SWORD: A SAT like Prover Using Word Level Information

Robert Wille, Görschwin Fey, Daniel Große, Stephan Eggersglüß, and 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.

1 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. Especially handling data paths is a difficult problem.

For this reason, approaches to exploit such high level information have been proposed in the past [5–7]. 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) [9–12] is more powerful since multiple provers are combined, but still structural information is not available. Related work is discussed in more detail in Section 2 and empirically compared in Section 6.

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 [11, 12].

The paper is structured as follows: Related work is discussed in more detail in the next section. The preliminaries and limits regarding Boolean SAT are reviewed in Section 3. Then, the basic algorithm of SWORD and the use of modules to effectively model a problem are introduced in Section 4. Section 5 discusses the advantages of this approach. Experimental evidence for the efficiency of SWORD in comparison to other prover paradigms is provided in Section 6. Finally, a summary and the conclusions are presented in Section 7.

2 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 [14–16, 4], several researchers investigated the combination of SAT with other proof techniques, i.e. *Satisfiability Modulo Theories* (SMT) [9–12]. 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 to different types of modules.

3 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.

3.1 Definition

The *Boolean satisfiability problem* (SAT problem) is to determine whether there exists an assignment α to an Boolean function f such that $f(\alpha) = 1$ (i.e. f is satisfiable) or to prove that no such assignment exists (i.e. f is unsatisfiable).

A 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 CNF formula is satisfied if all clauses are satisfied. A clause is satisfied if at least one of its

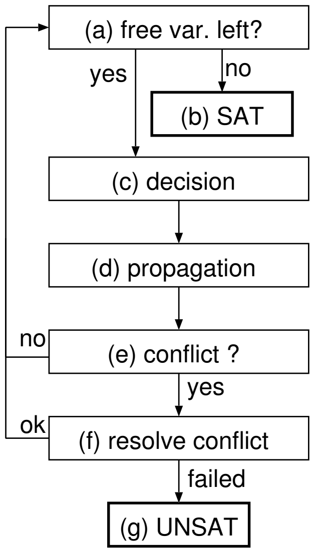


Fig. 1. DPLL algorithm in modern Boolean SAT solvers

literals is satisfied. A variable is satisfied when 1 is assigned to the variable, the negation of a variable is satisfied under the assignment 0.

3.2 Basic Algorithm

The basic search procedure to find a satisfying assignment is shown in Fig. 1 and has the structure of the DPLL algorithm [19, 3]. Instead of simply traversing the complete space of assignments, intelligent decision heuristics [16], conflict based learning [14] and sophisticated engineering of the implication algorithm [15] 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.

3.3 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* [20]. 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 [21], 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.

4 Using Word Level Information

In this section, we describe the architecture of SWORD and how word level information is 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.

4.1 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. By this, structural and semantical knowledge is available and can be exploited during the search process by special algorithms for each kind of module (we will explain this in more detail later).

Example 2. Fig. 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.

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.

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

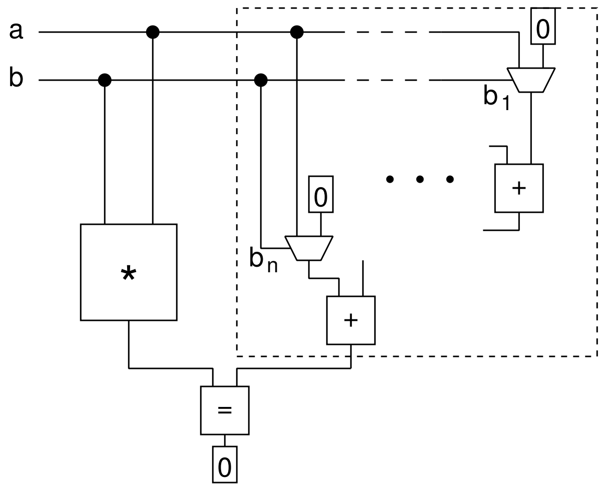


Fig. 2. Miter example of a multiplier

Besides providing word level information the representation in terms of modules has another advantage: The problem description of SWORD is much more compact than a CNF. To represent it for a classical SAT solver we need $\theta(n^2)$ clauses (see Example 1). Our representation consists of $2n + 2$ modules, only. Furthermore we need no auxiliary variables in total.

4.2 Overall Algorithm

The overall algorithm of SWORD is shown in Fig. 3. This algorithm is similar to the DPLL procedure as applied in standard SAT solvers (see Section 3.2): 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.

In the following, the global and local algorithms are described in more detail, respectively.

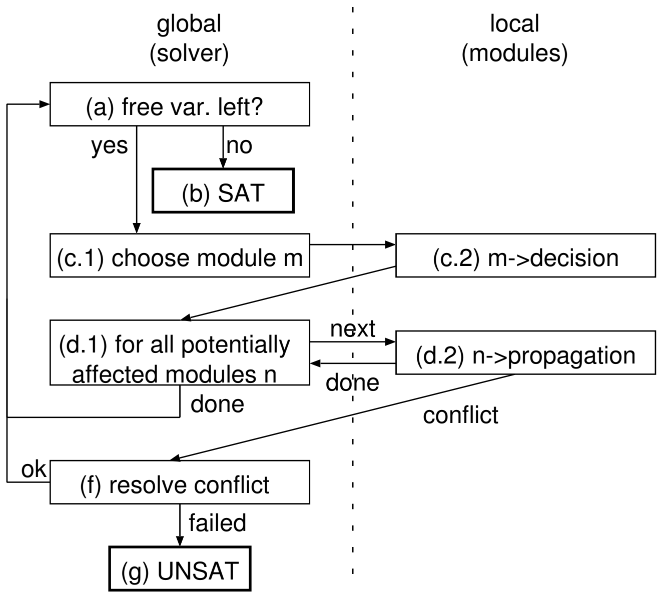


Fig. 3. Algorithm

4.3 Decision Strategies

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 Fig. 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.

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.

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.

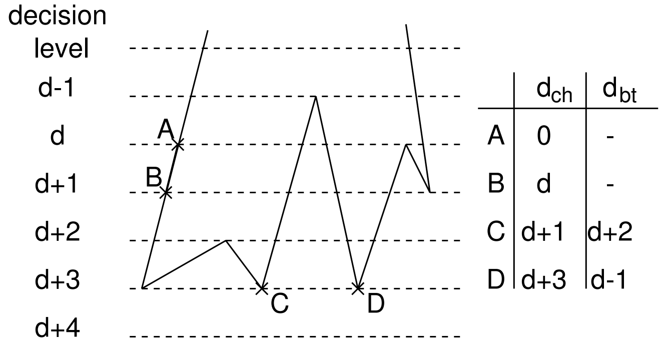
For an adder, assigning some variables a_i, b_i or o_i (with $n > i \geq 0$) while variables a_j, b_j or o_j (with $i > j \geq 0$) are still unassigned, often does not allow to imply values for the outputs since then, the value of the respective carry bits are unknown, too. In contrast, when all of the least significant bits of both operands are given, the corresponding bits of the outputs can be determined. Therefore, the variable representing the least significant unassigned bit is assigned first.

In the implmentation, 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 the next section).

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. Fig. 4 illustrates this mechanism. The global search tree is indicated by the plain line and the decision levels that are reached are also shown. A transition of the FSM of a module 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 Fig. 5; only state transitions are indicated, internal variables are not shown.



	d_{ch}	d_{bt}
A	0	-
B	d	-
C	d+1	d+2
D	d+3	d-1

Fig. 4. Search tree and decision levels

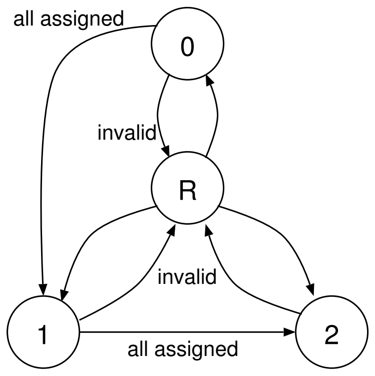


Fig. 5. FSM for an adder

4.4 Implication Engine

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

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.

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, then 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.

4.5 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).

Note, that the implication graph itself is more compact than the one of a Boolean SAT solver, because there are no auxiliary variables contained in the graph. As a result all clauses derived by conflict analysis consist of module variables and prune the search space of the original problem domain directly.

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 used 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.

5 Discussion

The first observation is that SWORD represents problems in a much more compact way than a CNF based solver. In contrast to modeling the internal structure of a module by clauses, the functionality is described on an algorithmic level. As already explained, this leads to a smaller number of variables and less constraints that have to be handled.

At the same time, this representation enables more efficient implications. Instead of a large number of clauses usually only the connecting modules have to be considered to imply a value for a particular variable. The implication procedures of particular modules are not as strong as possible (using the notation of [18] they are not *maximally implicative* and in the notation of constraint programming they are not *fully arc-consistent*). Of course, it is possible to create stronger implication procedures, but only at the cost of more complex modules and higher computation time. Currently, the implication procedures are crafted manually and exploit the knowledge about the decision order. By this, it is possible to trade-off between implicative power and efficiency. Investigating more powerful procedures that are automatically generated remains future work. One promising approach that starts from BDDs has been suggested in [22].

Implications and decisions are restricted to module variables. Therefore, in contrast to CNF based SAT, no auxiliary variables can occur in the implication graph. Thus, the size of the implication graph is reduced and, as a result, the time needed to traverse the implication graph is reduced. Similarly, the conflict clauses consist of module variables only. Therefore, instead of learning a large number of locally conflicting assignments, the overall search space is pruned.

Finally, structural information about predecessors or successors of modules is available within SWORD. Currently, this information is not fully exploited. Only the global decision heuristic evaluates the position of modules in a static preprocessing step. For Boolean SAT dynamic decision procedures have proven to be much more efficient. Thus, combining structural knowledge with heuristics from Boolean SAT is another direction for future work.

6 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 [23] as a representative of pure word level approaches, and the SMT solver Yices using the theory of bit vectors [11, 12]. All experiments have been carried out on an AMD Athlon64 3500+ (Linux, 2.2 GHz, 1 GB). Unless mentioned otherwise the time out was set to 500 CPU seconds.

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.

6.1 Parameter Studies

For selected representative instances Table 1 reports run times in CPU seconds to evaluate different features of SWORD. To demonstrate the influence of the parameters, alternative configurations are used. The last column *DEF* provides the run times of SWORD in the current configuration. The influence of the decision heuristic is studied in columns *RAND* and *MSB*. If the modules in the global decision heuristic are randomly selected, the numbers in column *RAND* are obtained. Column *MSB* gives the results, if the arithmetic modules are assigned from the higher to the lower bits, what is typically not clever. As can be seen, both approaches lead to high run times and even time outs in comparison with the heuristic shown in column *DEF*. In the remaining columns, learning strategies are evaluated. Column *CCLS* reports results for learning all conflict clauses regardless of the length and the origin. In column *30%* only short conflict clauses are learned that consist of up to 30% of the variables contained in the problem instance; clauses including variables from a multiplier are learned as well. The maximum relative length of 30% for learned clauses was experimentally determined. Finally, column *DEF* gives the run time of SWORD using all features: the global decision heuristic using priorities, the local decision heuristic that assigns least significant bits first, and conflict based learning of short clauses together with neglecting clauses coming from complex modules, like multipliers. As can be seen, *DEF* is the most robust approach and clearly outperforms all others. Therefore, this setting was used for the experiments in the following.

6.2 Comparison to Boolean SAT solver

Table 2 shows results in comparison to MiniSat. For each benchmark the number of variables to represent the problem, the number of clauses for MiniSat and

Table 1. Parameter studies

	Heuristic		*Learning*		
circuit	*RAND*	*MSB*	*CCLS*	*30%*	*DEF*
ec_mul_mul_10	53.97	47.95	>500	36.88	37.09
ec_mul_pp_9	125.83	187.06	45.51	45.29	15.54
ec_mul_gt_10	>500	>500	>500	>500	113.84
ec_mul_mul_li_10	59.32	48.19	>500	36.93	37.01
pc_arith_a_9	>500	>500	>500	37.57	37.83
pc_arith_b_13	>500	>500	363.42	30.68	30.91

the number of modules for SWORD are given in columns *var*, *cls* and *mod*, respectively. The memory requirements (in MB) and the CPU time (in seconds) are provided in columns *mem* and *time*. The improvement in run time of SWORD over MiniSat (i.e. the run time of SWORD divided to the run time of MiniSat) is shown in column *imp*. An *x* in column *sat* indicates whether the problem instance is satisfiable. Since for most satisfiable instances both solvers had small run times, we mainly report numbers for unsatisfiable instances here (satisfiability is studied in more detailed below).

SWORD is quite efficient regarding memory consumption. This is due to the problem representation. Especially word level problems are much more compact than a corresponding SAT instance. This benefit decreases only slightly when the problem is partially converted to gate level. Moreover, in contrast to the SAT solver, SWORD is quite robust with respect to larger bit widths of the data path. Considering run time, except for *pc_arith_b[10-13]*, SWORD significantly outperforms MiniSat on all benchmark circuits. For benchmarks in the table the improvement is always larger than a factor of two and in one case even three orders of magnitude.

Furthermore, we studied in more depth satisfiable instances. The instances are generated by removing or substituting a single Boolean gate in a multiplier circuit, i.e. all instances were derived from *ec_mul_gt_16*. In this manner over 4000 instances were generated. For all of them MiniSat and SWORD were started with a time out of 5 CPU seconds. Table 3 summarizes the results by giving in the number of instances where the solver took less than 0.01 seconds and where a time out occurred in row two and three, respectively.

Here, it can clearly be seen that SWORD solves most of the instances in almost no time and in addition has fewer time outs than MiniSat. For all instances where both solvers computed a solution within the given limit, Fig. 6 graphically shows the results. As can clearly be seen in the lower half of the diagram, there are many more instances that SWORD can handle in very low run time.

6.3 Comparison to Word Level Solvers

Table 4 provides run times for K*BMDs, SWORD and Yices. As expected K*BMDs performs very well on pure word level problems and outperform SWORD

Table 2. Comparison to MiniSat

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

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.

6.4 Summary

SWORD is very efficient in comparison to the most powerful available SAT solver on our verification benchmarks. This especially holds if the word level structure can be exploited. Furthermore, in contrast to other word level approaches that break down if Boolean operations are used, SWORD is very robust also in this

Table 3. Data for satisfiable instances

time	*MiniSat*	*SWORD*
<0.01	50	2183
>500	708	565

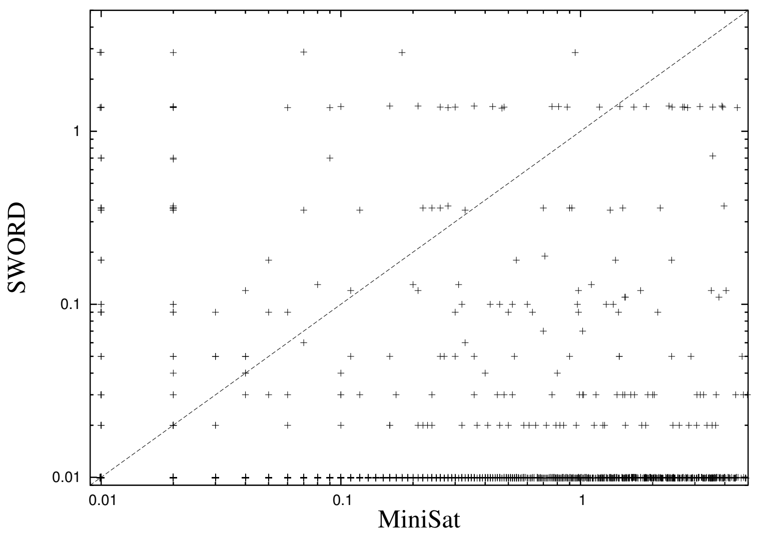


Fig. 6. Run time for satisfiable instances

case. This can be seen in the comparison with K*BMDs and Yices which often do not finish within the given time out.

7 Conclusions

We presented the 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. Considering a problem directly at the word level significantly reduces the size of the instances. Moreover, the word level information is exploited in all steps of the search process. In contrast to other word level solvers, SWORD is robust with respect to bit level operations on our benchmarks.

In future work, the efficiency of SWORD will be further improved by investigating more powerful decision heuristics and engineering the watching mechanisms for implication and backtracking. Furthermore, the local procedures for different types of modules are currently coded manually; an automatic approach to generate this code could be applied to study different version of the procedures for a single type of module. Finally, the application to other problem

Table 4. Comparison to K*BMDs and SMT

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

domains than verification is an important topic. As one example logic synthesis for reversible circuits with SWORD was introduced in [24].

Acknowledgments

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

References

1. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comp. **35** (1986) 677–691
2. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans. on CAD **21** (2002) 1377–1394
3. Davis, M., Logeman, G., Loveland, D.: A machine program for theorem proving. Comm. of the ACM **5** (1962) 394–397
4. Eén, N., Sörensson, N.: An extensible SAT solver. In: SAT 2003. Volume 2919 of LNCS. (2004) 502–518

5. Bryant, R., Chen, Y.A.: Verification of arithmetic functions with binary moment diagrams. In: Design Automation Conf. (1995) 535–541
6. Drechsler, R., Becker, B., Ruppertz, S.: K*BMDs: A new data structure for verification. In: European Design & Test Conf. (1996) 2–8
7. Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: ASP Design Automation Conf. (2002) 741–746
8. Thathachar, J.: On the limitations of ordered representations of functions. In: Computer Aided Verification. Volume 1427 of LNCS., Springer Verlag (1998) 232–243
9. Seshia, S.A., Lahiri, S.K., Bryant, R.E.: A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In: Design Automation Conf. (2003) 425–430
10. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: Computer Aided Verification. Volume 3114 of LNCS. (2004) 175–188
11. Dutertre, B., Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Computer Aided Verification. Volume 4114 of LNCS. (2006) 81–94
12. Dutertre, B., L.Moura: The YICES SMT Solver. (2006) Available at http://yices.csl.sri.com/.
13. Huang, C.Y., Cheng, K.T.: Using word-level ATPG and modular arithmetic constraint-solving techniques for assertion property checking. IEEE Trans. on CAD **20** (2001) 381–391
14. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. on Comp. **48** (1999) 506–521
15. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Design Automation Conf. (2001) 530–535
16. Goldberg, E., Novikov, Y.: BerkMin: a fast and robust SAT-solver. In: Design, Automation and Test in Europe. (2002) 142–149
17. Parthasarathy, G., Iyer, M., Cheng, K.T., Wang, L.C.: An efficient finit-domain constraints solver for circuits. In: Design Automation Conf. (2004) 212–217
18. Novikov, Y., Brinkmann, R.: Foundations of hierarchical SAT-solving. In: Int'l Workshop on Boolean Problems. (2004) 103–141
19. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 506–521
20. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part 2. (1968) 115–125 (Reprinted in: J. Siekmann, G. Wrightson (Ed.), Automation of Reasoning, Vol. 2, Springer, Berlin, 1983, pp. 466-483.).
21. Mano, M.M., Kime, C.R.: Logic and Computer Design Fundamentals. 3rd edn. Pearson Education (2004)
22. Höreth, S.: Compilation of optimized OBDD-algorithms. In: European Design Automation Conf. (1996) 152–157
23. Herbstritt, M.: 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.
24. Wille, R., Große, D.: Fast exact Toffoli network synthesis of reversible logic. In: Int'l Conf. on CAD. (2007) 60–64