Using Higher Levels of Abstraction for Solving Optimization Problems by Boolean Satisfiability

Robert Wille Daniel Große Mathias Soeken Rolf Drechsler
Institute of Computer Science
University of Bremen, 28359 Bremen, Germany
$\{$*rwille,grosse,msoeken,drechsle*$\}$*@informatik.uni-bremen.de*

Abstract

Optimization problems can be solved using Boolean Satisfiability by mapping them to a sequence of decision problems. Therefore, in the last years several encodings have been developed. Independently, also new solvers have been introduced lifting Boolean Satisfiability to higher levels of abstraction, e.g. SAT Modulo Theories (SMT) solvers and word level solvers. Both support bit-vector logic and thus allow more compact encodings of the problems.

In this paper we investigate the efficiency of these new solver paradigms applied to optimization problems. We show for two case studies – graph coloring and exact synthesis of reversible logic – that the resulting problem instances can be reduced with respect to the size. In addition for the synthesis problem significant run-time improvements can be achieved.

1. Introduction

In the last ten years techniques for solving instances of the *Boolean Satisfiability* (SAT) problem have been intensively studied. The SAT problem is to determine whether there exists a solution for a given Boolean formula or not. Taking the original DPLL algorithm [4] as a basis, researchers investigated conflict based learning [15], efficient implication routines [16], and strong heuristics [10] leading to well-engineered SAT solvers [8]. State-of-the art SAT solvers are able to handle instances with more than hundreds of thousands of variables and clauses. As a result SAT solvers are used today in different areas, e.g. in formal verification, automatic test pattern generation or logic synthesis.

However, common SAT solvers work on the Boolean level, i.e. the considered problems have to be encoded in terms of clauses. Due to the increasing complexity of these problems, recently several approaches have been studied which lift the problem instances to higher levels of abstraction (see e.g. [6]). This results in the development of *SAT Modulo Theories* (SMT) [3, 7] and *word level solvers* [5, 24, 9]. Since both solver paradigms support *bit-vector logic*, a more abstract problem description is possible. Hence, two advantages are expected:

1. A much more compact problem representation since instances can be encoded with the help of bit-vectors and bit-vector operations instead of Boolean variables and clauses, respectively.
2. More efficient algorithms since the higher level of abstraction can also be exploited for dedicated strategies and heuristics.

But even if a large number of new solvers have been introduced in this area (see e.g. [18]), until now their application to real world scenarios is still at the beginning. As one example in [2] SMT is used for the verification of software. In [14] the authors considered predicate abstraction with SMT while word level solvers have been applied e.g. for functional test generation in [27]. The encoding effort to represent circuit verification problems at different levels of abstraction has been studied in [20].

In this work we consider the application of the high-level SAT paradigm for optimization problems. Although for this kind of problems specialized techniques are available (see e.g. [17]), application of Boolean SAT has been shown to be very successful [12, 23]. Using SAT the optimization problem is solved by mapping it to a sequence of decision problems. Since many optimization problems are inherently non Boolean problems, we investigate if the usage of the higher level of abstraction is more efficient.

More precisely, we encode two representatives of optimization problems (the graph coloring problem and the exact synthesis of reversible logic using Toffoli gates) as instances of bit-vector logic and compare them with the respective representation in terms of clauses. It is shown that for both optimization problems significant reductions of the problem representation result, if the bit-vector encoding is used. Furthermore, while for graph coloring the overall run-time remains in the same range, speed-ups of up to three orders of magnitudes can be achieved for synthesis of reversible logic.

The paper is structured as follows: In the next section solvers for Boolean Satisfiability and the new abstractions (i.e. SMT and word level solvers) are introduced. After this, the main flow of solving optimization problems with the help of SAT is described in Section 3. Here also the importance of the used encoding is sketched. Section 4 provides the investigated case studies: (1) the problem is briefly introduced, (2) the problem formulation and encodings for SAT and bit-vector logic are discussed, and (3) experimental results are given. Finally, the paper is concluded in Section 5.

2. Different SAT Abstractions

In this section the basic concepts of SAT, SMT and word level solvers are introduced. The *Boolean Satisfiability* (SAT) problem is defined as follows:

Definition 1 *Let f be a Boolean function. Then SAT is to determine whether there exists a satisfying assignment α to all variables of f such that $f(\alpha) = 1$. In this case f is* satisfiable*; otherwise f is* unsatisfiable*.*

In general the Boolean function f is given in *Conjunctive Normal Form* (CNF), i.e. a product-of-sum representation. The CNF is a conjunction of clauses. A clause is a disjunction of literals and each literal is a propositional variable or its negation.

In past several (backtracking) algorithms (so called *SAT solvers*) have been proposed [4, 15, 16, 10, 8]. Most of them are based on three essential procedures: (1) The decision heuristic assigns values to free variables, (2) the propagation procedure determines implications due to the last assignment(s) and (3) the conflict analysis tries to resolve conflicts by backtracking that occur during the search. Advanced techniques like e.g. *efficient Boolean constraint propagation* [16] or *conflict analysis* [15] are common in state-of-the-art SAT solvers today.

The tremendous improvements in the performance of SAT solvers enables the consideration of problems with more than hundreds of thousands of variables and clauses. Thus, SAT is widly used in many application domains. Thereby, the real world problem is transformed into CNF and then solved by using a SAT solver as black box.

However, due to the increasing complexity of many problems, researchers investigated the use of higher levels of abstractions than CNF – by still exploiting the established SAT techniques. This leads to the development of *SAT Modulo Theories* (SMT) and dedicated *word level solvers*.

In SMT (e.g. [3, 7]) a traditional SAT solver is combined with decision procedures for decidable theories (e.g. linear arithmetic or bit-vector logic). While the SAT solver tries to find a satisfying assignment to an abstract representation of the problem (in CNF), the theory solver checks this assignment with respect to the underlying theory. Thus, advanced SAT techniques as well as higher problem abstractions are utilized. In contrast word level solvers either work directly on the word level of the problem (e.g. [5, 24]) or bitblast it to a traditional SAT solver (e.g. [9]).

In these solvers the higher level of abstraction is expressed e.g. by bit-vector logic. Since we use bit-vector logic in this paper we give some basic definitions in the following:

Definition 2 *A* bit-vector *is an element $\vec{b} = (b_{n-1}, \ldots, b_0) \in \mathbb{B}^n$. The* index $[\,] : \mathbb{B}^n \times [0, n) \to \mathbb{B}$ *maps a bit-vector $\vec{b}$ and an index i to the i^{th} component of the vector, i.e. $\vec{b}[i] = b_i$. Conversion from (to) a natural number is defined by* $\mathrm{nat} : \mathbb{B}^n \to N$ *(*$\mathrm{bv} : N \to \mathbb{B}^n$*) with $N = [0, 2^n) \subset \mathbb{N}$ and* $\mathrm{nat}(\vec{b}) := \Sigma_{i=0}^{n-1} b_i \cdot 2^i$ *(*$\mathrm{bv} := \mathrm{nat}^{-1}$*).*

Problems can be constraint by using bit-vector operations as well as arithmetic operations. Let $\vec{a}, \vec{b} \in \mathbb{B}^n$ be two bit-vectors. Then, the bit-vector operation $\circ \in \{\wedge, \vee, \ldots\}$ *is defined by $\vec{a} \circ \vec{b} := (\vec{a}[n-1] \circ \vec{b}[n-1], \ldots, \vec{a}[0] \circ \vec{b}[0])$. An* arithmetic operation $\bullet \in \{*, +, \ldots\}$ *is defined by* $\vec{a} \bullet \vec{b} := \mathrm{nat}(\vec{a}) \bullet \mathrm{nat}(\vec{b})$.

SAT solvers as well as SMT and word level solvers consider *decision problems*, i.e. problems which only allow yes or no (true or false, holds or fails, etc.) as answer. The next section describes how SAT can be used to solve *optimization problems* as well. After this, Section 4 shows by two case studies the benefits of applying higher levels of abstractions if optimization problems are considered.

3. Solving Optimization Problems Using SAT

A *decision problem* is to find an arbitrary solution such that a given question/formula is satisfied. In contrast *optimization problems* try to find the *best* of these solutions with respect to the resulting costs.

Definition 3 *Given a problem p with a set S of solutions for this problem. Let additionally $c(p, s) \in \mathbb{N}$ be the* costs *of the solution $s \in S$ for the problem p. Then, an* optimization problem *is to determine a solution s for the problem p such that $c(p, s)$ is minimal (or maximal, respectively).*[1]

For example, in graph coloring the number of colors and in logic synthesis the number gates is used as the cost criteria, respectively. Both problems are considered in more detail in the next section.

Each optimization problem can be formulated as a sequence of decision problems: In each step – i.e. for each decision problem – it is asked if there exists a solution for the problem with fixed costs. More precisely a decision problem is formulated asking if there is a solution s for the problem p with costs c and $c \in \mathbb{N}$.

Using a straight forward approach, first it is searched for a solution with costs $c = 0$. If no solution exists (i.e. if the decision problem is unsatisfiable), then the costs are incremented until one of the resulting decision problems becomes satisfiable. Thereby, minimality is guaranteed since c is incremented iteratively starting with $c = 0$. A pseudo-code of this approach can be found in Algorithm 1. We call this procedure *iterative approach* in the following.

However, this approach may lead to too many iterations. To avoid this an approximation can be used: By exploiting upper and lower bounds initial costs $c = c_i$ are determined. If there is a solution with costs c_i, then a better solution with

[1]In this work we only consider problems, where $c(p, s)$ has to be minimal. Finding solutions with maximal costs can be handled analogously.

Algorithm 1: iterativeApproach(p)

```
Data: problem instance $p$
Result: solution $s$ such that $c(p, s)$ is minimal
1  $c = 0$;
2  while true do
3      decProb = encodeDecProb($p, c$);
4      $s$ = solve(decProb);
5      if $s == UNSAT$ then
6          $c++$;
7      else
8          return $s$;
```

Algorithm 2: approximationApproach(p, l, u)

```
Data: problem instance $p$, lower bound $l$, upper bound $u$
Result: solution $s$ such that $c(p, s)$ is minimal
1  repeat
2      $c = \lceil \frac{l+u}{2} \rceil$;
3      decProb = encodeDecProb($p, c$);
4      $s$ = solve(decProb);
5      if $s == UNSAT$ then
6          $l = c$;
7      else
8          $u = c$;
9  until $l \neq u + 1$ ;
10 return $s$;
```

costs $c_{i'} < c_i$ is searched. Otherwise, the search continues with higher costs $c_{i'} > c_i$. The *best* solution (i.e. the solution with minimal costs) is found when (1) a solution with costs $c_{\min}$ is obtained and (2) it is proven that no solution with costs $c' < c_{\min}$ exists. One possible realization of this approach is given in Algorithm 2. We call this procedure *approximation approach* in the following.

In both approaches the respective checks (*Is there a solution with costs c?*) can be done by means of a SAT solver. Therefore, two steps are performed: First the decision problem is encoded as a SAT instance in CNF (line 3 in both algorithms). After this, the resulting instance is solved by using an off-the-shelf SAT solver (line 4 in both algorithms).

Thereby, the used encoding is crucial. To this day in many approaches, the respective problem formulation has been encoded as CNF (see e.g. [23] as a recent example). In general, this is possible in time and space linear in the size of the original problem formulation [22]. However, the resulting instances only allow implications at the Boolean level. Furthermore, often many auxiliary variables are necessary which increases the size of the resulting search space.

In this paper we evaluate the usage of higher levels of abstraction. In contrast to the existing approaches the respective decision problems are encoded in bit-vector logic which is supported by several SMT and word level solvers. As the next section will show, many problem formulations are naturally given on word level. Thus, using the bit-vector logic – to represent the word level – all bit-vector variables and most of the operators are preserved. On the one hand this simplifies the encoding of the problem. On the other hand improvements of the overall run-time can result.

4. Case Studies

In this section two representatives of optimization problems are considered, i.e. the graph coloring problem and exact synthesis of reversible logic. Both problems are solved using Boolean SAT and SMT with bit-vector logic, respectively.

First, each problem is briefly described. Then, the problem formulation as a decision problem is given and the encoding into a SAT/bit-vector logic instance is described. Finally, experimental results show the differences of both encodings with respect to the problem size and the needed run-time for solving. All experiments have been carried out on an AMD Athlon 3500+ with 1 GB of memory. If not stated otherwise, for the CNF instances we used the SAT solver MiniSat [8]. For solving the bit-vector logic instances the SMT solver Yices [7] has been applied.

4.1. Graph Coloring

The *graph coloring* problem is defined as follows: Given an undirected graph $G = (V, E)$, a *vertex coloring* is the assignment of a color to each vertex of G such that no two adjacent vertices have the same color. More formally, a coloring is defined as a mapping $\text{color} : V \rightarrow [1, k] \subset \mathbb{N}$ with $\text{color}(v_1) \neq \text{color}(v_2)$ for each $(v_1, v_2) \in E$. The *graph coloring problem* is to find a minimal number of colors that leads to a valid vertex coloring, i.e. $c := \min\{k \mid \text{color} : V \rightarrow [1, k] \textit{ and } k \in \mathbb{N}\}$ which is also called *chromatic number*. The graph coloring problem has applications e.g. in register allocation or high-level synthesis.

4.1.1. Problem Formulation. We formulated the decision problem *Is there a coloring for the graph $G = (V, E)$ with c colors?* as basis for the algorithms presented in the previous section as follows: For each vertex $v \in V$ a free variable x_v is introduced representing the color of this vertex. Constraints $x_v \leq c$ ensure that at most c colors are used. Furthermore, the additional constraints $x_v \neq x_w$ for each $(v, w) \in E$ guarantee that no adjacent vertices have the same color.

4.1.2. Bit-vector Encoding. Applying bit-vector logic the encoding of this formulation is straight forward. Each variable x_v can be represented by a bit-vector of length $\lceil \log_2 c \rceil$. For the constraints respective bit-vector operations are available. This leads to $|V|$ variables and at most $|E| + |V|$ constraints in total.

4.1.3. CNF Encoding. In contrast, to encode the problem into CNF for each variable x_v, $\lceil \log_2 c \rceil$ single Boolean variables are needed (i.e. $|V| \cdot \lceil \log_2 c \rceil$ in total). The constraints have to be transformed into certain sets of clauses. In total the constraint $x_v \leq c$ is encoded by $2^{\lceil \log_2 c \rceil} - c$ clauses for each vertex while the constraint $x_v \neq x_w$ is encoded by $2^{\lceil \log_2 c \rceil}$ clauses for each edge.

4.1.4. Experimental Results. The presented encodings of the decision problems have been used for both, the iter-

Table 1. Iterative approach for graph coloring

Benchmark	c	CNF Encoding (MiniSat) #vars	#clauses	time (s)	Bit-vector Encoding (Yices) #vars	#bv-constr	time (s)
anna	11	4,002	92,190	**6.00**	1,380	10,826	24.87
david	11	2,523	75,255	**4.14**	870	8,729	21.83
huck	11	2,146	56,030	**5.72**	740	6,538	22.06
jean	10	2,000	39,192	**0.80**	720	5,052	4.98
queen8_12	12	3,168	292,800	112.72	1,056	30,864	**87.82**
miles250	8	2,176	33,404	**0.14**	896	5,930	0.94
games120	9	2,520	75,688	**0.31**	960	10,808	0.99
myciel5	6	517	6,418	**28.38**	235	1,321	122.67

Table 2. Approximation approach for graph coloring

Benchmark	c	CNF Encoding (MiniSat) #vars	#clauses	time (s)	Bit-vector Encoding (Yices) #vars	#bv-constr	time (s)
anna	11	3,726	166,454	**6.32**	828	6,744	24.09
david	11	2,523	134,618	**3.93**	609	6,206	22.08
huck	11	1,776	64,310	**9.65**	444	4,056	35.90
jean	10	1,680	51,568	**1.28**	400	2,940	8.16
queen8_12	12	1,824	198,432	105.30	480	13,968	**85.85**
miles250	8	1,408	22,056	**0.19**	512	3,352	0.80
games120	9	1,680	62,928	**0.56**	480	5,464	1.30
myciel5	6	658	9,910	**28.44**	235	1,368	122.80

ative approach as well as the approximation approach described in Section 3. For the latter one we used the fixed value 2 as lower bound and the total number of vertices as upper bound, respectively. All graph coloring benchmarks haven been taken from [13].

The results are shown in Table 1 (for the iterative approach) and Table 2 (for the approximation approach), respectively. The first two columns denote the name of the benchmark as well as the resulting minimal costs needed for coloring the graph (i.e. the chromatic number). The next columns provide information about the size of the instances for the CNF encoding and the bit-vector encoding, respectively. Column #vars gives the number of variables. Column #clauses provides the number of clauses in the CNF encoding while column #bv-constr gives the number of bit-vector constraints of the bit-vector encoding. All these numbers have been calculated by summing up the respective numbers over all instances. Finally, in the last column the overall run-time of the algorithm (in CPU seconds) is given.

As can be seen for both approaches the bit-vector encoding leads to much more compact problem representations. On average the CNF encoding needs four times more variables as the bit-vector encoding. The difference in terms of constraints is even more significant: One order of magnitude more constraints are required by the CNF encoding in comparison to the bit-vector encoding. However, regarding the run-time the bit-vector encoding is outperformed in all benchmarks except *queen8_12*.

We see two reasons for this result:

1. In the last few years there was a significant progress in SAT solving techniques. We demonstrate this claim by additional experiments whose results are given in Table 3. Here, the same encodings are applied to the SAT solver siege [19] (in version 4 from 2003) instead of MiniSat (version 1.14 from 2005). As can be seen the bit-vector based approaches now outperform the SAT encoding. Since the bit-vector logic in the context of SMT has been considered for a very short time we expect similar improvements in the near future. This is approved by the latest results of the SMT competition [1].

Table 3. Comparison with another SAT solver

Benchmark	iterative siege_v4	Yices	approximation siege_v4	Yices
anna	47.57	**24.87**	44.83	**24.09**
david	29.22	**21.83**	29.31	**22.08**
huck	72.23	**22.06**	64.69	**35.90**
jean	13.21	**4.98**	11.82	**8.16**
queen8_12	203.67	**87.82**	196.35	**85.85**
miles250	0.96	**0.94**	**0.70**	0.80
games120	1.75	**0.99**	1.71	**1.30**
myciel5	153.77	**122.67**	153.78	**122.80**

2. We noticed that the considered underlying decision problem of the graph coloring problem is based on only a few bit-vector operators which additionally have very efficient CNF representations. In the next section another optimization problem – exact synthesis of reversible logic – is considered that requires much more complex constraints involving many different bit-vector operations. Here, the higher level of abstraction can be fully exploited which leads to significant reductions in run-time.

4.2. Exact Synthesis of Reversible Logic

The *exact synthesis problem for reversible logic* is defined as follows: Given a reversible function specification $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$ (i.e. a bijective Boolean function with n inputs and n outputs) the goal is to find a network realization of f with the minimal number of gates.

A network realization consists of a cascade of reversible gates. A widely used reversible gate is the Toffoli gate [21] which is also applied here. The idea of the Toffoli gate is to invert one input line (the target line) if the product of a set of control lines evaluates to true. To make the paper self contained we give a definition and a simple example.

Definition 4 *Let $X := \{x_0, \ldots, x_{n-1}\}$ be the set of domain variables and $\mathcal{T}^n := \{(\vec{c}, t) \in \mathbb{B}^n \times [0, n) \mid \vec{c}[t] = 0\}$. Then $g = (\vec{c}, t) \in \mathcal{T}^n$ is a* Toffoli gate *on n lines with control lines defined in $\vec{c}$ and the target line t. The gate maps $(x_0, \ldots, x_{n-1})$ to $(x_0, \ldots, x_t \oplus p, \ldots, x_{n-1})$ with $p := \bigwedge_{i \in M} x_i$ and $M := \{i \in [0, n) \mid \vec{c}[i] = 1\}$. If $M = \emptyset$, then $p := 1$.*

An example of a Toffoli network with its truth table is shown in Figure 1. A filled circle denotes a control line while an open circle denotes a target line. Thus, the network consists of two gates $g_1 = ((1, 0, 1), 1)$ and $g_2 = ((0, 1, 1), 2)$.

$x_2x_1x_0$	$\dot{x}_2\dot{x}_1\dot{x}_0$	$x_2x_1x_0$	$\dot{x}_2\dot{x}_1\dot{x}_0$
0 0 0	0 0 0	1 0 0	1 0 0
0 0 1	0 0 1	1 0 1	0 1 1
0 1 0	0 1 0	1 1 0	1 1 0
0 1 1	1 1 1	1 1 1	1 0 1

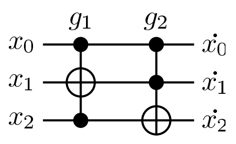

Figure 1. Toffoli network with assignment

Synthesis of reversible logic has applications in low-power design, optical computing and especially in quantum computing. In previous work this problem has been considered using pure Boolean SAT [11].

4.2.1. Problem Formulation. To solve the described optimization problem of exact synthesis the underlying decision problem is described as follows: *Is there a Toffoli network for the reversible function f with d gates?* Since a Toffoli network manipulates bit-vectors we formulate the decision problem directly in terms of bit-vector constraints:

The function $\text{control} : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$ evaluates to true, if all control lines of a gate $(\vec{c}, t)$ are true or there is no control line, respectively. More formally,

$$\text{control}(\vec{x}, \vec{c}) = \begin{cases} 1 & \text{iff } \vec{x} \wedge \vec{c} = \vec{c} \text{ or } \vec{c} = \text{bv}(0), \\ 0 & \text{else.} \end{cases}$$

where $\vec{x}$ is the input of the gate.

Based on this for each gate $g_k = (\vec{c}_k, t_k)$ $(0 < k \le d)$ with input $\vec{x}^{k-1}$ and output $\vec{x}^k$ in a network of size d the following constraint is added to the problem instance:

$$\vec{x}^k = \vec{x}^{k-1} \oplus \text{bv}\left(\text{control}\left(\vec{x}^{k-1}, \vec{c}_k\right) \cdot 2^{t_k}\right)$$

If the control function becomes 1 for the current input of a gate the 1 is shifted by the position of the target line and XOR'ed with the assignment of the gate. This directly corresponds to he behavior of a Toffoli gate (see Definition 4).

Since the equation has to hold for each row of the truth table, this constraint is duplicated 2^n times. Furthermore, for each truth table line the variable $\vec{x}^0$ $(\vec{x}^d)$ is assigned to the respective value of the input (output) of the function to be synthesized. In the overall bit-vector constraint the only free variables are the ones that define the gates, i.e. $g_1 = (\vec{c}_1, t_1), \ldots, g_d = (\vec{c}_d, t_d)$. If there exists a satisfying assignment for these variables a network realization with d gates for f has been found.

4.2.2. Bit-vector Encoding. A bit-vector encoding can be generated easily from the described formalization since all bit-vector operations are directly available.

4.2.3. CNF Encoding. A CNF encoding is built with the standard transformation from [22]. For a detailed description see [11].

4.2.4. Experimental Results. Since the approximation approach cannot be applied for synthesis of Toffoli networks (see [11]), only the iterative approach has been applied in our experiments. As benchmarks we used a selection of reversible functions from different domains (taken from [26]).

The results are given in Table 4. Columns BENCHMARK and d denote the name of the considered function to be synthesized and the resulting costs (i.e. the minimal number of gates). The sizes of the problem instances can be obtained by the values in column #VARS and in columns #CLAUSES or #BV-CONSTR, respectively (again as a sum over all instances). The run-time spent to solve all instances is given in column TIME. Furthermore, the improvement (with respect to run-time) of the bitvector encoding in comparison to the CNF encoding is provided in the last column.

In contrast to the graph coloring instances, more significant differences with respect to the sizes of the bitvector and the CNF encoding can be observed. For all benchmarks fewer variables and constraints are needed for the bitvector representation. For example, to encode the synthesis problem for *graycode6* nearly two millions of CNF variables (9 millions of clauses) are needed – in contrast to 4,458 bitvector variables (28,089 bitvector constraints).

Moreover, the higher level of abstraction have an effect on the time needed to solve. On average improvements of more than a factor of 30 can be documented. In the best case (*graycode6*) the run-time can be reduced by more than three orders of magnitudes.

5. Conclusions

In this paper we evaluated the usage of higher levels of abstraction for solving optimization problems by Boolean satisfiability. In two case studies we showed that encoding the problems in bit-vector logic results in a much more compact problem representation in comparison to the Boolean level. However, the higher level of abstraction can be fully exploited by the respective solver only when a problem is considered that requires many complex constraints. For graph coloring a run-time advantage for representation in CNF has been observed. In contrast, using bit-vector logic in the synthesis of reversible logic leads to speed-ups of more than three orders of magnitude in the best case.

In the future, we expect further progress in the development of more efficient SMT solvers as well as word level

Table 4. Iterative approach for synthesis of reversible logic

BENCHMARK		CNF ENCODING (MINISAT)			BIT-VECTOR ENCODING (YICES)			IMPROVEMENT
	d	#VARS	#CLAUSES	TIME (S)	#VARS	#BV-CONSTR	TIME (S)	
ham3	5	9,180	37,380	0.60	430	1,940	**0.33**	1.82
3_17	6	12,828	52,284	0.96	594	2,700	**0.93**	1.03
hwb4	11	254,540	1,156,672	47,111.40	3,476	18,238	**7,133.06**	6.60
mod5d1	7	624,036	2,955,904	2,061.59	2,968	17,080	**153.92**	13.39
graycode6	5	1,810,680	8,857,920	570.99	4,458	28,089	**1.88**	303.72
decod24	6	81,150	368,064	7.78	1,242	6,075	**3.11**	2.50
rd32-v0	4	38,716	175,360	4.01	628	2,966	**0.36**	11.14
rd32-v1	5	58,010	262,960	13.58	910	4,385	**1.35**	10.06
majority3	3	3,696	14,952	0.08	204	840	**0.03**	2.67
4mod5-v0	5	334,505	1,583,200	122.74	1,790	9,550	**7.96**	15.42
4mod5-v1	5	334,505	1,583,200	407.22	1,790	9,550	**22.44**	18.15
ALU-v0	6	468,147	2,216,320	1,970.77	2,442	13,242	**1,269.30**	1.55
Average								32.34

solvers. Thus, more optimization problems will benefit from higher level of abstractions by shorter overall runtimes. In addition exploiting the higher level representation for problem specific strategies has to be considered in future work. As shown for example in [25] this may lead to further improvements.

References

[1] The Satisfiability Modulo Theories Competition (SMT-COMP). `www.smtcomp.org`, 2007.

[2] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *SPIN*, pages 146–162, 2006.

[3] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. The MathSAT 3 System. In *Int. Conf. on Automated Deduction*, 2005.

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

[5] S. Deng, J. Bian, W. Wu, X. Yang, and Y. Zhao. EHSAT: An efficient rtl satisfiability solver using an extended dpll procedure. In *Design Automation Conf.*, pages 588–593, 2007.

[6] R. Drechsler. Using word-level information in formal hardware verification. *Automation and Remote Control*, 65(4):963–977, 2004.

[7] B. Dutertre and L. Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification*, volume 4114 of *LNCS*, pages 81–94, 2006.

[8] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[9] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, volume 4590, pages 519–531, 2007.

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

[11] D. Große, X. Chen, G. Dueck, and R. Drechsler. Exact SAT-based Toffoli network synthesis. In *Great Lakes Symp. VLSI*, pages 96–101, 2007.

[12] C. Haubelt, J. Teich, R. Feldmann, and B. Monien. SAT-based techniques in system synthesis. In *Design, Automation and Test in Europe*, pages 11168–11169, 2003.

[13] D. E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. ACM, New York, NY, USA, 1993.

[14] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. Smt techniques for fast predicate abstraction. In *Computer Aided Verification*, volume 4144, pages 424–437. Springer, 2006.

[15] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

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

[17] R. Nieuwenhuis and A. Oliveras. On SAT modulo theories and optimization problems. In *SAT*, volume 4121, pages 156–169. Springer, 2006.

[18] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.

[19] L. Ryan. Efficient Algorithms for Clause Learning SAT Solvers. Master's thesis, Simon Fraser University, Canada, 2004.

[20] A. Sülflow, U. Kühne, R. Wille, D. Große, and R. Drechsler. Evaluation of SAT like proof techniques for formal verification of word level circuits. In *IEEE 8th Workshop on RTL and High Level Testing (WRTLT'07)*, pages 31–36, 2007.

[21] T. Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644, London, UK, 1980. Springer-Verlag.

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

[23] M. N. Velev. Exploiting hierarchy and structure to efficiently solve graph coloring as SAT. In *Int'l Conf. on CAD*, pages 135–142, 2007.

[24] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. Sword: A SAT like prover using word level information. In *Int'l Conf. on Very Large Scale Integration*, pages 88–93, 2007.

[25] R. Wille and D. Große. Fast Exact Toffoli Network Synthesis of Reversible Logic. In *Int'l Conf. on CAD*, pages 60–64, 2007.

[26] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. RevLib: an online resource for reversible functions and reversible circuits. In *Int'l Symp. on Multi-Valued Logic*, 2008. RebLiv is available at http://www.revlib.org.

[27] Z. Zeng, K. R. Talupuru, and M. J. Ciesielski. Functional test generation based on word-level SAT. *Journal of Systems Architecture*, 51(8):488–511, 2005.