Arithmetic reasoning in DPLL-based SAT solving

Markus Wedler, Dominik Stoffel, Wolfgang Kunz
Dept. of Electrical & Computer Eng., University of Kaiserslautern/Germany
email: wedler@eit.uni-kl.de

Abstract

We propose a new arithmetic reasoning calculus to speed up a SAT solver based on the Davis Putnam Longman Loveland (DPLL) procedure. It is based on an arithmetic bit level description of the arithmetic circuit parts and the property. This description can easily be provided by the front-end of an RTL property checker. The calculus yields significant speedup and more robustness on hard SAT instances derived from the formal verification of arithmetic circuits.

1 Introduction

Bounded model checking (BMC) [2] has gained increased significance in Electronic Design Automation (EDA). It is used to verify that a digital circuit design meets the desired behavior. In BMC the design of a sequential circuit is unrolled for a finite number of time frames and augmented with the property under verification. This can be translated into a satisfiability (SAT) problem and is thus handled by standard SAT solvers. These solvers will either give a proof of unsatisfiability or a counter example for the property. Figure 1 shows the standard flow for property checking.

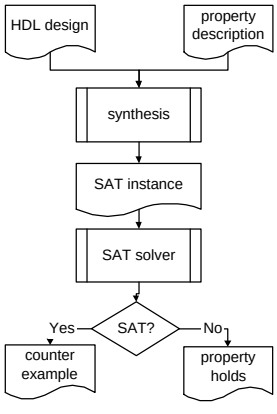

Figure 1. Standard flow for RTL property checking

It is well known that SAT solvers have problems when dealing with instances derived from the verification of arithmetic circuits. Hence, although SAT-based property checking can often be applied successfully to the control part of a design, it typically fails on data paths with large arithmetic blocks. One may resort to incomplete techniques like bit-slicing in order to find bugs in arithmetic units. However, they cannot prove the absence of a bug. Especially, it is very likely to miss errors in corner cases.

This forced the development of automatic bit width reduction techniques like in [6, 7]. But still the resulting model is often too large to be handled by a SAT solver and there are cases where no reduction is possible.

Another idea is to use word-level solvers for integer linear programming (ILP) [3, 5, 13] or constraint logic programming [12]. The problem with word-level solvers is that they do not incorporate the large variety of pruning techniques for the Boolean part of the problem. Therefore, they usually perform poorly on the control part of a design. Thus, a combination of word-level and Boolean solvers has to be developed. This problem is not simple because the different solvers cannot look into each other's non-solution areas. Two promising ways of integrating ILP and SAT have been proposed in [1, 4]. The first uses pseudo-Boolean constraints as clauses in a DPLL style solver and the second uses linear equations as propositions. The method described in this paper is integrated into a standard DPLL style SAT solver [8] but it seems likely that also a framework like [4] could benefit significantly from the proposed concepts. A general problem for all ILP-based solvers is that multiplication leads to non-linear constraints and linearization leads to large and hard to solve ILPs. Therefore, in this work we propose a new reasoning scheme that targets these hard to linearize cases.

The reasoning scheme is based on addition networks. In [10] a method for equivalence checking of multipliers based on arithmetic bit level description of the circuit was proposed. The arithmetic bit level contains partial products and addition networks. To verify that a circuit is a multiplier an arithmetic bit level representation is extracted from the gate netlist. For property checking this extraction is not necessary as the synthesis front-end can easily generate the arithmetic bit level information needed for our reasoning scheme. The arithmetic bit level information will be utilized to prune the search space of the SAT solver. Figure 2 shows the modified flow for property checking.

The rest of the paper is organized as follows: Section 2 introduces an arithmetic reasoning calculus. In Section 3 this calculus is integrated into the DPLL search procedure for SAT. Section 4 reports experimental results from our implementation of the described techniques. Section 5 concludes this paper.

1530-1591/04 $20.00 (c) 2004 IEEE

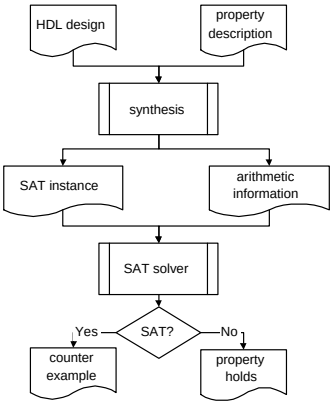

Figure 2. New flow for RTL property checking

2 Arithmetic Reasoning

Addition networks occur in many arithmetic circuits. For example one can find them in digital filters, multiply accumulate units or multipliers. In this section we study a calculus for carry propagation in such addition networks. We shall see that this kind of propagation is much stronger than conventional boolean constraint propagation (BCP). Throughout this paper BCP always refers to the boolean propagation derived by iteratively applying the unit literal rule on the corresponding conjunctive normal form (CNF) of the addition network.

Let us take a look at a multiplier, for example. Table 1 shows the multiplication of two 4 bit numbers. Partial products are added up columnwise. Each column n can produce c_{max}^{n+1} carries $carry_1, ..., carry_{c_{max}^{n+1}}$ that have to be added to the next column, $n+1$. The partial products and carries derived from the previous column are called *addends* of a column. Addends that are not carries are called *primary addends*.

n		8	7	6	5	4	3	2	1
	1	0	1	0	*	0	1	1	0
						0	0	0	0
	+				0	1	1	0	
	+			0	0	0	0		
	+		0	1	1	0			
$carry_1$	+	0	0	0	0	0	0		
$carry_2$	+		0	0	0	0			
$carry_3$	+			0	0				
		0	0	1	1	1	1	0	0

Table 1. Multiplication of two unsigned 4-bit numbers

In the following we consider value assignments on the set of addends A. A *value assignment* is a map $x : A \rightarrow \{0, 1, X\}$. The value of an addend a is called *unspecified* if $x(a) = X$, otherwise it is *specified*. A value assignment that leaves primary addends unspecified is referred to as *partial* assignment. Note that during a run of a SAT solver partial assignments will be generated. For each column n and each partial assignment $\underline{x}$ we denote the number of specified carries with $c_{\underline{x}}^n$. Let us suppose we have the partial assignment $X1X1$ and $1X1X$ on the inputs as shown in Table 2. At least two addends in column 4 are one, thus we know that at least one carry in column 5 has to be generated, hence we have $c_{\underline{x}}^5 = 1$. In a tabular representation

n		8	7	6	5	4	3	2	1
	X	1	X	1	*	1	X	1	X
						1	X	1	X
	+				X	X	X	X	
	+			1	X	1	X		
	+		X	X	X	X			
$carry_1$	+	X	X	X	**1**	X	X		
$carry_2$	+		X	X	X	X			
$carry_3$	+			X	X				
		X	X	X	X	X	X	X	X

Table 2. Partial assignments

as in Table 2 it is simple to derive this information. However, if the addition matrix is implemented by an addition network composed of half adders, this is no longer the case. The reason is that a specific implementation is based on a specific order in which the addends of a column are added up.

To illustrate this, consider the addition network of Figure 3 that could be part of some arithmetic block. The primary addends $A_{1,1}, \ldots, A_{1,4}$ are added up into column 1, the primary addends $A_{2,1}, \ldots, A_{2,3}$ and the carries $C_{1,1}$ and $C_{1,2}$ are added into column 2. As a result, further carries are generated that contribute to columns 3 and 4. $S_1, \ldots, S_4$ constitute the resulting column sums. Let us suppose we have the partial assignment $\underline{x}$ with $\underline{x}(A_{1,1}) = \underline{x}(A_{1,3}) = \underline{x}(A_{2,2}) = 0$ and $\underline{x}(A_{i,j}) = X$ otherwise. By BCP we *cannot* derive the necessary assignment $S_4 = 0$.

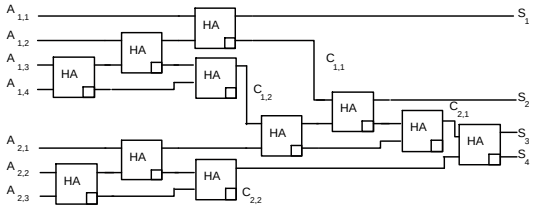

Figure 3. Addition network

In fact, the number of specified carries in column 2, $c_{\underline{x}}^2 = 1$, cannot be observed on the physical carry signals $C_{1,1}$ and $C_{1,2}$ using BCP. However, a key observation is that the structure of the addition network can be rearranged using commutative and associative laws such that BCP specifies $c_{\underline{x}}^2$ carries. In our case we have to swap addends $A_{1,1}$ with $A_{1,4}$ and $A_{2,3}$ with $C_{1,2}$. Then Boolean constraint propagation yields $S_4 = 0$. The resulting network is shown in Figure 4.

The example shows that reordering of the addends in each column of an addition network leads to stronger Boolean constraint propagation. The question is whether we can always find an order of the addends in column n such that $c_{\underline{x}}^{n+1}$ carries become specified (to either one or zero) by BCP in this column. Unfortunately, the answer is no. The addition network of Figure 5 is a counterexample.

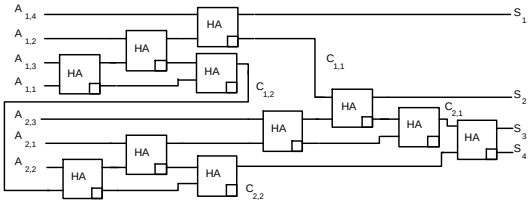

Figure 4. Addition network after swapping addends

No matter how we permute the addends $a_1, .., a_6$, a single zero assigned to one of them will never propagate to any of the carries $c_1, .., c_3$, even though it is mandatory that one of the carries has to be zero.

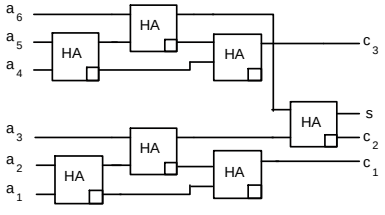

Figure 5. Addition network for counterexample

However, it is still possible to fix this problem. Fortunately, there always exists a functionally equivalent addition network and an order of the addends such that the number of carries identified by Boolean propagation is $c_{\underline{x}}^{n+1}$. In fact, a simple chain of half adders like in Figure 6 fulfills this requirement. This is formalized by Theorem 1 and Theorem 2.

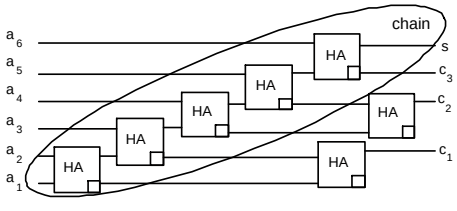

Figure 6. Addition chain

Theorem 1. *Let $A = \{a_1, ..., a_n\}$ be a set of addends of a column n. Further let $x : A \rightarrow \{0, 1, X\}$ be a partial assignment on A, $d := |\{a_i | x(a_i) \neq X\}|$ be the number of specified addends and $s := |\{a_i | x(a_i) = 1\}|$ be the number of addends that are 1. Then, there exists a functionally equivalent addition network such that c carries are identified by BCP, with*

$$c := \begin{cases} d/2 - 1, & \text{if n is odd, d is even and s is odd} \\ (d+1)/2, & \text{if n is even, d is odd and s is even} \\ \lfloor d/2 \rfloor, & \text{otherwise.} \end{cases}$$

Proof: W. l. o. g. we suppose $x(a_i) = 1, i = 1..s$, $x(a_i) = X, i = s + 1..l$ and $x(a_i) = 0, i = l + 1..n$. (Otherwise we permute the inputs of the addition network.) By assumption we know $n - l + s = d$.

As addition network we take a chain of half adders. Two succeeding $carry_{out}$ signals in this chain are excluding each other. Therefore we can add the $carry_{out}$ signals of every $2k$'s and $2k - 1$'s half adder. The sum-outputs of these additional half adders are $carry_{out}$ signals of the addition network. The carries are always zero and will never be used.

We will now show that in this addition network, with the above ordering of the addends, c carries are specified by Boolean propagation. The first $\lfloor s/2 \rfloor$ carries are only affected by the first $s + 1$ addends if s is even, and the first s carries if s is odd. As all of these inputs (except one in the even case) are 1 the ones are propagated to the carries by Boolean propagation. A similar analysis gives us the number of zeros propagated to carries at the end of the chain. Here we have to conduct a case split whether we have an even or odd number of half adders in the chain. We omit this for reasons of space. Please refer to [11]. □

Theorem 2. *Under the conditions of Theorem 1*

$$c_x^{n+1} = c = \begin{cases} d/2 - 1, & \text{if n is odd,d is even and s is odd} \\ (d+1)/2, & \text{if n is even, d is odd and s is even} \\ \lfloor d/2 \rfloor, & \text{otherwise} \end{cases}$$

holds, i.e., c is the number of carries that are specified to either one or zero.

Proof: Is omitted, please refer to [11].

In other words our theoretical results state that appropriate restructuring makes the forward implications in an addition network complete. Hence, backtracks in the SAT solver can only be caused by output constraints and dependencies of the addends.

Our approach heavily exploits the arithmetic bit level information for signal swapping. We have seen that it is not sufficient to only permute the addends of each column of the addition network. Instead we may additionally have to swap a certain addend with a certain partial sum. It has been shown in [9] that we can generate whatever architecture of the addition network is required using a series of signal swaps. In our context this means that we can also always generate the addition chains and addend orders required by Theorem 1. An attractive possibility is to let the front-end of the property checker generate addition chains rather than balanced trees like in the proof of Theorem 1. In this case we can expect that even conventional BCP will specify more carry signals than for non-chain architectures. This is also supported by our experimental results in Section 4. However, in order to identify the exact number of carries additional signal swaps have to be performed. In the chain architecture it is sufficient to consider swapping only addends of a column and not partial sums.

3 Integration into DPLL

Signal swaps change the functions of many internal nodes. If we want to integrate this into a SAT solver this implies that many of the learnt clauses become invalid whenever we perform a swap. Another drawback is that the swap itself is a complex task on a CNF.

Therefore, in our reasoning scheme we want to virtualize signal swapping. The information we want to calculate is how many carries are specified and what the values of the specified carries are. This can be done by counting the specified carries for each column. From the number of specified addends and the number of specified carries of the previous column we can calculate the number of specified carries and their values, like in the proof of Theorem 2.

In our example of Figure 3 we had the partial assignment $\underline{x}(A_{1,1}) = \underline{x}(A_{1,3}) = \underline{x}(A_{2,2}) = 0$ and $\underline{x}(A_{i,j}) = X$ otherwise. We know that two out of four addends in the first column are set to zero. This implies that one carry is specified to zero and the other is unspecified. Note that we do not need to know whether $C_{1,1}$ or $C_{1,2}$ is specified. In the second column we know that one addend and one carry is specified to zero. This implies again that one of the carries in this column is zero. Lastly, in the third column only these carries are added up and no carries are produced. Hence, the carry S_4 of this column has to be zero.

```
dpll() {
  preprocess();
  while (true) {
    decide_next_branch();
    status = deduce();
    if (status == conflict) {
      blevel= analyze_conflict();
      if (blevel == 0)
        return UNSAT;
      else backtrack(blevel);
    }elseif (status == SAT) return SAT;
    else break;
} }
```

Table 3. Pseudocode of the DPLL algorithm

Now we study how to integrate the above reasoning scheme into the DPLL procedure. Table 3 presents the pseudo-code of the top-level solving routine. We will describe how the key routines of this algorithm have to be modified to incorporate arithmetic reasoning. Our solver inherits all the features of modern SAT solvers.

3.1 Preprocessing

At the end of the preprocessing routine of the base solver we read the arithmetic bit level information prepared by the front-end of the property checker. We mark all the variables that are addends or sums in an addition network. In the following these variables are referred to as arithmetic variables. For each of them we store:

- the addition network the variable belongs to,
- the column of the network it belongs to,
- whether the variable is the result or an addend of the column.

For each addition network we set up a table storing the following information for each column:

maxSumCount	number of primary addends in the column.
maxCarryCount	number of carries derived from the previous column.
sum	number of primary addends set to 1 in the current assignment.
sumCount	number of primary addends set to either 0 or 1.
carrySum	number of carries from the previous column specifi ed to be 1 under the current assignment.
carryCount	number of carries from the previous column specifi ed under the current assignment.
result	expected result for the column.

These variables relate to our notions of Section 2 as follows: *maxCarryCount* $= c^n_{max}$, *sum* + *carrySum* $= s$, *sumCount* + *carryCount* $= d$ and *carryCount* $= c^n_x$. Note that *maxSumCount* and *maxCarryCount* do not change during the solving process but are fixed for a given addition network. For any addition network *carrySum* and *carryCount* of the first column are always zero. All the dynamic values are initialized with zero. The extra memory needed for these additional data structures is negligible compared to the memory used by the clause database. Moreover, it remains constant during the whole solver run.

3.2 Decision rule

When selecting the next branching variable, we prefer arithmetic variables if they occur among the first k variables that the original decision rule would take. k is a user defined value. A typical value is $k = 100$. Note that scanning such a small constant number of variables does not corrupt the overall runtime of the solver.

3.3 Update of the sum counts

Every SAT solver has some base routines *set_value(variable value)* and *unset_value(variable)* that are called during branching, BCP and backtracking whenever the value of a variable changes. The data structures used in the solver allow them to perform in constant time. Usually they only make a few assignments to internal memory that represents the state of a variable. At the end of these procedures we check whether the variable is marked to be arithmetic. In this case we update *sum* and *sumCount* in the corresponding table.

In the worst case these modifications lead to only a small number of additional operations. Therefore, they do not corrupt the runtime behavior of the solver. On a modern pipelined CPU the difference will not even be measurable.

3.4 Deduction

Deduction is based on the standard techniques employed in state-of-the-art SAT solvers. In Table 4 we refer to the deduction technique inherited from the base solver as *old_deduce()*. If *old_deduce()* did not find a Boolean conflict and we have not yet detected an arithmetic conflict, we update *carrySum* and *carryCount* for all columns of all addition networks. This is done in routine *calculate_carries()* and is based on the same analysis as in the proof of Theorem 2. We stop when the first arithmetic conflict is detected. This is the case if we detect a column that is full, i.e., all addends and carries are specified, and inconsistent, i.e., the sum does not match the result. After an arithmetic conflict is detected for the first time the current decision level is stored as *arith_dl*. Table 4 presents the pseudocode for the new deduction routine.

```
deduce(arith_status) {
  status = old_deduce();
  if (status != conflict and arith_status !=conflict)
    for each (addition network)
      for each (column ) {
        calculate_carries();
        if (column is full and inconsistent) {
          arith_dl = d_level();
          arith_status =conflict;
          return (status, arith_status);
} } }
```

Table 4. Pseudocode of deduction routine

3.5 Conflict analysis

The new conflict analysis has to handle two different kinds of conflicts. When a Boolean conflict occurs we can use the standard mechanisms to generate a conflict clause. When an arithmetic conflict occurs this is not the case. The problem is that there are no unsatisfied clauses. Hence, we have no starting clause for conflict analysis. Therefore, we generate a conflict clause that forbids the current assignment and perform chronological backtracking.

Figures 7 and 8 show two situations possible in the decision tree. For the situation shown in Figure 7 the arithmetic conflict is detected after one of the early decisions. Boolean conflict analysis at a deeper level of the search tree cannot detect this reason for the Boolean conflict. In this case we can benefit from the arithmetic conflict.

However, we also encounter situations as in Figure 8. Here the arithmetic conflict occurs only a few levels before the Boolean conflict and Boolean conflict analysis determines a backtrack level far beyond that level. In this case Boolean conflict analysis gives us much more benefit.

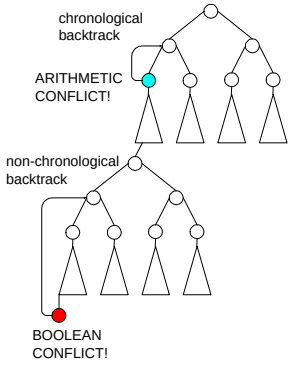

Figure 7. Decision tree

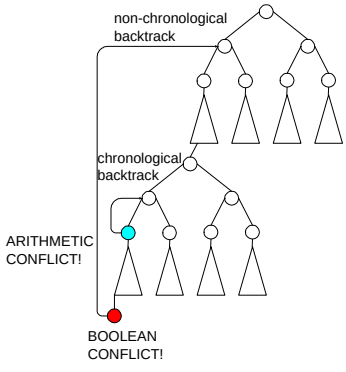

Figure 8. Decision tree

We would like to benefit in both situations. Therefore we do not backtrack when the arithmetic conflict occurs. Instead we remember the current decision level (*arith_dl* in *deduce()*) when the arithmetic conflict is detected for the first time and proceed deeper into the decision tree until a Boolean conflict occurs. After the Boolean conflict analysis we can decide whether it is more beneficial to backtrack to the arithmetic or the Boolean decision level.

4 Results

The ideas presented in this paper have been implemented on top of the well-known SAT solver zChaff [8]. In our experiments we took a 16x16 and a 20x20 multiplier with chain architecture and a 20x20 multiplier with tree architecture and tried to justify 200 randomly selected output vectors for all of them. Furthermore we created two miters containing 8x8 and 10x10 multipliers with chain architecture and checked the outputs for equivalence. Last we compared both solvers on 98 random sat formulas without any arithmetic information. Table 5 summarizes the results. It is organized as follows: Column 1 contains the name of the instance class and column 2 specifies whether the instances

are satisfiable. In column 3 the number of instances in the class is specified. Columns 4 and 5 specify the maximum speedup/slowdown of our solver compared to zChaff. Column 6 compares the sum of all run times in the class needed by our solver against the sum of the times needed by zChaff.

Class	Result	#	Max	Max	**overall**
			speedup	slowdown	**Speedup**
8x8Miter	UNSAT	16	3.3	1.4	**2.0**
10x10Miter	UNSAT	20	2.2	2.6	**1.8**
16x16chain	SAT	22	8.0	11.8	**1.4**
16x16chain	UNSAT	178	2.8	11.8	**1.4**
20x20chain	SAT	35	166.8	42.9	**4.9**
20x20chain	UNSAT	165	29.8	3.9	**3.5**
20x20tree	SAT	35	770.4	69.3	**14.1**
20x20tree	UNSAT	165	32.5	-1.7	**7.8**
random	SAT/UNSAT	98	1.1	1.5	**-1.03**

Table 5. Experimental results

Figures 9 visualizes the runtime distributions for both solvers. The instances are plotted in ascending order of the CPU times zChaff needed to complete the instances with chain architecture. We show results for chain as well as tree architecture instances. The plot clearly demonstrates that zChaff performs significantly better on the chain architecture than on the tree architecture of the addition network. However, even these results are clearly outperformed by the proposed extensions to the basic SAT procedure. The right margin of the plot impressively shows the increase in robustness. The results on the random sat instances indicate that there is no loss of performance when no arithmetic information is available.

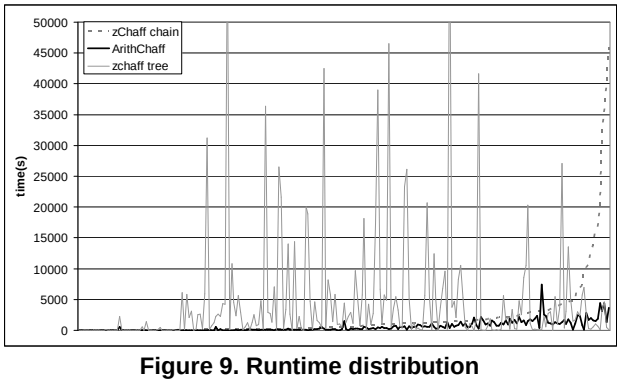

Figure 9. Runtime distribution

5 Conclusion

We have presented an arithmetic reasoning as a technique to speed up SAT solving for instances derived from circuits containing addition networks. Our results show that incorporating arithmetic reasoning into DPLL yields significant speedup and more robustness. The presented technique is orthogonal to all other pruning techniques used in modern SAT solvers. Furthermore it can be easily integrated into current verification environments. In those cases where no arithmetic information is available the solver exactly performs like its base solver without any run time penalty.

References

[1] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. Conference on Automated Deduction (CADE)*, pages 195–210, 2002.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. Intl. Design Automation Conference (DAC-99)*, pages 317–320, June 1999.

[3] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. Asia and South Pacific Design Automation Conference (ASPDAC-02)*, Bangalore, India, 2002.

[4] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. Design Automation Conference (DAC-03)*, pages 830–835, 2003.

[5] F. Fallah, S. Devadas, and K. Keutzer. Functional vector generation for HDL models using linear programming and boolean satisfiability. *IEEE Transactions on CAD*, CAD-20(8), 2001.

[6] P. Johannsen. BOOSTER: Speeding up RTL property checking of digital designs by word-level abstraction. In *Proc.Intl. Conf. Computer Aided Verification(CAV-01)*, pages 373–377, July 2001.

[7] P. Johannsen and R. Drechsler. Formal verification on the RT level computing one-to-one design abstractions by signal width reduction. In *Proc. IFIP International Conference on Very Large Scale Integration(IFIP VLSI-SOC 2001)*, Montpellier, France, 2001.

[8] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Intl. Design Automation Conference (DAC-01)*, pages 530–535, June 2001.

[9] I. Neumann, D. Stoffel, M. Berkelaar, and W. Kunz. Layout driven synthesis of data path circuits using arithmetic reasoning. In *Proc. International Workshop on Logic and Synthesis*, pages 1–6, Laguna Beach, California, USA, May 2003.

[10] D. Stoffel and W. Kunz. Verification of integer multipliers on the arithmetic bit level. In *Proc. International Conference on Computer-Aided Design (ICCAD-01)*, pages 183–189, San Jose, CA, November 2001.

[11] M. Wedler, D. Stoffel, and W. Kunz. Arithemetik reasoning in DPLL-based SAT solving. Technical Report EIS-11-03-1, Dept. of Electrical and Computer Engineering, University of Kaiserslautern, Germany, 2003.

[12] Z. Zeng, M. Ciesielski, and B. Rouzeyre. Functional test generation using constraint logic programming. In *Proc. IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2001)*, Montpellier, France, 2001.

[13] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A unified approach to RTL satisfiability. In *Proc. Conference on Design, Automation and Test in Europe (DATE-01)*, Munich, Germany, 2001.