (princ (format nil "Chunks-to-Indices:~%~a~%" chunk-to-indices)
stream)
              (princ (format nil "Equivalence Classes:~%") stream)
              (loop for cl in classes do
                    (princ (format nil "~a,~%" cl) stream))))
      (loop for c in classes do
            ;; sometimes better: other order!
            (unless (null c)
              (let ((a (find-via-assoc (car c) alist)))
                (loop for b in (cdr c) do
                      (fs-merge a b)))))
        (obs-fs (princ (format nil "BDD-Embedding:~%~a~%Alist: ~a~%->
Combining result~%" num-to-bdd alist) stream))
      (let ((i -1))
        (loop for origc in rep-llist collect
              (progn
                (obs-fs (princ (format nil "Column-list: ~a~%" origc) stream))
                (make-bv-composition-from-list
                 (loop for column in origc collect
                       (progn
                         (incf i)
                         (lift-bdd-if
                             (or (find-via-assoc (car (nth i classes)) (append alist
num-to-bdd))
                              (make-fresh-bv-var (bv-length (car column)))))))))))))


(defun find-via-assoc (el alist &key (test #'equal))
  (let ((res (assoc el alist :TEST test)))
    (if (null res)
        el
      (find-via-assoc (cdr res) alist))))

(defun find-via-rassoc (el alist &key (test #'equal))
  (let ((res (rassoc el alist :TEST test)))
    (if (null res)
        el
      (find-via-rassoc (car res) alist))))


(defun fixed-bv-content (term)
  (cond
   ((rec-bv-var? term) term)
   ((rec-bv-extraction? term) (bv-extraction-bv term))
   (t (error-misc "fixed-bv-content" term "not caught."))))

(defun pad-fixed-bv-if (term pattern &key (fresh-var-call nil))
  ;;; returns a eventually padded bv-term with >term< in the middle
  ;;; according to pattern (possibly an extraction)
  (cond
   ;; no padding
   ((rec-bv-var? pattern)
    term)
   ((rec-bv-extraction? pattern)
    (let ((len   (bv-length (bv-extraction-bv pattern)))
          (left  (bv-extraction-left  pattern))
          (right (bv-extraction-right pattern)))
      (make-bv-composition-from-list
      (list (make-fresh-bv-var (- len left 1) :FRESH-VAR-CALL fresh-var-call)
            term
            (make-fresh-bv-var right :FRESH-VAR-CALL fresh-var-call)))))
   (t (error-misc "pad-fixed-bv-if" pattern "not caught"))))

(defun same-pair? (x y)
  (or (equalp x y)
      (and (equalp (car x) (cdr y))
           (equalp (car y) (cdr x)))))


;;; ********************
;;;  Examples
;;; ********************

(defun ex-1 ()  ;;; TAUTOLOGY
   (fixed-bv-solve
   '(BV-EQUAL
     (bv-addition 4 (bv-var x 4) (bv-var y 4))
     (bv-addition 4 (bv-var y 4) (bv-var x 4)))))

(defun ex-2 ()  ;;; UNSATISFIABLE
  (fixed-bv-solve
     '(BV-EQUAL  (bv-compose  (bv-var  x  16)  (bv-extract  (bv-var  x  16)
(tupcons 11 0)))
            (bv-compose (bv-var y 4) (bv-const 0 12) (bv-const 0 12)))))

(defun ex-3 () ;; Satisfiable but not valid
  (fixed-bv-solve
   '(bv-equal
     (bv-and (bv-var x 3)
             (bv-var y 3))
     (bv-var x 3))))
```

Bibliography

[Bar93] Jon Barwise, editor. *Handbook of mathematical logic*, volume 90 of *Studies in logic and the foundations of mathematics*. North-Holland, Amsterdam, reprint edition, 1993.

[BBC+] N. S. Bjørner, A. Browne, E.C. Chang, M. Col'on, A. Kapur, Z. Manna, Sipma. H.B., and T.E. Uribe. Step: Deductive-algorithmic verification of reactive and real-time systems. In *Computer Aided Verification; 8th International Conference, CAV '96, New Brunswick, NJ, USA. July 31 - August 3, 1996; Proceedings.*

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem.* Springer, Berlin;Heidelberg;New York, 1997.

[BK95] David A. Basin and Nils Klarlund. Hardware verification using monadic second-order logic. In Pierre Wolper, editor, *Computer Aided Verification; 7th International Conference, CAV '95, Li'ege, Belgium, July 3-5, 1995; Proceedings*, volume 939 of *Lecture Notes in Computer Science*, pages 31–41, Berlin;Heidelberg;New York, January 1995. Springer.

[BLW95] Beate Bollig, Martin Löbbing, and Ingo Wegener. Variable orderings for obdds, simulated annealing, and the hidden weighted bit function. Technical report, University of Dortmund, FB Informatik, LS II, 1995. tr-528, to obtain at `http://www.informatik.uni-dortmund.de/papers/tr-528.ps.gz`.

[BP98] Nikolaj S. Bjørner and Mark Pichora. Deciding fixed and non-fixed size bit-vectors. To appear in the TACAS workshop, April; obtained at `http://theory.stanford.edu/people/nikolaj/`, 1998.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry91] R.E. Bryant. Vlsi complexity: On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(02):205, 1991.

[Bry92] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. In *Computing Surveys*, volume 24(3), pages 293–318, September 1992.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 Congress*, Logic, Methodology and Philosophy of Science, pages 1–11. Stanford University Press, Stanford CA, 1962.

[BW96] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 45(09):993–1002, 1996.

[CLS96] David Cyrluk, Patrick Lincoln, and Natarajan Shankar. On shostak's decision procedure for combinations of theories. In McRobbie and Slaney [MS96], pages 463–477.

[CMR96] David Cyrluk, Oliver Möller, and Harald Rueß. An efficient decision procedure for a theory of fix-sized bitvectors with composition and extraction. Ulmer Informatik-Berichte 96-08, Fakultät für Informatik, Universität Ulm, 1996.

[CMR97] David Cyrluk, Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *Computer Aided Verification. 9th International Conference (CAV97). Haifa, Israel, June 22-25, 1997: Proceedings*, volume 1254 of *Lecture Notes in Computer Science LNCS*, pages 60–71, Berlin - Heidelberg - New York, 1997. Springer.

[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.

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

[GHR93] Cov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming Vol.1: Logical Foundations.* Handbooks of Logic in Computer Science and Artificial Intelligence and Logic Programming. Oxford University Press, Oxford, 1993.

[Hak85] Armin Haken. The intractability of resolution. In TCS-39-1 [TCS84], pages 297–308. Journal.

[HJJ+96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996. Also available through http://www.brics.dk/~klarlund/Mona/main.html.

[JAC90] Journal of the acm, January 1990. Journal.

[Jaf90] Joxan Jaffar. Minimal and complete word unification. In JACM-37-01 [JAC90], pages 47–85. Journal.

[Mak92] G. S. Makanin. Investigations on equations in a free group. In Schulz [Sch92b], pages 1–11.

[MS95] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. ieeecs.

[MS96] M. A. McRobbie and J. K. Slaney, editors. *Automated Deduction - CADE-13; 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996; Proceedings*, volume 1104 of *Lecture Notes in Computer Science;I.2.3, F.4.1-2*, Berlin;Heidelberg;New York, 1996. Springer.

[ND88] G. Sivakumar N. Dershowitz, M. Odaka. Confluence of conditional rewrite systems. In Stephane Kaplan and Jean-Pierre Jouannaud, editors, *Conditional Term Rewriting Systems. 1st International Workshop Orsay, France, July 8-10, 1987*, volume 308 of *Lecture Notes in Computer Science*, pages 31 – 44, Berlin;Heidelberg;New York, 1988. Springer.

[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.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, Massachusetts - Menlo Park, California - New York, 1994.

[Pre29] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welcher die addition als einzige operation hervortritt. In *Comptes Rendus du I^{er} Congrès des Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley, Chichester;New York;Brisbane, 1986.

[Sch92a] Uwe Schöning. *Theoretische Informatik kurz gefaßt.* BI Wissenschaftsverlag, Mannheim - Leipzig - Wien - Zürich, 1992.

[Sch92b] Klaus Ulrich Schulz, editor. *Word Equations and Related Topics. 1st International Workshop IWWERT '90, Tübingen, Germany, October 1-3, 1990*, volume 572 of *Lecture Notes in Computer Science*, Berlin;Heidelberg;New York, 1992. Springer.

[Sho78] R.E. Shostak. An Algorithm for Reasoning about Equality. *Communications of the ACM*, 21(7):583–585, July 1978.

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

[TCS84] Theoretical computer science, July 1984. Journal.

[TH] J-L. Lassez T. Huynh, C. Lassez. Fourier algorithm revisited. In *Algebraic and Logic Programming*, pages 117–131. Springer.

[vL90a] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science. Vol. A: Algorithms and Complexity.* Elsevier, Amsterdam - New York - Oxford, 1990.

[vL90b] Jan van Leeuwen. *Handbook of Theoretical Computer Science. Vol. B: Formal Methods and Semantics.* Elsevier, Amsterdam - New York - Oxford, 1990.