A Novel SAT All-Solutions Solver for Efficient Preimage Computation [*]

Bin Li
Department of ECE
Virginia Tech.
Blacksburg, VA, 24061

Michael S. Hsiao
Department of ECE
Virginia Tech.
Blacksburg, VA, 24061

Shuo Sheng
Mentor Graphics Corporation
8005 SW Boeckman Rd.
Wilsonville, OR 97070

Abstract

In this paper, we present a novel all-solutions preimage SAT solver, SOLALL, with the following features: (1) a new success-driven learning algorithm employing smaller cut sets; (2) a marked CNF database non-trivially combining success/conflict-driven learning; (3) quantified-jump-back dynamically quantifying primary input variables from the preimage; (4) improved free BDD built on the fly, saving memory and avoiding inclusion of PI variables; finally, (5) a practical method of storing all solutions into a canonical OBDD format. Experimental results demonstrated the efficiency of the proposed approach for very large sequential circuits.

1. Introduction

Preimage and image computations are important operations in formal verification. Conventional methods for computing images and preimages have relied on Ordered Binary Decision Diagrams (OBDDs). However, since OBDDs are very sensitive to the problem size in that the memory required can grow exponentially with the size of the function, the resulting BDDs can easily exceed the memory available. Although methods to improve variable ordering [1, 2, 3] and partitioning of the OBDDs [4, 5] have been proposed to reduce memory explosion, they inherently are still limited by space. To alleviate this problem, methods that are less vulnerable from memory explosion can be used. Such methods include automatic test pattern generation (ATPG) and satisfiability (SAT) solvers. However, because conventional ATPG and SAT solvers are targeted at solving a single solution, extending them to solve image/preimage will require that they return *all* solutions for a given initial state. A naive way of solving all solutions is by simply enforcing the solver to continue the search after obtaining each solution. However, this will result in temporal explosion, in which exponential time will be needed. Therefore, methods to overcome these hurdles will be crucial in the success of an all-solutions ATPG or SAT solver.

SAT solvers and ATPGs have their individual merits. While modern SAT solvers employ conflict-driven learning [6, 7, 8, 9] to dynamically learn from mistakes, ATPGs use testability measures and circuit structure to guide the search [10]. Instead of targeting both ATPG and SAT solvers, in this paper, we will focus only on SAT to extend it to efficiently solve all solutions.

A previous work has been proposed that combines the advantages from BDD and SAT/ATPG solvers [11]. A SAT solver is used to partition the search space first. Then, BDDs-based methods are applied to obtain all solutions in each of the subspaces. Note that this approach may still be limited by the space requirement of individual BDDs. Recently, quantifier elimination combined with a SAT-based solver has been proposed for symbolic model checking [12]. In a separate approach [13], satisfiability cubes are generated to narrow the search space. In both of these methods, a new “block clause” [12] or “satisfiability cube” [13] is created as soon as SAT solver finds a new solution. The block clauses (or satisfiability cubes) are added to prevent the SAT solver from entering the solution spaces identified so far. In [12] the efficiency comes from maximizing the cubes by quantification, thus fewer literals remain in the block clause. For the rest of the paper, we will call this kind of learning **Type-I success-driven learning**, since clauses are added based on successes.

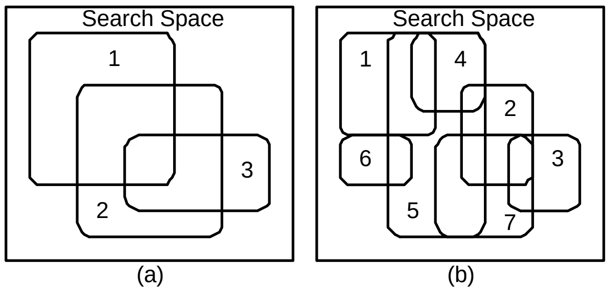

Figure 1. Effect of Cube Size On Solutions

Figure 1 illustrates why solutions cube sizes can influence the SAT performance mentioned above. For instance, in the Figure 1(a), where every cube has its maximal size, three solution cubes are sufficient to cover the entire solution space. On the other hand, in the Figure 1(b), if solutions

[*]This research is supported in part by NSF Grants CCR-0196470 and CCR-0305881.

cube size are smaller, potentially many more solutions can result. Bigger cubes also constrain the SAT-solver better from entering the solution space again.

Another recent all-solutions SAT solver [14] also adds solutions into CNF database. However, it uses *Espresso* to combine small cubes to form larger ones. In addition, an unsatisfied clause database is stored at each step to prune future searched space. Because storage of many temporary clause database requires large memory spaces, its learning may be limited.

On the ATPG side, an all-solutions ATPG by using a different type of success-driven learning to prune redundant search space was recently proposed [15]. Rather than using the solutions to form block clauses as in [12], cutsets in the solution space are formed to narrow the search space. A free BDD is generated at the end, representing the computed preimage. Because this success-driven learning is different from [12], we call it **Type-II success-driven learning** in this paper.

However, type-II success-driven learning in the ATPG-context [15] also has its limitations. First, it does not have conflict-driven learning such as in modern SAT solvers to learn from mistakes. Second, it does not offer a method to quantify the primary input (PI) variables from the final preimage represented as a free BDD.

In this paper, our main contribution is on building an efficient all-solutions SAT solver for preimage computation, which we call "SOLALL". Many novel features are included in our framework in order to achieve this very challenging task. We note that these features can also be applied on image computation also, although we will focus on preimage computation in this paper. It uses type-II success-driven learning rather than "block clauses" [12] or "satisfiability cubes" [13] because we would not need to insert a clause for every unique solution obtained; instead, only cutsets for solution subspaces are computed and stored. Because the size of cut sets for type-II success-driven learning has a direct impact on the overall performance, we propose a new pruning method that results in smaller cut sets than those obtained in [15], thus saving additional memory usage. Next, we combine type-II success-driven learning with conflict-driven learning via a marked CNF database to enhance its performance. We show that combining conflict-driven and type-II success-driving learning is a non-trivial task in this paper. In order to remove the PI variables from the resulting preimage, SOLALL uses a "quantified jump back" technique to dynamically remove the PI variables on the leaves of decision tree. In doing so, the search space that needs to be explored is also reduced. Finally, we present a practical method to store all solutions for the preimage into a canonical OBDD format, which helps us verify the result. Experimental results demonstrated the efficiency of the proposed approach for very large sequential circuits.

The rest of this paper is organized as follows: Section 2 gives the preliminaries and search space reduction. Section 3 describes the optimized cut sets. Section 4 explains the method for combining conflict-driven and success-driven learning. Section 5 provides the practical method for removing PIs. Section 6 demonstrates how to transfer our result into a OBDD for future verification. Experimental results are reported in Section 7 and Section 8 concludes the paper.

2. Preliminaries

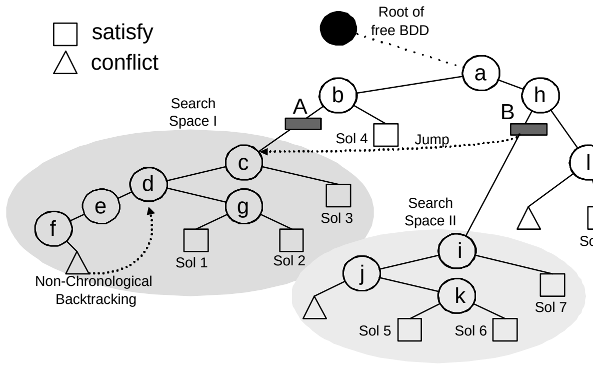

Figure 2. Conflict/Success-Driven Learning

In recent years, conflict-driven learning [6] has been added to enhance SAT solvers' performance. An example is shown in Figure 2 to briefly illustrate the basic concepts of conflict-driven learning [6]. In this figure, when the SAT solver encounters a conflict at node f, after performing conflict analysis (which indicates the conflict is introduced by an error decision at node d), the SAT solver then backtracks directly to node d rather than the previous-level node e. This is termed *non-chronological backtracking* [6]. The SAT solver obtains a *conflict clause* from that conflict by a traversal of the implication graph and adds it into CNF database. Conflict clauses can allow the SAT solver to avoid future conflicts (when it enters the same conflict space again) and allow for backtracks sooner.

Conventional SAT solvers will stop as soon as they find the first solution *sol* 1 at node g. If it continues to search the rest of the space, it can obtain all solutions given enough time. However, because many solutions can overlap, it is possible that different decisions in the decision tree can lead to the same search space. Using the same example shown in Figure 2, the entire search space I under point A can be the same as search space II under point B. We can avoid re-exploring the search space II by noting that the search state is the same, and simply add a pointer linking B to A [15]. Identification of the equivalence of space I and II is performed by computing a *cut set* in the circuit each time a new decision is made. Because the cut set under point B is identical to the cut set under point A, we can safely conclude that the solution space for point B will be identical to that under A. The added pointer between B and A is termed *jump*. All solutions that result from this process can simply be stored in a pruned graph that each leaf has either a solution or conflict. This graph is essentially a *free BDD*. This is why we call it *type-II success-driven learning*, as the learning is invoked by noting regions that contain solutions (successes).

2.1. Static & Dynamic Search Space Reduction

Similar to [16], structural information embedded in the CNF can be used to enhance the variable selection. With the help of this inherent structural information, we just need to justify variables that belong to the corresponding Objective Cone. An *Objective Cone* for a given objective is the fan-in cone from the target objective (primary output and next state variables). If there are several objectives, we can easily merge all the objective cones into one region.

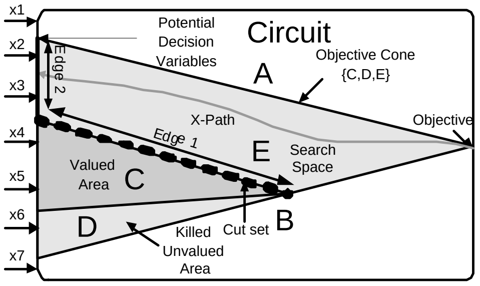

Figure 3. Decision Variables and Cut Set

Regions $\{C, D, E\}$ in Figure 3 together form the objective cone for the objective shown. Variables $x2$, $x3$, ... $x6$ are the only PI and FF (flip-flop) decision variables that need to be considered. $x1$ and $x7$ can be ignored, leaving some variables within regions $\{A, B\}$ unspecified. Because the objective cone is computed a priori, we call this space reduction process *static analysis*.

In addition to reducing the size, not every variable in the Objective Cone needs to be valued. Suppose $x4$ and $x5$ have already been valued, and all variables in area C have been implied, then we do not need to justify area D (even though D is also in the objective cone). In essence, the individual solutions in area D will not affect the other areas $\{C, E\}$ in the objective cone and the solution cubes we have obtained have already covered the solutions in D. The existence of areas such as D depends on current variable values, and it may change during the searching procedure. Thus, we call this space reduction process *dynamic analysis*.

3. Proposed Cut Set Reduction

The observation that the same solution subspace (contains multiple solutions) can be reached multiple times prompted the use of type-II success-driven learning [15]. A cut set for the search state is stored so that it can check if the current search subspace has been reached before. In [15], each cut set is composed of (1) all unvalued PIs and FFs at the end of X-paths (a path of unvalued variables from the objective ($Edge2$ in Figure 3)), and (2) the frontier of all valued gates in the objective cone ($Edge1$ in Figure 3). Note that there must exist X-paths from the each valued node at the frontier to the target objective.

In a circuit with a target objective, we will show that the frontier valued nodes in the objective cone ($Edge1$) is sufficient to uniquely identify the search subspace. The following discussion is used to illustrate our proposed cutset reduction.

Definition 1 *The space containing all feasible solutions is called the* **search space** *for a given problem.*

Definition 2 *If a set of valued variables can satisfy all clauses for a given CNF formula, then that set of valued variables is a* **candidate solution** *for this problem. The candidate solution can be represented as a vector, $\overrightarrow{v}$, consisting of the valued variables.*

Definition 3 *An* **unjustified gate** *is a gate whose output value is specified but is not justified by its input values.*

Observation 1 *At any given time during our search process (before the objective is satisfied), all gates other than the objective gate are either justified or un-valued, and the objective gate is the only unjustified node.*

This observation can be explained as follows. Initially, the objective variable is specified but unjustified because all nodes in the objective cone have not been valued. Because we select the variables according to the constructed circuit structure (explained in Section 2), we will always start selecting from the PI and FF variables; consequently, any newly valued variable θ will always be justified. Although θ may imply other variables to take on a certain value, the implied variables are also justified by θ due to the method with which we select our decisions. Therefore, at any time before the entire CNF is satisfied, only the variable corresponding to the objective gate is unjustified.

Lemma 1 *Only the un-valued PIs and FFs at the end of the X-paths from the objective can determine whether the objective can be satisfied.*

Proof: We will prove this by contradiction. Based on Observation 1, the objective variable can be the only unjustified variable during the search process before a solution is found. Further, there exist a frontier corresponding to the current valued PI and FF variables. In order to advance this frontier, at least one of the PI and FF variables at the end of X-paths from the objective must be valued. If a valuation on *all* the PI and FF variables at the end of the X-paths from the objective neither satisfies the objective nor causes a conflict, then there must still exist at least one X-path from the objective node to a PI or FF. But since *all* PIs and FFs that lie on X-paths in the objective cone have been valued, there can no longer exist any more X-paths from the objective node to a PI or FF, thus a contradiction. ⋄

Proposition 1 *The search space at any point during the search process needs to include only those nodes with at least one X-path to the objective.*

This proposition follows from the fact that if a node, n, does not have an X-path to the objective gate, then any value on n will not propagate to any node that can affect the target objective node. Using Figure 3, region E is the current search space, if variables in region C have been valued.

Lemma 2 *Let region R denote the search space obtained from Proposition 1. Any CNF clause that does not contain at least one variable in region R can be removed from the CNF database without changing the search space for the target objective.*

Proof: If a clause, α, contains no variables in R, then α must not be associated with any gate within R. Thus, there does not exist any X-path between any variable in α to the objective node. $\diamond$

Lemma 3 *If the first variable in a CNF clause is not inside region R, this clause can be removed from the CNF database without changing the search space for the target objective.*

Proof: Because all clauses are created based on the circuit structural information, there is always a variable in each CNF clause that indicates the output variable for a gate. The variables in a clause is arranged such that the output is placed as the first literal. If the output of the gate is not within region R, it will not have a path to the objective since it is not in the objective cone. $\diamond$

Theorem 1 *The* frontier *of all valued nodes inside the objective cone can uniquely determine the search space.*

Proof: We will prove this by contradiction. Suppose there are two different search subspaces R and R' for the same frontier f such that $R \neq R'$. Let a variable $\theta \in R$ but $\theta \notin R'$. Because $\theta \in R$, there must exist an X-path ϕ_1 from θ to the objective o. Because $\theta \notin R'$, there must not exist an X-path, ϕ_2, from θ to the same objective o in the same circuit. However, since we know that there exists at least one X-path ϕ_1 from θ to o, θ must also belong to R' as well, thus a contradiction. $\diamond$

Based on Theorem 1, the cutsets (frontiers) we obtain for the search subspaces can be significantly smaller than those obtained in [15].

4. Combined Learning

While conflict-driven learning in state-of-art SAT solvers can be very powerful, conflict-driven learning cannot directly combine with type-II success-driven learning in an all-solutions SAT solver. This is because the knowledge learned by conflict-driven learning can interfere with the computation of the cut sets for the search subspaces. We will use Figure 4 to illustrate this.

In Figure 4 (a), the cut set and decision variables are obtained after a subset of variables has been valued. It is possible that later, as shown in Figure 4 (b), that the solver enters the same search space again. However, this time it may have learned more knowledge (implications) from conflict-driven learning via the added conflict clauses (shown by the circle within the objective cone). Consequently, the cut sets that correspond to the same search space may no longer match any more with the previously computed cut set. This can significantly impair the success-driven learning.

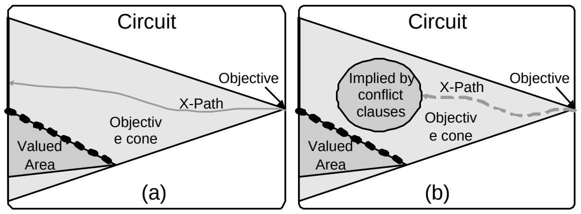

Figure 4. Conflict Clauses Effecting

We solve this problem by a marked CNF database. All original clauses are marked at the beginning. Conflict-driven learning is performed on the *entire* database (marked and unmarked clauses); however, the conflict-induced clauses are never marked. By using only the marked clauses for cut set computation and next decision variable selection, the cut sets corresponding to the current search space will always be the same without being interfered by the intermediately implied values.

5. Remove PIs From Preimage

In preimage computation, we only want the FFs in our final solution, and the PIs should not be included in the preimage. In other words, two different solutions that differ only in the PI variables can be viewed as the same solution. In SOLALL, we select decision variable by favoring FFs over PIs. Thus, the variables near the top of the decision tree will belong to FF variables, while PIs are always lower in the decision tree.

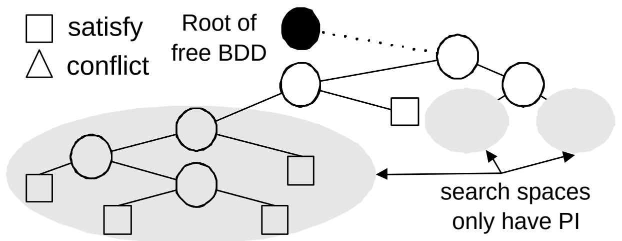

Figure 5. Quantified Jump Back

For example in Figure 5, the variables inside the gray regions denote PI variables. Then, as soon as one solution is found that involves at least one PI variable, the solver can jump out of the corresponding gray area, and directly continue to search starting from the earlier FF variable. We call this *Quantified Jump Back*(QJB), which is similar with the work in [12]. However, QJB can further reduce the search space because not all solutions in the gray region needs to be fully explored for the preimage. In addition, it allows for the removal of PI variables on the leaves of the decision tree to save memory, and reduce total number of solutions as well.

QJB may not guarantee that all PIs will be quantified because some PIs may be implied early on by some FFs.

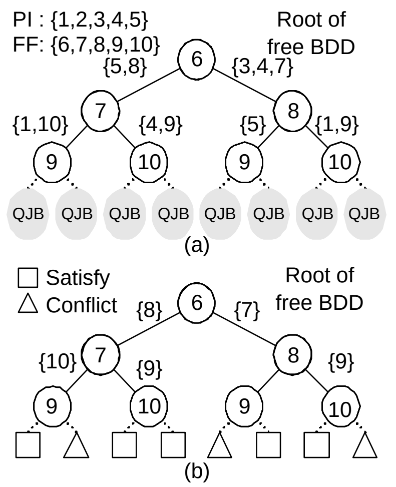

Figure 6. Remove PIs from Free BDD

In order to remove these implied PIs, we use the following approach: decision variables are stored in the BDD *nodes*, while implied variables are stored on the *edges* of the BDD. For example, in Figure 6, there are ten variables. Variables 1 to 5 are PIs, 6 to 10 are FFs. Since FFs have the highest priority, a free BDD for the preimage should only store FF variables $\{6, 7, 8, 9, 10\}$. In Figure 6 (a), decision to the left branch of node 6 implies $\{5, 8\}$. Decision to the right branch of node 6 implies $\{3, 4, 7\}$. These implied variables are associated with their corresponding edges. Now we can simply remove implied PI variables on the BDD edges without changing the free BDD structure. (Note that the implied FF variables are still on the edges.) Those PIs in gray leaves are only those that have not been implied earlier, and via QJB they can be easily removed from the BDD. The final free BDD without PIs is shown in Figure 6 (b). Clearly, the total number of solutions is the same for both (a) and (b) of the figure.

6. Convert Free BDD to an OBDD

So far, all solutions computed are stored into a free BDD. Ordered BDD (OBDD) is a BDD in which variables on every path follow the same variables order. Although the size of an OBDD can be larger than a free BDD, OBDDs have many nice properties such as allowing the comparison of two preimage BDDs, etc. However, converting a free BDD into an OBDD is known to be a NP-hard problem.

In this work, we propose a simple and practical method to obtain an OBDD from the result of SOLALL. Given a variable order, SOLALL can be forced to select next decision variable accordingly. Finally, decision variables on all paths in the free BDD now have the same variable order. The free BDD we obtain in SOLALL can easily be converted into an OBDD using the following three-step process. Each step can keep the total solutions number unchanged.

- Step 1: In Figure 2, Since cut-set at A is the same as the cut set at B, we can simply copy all the sub-trees under cut set A to cut set B by converting the free BDD to a tree-like BDD.
- Step 2: In the tree BDD, there may still exist implied variables on the edges of the BDD. These implied variables are pushed down to their corresponding position based on a given variables order. This is done recursively by pushing the variable down one level at a time.
- Step 3: Finally, we convert the implied variables on the edges into BDD nodes. Since variables have been ordered in step 2, this BDD is the final OBDD.

The OBDD obtained from the above 3-step process will preserve the correctness in the original free BDD. The proofs are lengthy and thus are omitted here.

7. Experimental Results

The proposed SOLALL is written in C++. All our experiments were conducted on 1.7 GHz Pentium 4 PC with 512MB RAM, running RedHat Linux 7.3. The effectiveness of our proposed method is evaluated for the ISCAS89 sequential benchmark circuits using EG properties involving conjunction of 10 random flip-flops for circuits larger than s1423, while conjunction of 5 random flip-flops were used for the smaller circuits.

In order to compare the efficiency of the two types of success-driven learning, we build another SAT based solver with *success-driven Type-I*, similar to [12], in which quantification on the solution is performed before adding the block clause into the CNF database. This solver is denoted as *Type I*, Our proposed SAT solver is denoted as *Type II*.

Table 1. Type-I vs. Type-II Learning

	Type-I Success + Conflict		SOLALL (Type-II Success + Conflict)	
circuit	time(s)	mem(M)	time(s)	mem(M)
s344	0.004	1.3	**0.002**	**1.3**
s1196	**0.002**	**1.3**	0.004	1.3
s1423	64.4	93.2	**0.315**	**3.2**
s5378	0.412	4.34	**0.033**	**3.47**
s9234	Abort(>231)	>128	**0.054**	**6.3**
s13207	4.02	44	**0.027**	**2.53**
s15850	32.4	71.2	**0.158**	**10.0**
s35932	Abort(>141)	>128	**0.20**	**13.0**
s38417	Abort(>136)	>128	**0.055**	**9.23**
s38584	Abort(>204)	>128	**0.071**	**14.1**

We report results in Table 1. For each circuit, the execution time and memory used are reported for each circuit for both Type-I and Type-II success-driven learning. Note that conflict-driven learning is also incorporated in both methods. According to the results, *Type-I* cannot complete for several of larger circuit, while our *Type-II* (SOLALL) can return all solutions for the preimage of all circuits. In addition, SOLALL is generally faster and uses less memory(exclude s1196); in particular, a maximum memory size of 14.1MB is ever encountered, and less than one second of execution are needed for *all* circuits.

Table 2. SOLALL vs. Others for s5378

	ATPG-based [15]		BDD-based [17]		Type-I Success + Conflict		SOLALL (Type-II Success + Conflict)	
property	time(s)	mem(M)	time(s)	mem(M)	time(s)	mem(M)	time(s)	mem(M)
1	0.62	10	14.7	27	0.412	4.34	**0.033**	**3.47**
2	0.42	10	14.2	27	0.121	14.1	**0.08**	**4.56**
3	**0.07**	10	14.7	27	Abort(>105.7)	>128	0.141	**4.21**
4	0.51	10	14.8	27	0.071	2.12	**0.026**	**1.80**
5	**0.1**	10	14.7	27	Abort(>100.3)	>128	0.245	**4.36**
6	1.03	10	14.8	27	0.752	4.41	**0.093**	**4.21**
7	1.1	10	14.7	27	0.098	3.75	**0.004**	**1.39**
8	1.97	10	14.7	27	Abort(>79.76)	>128	**0.168**	**4.72**
9	0.98	10	14.8	27	0.041	2.12	**0.021**	**1.39**
10	1.71	10	14.7	27	1.202	4.33	**0.027**	**3.81**

Note 1: the BDD-based method was run on 200MHz Sun workstation

Note 2: the BDD-based method could not handle larger ISCAS89 circuits, such as s38417, though not shown here

In the second experiment, we compared SOLALL with the success-driven learning ATPG engine [15] and a BDD-based approach [17] for 10 EG properties for circuit s5378. Table 2 reports the results. Again, execution time and memory used are reported for each property. Although SOLALL has not incorporated circuit testability measures, from the results it can be seen that its execution time is on the same order with [15], despite that [15] uses sequential implication learning to prune additional search space. Memory usage for SOLALL is also less than [15]. When compared with BDD-based solver BINGO, SOLALL consistently performed better. However, note that BINGO fails larger problem instances, as reported in [15], while both ATPG and SAT-based methods scale much better.

8. Conclusions

We have presented a novel and efficient all-solutions SAT solver, SOLALL. SOLALL can be applied to efficiently compute preimage. The search space combined success/conflict-driven learning. In addition, a method for computing smaller cut sets is proposed. Finally, a practical method to convert SOLALL results into an OBDD without PIs is given. Experimental results show that SOLALL is very efficient in returning all solutions and is also very memory-efficient. Application of SOLALL to large sequential circuits showed that SOLALL can successfully return all solution in less than one second. We believe this work holds potential for formal verification of large-scale sequential problems.

References

[1] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. *Proc. ICCAD*, pages 42–47, Nov. 1994.

[2] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. *Proc. ICCAD*, pages 628–631, 1994.

[3] C. Meinel and C. Stangier. Speeding up symbolic model checking by accelerating dynamic variable ordering. *Great Lake Symp. on VLSI*, pages 39–42, 2000.

[4] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned robdds: A compact, canonical and efficiently manipulable representation for boolean functions. *Proc. ICCAD*, pages 547–554, Nov. 1996.

[5] I.-H. Moon, H. H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. *Proc. DAC*, pages 23–28, 2000.

[6] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. On Computers*, 48(5):506–521, May 1999.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. *Proc. DAC*, pages 530–535, 2001.

[8] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. *Proc. ICCAD*, pages 279–285, Nov. 2001.

[9] E. Goldberg and Y. Novikov. Berkmin: a fast and robust sat-solver. *Proc. DATE*, pages 142–149, 2002.

[10] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, Piscataway, New Jersey, 1990.

[11] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik. Partition-based decision heuristics for image computation using sat and bdds. *Proc. ICCAD*, pages 286–292, 2001.

[12] K. L. McMillan. Applying sat methods in unbounded symbolic model checking. *Proc. CAV*, pages 250–264, 2002.

[13] L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified boolean formula evaluation. *Proc. of 8th Int'l Conf. on Principles and Practice of Constraint Programming*, pages 200–215, Sep. 2002.

[14] H.-J. Kang and I.-C. Park. Sat-based unbounded symbolic model checking. *Proc. DAC*, pages 840–843, Sep. 2003.

[15] S. Sheng and M. S. Hsiao. Efficient preimage computation using a novel success-driven atpg. *Proc. DATE*, pages 822–827, Sep. 2003.

[16] A. Gupta, A. Gupta, Z. Yang, and P. Ashar. Dynamic detection and removal of inactive clauses in sat with application in image computation. *Proc. DAC*, pages 536–541, 2001.

[17] H. Iwashita and T. Nakata. Forward model checking techniques oriented to buggy designs. *Proc. ICCAD*, pages 400–404, Nov. 1997.