SWORD: A SAT like Prover Using Word Level Information

Robert Wille Görschwin Fey Daniel Große Stephan Eggersglüß Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{rwille,fey,grosse,segg,drechsle}@informatik.uni-bremen.de

***Abstract*— Solvers for Boolean *Satisfiability* (SAT) are state-of-the-art to solve verification problems. But when arithmetic operations are considered, the verification performance degrades with increasing data-path width. Therefore, several approaches that handle a higher level of abstraction have been studied in the past. But the resulting solvers are still not robust enough to handle problems that mix word level structures with bit level descriptions.**

In this paper, we present the satisfiability solver SWORD – a *S*AT like solver that facilitates *word* level information. SWORD represents the problem in terms of modules that define operations over bit vectors. Thus, word level information and structural knowledge become available in the search process. The experimental results show that on our benchmarks SWORD is more robust than Boolean SAT, K*BMDs or SMT.

I. Introduction

The number of elements integrated within digital circuits grows exponentially and this trend is going to continue for at least another 10 years. Already today millions of gates are integrated in a single circuit. Throughout the design flow for such complex systems, techniques to represent and manipulate the function are needed. In particular, to formally verify the correctness of a circuit with respect to all design states and input sequences, techniques for symbolic function manipulation are applied.

Current state-of-the-art tools for formal verification use Boolean techniques like *Binary Decision Diagrams* (BDDs) [1], *AND-Inverter-Graphs* [2] and provers for *Boolean Satisfiability* (SAT) [3], [4]. No word level information such as knowledge about arithmetic operations or structural knowledge is directly used for function manipulation. As a result, the performance of verification tools degrades with increasing data-path width.

For this reason, approaches to exploit such high level information have been proposed in the past [5], [6], [7]. But pure word level approaches suffer from complexity problems when irregularities in the word level structure occur, e.g. bit slicing [8]. The recent concept of *Satisfiability Modulo Theories* (SMT) [9], [10], [11], [12] is more powerful since multiple provers are combined, but still structural information is not available. Related work is discussed in more detail in Section II and empirically compared in Section V.

In this paper, we propose SWORD – a *S*AT-like prover that uses *word* level information and also resembles the structure of the original problem. Internally, the problem is represented as a composition of modules; each module is defined over bit vectors and enforces the constraints for a word level operation on the corresponding Boolean variables. The main advantages of this approach are the following:

- *Compact problem representation:* The composition of word level modules is a much more compact representation than the transformation to Boolean constraints.
- *Knowledge about structure and semantics:* This knowledge is determined by the position of a module within the problem instance and the type of a module. Such information helps to predict the impact of a decision or of learned information during the search process more accurately.
- *Efficient reasoning:* Different types of modules require different reasoning procedures and decision heuristics to allow for an efficient search procedure. These procedures are designed for each type of module individually in the proposed framework.

Thus, SWORD combines the advantages of a Boolean proof procedure with the power of word level knowledge. The proposed solver is empirically compared to K*BMDs [6] as a word level decision diagram, the Boolean SAT solver MiniSat [4] and the SMT solver Yices [12].

II. Related Work

Several approaches to incorporate word level information in the proof process have been proposed so far. BDDs have been generalized to the word level quite early [5] resulting in K*BMDs [6] as a very general form. These diagrams can represent word level multiplications very efficiently, but whenever bit nibbling occurs – as is common practice in circuit descriptions – the performance degrades. In fact, *BMDs may be exponentially large for certain functions [8].

A different approach is the transformation of the problem into *Integer Linear Programming* (ILP) constraints [7]. But the same limitations to pure word level descriptions have been observed. A pure ILP-based approach is often too slow for real world applications.

Combining Boolean provers and word level provers seems to be more promising. The framework proposed in [13] is based on an ATPG engine that is enhanced by arithmetic word level primitives. An arithmetic constraint solver is applied to validate bit level assignments on the circuit. But the powerful learning concepts known from Boolean SAT are not incorporated.

Due to the tremendous improvements in the performance of provers for Boolean SAT in the recent past [14], [15], [16], several researchers investigated the combination of SAT with other proof techniques, i.e. *Satisfiability Modulo Theories* (SMT) [9], [10],

[11], [12]. An SMT solver integrates a Boolean SAT solver with another solver (or multiple solvers) for specialized theories. Usually, the SAT solver works on an abstract representation of the problem and steers the overall search process. Each satisfiable assignment for the Boolean SAT problem has to be validated on the concrete problem using the theory solver. The solver proposed in [17] can be seen as a specialized SMT solver for bit vector logic. Tightly coupling the different solvers, especially to enforce learning due to conflicts resulting from partial assignments and to efficiently carry out implications, is a challenge in this area. Usually, validating a given SAT assignment by using the theory solver is very time consuming. Therefore the overall performance is limited by the performance of the theory solver. In our framework no theory solvers are needed. Moreover, structural information about the original problem is available.

A very general theoretical framework for hierarchical SAT solving was presented in [18]. There, the problem is also decomposed into modules, where each module may have different implication procedures. But no experimental evidence was given and no hints for an implementation were provided.

Nonetheless our solver works similar to such a hierarchical solver. Besides specialized implication procedures also dedicated decision heuristics are applied for different types of modules.

III. Boolean SAT Solving

Our algorithm inherits the basic structure of a classical algorithm to solve a problem instance of Boolean *Satisfiability* (SAT) [14]. Therefore we briefly review the techniques applied in Boolean SAT solvers.

A. Basic Algorithm

The SAT instance is represented as a Boolean formula in *Conjunctive Normal Form* (CNF), which is given as a set of clauses; each clause is a set of literals and each literal is a propositional variable or its negation.

The basic search procedure to find a satisfying assignment is shown in Figure 1 and has the structure of the DPLL algorithm [3]. Instead of simply traversing the complete space of assignments, intelligent decision heuristics, conflict based learning and sophisticated engineering of the implication algorithm lead to an effective search procedure. The description follows the implementation of the procedure in modern SAT solvers. While there are free variables left (a), a decision is made (c) to assign a value to one of these variables. Then, implications are determined due to the last assignment by *Boolean Constraint Propagation* (BCP) (d). This may cause a conflict (e) that is analyzed. If the conflict can be resolved by undoing assignments from previous decisions, backtracking is done (f). Otherwise the instance is unsatisfiable (g). If no further decision can be done, i.e. a value is assigned to all variables and this assignment did not cause a conflict, the CNF is satisfied (b). In the following the *decision level* d denotes the number of variables assigned by decisions in the current partial assignment, i.e. neglecting variable assignments due to implications.

B. Limits of Boolean SAT

Due to the translation of the problem into CNF, the power of BCP as an implication engine and the efficiency of learning are limited. In the verification domain, the original problem is usually given at the word level. Operations are defined over bit vectors. Each Boolean variable that is visible in a bit vector at this level is called *module variable* in the following. The translation of word level operations over bit vectors of *module variables* into CNF involves the creation of a large number of *auxiliary variables* [19]. The dependencies between these variables are modeled by constraints in terms of clauses.

Example 1. *Consider an $n \times n$-multiplier. On the word level, $4n$ module variables are needed for the bit vectors of the operands and the result.*

On the other hand, the multiplier can be represented by n^2 AND gates [20], i.e. the number of auxiliary variables is in $\theta(n^2)$. A single gate can be modeled by three clauses for each element. Therefore the multiplier can be represented by a CNF with $\theta(n^2)$ clauses[1].

Simplified, all these auxiliary variables have to be considered during BCP; but implications on auxiliary variables do not yield a reduction of the search space for the original problem. Moreover, conflict clauses may be derived, that are defined over auxiliary variables only – again without pruning the search space of the original problem. In principle, this problem can be prevented by introducing additional clauses, that describe the implications on module variables directly, but then the translation becomes inefficient due to a large number of clauses.

IV. Using Word Level Information

In this section we describe the architecture of SWORD and how word level information can be used during the solve process. Therefore, we first explain the representation of the problem and present the overall algorithm. Afterwards the utilization of word level information in decision making, the implication engine and conflict analysis are explained in more detail.

A. Representation

SWORD represents the problem in terms of so called *modules*. Each module defines an operation over bit vectors of *module variables*. Each module variable is a Boolean variable.

Example 2. *Figure 2 shows an equivalence checking problem in terms of a miter circuit. A multiplier is compared to a realization that sums up the partial products.*

[1]More efficient translations may be available, but the problem instance still grows.

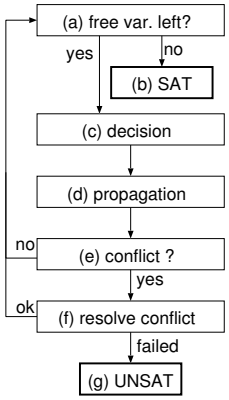

Fig. 1. DPLL

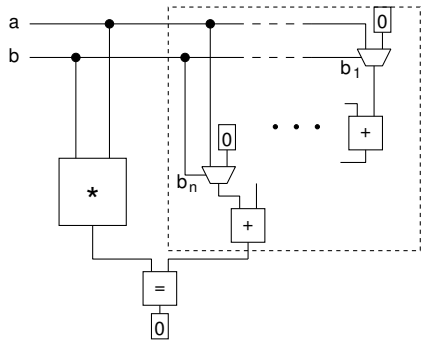

Fig. 2. Miter for a multiplier

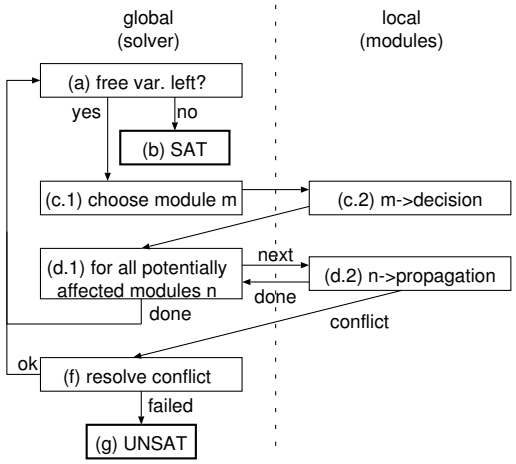

Fig. 3. Algorithm

SWORD represents this problem by using one module representing a multiplier, $n-1$ modules representing an adder, n modules representing a multiplexor and one module representing a comparator. No auxiliary variables are needed.

B. Overall Algorithm

The overall algorithm of SWORD is shown in Figure 3. This algorithm is similar to the DPLL procedure as applied in standard SAT solvers: While free variables remain (a) a decision is made (c), implications resulting from this decision are carried out (d), and if a conflict occurs, it is analyzed (f). The important difference is that SWORD has two operation levels: the *global* algorithm controls the overall search process and calls the *local* procedures of modules for decision and implication. Thus, decision making and implication engine can be adjusted for each type of module.

In more detail, the solver first chooses a particular module based on a *global decision heuristic* (c.1). Then, this module chooses a value for one of its variables according to a *local decision heuristic* (c.2). Afterwards, the solver calls the *local implication procedures* (d.2) of all modules that are potentially affected (d.1) by the previous decision or implication. Here a *variable watching scheme* similar to the one presented in [15] is used, which can efficiently determine these modules. The chosen modules imply further assignments and detect conflicts.

C. Decision Strategies

1) Global Decision: The global decision procedure chooses a module, that assigns a value to one of its connected module variables. So the global decision procedure has to decide, which module will make the best decision, i.e. which decision of a module leads to as many implications as possible. Therefore a (global) heuristic is employed to decide which modules are "more important" than others. To determine the importance of a particular module, semantical information such as the type or structural information such as the position within the overall problem are available.

Example 3. *Again, consider the miter circuit shown in Figure 2. In this example the primary inputs and the outputs of the multiplier module are considered more important than, for example, the select input of one of the multiplexors. Therefore, the global decision heuristic selects the multiplier module first.*

To realize this efficiently, the global decision heuristic currently uses a static priority based on the type of the module. Here, more complex modules (e.g. multipliers) are considered as being more important and, therefore, are selected for a decision with a higher priority than less complex modules. The complexity is measured in the number of two-input gates needed to describe a module. Furthermore the priority of a particular module can be increased/decreased when it is located near to the primary inputs/outputs or the objective. By this, each global decision can be done very efficiently, because no complex data manipulation is necessary.

2) Local Decision: The local decision procedure of a module assigns a value to one of its module variables. The impact of a particular decision depends on the type of a module. Therefore different strategies are applied for different types of modules. For example, a module representing a multiplier uses a different heuristic than a module representing an AND gate. In the following an adder exemplifies the local decision procedures of SWORD. This type of module is simple enough to be explained within the page limitation, but provides some interesting insight.

An n-bit adder $\texttt{ADD} : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}^{n+1}$ is considered, which is represented by a module in SWORD. The module variables connected to this module are given by $a_{n-1}, \ldots, a_0$ and $b_{n-1}, \ldots, b_0$ that represent the inputs of the adder and $o_n, \ldots, o_0$ that represent the outputs.

For an adder, assigning some variables a_i, b_i or o_i (with $n > i \geq 0$) while variables a_j, b_j or o_j (with $i > j \geq 0$) are still unassigned, often does not allow to imply values for the outputs. In contrast, when all of the least significant bits of both operands are given, the

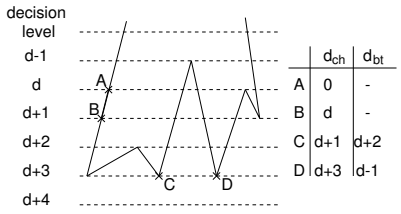

Fig. 4. Search tree and decision levels

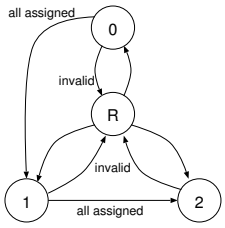

Fig. 5. FSM for an adder

corresponding bits of the outputs can be determined. Therefore the variable representing the least significant unassigned bit is assigned first.

From an implication point of view, the local decision procedure is realized as a *Finite State Machine* (FSM). This allows to carry out decisions efficiently. The FSM has $n+1$ states and is in state i ($n > i \geq 0$) when all variables with lower significance than i are assigned, i.e. a_j, b_j and o_j ($i > j \geq 0$) are assigned. Thus, if the FSM is in state i, only the variables a_i, b_i and o_i are considered. If all of these variables are assigned, the FSM proceeds to state $i+1$. Otherwise at least two of these variables are unassigned (because an implication is carried out when only one variable is unassigned, as explained in Section IV-D.2).

An additional state R is needed to recalculate the state when it was invalidated: Due to backtracking the state of the local FSM of a module may be invalidated because currently assigned variables may become unassigned. This is recognized by tracking the decision level. The decision level of the last state transition, i.e. since the last change of a state, is stored in d_{ch} and the lowest decision level that has been reached after a backtrack intermediately is stored in d_{bt}. The state of the FSM may only be invalidated when $d_{bt} < d_{ch}$.

Example 4. *Figure 4 illustrates this mechanism. The search tree is indicated by the plain line and the decision levels that are reached are also shown. A transition of the FSM is indicated by a cross. The table shows the values of d_{ch} and d_{bt} before the transition is done. The first transition occurs at* A *and d_{ch} is changed from 0 to d; d_{bt} is uninitialized. At* B *the decision level has increased; the state is still valid; d_{ch} is updated to $d+1$. Due to a backtrack d_{bt} is set to $d+2$. Thus, at C the state from decision level $d+1$ is still valid. In contrast, when transition D is done, the state is potentially invalid and has to be recalculated.*

The resulting FSM for a 3-bit adder is shown in Figure 5; only state transitions are indicated, internal variables are not shown.

D. Implication Engine

The implication engine is also divided into a global part and local procedures that are dedicated to the type of a module.

1) Detection of Affected Modules: Globally those modules that may be affected by a previous decision or implication have to be identified. This is done by a variable watching scheme. Currently, a conservative approach is applied: the local propagation procedure of each module that contains a variable that has been assigned is called. Such a static scheme is efficient, because module variables usually only connect to a few modules – often only two modules.

2) Local Implication: The local implication procedures only consider the connected module variables for the propagation of values. For efficiency these procedures do not determine all implications that are possible, but only those that can be derived efficiently. Again, the local implication procedure of an adder exemplifies the local implication procedures.

The implication procedure works similar to the decision procedure: If, for example, the input bit a_i and the output bit o_i and all less significant input bits (a_j and b_j with $i > j \geq 0$) are assigned, the third variable (b_i in the example) can be implied. This implication procedure does not guarantee to detect implications on higher significant bits and is therefore not too powerful. But in most cases implications on these bits are improbable.

The implication procedure relies on the same FSM that is used for decisions. Additionally, the carry bits $c_{n-1}, \ldots, c_0$ are internally updated at each state transition. In state i ($n > i \geq 1$) carry bit c_{i-1} is also given. Therefore an implication can be carried out efficiently based on the current state i, the value of the carry bit c_{i-1} and the values of the module variables a_i, b_i, o_i.

Note, due to the implication procedure a conflicting assignment may not be detected directly. But when the FSM reaches state n, i.e. all module variables are assigned, the consistency of the assignment will be validated. However, due to the order of decisions conflicts are usually detected early. The mechanisms for conflict analysis are explained in detail in the next section.

E. Conflict Analysis

In SWORD conflict analysis and learning are quite similar to the classical approach of a SAT solver. Upon detection of a conflict, the module returns the conflicting variables to the global solve process. Then, conflict analysis is carried out. Currently we adapted the implementation of MiniSat [4]. Because SWORD does not work in terms of clauses, a separate *implication graph* is stored globally. Each module updates this graph when an implication is carried out. The learned information is stored in terms of clauses as in standard SAT solvers. Therefore an additional clause module exists, which handles all clauses generated by conflict

analysis (and applies the known state-of-the-art SAT techniques).

The conflict graph keeps track of the reasons for a particular assignment. Thus, the identification of a reason is crucial in this context. The smaller the reason, the smaller the conflict clauses and the more effectively the search space is pruned. Again, an adder is facilitated to give an idea of how the implication graph is created.

Example 5. *Assume, o_i is implied based on the internal value of c_{i-1} and the module variables a_i and b_i. Furthermore, due to previous assignments $a_{i-1} = 0$ and $b_{i-1} = 0$, the reasons for these assignments are already stored in the implication graph. In this case input bits with lower significance than $i-1$ do not influence the value of o_i, because no carry bit is propagated beyond $i-1$. Thus, the four variables a_i, b_i, a_{i-1} and b_{i-1} are identified as the reason for the implication on o_i. The four edges (a_i, o_i), (b_i, o_i), (a_{i-1}, o_i) and (b_{i-1}, o_i) are added to the implication graph. Note, that the reasons for $a_{i-1} = 0$ and $b_{i-1} = 0$ are already stored in the graph.*

Like in standard SAT solvers, only conflict clauses up to a certain length are learned. The ratio behind this heuristic is that short clauses prune a large part of the search space while longer clauses are less valuable.

Semantical knowledge is also exploited in this process. For example, a conflict clause is not learned if it contains variables that are associated to a complex module like a multiplier – in this case only backtracking is carried out. This heuristic is motivated by the observation that usually a large number of clauses is learned that describe the behavior of a multiplier which causes memory overhead but does not speed up the search.

V. Experimental Results

This section provides experimental results for SWORD in comparison to the Boolean SAT solver MiniSat [4], K*BMDs [6] using the package of [21] as a representative of pure word level approaches, and the SMT solver Yices [11], [12].

All experiments were carried out on an AMD Athlon64 3500+ (Linux, 2.2 GHz, 1 GB). We considered different benchmark problems. In the following, the name indicates the type of the problem. The prefix *ec_* indicates equivalence checking of a multiplier (*mul_*) on the word level with another multiplier that is given as word level module (*mul_*), as sum of partial products (*pp_*), or as gate level description (*gt_*), respectively. Thereby, a miter circuit is used. In some cases the least significant bit was ignored in the miter (indicated by *li_*) and in other cases a fault was injected at the gate level to create a satisfiable instance (indicated by *ft_*). The prefix *pc_arith* indicates a property checking problem that contains arithmetic modules. Finally, a number indicates the bit width of the data path.

Table I provides run times for K*BMDs, SWORD and Yices, while Table II shows results in comparison to MiniSat. An *x* in column *sat* indicates whether the problem instance is satisfiable. For each benchmark the number of variables to represent the problem, the number of clauses for MiniSat and the number of modules for SWORD are given in columns *var*, *cls* and *mod*, respectively. The memory requirements and the CPU time in seconds are provided in columns *mem* and *time*. Finally, the improvement in run time of SWORD over MiniSat is shown in column *imp*.

TABLE I
Comparison to word level solver

circuit	*K*BMD*	*SWORD*	*SMT*
ec_mul_mul_7	<0.01s	0.35s	<0.01s
ec_mul_mul_8	<0.01s	1.67s	<0.01s
ec_mul_mul_9	<0.01s	8.02s	<0.01s
ec_mul_mul_10	<0.01s	37.09s	<0.01s
ec_mul_pp_7	0.01s	0.62s	15.83s
ec_mul_pp_8	0.01s	3.10s	105.56s
ec_mul_pp_9	0.01s	15.54s	>500s
ec_mul_pp_10	0.01s	59.85s	>500s
ec_mul_gt_7	3.48s	0.91s	10.93s
ec_mul_gt_8	13.60s	4.69s	82.40s
ec_mul_gt_9	53.45s	23.20s	>500s
ec_mul_gt_10	202.31s	113.48s	>500s
ec_mul_mul_li_7	>500s	0.34s	0.29s
ec_mul_mul_li_8	>500s	1.66s	1.96s
ec_mul_mul_li_9	>500s	7.95s	58.15s
ec_mul_mul_li_10	>500s	37.01s	>500s
pc_arith_a_6	0.5s	0.36s	<0.01s
pc_arith_a_7	2.1s	1.72s	<0.01s
pc_arith_a_8	8.7s	8.21s	<0.01s
pc_arith_a_9	35.8	37.83s	<0.01s
pc_arith_b_10	1.69s	1.42s	0.07s
pc_arith_b_11	3.18s	4.68s	0.15s
pc_arith_b_12	6.36s	12.24s	0.34s
pc_arith_b_13	12.82	30.91s	0.96s

As expected K*BMDs performs very well on pure word level problems and outperform SWORD in this case (e.g. benchmark set *ec_mul_mul*). But when the description is provided at the bit level the performance degrades significantly (*ec_mul_gt*). Furthermore, bit level operations cannot be handled efficiently (*ec_mul_mul_li*). Yices also handles the pure word level problems extremely efficient. But again, when word level and lower level descriptions are mixed, the performance degrades. On these benchmarks SWORD is more robust. In comparison to MiniSat SWORD requires less memory and is significantly faster (except benchmark set *pc_arith_b*). In the best case up to three orders of magnitude can be achieved.

VI. Conclusions

We presented the satisfiability solver SWORD that uses a SAT like algorithm and exploits word level information in the search process. SWORD works on a representation of the problem in terms of modules. This yields a powerful framework for decision making, implications and conflict analysis. Experimental results show on our benchmarks, that SWORD is more robust than other approaches that were considered here.

A task for future work is developing techniques for automating the creation of modules for SWORD. Furthermore, the application to other problem domains

TABLE II
COMPARISON TO BIT LEVEL SOLVER

		MiniSat				*SWORD*				
circuit	sat	*var*	*cls*	*mem*	*time*	*var*	*mod*	*mem*	*time*	*imp*
ec_mul_mul_7		519	1766	3.98MB	2.02s	43	3	2.73MB	0.35s	5.77
ec_mul_mul_8		687	2348	4.50MB	10.79s	49	3	2.73MB	1.67s	6.46
ec_mul_mul_9		879	3014	5.65MB	54.96s	55	3	2.73MB	8.02s	6.85
ec_mul_mul_10		1095	3764	8.45MB	461.44s	61	3	2.73MB	37.09s	12.44
ec_mul_pp_7		1012	3381	4.24MB	3.98s	228	17	2.73MB	0.62s	6.41
ec_mul_pp_8		1331	4460	5.00MB	25.76s	292	19	2.73MB	3.10s	8.30
ec_mul_pp_9		1694	5689	6.93MB	189.24s	364	21	2.73MB	15.54s	12.17
ec_mul_pp_10		2101	7068	>10.16MB	>500s	444	23	2.86MB	59.85s	>8.35
ec_mul_gt_7		519	1766	3.98MB	2.02s	274	246	2.73MB	0.91s	2.21
ec_mul_gt_8		687	2348	4.50MB	10.79s	360	328	2.86MB	4.69s	2.30
ec_mul_gt_9		879	3014	5.65MB	54.96s	458	422	2.86MB	23.20s	2.36
ec_mul_gt_10		1095	3764	8.45MB	461.44s	568	528	2.86MB	113.84s	4.05
ec_mul_mul_li_7		518	1761	3.99MB	2.03s	43	3	2.73MB	0.34s	5.97
ec_mul_mul_li_8		686	2342	4.36MB	7.95s	49	3	2.73MB	1.66s	4.78
ec_mul_mul_li_9		878	3009	5.90MB	88.88s	55	3	2.73MB	7.95s	11.17
ec_mul_mul_li_10		1094	3759	8.11MB	409.51s	61	3	2.73MB	37.01s	11.06
ec_mul_gt_ft_18	x	3687	12788	17.16MB	70.58s	1880	1808	3.12MB	<0.01s	>7058.00
ec_mul_gt_ft_19	x	4119	14294	16.84MB	54.88s	2098	2022	3.29MB	0.01s	5488.00
ec_mul_gt_ft_21	x	4575	15884	20.10MB	73.91s	2328	2248	3.30MB	<0.01s	>7391.00
ec_mul_gt_ft_22	x	5055	17558	24.91MB	111.03s	2570	2486	3.43MB	0.03s	3701.00
pc_arith_a_6		572	1980	4.11MB	3.78s	55	10	2.73MB	0.36s	10.50
pc_arith_a_7		740	2562	5.00MB	28.52s	61	10	2.73MB	1.72s	16.58
pc_arith_a_8		932	3228	6.93MB	196.98s	67	10	2.73MB	8.21s	23.99
pc_arith_a_9		1148	3978	>10.16MB	>500s	73	10	2.73MB	37.83s	>13.21
pc_arith_b_10		250	852	3.60MB	0.01s	77	17	3.89MB	1.42s	<0.1
pc_arith_b_11		268	911	3.61MB	0.01s	82	17	4.68MB	4.68s	<0.1
pc_arith_b_12		286	970	3.59MB	0.01s	87	17	6.70MB	12.24s	<0.1
pc_arith_b_13		304	1029	3.59MB	0.01s	92	17	7.70MB	30.91s	<0.1

than verification is an important topic. As one example logic synthesis for reversible circuits with SWORD was introduced in [22].

ACKNOWLEDGMENTS

We wish to thank João Marques-Silva and Paulo Jorge Matos for many helpful discussions in the area of SMT.

REFERENCES

[1] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on CAD*, vol. 21, no. 12, pp. 1377–1394, 2002.

[3] M. Davis, G. Logeman, and D. Loveland, "A machine program for theorem proving," *Comm. of the ACM*, vol. 5, pp. 394–397, 1962.

[4] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT 2003*, ser. LNCS, vol. 2919, 2004, pp. 502–518.

[5] R. Bryant and Y.-A. Chen, "Verification of arithmetic functions with binary moment diagrams," in *Design Automation Conf.*, 1995, pp. 535–541.

[6] R. Drechsler, B. Becker, and S. Ruppertz, "K*BMDs: A new data structure for verification," in *European Design & Test Conf.*, 1996, pp. 2–8.

[7] R. Brinkmann and R. Drechsler, "RTL-datapath verification using integer linear programming," in *ASP Design Automation Conf.*, 2002, pp. 741–746.

[8] J. Thathachar, "On the limitations of ordered representations of functions," in *Computer Aided Verification*, ser. LNCS, vol. 1427. Springer Verlag, 1998, pp. 232–243.

[9] S. A. Seshia, S. K. Lahiri, and R. E. Bryant, "A hybrid SAT-based decision procedure for separation logic with uninterpreted functions," in *Design Automation Conf.*, 2003, pp. 425–430.

[10] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *Computer Aided Verification*, ser. LNCS, vol. 3114, 2004, pp. 175–188.

[11] B. Dutertre and L. Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *Computer Aided Verification*, ser. LNCS, vol. 4114, 2006, pp. 81–94.

[12] B. Dutertre and L.Moura, *The YICES SMT Solver*, 2006, available at http://yices.csl.sri.com/.

[13] C.-Y. Huang and K.-T. Cheng, "Using word-level ATPG and modular arithmetic constraint-solving techniques for assertion property checking," *IEEE Trans. on CAD*, vol. 20, no. 3, pp. 381–391, 2001.

[14] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. on Comp.*, vol. 48, no. 5, pp. 506–521, 1999.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conf.*, 2001, pp. 530–535.

[16] E. Goldberg and Y. Novikov, "BerkMin: a fast and robust SAT-solver," in *Design, Automation and Test in Europe*, 2002, pp. 142–149.

[17] G. Parthasarathy, M. Iyer, K.-T. Cheng, and L.-C. Wang, "An efficient finit-domain constraints solver for circuits," in *Design Automation Conf.*, 2004, pp. 212–217.

[18] Y. Novikov and R. Brinkmann, "Foundations of hierarchical sat-solving," in *Int'l Workshop on Boolean Problems*, 2004, pp. 103–141.

[19] G. Tseitin, "On the complexity of derivation in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, 1968, pp. 115–125, (Reprinted in: J. Siekmann, G. Wrightson (Ed.), Automation of Reasoning, Vol. 2, Springer, Berlin, 1983, pp. 466-483.).

[20] M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, 3rd ed. Pearson Education, 2004.

[21] M. Herbstritt, *wld: A C++ library for decision diagrams*, Institute of Computer Science, Albert-Ludwigs-University, Freiburg im Breisgau, 2000, http://ira.informatik.uni-freiburg.de/software/wld.

[22] R. Wille and D. Große, "Fast Exact Toffoli Network Synthesis of Reversible Logic," in *Int'l Conf. on CAD*, 2007.