Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 144 (2006) 3–14

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Encoding RTL Constructs for MathSAT: a Preliminary Report

Marco Bozzano[a], Roberto Bruttomesso[a], Alessandro Cimatti[a]
Anders Franzén[a,c], Ziyad Hanna[b], Zurab Khasidashvili[b],
Amit Palti[b], and Roberto Sebastiani[c]

[a] *ITC-IRST, Via Sommarive 18, 38050 Povo, Trento, Italy*
{bozzano,bruttomesso,cimatti,franzen}@itc.it

[b] *Logic and Validation Technologies, Intel Architecture Group of Haifa, Israel*
{ziyad.hanna,amit.palti,zurab.khasidashvili}@intel.it

[c] *DIT, Università di Trento, Via Sommarive 14, 38050 Povo, Trento, Italy*
roberto.sebastiani@dit.unitn.it

Abstract

Formal checking at Register-Transfer Level (RTL) is currently a fundamental step in the design of hardware circuits. Most tools for formal checking, however, work at the boolean level, which is not expressive enough to capture the abstract, high level (e.g., structural, word level) information of RTL designs. Tools for formal checking are thus confronted with problems which are "flattened" down to boolean level, so that a predominant part of their computational effort is wasted in performing useless boolean search on the bitwise encoding of integer data and arithmetical operations. In this paper we present a way of encoding RTL constructs into SMT formulas, that is, boolean combinations of boolean variables and quantifier-free constraints in Integer Linear Arithmetic. Such formulas can be handled by the MathSAT tool (and others) *directly*, without flattening to boolean level, so that to reduce drastically the computational effort.
We propose a mixed boolean/ILP encoding, in which control variables are encoded as boolean variables, datapath variables as integer variables; control constructs are handled as boolean combination of control variables and predicates over datapath variables, and datapath constructs are encoded, as much as possible, as linear arithmetical constraints over datapath variables.

Keywords: Satisfiablity Modulo Theories, Decision Procedures, Verification of RTL designs

1571-0661/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2005.12.001

1 Introduction

Formal methods are widely applied as powerful verification and early debugging techniques in the development of complex industrial systems. In particular, formal checking at Register-Transfer Level (RTL) is currently a fundamental step in the design of hardware circuits. A number of techniques and tools have been developed, able to carry out equivalence checking and property checking of combinational and sequential circuits.

Most tools for formal checking, however, work at the boolean level, which is not expressive enough to capture the abstract, high level (e.g., structural, word level) information of RTL designs. Tools for formal checking are thus confronted with problems which are "flattened" down to boolean level (e.g., integer data values are encoded and manipulated as arrays of booleans), so that a predominant part of their computational effort is wasted in performing useless boolean search on the bitwise encoding of integer data and arithmetical operations (e.g., up to a 2^{32} factor in the amount of boolean search for a 32-bit integer value). In particular, notice that boolean solvers are "bad at mathematics", in the sense that reasoning on the boolean encoding of arithmetical operations (e.g., sums) causes a blowup of the computational effort.

We want to investigate enhanced SAT-based techniques for RTL formal checking and to deliver better verification tools for RTL designs. These tools will avoid flattening by working directly at a level of expressivity higher than boolean reasoning, and will be able to analyze larger scale RTL designs.

The first step in this direction is the development of the MathSAT tool, an efficient SAT-based tool for satisfiability modulo theories (SMT), able to solve boolean combinations of boolean atoms and quantifier-free atoms in the theories of equality and uninterpreted functions (EUF), difference logic (DL), linear arithmetic on the Reals ($LA(\mathbb{R})$) and on the integers ($LA(\mathbb{Z})$), and on combined theories $EUF + DL$, $EUF + LA(\mathbb{R})$, $EUF + LA(\mathbb{Z})$ [4,3].

As a second step in this direction, in this paper we present a way of encoding RTL constructs into SMT formulas in $LA(\mathbb{Z})$ or $EUF + LA(\mathbb{Z})$. Such formulas can be handled by the MathSAT tool (and others) *directly*, without flattening to boolean level, so that to reduce drastically the computational effort.

The main idea is to partition an RTL circuit design into *control* and *data-path* components, and to encode the former into boolean *formulas*, so that to be handled by the SAT solver embedded in MathSAT, and to encode (as much as possible) the latter into *terms* in $LA(\mathbb{Z})$, so that to be handled directly by the $LA(\mathbb{Z})$ solver in MathSAT, thus avoiding bit-blasting as much as possible.

2 Basic Notation and Assumptions

In the following we use capital symbols A, B, ... to represent boolean variables (bits), underlined capital symbols $\underline{A}$, $\underline{B}$, ... to represent word variables (i.e., bit arrays), and lower case symbols a, b, ... to represent integer variables. If not otherwise specified, we assume that words have N bits (e.g., $N = 32$), and that integer variables can assume positive integer values within the range $[0, ..., 2^N - 1]$. In the following, if not otherwise specified, we assume that l, m are integer values s.t. $0 \leq l \leq m \leq N - 1$.

If $\underline{A}$ is a word, we denote as $\underline{A}_{[i]}$ the boolean variable representing the i-th bit of $\underline{A}$; [1] we denote by $\underline{A}_{[m:l]}$ the $(m-l+1)$-bit subword of $\underline{A}$ from the l-th to the m-th bit, that is, the word obtained by concatenating $\underline{A}_{[m]}\underline{A}_{[m-1]} \cdots \underline{A}_{[l+1]}\underline{A}_{[l]}$.

If a is an integer variable, we denote by $\underline{A}$ (same symbol, capitalized and underlined) the word variable corresponding to the bitwise encoding of a. Consequently, we denote by $\underline{A}_{[i]}$ a boolean variable representing the i-th bit of the word corresponding to the bitwise encoding of a; we denote by $a_{[i]}$ an integer variable s.t. $0 \leq a_{[i]} \leq 1$ representing (the integer value of) the i-th bit of its word corresponding to the bitwise notation of a, that is, $a = \sum_{i=0}^{N-1} 2^i \cdot a_{[i]}$; we denote by $a_{[m:l]}$ the integer variable in $[0, ..., 2^{m-l+1} - 1]$ representing the word $\underline{A}[m : l]$.

In sequential circuits, we use unprimed symbols like a, C and primed symbols like a', C' to indicate *current* and *next* values of either bits of words.

3 Encoding linearizable constructs

We partition an RTL circuit design into *control* and *datapath* components.

- Control variables are encoded as boolean variables (i.e., atomic propositions). Control constructs are handled as boolean combination of control variables and predicates over datapath variables. (E.g., an "AND" gate is represented by the boolean connective "$\wedge$".)
- Datapath variables are encoded as integer variables in $[0, 2^N - 1]$, N being the size of the words in the datapath. Datapath constructs are encoded, as much as possible, as linear arithmetical constraints over datapath variables. (We call these constructs, *linearizable.*) The few other constructs which are not linearizable are encoded by bit-blasting, or by means of uninterpreted functions (see §4).
- Some other constraints, which we call *interface constraints*, are needed to

[1] We start counting from 0, from the right to the left. This means, e.g., that $\underline{A}_{[0]}$ is the least significant bit of $\underline{A}$ and that $\underline{A}_{[N-1]}$ is the most significant bit of $\underline{A}$.

represent the interface between the control and datapath lines. (E.g., in case of comparators, multiplexers, or of bit-blasting.)

Notice that often the i-th bit of a datapath word $\underline{A}$ for some i can be also an input or the output of control gates. If so, the bit requires two representations: an integer variable $a_{[i]}$, representing it as (one bit of) a datapath word, and an atomic proposition $\underline{A}_{[i]}$, representing it as a "control" wire (which can be the input or output of some gate). Then it is added an interface constraint in the form:

(1) $\underline{A}_{[i]} \leftrightarrow (a_{[i]} = 1)$,

which relates the respective values of the two representations. [2]

3.1 Encoding Word Operations

If a word $\underline{A}$ of length l is represented by an integer a, then it is necessary to add a subformula representing the *range* of a:

(2) $(0 \leq a) \wedge (a < 2^l)$.

In the following, for every integer variable added (included dummy ones) we assume an implicit subformula like (2), which will not be reported for short.

3.1.1 Concatenation of words

Let $\underline{W_1}$,...,$\underline{W_K}$ be words of length l_1,...,l_k respectively, and let $\underline{W}$ be the result of concatenating the words $\underline{W_K}$,..,$\underline{W_1}$, that is, $\underline{W}$:=$\underline{W_K}$...$\underline{W_1}$. Given the integer variables w_1, ... w_k, the integer variable w is built as follows:

(3) $(w = \sum_{j=1}^{k} 2^{(\Sigma_{i=1}^{j-1} l_i)} \cdot w_j)$.

3.1.2 Extraction of subwords

We need extracting the integer variable $a_{[m:l]}$ from the integer variable a (i.e., to represent the subword $\underline{A}_{[m:l]}$ of a word $\underline{A}$). Depending on the values l and m, we have to distinguish different cases:

(i) **Extract the least [most] significant subword** $a_{[m:0]}$ $[a_{[N-1:m+1]}]$ We introduce two variables $a_{[m:0]}$ and $a_{[N-1:m+1]}$, and add the following formulas:

(4) $(a = a_{[m:0]} + 2^{m+1} \cdot a_{[N-1:m+1]})$.

[2] Notice that $\underline{A}_{[i]}$ can be eliminated from the final formula by substituting all occurrences of $\underline{A}_{[i]}$ with $(a_{[i]} = 1)$ and by eliminating (1), that is, $\phi \wedge (\underline{A}_{[i]} \leftrightarrow (a_{[i]} = 1)) \implies \phi[\underline{A}_{[i]}|(a_{[i]} = 1)]$.

(ii) **Extract an intermediate subword** $a_{[m:l]}$ We introduce three variables $a_{[l-1:0]}$, $a_{[m:l]}$ and $a_{[N-1:m+1]}$ and add the following formulas:

(5) $(a = a_{[l-1:0]} + 2^l \cdot a_{[m:l]} + 2^{m+1} \cdot a_{[N-1:m+1]})$.

Notice that $a_{[N-1:m+1]}$, $[a_{[m:0]}]$ in (4) and $a_{[l-1:0]}$ and $a_{[m:l]}$ in (5) may be a dummy variables, unless they are explicitly required.

3.1.3 Assignment

If a N-bit word $\underline{A}$ is assigned the value of another N-bit word $\underline{B}$, this fact is simply encoded as

(6) $(a = b)$.

If the assignment is performed between subwords, e.g. a N-bit word $\underline{A}_{[m_1:l_1]}$ is assigned the value of another N-bit word $\underline{B}_{[m_2:l_2]}$ such that $m_1 - l_1 = m_2 - l_2 = N$, then it is first necessary to extract $a_{[m_1:l_1]}$ and $b_{[m_2:l_2]}$ as in (4),(5).

3.1.4 ITE's and Multiplexers

An ITE command in the form "$\underline{A}$=ITE(D,$\underline{B}$,$\underline{C}$), D being a bit and $\underline{A}$, $\underline{B}$ and $\underline{C}$ being N-bit words, is encoded simply as:

(7) $(D \rightarrow (a = b)) \wedge (\neg D \rightarrow (a = c))$.

If $\underline{B}$ and $\underline{C}$ are mutually exclusive values for $\underline{A}$ (e.g., $\underline{B}$ and $\underline{C}$ are different constant values) it may be of help to explicitly add a mutex condition:

(8) $(D \rightarrow (a = b)) \wedge (\neg D \rightarrow (a = c)) \wedge (\neg(a = b) \vee \neg(a = c))$,

as the latter forces (a=c) to $\perp$ and D to $\top$ as soon as (a=b) is assigned to $\top$, and vice versa. (From now on we will call the combination of an ITE statement with a mutex condition as in (8), an "ITEX" statement.)

In case of nested ITE's, like, e.g., "$\underline{A}$=ITE(D,ITE(E,$\underline{B}$,$\underline{C}$),ITE(F,$\underline{G}$,$\underline{H}$))", D, E and F being bits and $\underline{A}$, $\underline{B}$, $\underline{C}$, $\underline{G}$ and $\underline{H}$ being N-bit words, it is efficient to encode multiple conditions directly:

(9) $((D \wedge E) \rightarrow (a = b)) \wedge ((D \wedge \neg E) \rightarrow (a = c)) \wedge$
$((\neg D \wedge F) \rightarrow (a = g)) \wedge ((\neg D \wedge \neg F) \rightarrow (a = h))$.

If the ITE is performed among subwords, then it is first necessary to extract the subwords, as described above.

3.1.5 Latches

In sequential circuits, the output $\underline{L}$ of a word latch is updated to its input value $\underline{D}$ at every clock tick. If we keep the clock signal implicit, this fact can be encoded simply as

(10) $(l' = d)$.

If we use an explicit representation of clock CK, we can encode this by

(11) $l' = ITE(CK', d', l)$.

(In general, there are many types of state elements, but all can be reduced to latches plus some combinational logic.)

3.1.6 Right and Left Shift

Let $\underline{B}$ be the result of right shifting $\underline{A}$ of k bits, $k \leq N$. $\underline{B}$ is the concatenation

$$\overbrace{0...0}^{k\ times}\ \underline{A}_{[N-1:k]}$$

and thus it can be encoded simply as:

(12) $(b = a_{[N-1:k]})$.

where $a_{[N-1:k]}$ is extracted as in (4).

Let $\underline{B}$ be the result of left shifting $\underline{A}$ of k bits, $k \leq N$. $\underline{B}$ is the concatenation

$$\underline{A}_{[N-k-1:0]}\ \overbrace{0...0}^{k\ times}$$

and thus it can be encoded simply as:

(13) $(b = 2^k \cdot a_{[N-k-1:0]})$,

where $a_{[N-k-1:0]}$ is extracted as in (4).

3.2 Encoding Mixed Bit&Word Operations

3.2.1 Bit Composition (Bit-blasting)

To relate an integer variable a with the boolean variables of its bitwise decomposition $\underline{A}_{[N-1]}...\underline{A}_{[1]}\underline{A}_{[0]}$, first we need introducing N auxiliary integer variables $a_{[N-1]},...,a_{[0]}$ s.t. $a_{[i]} \in [0, 1]$ for every i, and then add the constraints:

(14) $(a = \sum_{i=0}^{N-1} 2^i \cdot a_{[i]}) \wedge$

(15) $\bigwedge_i (\underline{A}_{[i]} \leftrightarrow (a_{[i]} = 1))$.

Notice that, like in (1), the i-th bit has two representations: the integer variable $a_{[i]}$ represents it as (one bit of) a datapath word, whilst the atomic proposition $\underline{A}_{[i]}$ represents it as a "control" wire. The interface constraints (15) relate the respective values of the two representations. [3]

[3] An alternative encoding for (15) is "$\bigwedge_i (a_{[i]} = ITEX(\underline{A}_{[i]}; 1; 0))$".

3.2.2 Extraction of single bits

The extraction of the i-th bit of a word can be performed by extracting first the least significant subword $a_{[i:0]}$ and hence extracting the most significant bit of $a_{[i:0]}$:

(16) $(a = a_{[i:0]} + 2^{i+1} \cdot a_{[N-1:i+1]}) \wedge$

(17) $(\underline{A}_{[i]} \leftrightarrow (a_{[i:0]} \geq 2^i))$.

3.2.3 Addition and multiplication by constant

Let $\underline{D}$ be the result of the sum between $\underline{A}$ and $\underline{B}$ and the carry-in bit CIN, and let $COUT$ be the carry-out bit. To encode this, we need an extra integer value $cin \in [0:1]$:

(18) $CIN \leftrightarrow (cin = 1)$

(19) $COUT \leftrightarrow (a + b + cin \geq 2^N)$

(20) $d = ITE(a + b + cin \geq 2^N, a + b + cin - 2^N, a + b + cin)$

Notice that the interface constraints (18) and (19) are necessary only if we need carry-in and carry-out being explicitly represented as boolean variables. If not, an alternative encoding [5] is

(21) $(d = a + b + cin - \sigma \cdot 2^N), \quad \sigma \in [0, 1]$.

This is particularly convenient because it prevents MathSAT from splitting on the condition $(a + b + cin \geq 2^N)$, as in (20). Another advantage of (21) is that it can be easily generalized to the case of multiple sums of k words $\underline{D} = \Sigma_{i=1}^{k}\ \underline{A}_i$:

(22) $(d = \Sigma_{i=1}^{k} a_i - \sigma \cdot 2^N), \quad \sigma \in [0, k-1]$.

Similarly, let $\underline{D}$ be the result of the product between $\underline{A}$ and a constant $\underline{C}$. We can encode it as:

(23) $(d = a \cdot c - \sigma \cdot 2^N), \quad \sigma \in [0, c-1]$.

In (21), (22), (23), if a boolean variable C representing overflow is explicitly needed, we can encode this fact by the additional interface constraint

(24) $C \leftrightarrow (\sigma \geq 1)$.

3.2.4 Unary And, Or and Not

The unary and of the word $\underline{A}$, written $\&\underline{A}$, is true iff all the bits of $\underline{A}$ are true. We can encode this fact by:

(25) $(a = 2^N - 1)$.

Similarly, the unary or of the word $\underline{A}$, written $|\underline{A}$, is true iff at least one of the bits of $\underline{A}$ is true. We can encode this fact by:

(26) $(a \geq 1)$.

The unary not of the N-bit word $\underline{A}$, written $!\underline{A}$, is the N-bit word obtained by complementing all bits in $\underline{A}$. We can encode this fact by the expression

(27) $(2^N - 1 - a)$.

Thus, e.g., if $\underline{B}$ is $!\underline{A}$, then we write "$b = (2^N - 1 - a)$".

3.2.5 Bitwise operations by constants

Let $\underline{W_1} := \underline{W}\ op\ \underline{C}$ be the result of a bitwise operation $op \in \{$`&&`, `||`, `^^`$\}$ between a word $\underline{W}$ and a *constant* word $\underline{C}$. We can decompose $\underline{C}$ into a concatenation $\underline{C}^1_k\underline{C}^0_k \ldots \underline{C}^1_0\underline{C}^0_0$ of sequences of 1's and 0's of lengths $l^1_k l^0_k \ldots l^1_0 l^0_0$ respectively,

$$\underbrace{\overset{m^1_k+l^1_k-1}{111\ldots}\quad\overset{m^1_k}{111}}_{\underline{C}^1_k}\underbrace{\overset{m^0_k+l^0_k-1}{000\ldots}\quad\overset{m^0_k}{000}}_{\underline{C}^0_k}\ \cdots\ \underbrace{\overset{m^1_0+l^1_0-1}{111\ldots}\quad\overset{m^1_0}{111}}_{\underline{C}^1_0}\underbrace{\overset{m^0_0+l^0_0-1}{000\ldots}\quad\overset{m^0_0}{000}}_{\underline{C}^0_0},$$

with the intended meaning that $m^j_i + l^j_i - j$ and m^j_i are the indexes respectively of the msb and lsb of $\underline{C}^j_i$, for every i, j, and that $l^j_i = 0$ represents the fact that $\underline{C}^j_i$ is an empty sequence.

If *op* is the bitwise and "`&&`", then we have

(28) $$(w_1 = \sum_{i=0\ldots k,\ l^1_i \geq 0} 2^{m^1_i} \cdot w_{[m^1_i+l^1_i-1:m^1_i]}).$$

If *op* is the bitwise or "`||`", then we have

(29) $$(w_1 = \sum_{i=0\ldots k,\ l^1_i \geq 0} 2^{m^1_i} \cdot (2^{l^1_i} - 1) + \sum_{i=0,\ l^0_i \geq 0}^{k} 2^{m^0_i} \cdot w_{[m^0_i+l^0_i-1:m^0_i]}).$$

If *op* is the bitwise xor "`^^`", then we have

(30) $$(w_1 = \sum_{i=0\ldots k,\ l^1_i \geq 0} 2^{m^1_i} \cdot (2^{l^1_i} - 1 - w_{[m^1_i+l^1_i-1:m^1_i]}) + \sum_{i=0,\ l^0_i \geq 0}^{k} 2^{m^0_i} w_{[m^0_i+l^0_i-1:m^0_i]}).$$

4 Encoding non-linearizable operations

For some RTL operators (e.g., shift by variable, bitwise operations between two variables), or for complex sub-circuits, finding an encoding into integer linear arithmetic is not possible. When so, we have two alternatives: to use a mixed booleanization/linear encoding approach, or to represent the operator as an uninterpreted function.

4.1 Mixed linear encoding and booleanization

When an operator (or sub-circuit) is not directly or fully expressible with linear mathematical constraint, the effort is that of minimizing the booleanization by extracting subparts that can be expressed with linear mathematical constraints, and hence booleanize the remaining part.

A noteworthy example is multiplication. Multiplication is not a linear mathematical operation. We can "linearize" it by extracting the bits of one of the operand and apply shift-and-add. Let $\underline{A}$ a N-bit word, let $\underline{B}$ be a M-bit word and let $\underline{C}$ a (N+M) bit word, $M \geq N \geq 0$. (If $|\underline{A}| \geq |\underline{B}|$, we switch the operands.) First, we decompose $\underline{A}$ into bits as in (14). Then we define the product $\underline{C} = \underline{B} \cdot \underline{A}$, by introducing N auxiliary integer variables sum_i representing $b \cdot a_{[i]}$.

$$(31)\ \ (c = \sum_{i=0}^{N-1} 2^i \cdot sum_i) \wedge \bigwedge_{i=0}^{N-1} (sum_i = ITE(\underline{A}_{[i]}; b; 0)).$$

4.2 Uninterpreted functions

Often, some properties do not depend on the particular semantics of some subcircuit or operator in the circuit. Thus, another alternative (see, e.g., [10]) is trying to encode a non-linear operator (or a generic sub-circuit) as an *uninterpreted function*, that is, to abstract away any information related to the semantics of the operator, and assume only the congruence constraint of the function (that is, every function maps equal values into equal values).

Let $\underline{D}$, $\underline{A}_1$,...,$\underline{A}_k$ be words of size l, l_1,...,l_k respectively, and let $\underline{D} = OP(\underline{A}_1, ..., \underline{A}_k)$, OP being the operator we want to encode. If a property is proved, then it is a correct property, but not vice versa. For every subcircuit described in the form "$\underline{D} = OP(\underline{A}_1, ..., \underline{A}_k)$", OP being the operator we want to encode and $\underline{D}$, $\underline{A}_1$,...,$\underline{A}_k$ being words of size l, l_1,...,l_k respectively, we encode it as the constraint

$$(32)\ \ (d = f_{op}(a_1, ..., a_k)),$$

f_{op} being a fresh k-ary uninterpreted function. This fact guarantees the preservation of congruence, that is, if $(d = f_{op}(a_1, ..., a_k))$, $(e = f_{op}(b_1, ..., b_k))$ and $(a_i = b_i)$ hold for every i, then $(d = e)$ holds.

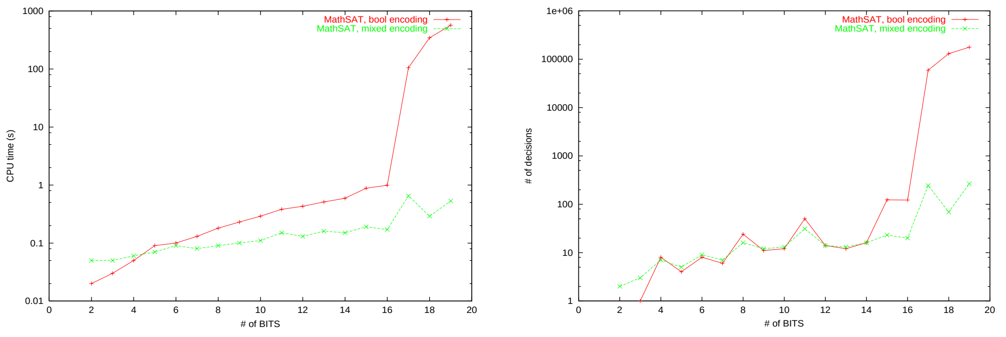

Fig. 1. MathSAT on the $a \cdot b = p^2$ problem. X axis: number of bits. Y axis: CPU time in seconds (left), number of of boolean decisions (right).

5 A simple example

In order to illustrate the benefits of our encoding, we consider the example of a simple RTL circuit, which we call "$ab = p^2$": two integers a and b in $[0..2^N - 1]$ are given; a is decomposed into its N-bit decomposition $\underline{A}$ as in (14); $\underline{A}$ and b are given in input to a shift-and-adder multiplier (31) and the output is compared with the constant integer value p^2, p being the biggest prime number strictly smaller than 2^N. As a comparison, we have provided also a purely boolean version of the encoding, where all the integer values are substituted by arrays of boolean variables, and the sums are substituted by bitwise adders.

We have chosen this example for a few reasons. First, it is extremely simple. Second, it is parametric (and scales up with) the number N of bits of the words used. Third, it belongs to the category of mixed encodings of Section 4.1, so that, it is not an "ideal" situation for our encoding. In fact, it mixes a part of linear arithmetic, due to the sequence of sums performed, with a significant component of boolean control reasoning, due to the booleanization of the operand a which controls the shift-and-add mechanism. Fourth, although the resulting formula is satisfiable, it has only one solution given by $a = b = p$ and by their corresponding bit values, which makes the problem of finding such solution hard enough.

In Figure 1 we compare the performance of MathSAT on the two encodings, in terms of total CPU time (left) and number of boolean decisions taken (right), for increasing values of N. [4] When invoked on a purely boolean formula, MathSAT does not instantiate or call any theory solver, so that, apart for a little overhead in parsing, it behaves like its underlying SAT solver Min-

[4] All tests have been run on a bi-processor `XEON 2.4GHz 1GB RAM` machine on `linux RedHat 9`. We have used MathSAT version 3.2.1 with the default options.

iSAT [8].

We notice that the performances are comparable for small N ($N \leq 16$), as the two encodings force about the same amount of decisions (≤ 100). For bigger N, with the boolean encoding the SAT solver starts branching heavily on boolean variables encoding datapath variables, and performance degrades abruptly; with the mixed encoding, instead, no branching on datapath boolean variables is performed, and performance does not degrade dramatically with N. Comparing the two plots of Figure 1, we notice that the CPU time curve follows directly that of the number of decisions. Thus, the boolean search dominates the global performance, so that the overhead due to handling the linear arithmetic constraints is nearly negligible wrt. that of handling the extra boolean search due to booleanization.

6 Related work and discussion

The problem of RTL verification trying to avoid *bit-blasting*, i.e. the uniform reduction to a problem of propositional satisfiability, has received significant attention in the last years.

The approaches proposed in [5,11] are based on the encoding of bits, bit-vectors and their operators into integer linear programming (ILP) expressions. An ILP procedure is then used to solve the problem. The main difference with our approach is that single bits and relative operators are also handled within the ILP.

In [6,9,7,2], the decision procedure is engineered to be a component of more general reasoning frameworks. For this reason, they rely on various combination schemata (e.g. Nelson-Oppen, Shostak), which makes our approach somewhat simpler. Another other remarkable differences is that we use of a SAT-based technology, while [9,7] are based on extensions of BDDs.

A completely different approach is followed in [10,1]: abstract representations of an RTL circuit are generated by abstracting away information on the datapath, and the resulting encoding is then fed into a propositional SAT tools. The approach in [1] is subject to loss of information, and iterative refinement may be required.

References

[1] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In *Proc. DAC '04*. ACM Press, 2004.

[2] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. DAC '98*. ACM Press, 1998.

[3] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P.van Rossum, S. Ranise, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *Proc. CAV 2005.*, LNCS. Springer, 2005. To appear.

[4] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Proc. TACAS 2005*, volume 3440 of *LNCS*, 2005.

[5] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. ASP-DAC 2002*, pages 741–746. IEEE, 2002.

[6] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Proc. CAV '94*, volume 818 of *LNCS*. Springer, 1994.

[7] D. Cyrluk, M. Oliver Möller, and H. Ruess. An efficient decision procedure for the theory of fixed-sized bit-vectors. pages 60–71, 1997.

[8] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[9] M. Oliver Möller and Harald Ruess. Solving bit-vector equations. pages 36–48, 1998.

[10] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions. In *Proc. 40th Design Automation Conference (DAC)*, 2003.

[11] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: a unified approach to RTL satisfiability. In *Proc. DATE '01.* IEEE Press, 2001.