Efficient Preimage Computation Using A Novel Success-Driven ATPG [*]

Shuo Sheng
Department of ECE, Rutgers University
Piscataway, NJ, 08854
shuo@ece.rutgers.edu

Michael Hsiao
Bradley department of ECE, Virginia Tech.
Blacksburg, VA, 24061
hsiao@vt.edu

Abstract

Preimage computation is a key step in formal verification. Pure OBDD-based symbolic method is vulnerable to the space-explosion problem. On the other hand, conventional ATPG/SAT-based method can handle large designs but can suffer from time explosion. Unlike methods that combine ATPG/SAT and OBDD, we present a novel success-driven learning algorithm which significantly accelerates a ATPG engine for enumerating all solutions (preimages). The algorithm effectively prunes redundant search space due to overlapped solutions and constructs a free BDD on the fly so that it becomes the representation of the preimage set at the end. Experimental results have demonstrated the effectiveness of the approach, in which we are able to compute preimages for large sequential circuits, where OBDD-based method fail.

1 Introduction

Preimage and image computation is an important step in many formal verification and ATPG applications. We focus on preimage computation in this paper, in which the problem is defined as finding all the states that can reach a set of present states in one or more transitions.

Symbolic methods based on Ordered Binary Decision Diagrams (OBDDs) for preimage computation have been successful for small and medium sized circuits. In these methods, both the present states and the circuit transition function are represented by OBDDs. Through the existential quantification procedure, the set of preimages can be efficiently computed and coherently represented as another OBDD. However, this approach works well only when BDD construction is possible for both present state sets and circuit transition relations. For large designs, OBDD representation for the entire transition relation usually cannot be constructed. Even when it can be constructed using methods such as partitioned BDDs [5, 6], the existential quantification operation can still cause memory explosion.

On the other hand, an ATPG or SAT-solver engine can be used to manipulate the circuit transition function instead of BDDs. The advantage is that they can handle much larger circuits without memory explosion; the disadvantage is that, due to their branch-and-bound nature, many subproblem computations are repeated. In essence, ATPG/SAT trades off time with space. In [7], a method of combining SAT-solver and OBDD for image computation was proposed. The circuit transition function is represented by a CNF formula. SAT-solver performs a high-level decomposition of the search space and BDD is used to compute all solutions below the intermediate points in SAT decision tree, which is referred to as "BDDs at SAT leaves". As an extended work, a decision heuristic based on separator-set induced partitioning for SAT-solver was proposed in [8], which yielded simpler BDD subproblems. However, these two work still fall into the framework of partitioned BDDs: SAT-solver is used to compute disjunctive decomposition of the problem and the decomposed problems are handled by OBDDs.

To our knowledge, there has not been much work on efficient application of pure ATPG or SAT solver as the main engine for preimage/image computation. The reason that prevents it is: preimage/image computation requires the engine to enumerate all solutions; the decision procedure employed in SAT-solver and ATPG are usually based on branch-and-bound algorithms (e.g., Davis-Putnam, PODEM, etc.); these algorithms are optimized to derive one solution at a time, as opposed to BDD which captures all solutions simultaneously. To use SAT or ATPG to enumerate multiple solutions, backtracks are enforced so that the algorithm can continue searching for the next solution when one is found. If the preimage set is large, e.g. it contains millions of solutions, then enumerating and storing them one at a time is obviously impossible due to both time and memory limitation. The ATPG/SAT engine, in this case, will suffer from both time and space explosion problems. We call this phenomenon as *solution explosion*. This is also the reason why the authors of [7] avoid having the SAT-solver run to a leaf node of the decision tree, but instead they let BDDs finish off the decision tree starting at intermediate nodes.

In a recent work [9], a SAT-solver is employed to perform quantifier elimination for symbolic model checking. The problem is similar to our problem of enumerating all solutions. The author relied on CNF representation of circuit and modified zCHAFF [4] such that whenever a solution is found, a "blocking clause" is constructed to prevent the SAT-solver from being trapped into the same solution again. This is equivalent to

[*]Supported in part by NSF Grant CCR-0196470 and NJ Commission on Science & Technology.

enforcing a implication and a backtrack to drive the SAT engine for enumerating all solutions. However, it is unclear in the paper if the aforementioned "solution explosion" problem has been addressed. Therefore, we suspect that this method is applicable only to the case where nice CNF representations of the problem and result exist.

In this paper, our main contribution is on developing a method to solve the "solution explosion" problem so that we are able to apply a ATPG engine to compute preimages efficiently. In contrast to *conflict(failure)-driven learning* employed in GRASP [3] and CHAFF [4], we developed a *success-driven learning* algorithm on top of a deterministic combinational ATPG engine for enumerating all solutions. It employs *search state equivalence* analysis (in contrast to conflict analysis) to capture and reuse the knowledge it learns from previous solutions so that many subproblems are skipped during the search for multiple solutions. A free BDD that stores every derived solution is gradually formed as the ATPG decision process proceeds so that it becomes the representation of the preimage set at the end. Notice that our approach calls for neither OBDD construction of circuit transition function nor existential quantification, and BDD is only used to represent the present and the computed preimage state sets, the memory explosion problem is significantly reduced. Solution-based, or satisfiability-directed learning has been proposed in [10], in which the authors worked on a QBF-solver. However, the idea employed there is mainly for pruning repeated searching for universal quantified variables and the ultimate goal is to decide the satisfiability of a QBF formula rather than to enumerate all solutions.

The rest of the paper is organized as following: Section 2 explains the basic idea using an example; Section 3 introduces the search state equivalence concept; Section 4 details the overall success-driven learning algorithm; Section 5 describes how BDD is constructed during the ATPG process for representing the preimage set; experimental results are reported in Section 6 and Section 7 concludes the paper.

2 Basic Idea

The prototype ATPG algorithm we use is PODEM [1]. A naive way of using PODEM to compute preimages is to enforce a backtrack whenever a solution is found so that the algorithm could continue to search for the next solution until the entire search space is implicitly enumerated. Various search-space pruning techniques have been proposed [3, 4, 10–12] for improving combinational ATPGs and SAT-solvers. However, all these methods target pruning *conflict spaces*; in other words, they *learn from mistakes*, e.g. conflict-driven learning, dependency-directed non-chronological backtracking, conflict clauses, etc. In preimage computation scenario, this is far from being sufficient because solution subspaces can overlap heavily. Further, since these subspaces contain solutions, they do not cause any conflicts with one another. Figure 1 shows that multiple solutions exist and overlap in the search space, where each circle represents a solution. A solution may not be fully specified. For instance, two solution cubes exist to justify a logic 0 for the output of an AND gate: {0X, X0}. The size of the circles in Figure 1 reflects the number of unspecified primary inputs or flip-flops present in a given solution; larger circles indicate more don't-cares PIs/FFs in the solution.

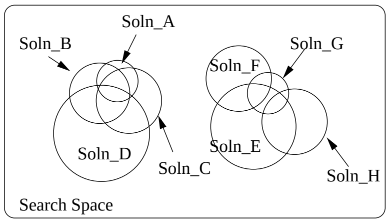

Figure 1. Solution cubes in search space.

When the solution-overlap phenomenon is mapped to the decision tree, it shows as the duplication of many solution structures. We illustrate this in Figure 2 for an ISCAS89 benchmark circuit *s5378*. We show multiple solutions found by ATPG for a target objective in *s5378* in the figure. A circle with a label denotes a decision node; the label is the gate ID in the circuit; the left branch corresponds to the decision of 0 and the right branch corresponds to 1; a triangle denotes a conflict subspace (no solution exists); a rectangle marks the terminal node of a solution. From this figure, we see that there are three solutions under the left branch of decision node 107, marked as Soln#1, Soln#2 and Soln#3. They specify three paths in the decision tree, which actually characterize three *solution cubes*. In addition, all the three solutions have common variable assignments for those variables near the top of the decision tree. The second and third solutions were found by simply enforcing a backtrack and making ATPG continue after the first solution has been found.

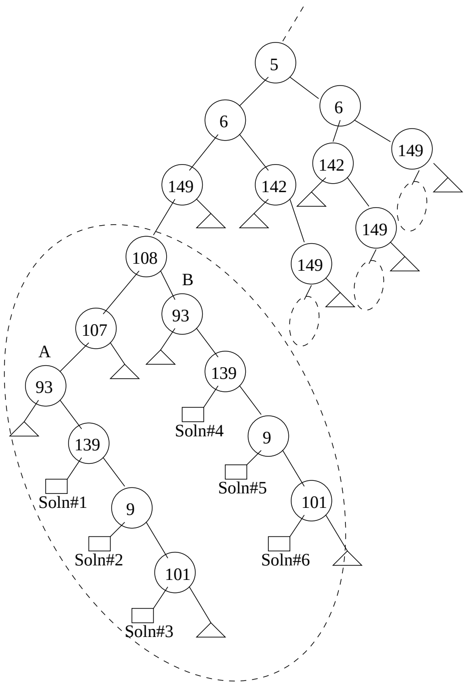

Figure 2. Decision Tree in ATPG.

However, when the ATPG continued backtracking to find the fourth, fifth and the sixth solutions (the three lying under the right branch of node 108, Soln#4, Soln#5 and Soln#6), we observed an interesting phenomenon: the same *partial* assignment for decision nodes $\{93, 139, 9, 101\}$ is repeated. This implies that the search space immediately before the two 93 nodes (marked as A & B in the figure) are "equivalent". In other words, earlier decision node assignments $\{5 = 0, 6 = 0, 149 = 0, 108 = 0, 107 = 0\}$ and $\{5 = 0, 6 = 0, 149 = 0, 108 = 1\}$ resulted the same circuit state. Therefore, if we could learn from the first three solutions, we would be able to skip the search and directly return solutions #4, #5, and #6. When we advanced the search further, such phenomenon was observed again. We found that the entire subtree under the left-most decision node 149 (within the big dotted circle) was repeated again under the other three 149 nodes on the right (denoted by the three small dotted circles). Therefore, if we could learn the structure in the area inside the largest dotted circle (containing 6 solutions) and its corresponding "search space", we would be able to avoid digging into the complete decision tree for the other three by backtracking earlier and return all related solutions. The savings for the "enforced backtracks" for these solutions will be enormous when there are large number of overlapped solutions (preimages) in the search space. Since this learning is invoked by solutions, we call it *success-driven learning*. The subsequent sections will explain our notion of "search state equivalence" and how they are implemented into a "success-driven learning algorithm" and how a free BDD is constructed to represent all the solutions.

3 Search State Equivalence

In a high performance ATPG or SAT-solver, *learning* plays a very important role: the knowledge learned is stored and used for pruning search space in the future, e.g. in SOCRATES [2] the knowledge is in the form of implications and in GRASP [3] and CHAFF [4] it is in the form of conflict clauses. In our approach, the knowledge is equivalent search state.

As shown in Figure 2, we discovered that different complete solutions may share the same partial solution. We will explain this phenomenon again by an example via the circuit fragment shown in Figure 3. There are four PIs (decision nodes) in this circuit: a, b, c, d. The OR-gate z is the PO. Let us assume that we wish to derive all solutions for the objective $z = 1$. It is observed that two different partial PI assignments, $\{a = 0, b = X, c = 1\}$ and $\{a = 0, b = 0, c = X\}$, will result in the same internal circuit state $\{g = 0, f = 0, h = X\}$. Then, to satisfy the objective $z = 1$, we need to set $d = 1$ in both cases, which corresponds to the repeated partial solution at the bottom of the decision tree. We characterize an *equivalent* search state by its cut set with respect to the objective. In this example, the cut set is simply $\{g = 0, f = 1, d = X\}$, which is a unique *internal* circuit state.

The cut set $\{g = 0, f = 1, d = X\}$ consists of two parts: internal specified gates $\{g = 0, f = 1\}$ and unspecified PI gates $\{d = X\}$. They are obtained by the following procedures: 1. beginning from the objective site z (currently unsatisfied), backtrace to PIs along all the X-paths in its fanin cone; 2. record every specified input to all unspecified gate-output encountered in this depth-first search (in our example, there is only one X-path $z - h - d$, $f = 1$ is the specified input of the unspecified gate h and $g = 0$ is the one for unspecified gate z.); 3. record the unspecified PIs at the end of each X-Path ($d = X$ in the example). After performing this depth-first-search all the marked gates (specified gates and unspecified PIs) and their values (1,0,X) define a cut set of the circuit state (shown in dashed line in Figure 3). Notice that *this cut set is rather with respect to the objective than to the entire circuit*. Cut sets are stored in a hash table which is managed as the knowledge database. The algorithm uses this database to determine if an equivalent search space is encountered, and if so, it skips this search space.

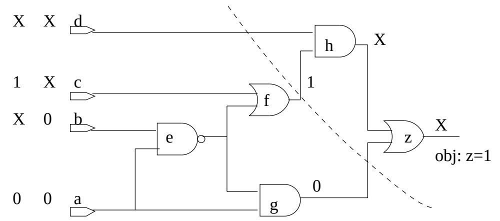

Figure 3. Search State Equivalence.

The idea of employing search state equivalence has been proposed in [13]. However, there are fundamental differences: (1) the state-equivalence in [13] is used to find a single solution while ours targets finding *all* solutions for a single objective. (2) five-value logic is used in [13] to find *a single* solution for a stuck-at fault while we use three-value logic to find *all solutions*; (3) the cut sets in [13] do not include PIs with X values while cut sets in our approach do, because the PIs with don't care values make up the partial solutions at the bottom of the decision tree and thus bound the remaining search space (they need to be remembered to recover the complete solutions); (4) finally, the novelty of our work also lies in that we combine a BDD data structure with the success-driven learning ATPG algorithm to efficiently represent the preimage set.

4 Success-Driven Learning

The success-driven learning algorithm controls when to invoke the search state equivalence learning and when to reuse it. It is built into a basic recursive PODEM algorithm that enumerates the entire search space, shown in Figure 4. Notice that unlike conventional PODEM which returns either SUCCESS or FAIL, this function does not have a return value because we always enforce a backtrack when a solution is found so that it can continue to search for the next solution. However, we do remember how many solutions are found under each node in the decision tree. We keep a *success-counter* (SC) pair for each decision node in the decision stack. This pair, $SC^0(i)$ and $SC^1(i)$, counts the total number of solutions found for the 0 and 1 branch of node i in the decision tree. For example, in Figure 2, for the left most decision node 93 (marked 'A' in figure), $SC^0(93) = 0$ and $SC^1(93) = 3$. The subroutine $update\_success\_counter()$

```
function success_driven_podem() {
    // step 1.  found a solution
    if (objective_satisfied) {
        update_solution_BDD();
        update_success_counter();
        return;
    }
    // step 2. reuse knowledge learned before
    deduce_current_search_state();
    if (lookup_search_state_dateBase()== HIT) {
        update_solution_BDD();
        update_success_counter();
        return;
    }
    // step 3. get a new decision node
     next_decision_node = get_obj();
    if (next_decision_node == NULL)  return;
    // step 4. try left branch of the decision node
     next_decision_node = 0;
    if (imply()== CONFLICT)
          nBackTrack++;
    else
          success_driven_podem();
    // step 5. try right branch of the decision node
     next_decision_node = 1;
    if (imply()== CONFLICT)
          nBackTrack++;
    else
          success_driven_podem();
   // step 6. pop decision node out of decision stack
   if (check_success_counter() > 0) {
          update_search_state_datebase();
   } }
```

Figure 4. Success-Driven Learning Alg.

at step 1 in Figure 4 is called every time a solution is found. It increments all the success counters in the current decision stack. Using the same example in Figure 2, when the solution at the 0 (left)-branch of the left-most decision node 139 (below point 'A') is found, $update\_success\_counter()$ will increment the following success counters: $SC^0(139), SC^1(93)$, $SC^0(107), SC^0(108), SC^0(149)$, $SC^0(6), SC^0(5)$ and all such counters above decision node 5 till the top of the decision tree. Next, when the solution at the left branch of succeeding decision node 9 is found, $update\_success\_counter()$ will perform the same update again, except that $SC^1(139)$ rather than $SC^0(139)$ is updated, since it is on the right branch of node 139. In addition, $SC^0(9)$ is initialized to 1. Note that through this mechanism, all the success counters are synchronized as they are dynamically maintained. When a decision node is popped out of the decision stack, it indicates that the subspace below it has been *fully* explored. Therefore, the two success counters for this node should have the *final* numbers of all solutions in the subspace below it. If any of them is greater than 0 (indicating that there exist at least one solution), we then perform the search state equivalence learning by computing the corresponding cut set and delete the success counters in the subtree. This operation is performed by subroutine $check\_success\_counter()$ and $update\_search\_state\_datebase()$ at step 6. Because we only compute the equivalent search space when there is at least one success (solution), this algorithm is called "success-driven learning". Note that since we only allocate success counters for decision nodes that are active in the current decision stack and since only PIs and FFs can be decisions in PODEM, the maximum number of success counters managed is $2 \times |PIs + FFs|$.

At step 2 of this algorithm, the function computes the current search state and look it up in the knowledge database. If it detects a Hit, then it immediately returns and the entire subspace below is pruned, with solution BDD updated. At steps 4 and 5, the subroutine $imply()$ performs logic simulation and implication when a new decision variable is assigned a specified value. If it detects a conflict, it increments the counter $nBackTrack$ and skips the exploration of the subspace below. At step 6, $update\_search\_state\_datebase()$ creates a new entry in the hash table if a new search state cut set is found. The subroutine $update\_solution\_BDD()$ at steps 1 and 2 constructs the BDD to record every partial solution found by the function. This BDD grows from *partial* to *complete* when the entire search space has been explored and the last decision node in decision stack is popped out. It is explained next.

5 Integrating BDD into ATPG

To avoid solution explosion, a BDD representation of the preimage set is desirable. In this section, we describe how we integrate a BDD data structure into our ATPG.

5.1 Constructing BDD

The main idea comes from the observation that the decision tree in Figure 2 resembles a free BDD, although this BDD may not be canonical. Based on this, we construct the solution-BDD using the success-driven learning procedure.

We define the BDD node data structure in Figure 5. There are six members in the structure. The first is the decision node label, which identifies the PI. The second is the address of the left child. The third is the number of total solutions under the left branch (value passed from the success counter SC_0 during success-driven learning). The next two fields are for the right branch. The last member is the index to the hash table that identifies the search state cut set.

```
typedef struct _BDD_NODE {
     int decision_node_id;
     int left_branch_successor;
     int num_solutions_in_left_branch;
     int right_branch_successor;
     int num_solutions_in_right_branch;
     int search_state_hash_table_index;
} BDD_NODE;
```

Figure 5. BDD Node Data Structure.

Based on our ATPG algorithm, our BDD is built in a *bottom-up* fashion. Whenever a new solution is found, a BDD node is created for the leaf decision node and its associated search state. Its parent nodes are created when the ATPG backtracks to higher levels in the decision tree and detects there are solutions beneath by success counters. Figure 6 shows how the solution BDD is constructed in chronological order for the solutions found in the decision tree in Figure 2. The BDD grows gradually from a single node (Figure 6(a)) to a complete graph (Figure 6(f)), in which each path corresponds to a solution. When a search state equivalence is detected and this knowledge is reused, instead of creating a brand new BDD node, subroutine $update\_solution\_BDD()$ will link the current decision node to an existing BDD node, as shown by those dashed edges in Figure 6. Through this reduction, significant space is saved for storing the overlapping portion of solutions.

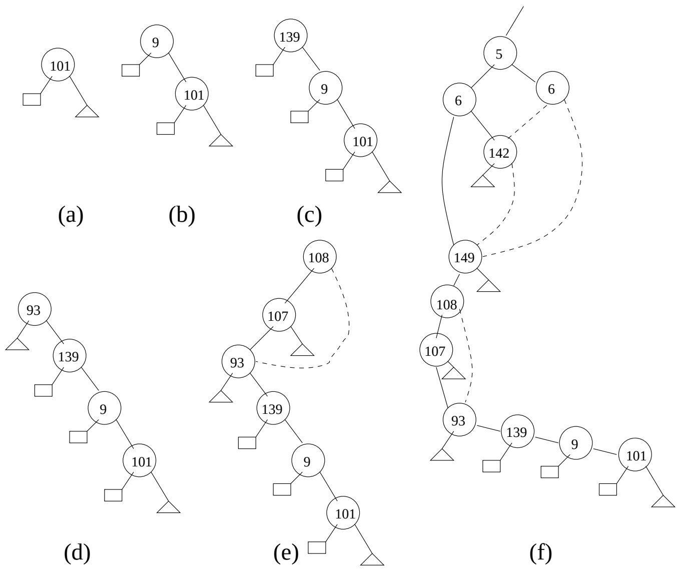

Figure 6. BDD constructed chronologically.

5.2 Variable Ordering

An interesting question would be what decides the variable ordering for this free BDD. As a matter of fact, the variable ordering is implicitly decided by the ATPG decision procedure when it picks a new decision variable. It is dynamically generated by the subroutine $get\_obj()$ at step 3 in the function $success\_driven\_podem()$. The function $get\_obj()$ is a standard function in ATPG that uses controllability and observability as heuristics to find next decision PIs through backtracing along X-paths [1]. Notice that unlike SAT-solvers in which the variable that directly satisfies the largest number of clauses is chosen [3], ATPG's decision variable selection is more guided by the structural information of the circuit and the objective. It has been shown in [14] that such testability-based heuristics often yield a good variable ordering for constructing compact shared ROBDDs.

6 Experimental Results

The success-driven learning algorithm together with a basic PODEM and two ATPG enhancements (improved backtrace with conflict check [12], conflict analysis [11]) were implemented in 5,000 lines of C code. We conducted ATPG experiments on a Pentium 4 1.7 GHz machine with 512MB RAM and Mandrake Linux OS 8.0. The experiments were designed to evaluate the effectiveness of the proposed method and compare its performance to a CUDD-based BDD package, BINGO [15]. The BINGO experiments were conducted on a SUN Ultra-1 200MHz machine with 512MB RAM.

The first set of experiments we conducted was computing 1-cycle preimages for some "properties" of ISCAS benchmark circuits *s5378*. The properties we used are random conjectures of 10 state variables. Therefore, each property actually specifies a set of target states. We applied our success-driven learning algorithm to compute all states which not only can reach the target states in a single transition but also contain the property themselves. This is a typical step for checking *liveness (EG) properties* in model checking, where the property needs to be included in every state in the path.

In Table 1 we show the results for *s5378*, which contains 3043 gates and 179 FFs. Three methods were compared: PODEM without success-driven learning (denoted as NO_SUCC), PODEM with success-driven learning (denoted as SUCC) and BINGO. For the first two ATPG experiments, a backtrack limit of 100,000 is imposed. Under column of "# of soln", we report the number of solution cubes found by the ATPG engine. If the ATPG can not exhaust all solutions within the 100,000 backtrack limit (e.g. for property 4, 6, 7, 8, 10 with the first method) then this number reflects the maximum number of solutions it can enumerate within that backtrack limit. In the middle five columns, since our success-driven learning algorithm constructs a BDD, we also include the "bdd-size" (number of BDD nodes) and the "mem" (peak memory) columns so as to compare with BINGO results in the right-most three columns.

From the table we can see that success-driven learning significantly reduced the execution time (about 2 to 3 orders of magnitude speedup) for finding all solutions when compared to ATPG without success-driven learning, while being memory efficient than BINGO. For example, for property #2, both NO_SUCC and SUCC found all 7,967 solutions; however, NO_SUCC consumed more than eighteen thousand backtracks and 22.6 sec, while SUCC took only 200 backtracks and 0.42 sec in time and 10Mb in memory to finish the job. BINGO took 14.7 sec and 27Mb memory to do the job. The bdd size for BINGO is 230 nodes while our general BDD only contained 139 nodes. For property #6, NO_SUCC could not exhaust all solutions; it only found 22,266 solutions within the 100,000 backtrack limit and took 60.49 sec; SUCC can enumerate all 67,379,200 solutions in 1.03 sec, using only 806 backtracks and 739 nodes to represent these solutions. Note that we do not need a seperate graph traversal to obtain those solution numbers - we just need to add the two solution counters in the root node in our BDD because they have been synchronized to reflect the number of total solutions in the subtrees. Also, even though ATPG trades off time for space, our ATPG's performance (less than 2 sec. on a 1.7 GHz Pentium machine) was on the same order in execution time with BINGO (about 14 sec on a 200MHz Sun UltraSparc). For all ten properties, our ATPG completed the preimage computation using only 10MB for each property, while BINGO required 27MB for each property.

We also performed a similar experiment for a property of circuit *s38417*, which contains 1636 FFs and 23950 gates. The

	NO_SUCC			SUCC					BINGO		
property	# of soln.	# of bktrack	time(s)	# of soln.	# of bktrack	bdd size	time(s)	mem	bdd size	time(s)	mem
1	2,990	12,034	20.1	2,990	319	250	0.62	10M	229	14.7	27M
2	7,967	18,117	22.67	7,967	200	139	0.42	10M	230	14.2	27M
3	5	67	0.08	5	29	23	0.07	10M	219	14.7	27M
4	14,165	>100,000	115.34	250,880	509	283	0.51	10M	1689	14.8	27M
5	1,024	8,218	4.83	1,024	77	76	0.1	10M	1075	14.7	27M
6	22,266	>100,000	60.49	67,379,200	806	739	1.03	10M	3064	14.8	27M
7	14,630	>100,000	44.18	31,928	611	606	1.1	10M	965	14.7	27M
8	16,331	>100,000	61.54	8,630,272	3,555	3551	1.97	10M	2796	14.7	27M
9	4,008	19,826	30.7	4,008	517	395	0.98	10M	3893	14.8	27M
10	20,626	>100,000	77.04	22,750	1,149	494	1.71	10M	506	14.7	27M

Table 1. Compute Preimages for s5378 (179 FFs, 3043 gates)

	NO_SUCC			SUCC					BINGO		
property	# of soln.	# of bktrack	time(s)	# of soln.	# of bktrack	bdd size	time(s)	mem	bdd size	time(s)	mem
1	24,938	>100,000	517	129,171,456	440	438	0.77	187M	abort	abort	>512M

Table 2. Compute Preimages for s38417 (1636 FFs, 23950 gates)

results are shown in Table 2. For this circuit, BINGO failed to construct the transition function due to the 512Mb memory limitation while our proposed method successfully finishing enumerating all 129,171,456 solutions within 0.77 sec, using only 187Mb memory. In addition, we iterated the preimage computation procedures for two cycles using the "BDD-bounding" techniques proposed in [7] and proved that this property hold as a EG property. "BDD-bounding" is a technique that uses the preimage set BDD resulted from last iteration as a constraint for ATPG/SAT-solver in the next iteration so that the preimage computation can be extended to multiple cycles.

7 Conclusion and Future Work

We presented a novel success-driven ATPG algorithm to efficiently compute preimages. ATPG and BDD are combined in such a way that the circuit function is manipulated by ATPG engine while the results are encapsulated by a BDD data structure (therefore ATPG is invisible to the outside). The procedures can be iterated to multiple cycles using BDD-bounding. Experimental results showed that the proposed method can achieve 2 to 3 orders of magnitude speed-up over convential ATPG in preimage computation and at the same time it consumes much less memory than pure BDD-based methods. We believe this method has the potential for solving large-scale hardware verification problems. To extend this work, an efficient way other than BDD-bounding for iterating this preimage computation procedure to multiple cycles is needed. We will investigate on this direction and the target is a new unbounded model-checking tool based on ATPG.

Acknowledgment We want to thank Koichiro Takayama at Advanced CAD research, Fujitsu Labs. of American, Sunnyvale, CA, for providing the BDD experimental results.

References

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

[2] M. H. Schulz, E. Trischler and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", *IEEE Trans. CAD*, vol.7, no.1, pp. 126-137, 1988.

[3] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506-521, May, 1999.

[4] L. Zhang, C. F. Madigan, M. H. Moskewicz and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver", *Proc. ICCAD*, 2001, pp. 279-285.

[5] J. M. Burch, E. M. Clarke, D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking", *Proc. DAC*, 1991, pp. 403-407.

[6] N. Narayan, J. Jain, M. Fujita, A. Sangiovanni-Vincentelli, "Partitioned ROBDDs: A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions", *Proc. ICCAD*, 1996, pp. 547-554.

[7] A. Gupta, Z. Yang, P. Ashar and A. Gupta, "SAT-based Image Computation with Application in Reachability Analysis", *Proc. FMCAD*, 2000.

[8] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-Based Decision Heuristics for Image Computation using SAT and BDDs", *Proc. ICCAD*, 2001, pp. 286-292.

[9] K. L. McMillan, "Applying SAT methods in Unbounded Symbolic Model Checking", *Proc. CAV*, 2002.

[10] L. Zhang and S. Malik, "Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver", *Proc. 8th Intl. Conf. on Principles and Practice of Constraint Programming* (CP2002), 2002.

[11] J. P. Marques and K. A. Sakallah, "Dynamic Search-Space Pruning Techniques in Path Sensitization", *Proc. DAC*, 1994.

[12] I. Hamzaoglu and J. H. Patel, "New Techniques for Deterministic Test Pattern Generation", *Proc. VTS*, 1998, pp. 446-452.

[13] J. Giraldi and M. L. Bushnell, "EST: The New Frontier in ATPG", *Proc. DAC*, 1990, pp. 667-672.

[14] P. Chung, I. N. Hajj and J. H. Patel, "Efficient Variable Ordering Heuistics for Shared ROBDD", *Proc. ISCAS*, 1993, pp. 1690-1693.

[15] H. Iwashita and T. Nakata, "Forward Model Checking Techniques Oriented to Buggy Designs", *Proc. ICCAD*, pp.400-404, Nov. 1997.