TELKOMNIKA, Vol. 11, No. 7, July 2013, pp. 4077 ~ 4085
e-ISSN: 2087-278X

Learning Techniques for Automatic Test Pattern Generation using Boolean Satisfiability

Liu Xin
School of Electrical & Electronic Engineering, Hubei University of Technology, Wuhan, China
e-mail: dolce_lx@sina.com

Abstract

Automatic Test Pattern Generation (ATPG) is one of the core problems in testing of digital circuits. ATPG algorithms based on Boolean Satisfiability (SAT) turned out to be very powerful, due to great advances in the performance of satisfiability solvers for propositional logic in the last two decades. SAT-based ATPG clearly outperforms classical approaches especially for hard-to-detect faults. But its inaccessibility of structural information and don't care, there exists the over-specification problem of input patterns. In this paper we present techniques to delve into an additional layer to make use of structural properties of the circuit and value justification relations to a generic SAT algorithm. It joins binary decision graphs (BDD) and SAT techniques to improve the efficiency of ATPG. It makes a study of inexpensive reconvergent fanout analysis of circuit to gather information on the local signal correlation by using BDD learning, then uses the above learned information to restrict and focus the overall search space of SAT-based ATPG. The learning technique is effective and lightweight. Experimental results show the effectiveness of the approach.

Keywords*: test pattern generation, Boolean satisfiability, conjunctive normal form, binary decision diagram, learning technique*

Copyright © 2013 Universitas Ahmad Dahlan. All rights reserved.

1. Introduction

According to Moore's Law, the design sizes have been growing continuously over the last decades. Since it is likely that this trend is going on over the next years, algorithms in the field of Electronic Design Automation (EDA) have to be improved. Tools for Automatic Test Pattern Generation (ATPG) have to cope with this growth as well. Classical algorithms working on a circuit representation reach their limits. The number of faults they cannot classify grows and thus the fault coverage decreases rapidly. This leads to a loss of quality. Therefore, new efficient techniques have to be developed.

Due to improvements made for solving the Boolean Satisfiability (SAT) problem in the last two decade, SAT-based ATPG has been found to be a promising alternative to the classical approaches. In SAT-based ATPG, the problem of finding a test pattern for a particular fault is transformed into a propositional satisfiability problem. The problem is satisfiable if, and only if, the fault is testable. The search for a satisfying assignment is thus the search for a test pattern. SAT-based methods proved to be highly advantageous particularly for hard-to-solve problem instances.

However, SAT-based ATPG algorithms suffer from a major drawback. Most state-of-the-art SAT solvers require problem instances given in Conjunctive Normal Form (CNF). Hence, the ATPG problem needs to be converted into a CNF representation. This generation is a vital issue because most of the structural information is lost during the transformation of the original problem into CNF. Generic ways are to add clauses for any circuit-based problem. Even more problem-specific knowledge can be provided to the SAT solver as guidance. This has been suggested for the SAT-based test pattern generator TEGUS [1]. Improvements on the justification and propagation have been proposed in [2-6].

SOCRATEST [7] was one of the first tools incorporate learning into the standard ATPG flow. Implications on reconvergent circuit structures were statically added during a preprocessing step, but only certain types of implications were detected. Subsequently, the more general concept of recursive learning [8] was applied in the tool HANNIBAL. Recursive learning is complete, in principle, i.e. all implications resulting from a partial assignment of

Received February 10, 2013; Revised April 30, 2013; Accepted May 12, 2013

values to signals in a circuit can be learned. However finding the cause for a certain implication on the circuit structure requires too many backtracking steps in practice. Therefore, this technique is not feasible for large circuits.

In traditional ATPG tools learning typically means an overhead that has to pay off while generating test patterns. In contrast, the SAT solver at the core of a SAT-based ATPG tool uses learning in the core algorithm to traverse the search space more effectively.

A SAT solver learns information from a SAT instance each time a non-solution subspace is found. Conflict clauses serve as an efficient data structure to store this information. Techniques to reuse learned information for similar SAT instances have been proposed [9]. The challenge for reuse lies in the creation of the SAT instance and storing the learned information in a database. Domain specific knowledge is needed to allow for efficient reuse of this information. This specific knowledge is just the circuit structural information.

Classical ATPG has structural information guide its search process, and easy to handle multi-value logic. In contrast, SAT-based ATPG have more elegant and unified model, as well as potentially faster and more precise implication process based on efficient learning techniques. But due to inaccessibility to structural information and don't care, there exists the over-specification problem of input patterns.

In order to overcome these problems, we use an added circuit structure layer on the generic SAT solve engine to maintain circuit-related information and value justification relations. In addition, we explore a new way to improve the efficiency of SAT-based ATPG with the help of "cheap" and affordable BDD learning, which similar to the contribution of "conflict clauses" in leading SAT tools [9].

This paper is structured as follows. Firstly, the basics of SAT algorithms and the SAT-based ATPG are briefly reviewed and an introductory example is given. Next, we detail the approach of learning from BDD-based analysis. In Section 4, we present our ATPG framework made for overcoming the over-specification problem. In Section 5 offers experimental results on the ISCAS benchmark circuits. Finally, conclusions and directions for future research are given.

2. Preliminaries

2.1. Boolean Satisfiability

The input to the SAT solver is a formula in CNF. A CNF formula is a set of clauses; each clause is a set of literals; each literal is either a variable or its negation. The function of a clause is the disjunction of its literals, and the function of a CNF formula is the conjunction of its clauses. An assignment to the set of variables $V \subseteq \{0, 1\}^n$ of formula j is a mapping from V to {true, false}. A satisfying assignment for formula j is one that causes j to evaluate to true.

Many successful SAT solvers are based on the Davis–Putnam (DP) procedure (DPLL), whose modern incarnations are described by the pseudo code of Figure 1.

```
SAT_Solve ( ) {
    while (decide ( ) != done ) {
        while (deduce ( ) == conflict) {
            blevel = analyze_conflict ( );
            if ( blevel == 0 )
                return UNSATISFIABLE;
            else
                backtrack ( blevel );
        }}
    return SATISFIABLE;
}
```

Figure 1. GRASP_DPLL algorithm

The solver maintains a current partial assignment that is extended until it either becomes a total satisfying assignment, or becomes conflicting. While extending the partial assignment, the deduce procedure (also known as BCP: Boolean constraint propagation) tries to detect as many implications as possible by using asserting clauses. A conflict occurs whenever the simultaneous implication of opposite Boolean values on a variable.

Modern SAT solver is based on conflict analysis and conflict driven learning (shown as Figure 1). When the search reaches an assignment causing a conflict in satisfying the original objective, the conflict is analyzed and a conflict clause is produced to prevent the search entering the same non-solution space. In a way, accumulating conflict clauses is a process of pruning the search space. When the entire space is pruned, the problem can be proved unsatisfiable.

2.2. ATPG problem

Test generation based on Boolean satisfiability can be divided into two independent steps: extraction of the CNF formula and identification of a satisfying assignment. Different fault models require different CNF formula extractors, but can use identical satisfiability solvers. The CNF formula extractor for a single stuck-at fault (SSF) is detailed in [1] and [2].

We start by introducing unified representations for the ATPG problems that will be used in what follows. A combinational circuit ***C*** is represented as a directed acyclic graph ***C*** = (***V***, ***E)***, where ***V*** denote the circuit nodes, ***E*** identifies gate input-output connection relations. A single stuck-at fault $y = y(w, v)$ is one that causes a net *w* on ***C*** to be permanently stuck at a logic value $v \in \{0, 1\}$. Faulty circuit denoted by C_f , is then ***C*** operation with the fault $y = y(w, v)$.

Here, notation C_y^{fo} represents the fault shadow subcircuit, the parts of the circuit that are structurally influenced by the fault site are calculated by a depth first search, which consists of the transitive fan-outs of the net *w* on C_f. Notation $C_y^{\sup}$ denotes the fault support subcircuit made up of the transitive fan-ins of the transitive fan-outs of the wire *w* of ***C***. Then ATPG problem can be cast as a satisfiability problem made up of the circuit corresponding to the pairwise xor of the output of C_y^{fo} and $C_y^{\sup}$ (see Fig.2 (a)). This construction is similar to a miter circuit. The set of all satisfiable assignments for ATPG problem precisely gives the set of all input vectors to detect the fault y . It can be formulated as an instance of Boolean Satisfiability in CNF. This formula consists of the three parts, i.e., the formula j_f for C_y^{fo}, and the formula j_g for $C_y^{\sup}$, and the formula j_{xor} for the xor of the output of the former two. The formula is shown as :

$$j_y = j_f \wedge j_g \wedge j_{xor} \tag{1}$$

Taking an example, supposed that there exists the circuit such as Figure 2 (b) with a SSF $y(h, 1)$. Then, the problem instance circuit for this ATPG solution is illustrated in Figure 2(c).

Consider the example circuit in Figure 2(c). The formula (1) expands on as following:

$$\begin{aligned} j_g &= [h = \neg(b+c)] \wedge [i = de] \wedge [j = \neg(ah)] \wedge [k = i + j] \\ &= (\neg b + \neg h) \wedge (\neg c + \neg h) \wedge (b + c + h) \\ &\wedge (d + \neg i) \wedge (e + \neg i) \wedge (\neg d + \neg e + i) \\ &\wedge (a + j) \wedge (h + j) \wedge (\neg a + \neg h + \neg j) \\ &\wedge (\neg i + k) \wedge (\neg j + k) \wedge (i + j + \neg k) \end{aligned} \tag{2}$$

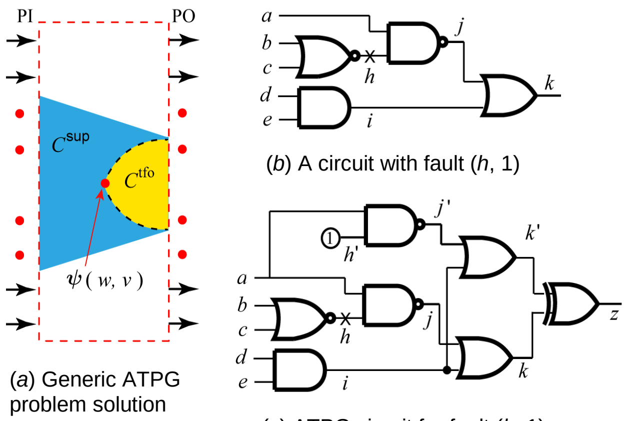
$$\begin{aligned} j_f &= [k' = i + j'] \wedge [j' = \neg(ah')] \\ &= (\neg i + k') \wedge (\neg j' + k') \wedge (i + j' + \neg k) \\ &\wedge (a + j') \wedge (h' + j') \wedge (\neg a + \neg h' + \neg j') \end{aligned} \tag{3}$$

$$\begin{aligned} j_{xor} &= [z = k \oplus k'] \\ &= (k + \neg k' + z) \wedge (\neg k + k' + z) \\ &\wedge (k + k' + \neg z) \wedge (\neg k + \neg k' + \neg z) \end{aligned} \quad (4)$$

In order to speed up the CNF solver, we in practical augment the additional causes that explicitly state structural information of circuit. The first are the activating fault site causes $j_w = (w^{1-v}) \wedge (w_f^v)$, where define $w^1 = w$, $w^0 = \neg w$. The second are the active clauses $j_d = j(G_d \to (G_g \neq G_f))$, where G_g and G_f denote the logic gate on the circuit C_y^{sup} and C_y^{tfo} respectively, and G_d is on the path from the fault site to a primary output. Adding this kind of information can speed up the solver by an order of magnitude because added restrictive clauses avoiding portions of the search space. Therefore, the formula of detecting the fault y in CNF can be written as:

$$j_y = j_f \wedge j_g \wedge j_d \wedge j_w \wedge j_{xor} \quad (5)$$

A satisfying assignment to the formula (5) directly determines the values for the primary inputs to test the fault. Hence, a test vector for detect the fault $y = y(h, 1)$ is {*a*, *b*, *c*, *d*, *e*} = {1, 1, 1, 0, 1}.



(*a*) Generic ATPG problem solution

(*b*) A circuit with fault (*h*, 1)

(*c*) ATPG circuit for fault (*h*, 1)

Figure 2. ATPG problem solution and an example circuit

3. BDD Learning Engine

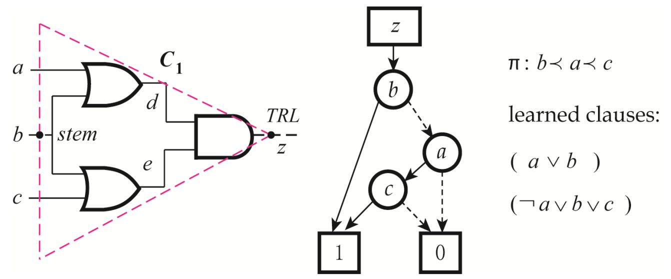
3.1. BDD Learning

A BDD [10] is a directed acyclic graph with two terminal nodes, 1 and 0, representing logical function 1 and 0, respectively. An internal node associated to an input variable has one or more incoming edges and exactly two outgoing edges. A path from root node to terminal node 1 represents a cube for the logic function; that is, BDD is an intelligent way of encoding all the cubes. The graph is levelized and each level is indexed by a support variable. Formal treatments on BDD can be found in [10]. Figure 3 shows an example circuit and its BDD structure.

Our approach focuses on the use of learned clauses generated by a BDD-based analysis of the circuit structure – we call this BDD learning. Essentially, a BDD is used to capture the relationship between Boolean variables of (a part of) the SAT problem, in the form of a characteristic function. In such a BDD, each path to a "0" node denotes a conflict. A learned clause corresponding to this conflict is easily obtained by negating the literals that define the path. Since a BDD captures all paths to 0, i.e. all possible conflicts among its variables, the potential advantage is that multiple learned clauses can be generated and added to the SAT solver at the same time. In contrast, conflict driven learning [9] typically analyzes a single conflict at a time. An example with multiple learned clauses generated from a BDD is shown in Figure 3(b).

3.2. Learning Node Selection

In ATPG or any circuit-based SAT application, the bulk of Boolean constraints arise from the circuit structure. Therefore, it is a natural candidate to exploit the circuit structure graph to create useful BDDs. The main goal for our BDD learning technique is to be effective but lightweight, i.e. it should improve the performance of SAT solver, but without overwhelming the SAT solver heuristic. This rule out the possibility of learning on a global scale, i.e. creating a BDD and learning clauses for every node in the circuit graph, which is to be too expensive. Therefore we perform learning selectively, i.e. by selecting signal nodes of the circuit graph around which to perform learning.



(*a*) TRL *f* and its stem (*b*) BDD for *f* and its learning

Figure 3. An example circuit and its BDD learning

How to select learning nodes? It is well known that ATPG would be quite simple for combinational circuits if it were not for something called reconvergent fanout. A fanout stem is the point at which a circuit node drives more than one other node. When successor nodes of the same signal come together at a subsequent node, such as the node is called the point of reconvergence. Each reconvergent fanout introduces dependencies in the values that can be assigned to nodes. It is these signal value assignments correlation to each other that cause conflict in the ATPG process.

Now that reconvergent fanout is a fundamental cause of the difficulty in testing for circuits. How to compute data correlation? Let us take a look at a Total Reconvergence Line (TRL, also called as dominator) used in the TOPS algorithm [11]. For simplicity of the illustration, we present the concepts using an example circuit shown in Figure 3 (a). As shown in the figure, a TRL is the output f of a subcircuit C_1 such that all paths between any line in C_1 and any *PO* (primary output) go through *z*. In other words, cutting *z* would isolate C_1 from the rest of the circuit. As shown in Figure 3 (a), there are a stem *b* and a TRL *z.* TRLs can be identified in linear time with a single traversal of the network. As we see, it is the transitive fanin cone between the TRL and stem that introduce the signal correlation due to more than one fanout branch. If we could compute the corresponding signal correlation of the transitive fanin cone between TRL and stem in advance, then we can avoid many conflicting value assignments. Therefore, we select TRLs as the learning nodes around which to perform learning.

Once learning node selection is done, we need to create BDDs that capture relationships among variables in the circuit region around the learning node. Here, we created BDDs across very few logic levels of the fanin cone of TRL, typically 5-10, in order to avoid BDD size blowup. Keeping the BDD sizes small has the additional benefit of creating shorter paths in the BDD, thereby resulting in shorter clause lengths. In general, shorter learned clauses are likely to be more beneficial than longer learned clauses, since they require less number of assignments before resulting in an implication. The potential benefit of BDD learning is in its ability to perform non-local learning around the learning nodes.

3.3. Generation of Learned Clauses

After a BDD has been created, we need to generate learned clauses, i.e. dump BDDs as CNF formulas. Given a BDD representing a function *f* in monolithic or conjunctive form, it creates clauses starting from *f* corresponding to the off-set of the function f. Within the BDD for *f*, such clauses are found by following all the paths from the root node of the BDD to the constant node 0. In order to favor shorter clauses, only those cubes that are shorter than a given maximum clause length, typically 5-10, are used for generating learned clauses. Since the maximum clause length varies from 5 to 10, traversal is very fast.

3.4. Adding Learned Clauses to SAT Solver

For the time being, we only perform static learning. This means that the learning nodes are selected using static information of the circuit structure, and learned clauses are added statically, before the SAT solver starts the search. This is relatively straight forward, since all implications due to the learned clauses occur at the starting level. We use a BDD learning engine to encapsulate the essential tasks of learning node selection, creation of BDDs, and generation of learned clauses (see Fig.5). This engine is integrated with a generic SAT solver. BDD learning is especially performed in conjunction with other learning mechanisms in the SAT solver, e.g. conflict-driven learning [5]. Furthermore, a learned clause j_{TRLi} generated by the BDD learning engine is treated similar to other learned clauses by the SAT solver [12]. For example, scores of variables related to these learned clauses are incremented. After integration of BDD learning, the formula for detecting the ault $y = y(w, v)$ is the conjunction of formula (5) and j_{TRLi}, i.e.

$$j_y = j_y \wedge j_{TRLi} \tag{6}$$

4. Layered Approach for Solving the SAT Instances of ATPG Problem

Assigning values to those variables that correspond to the primary inputs is sufficient. Once these assignments are, BCP will imply the values of the internal gates and outputs. However, the SAT solver usually does not have this information in advance. Decisions may be made for internal gates as well. But, when the SAT solver finally returns a satisfying assignment, the values of primary inputs must be consistent with those of internal gates, since a combinational circuit is considered.

Typically, the SAT solver does not check whether further assignments are necessary to satisfy all clauses, but stops after assigning values to all variables without finding a conflict. For ATPG, this means that there are no don't care values contained in the test patterns, but all inputs have a fixed value.

As far as a test pattern for a fault is concerned, it is an input vector that sensitizes the fault under consideration and propagates the fault effects to a primary output or an observable point. A test pattern is found if and only if both the fault and a propagation path to an observable point are sensitized and all unjustified lines are justified. If one of these conditions can’t be satisfied, the fault is proved as undetectable. This definition considerably increases the efficiency of the SAT-based ATPG algorithms. For example, the previous SAT-based ATPG algorithms [1, 2] consider that a test pattern is found when the characteristic CNF formula is satisfied, i.e., all clauses in the formula evaluate to 1. This approach potentially increases the complexity of the ATPG problem. The recent SAT-based ATPG algorithms also check for an empty J-frontier instead of whether all clauses in the CNF formula are satisfied [13].

Similar to the one proposed in [13], we augment a circuit structure layer on the generic SAT solver engine to maintain circuit-related information and value justification relations. Figure 4 represents the layered structure of our test generation algorithms. As shown in Figure 4, the BDDs learning are on the top of the hierarchy schema, which built on circuit structure layer.

Now we can adapt a generic SAT algorithm (shown as Figure 1) to allow proper maintenance of justification information. The required modifications are function deduce () and analyze-conflict (), now they must invoke dedicated procedures for updating node justification information. Additionally, function decide () now tests for satisfiability by checking for an empty J-frontier. Our decision heuristic uses a VSIDS heuristic [8], where unassigned literals with the highest appearance in the formula are weighted more. However, this weighting is exponentially decayed, to ensure that newly learned clauses get more precedence. If all the generated conflict clauses are satisfied, the decision variables are selected by the order of the J-frontier currently trying to satisfy. The reason is for hard problems, as the conflict clauses tend to accumulate fast during search, the decisions will gradually be dominated by the conflict clauses. Therefore, the performance will not degrade much. If no conflict, the condition of test pattern found is J-frontier become empty.

Unlike the structural ATPG, justification and value consistency are formally dissociated in our approach, i.e., the SAT algorithm handles value consistency, and the added layer handles justification.

The justification conditions of a gate are as follows:

If its output is at an output controlling, and if any of the inputs are at a controlling value;

If its output is at an output non-controlling value, and if all inputs are at non-controlling values;

If it is a XOR gate, and all inputs are assigned.

A controlling value ***c*** is an input value that uniquely determines the output value of a gate. If the output of the gate is a negated primitive Boolean function, then its inversion parity *i* = 1. The output non-controlling value of a gate is $\overline{c \oplus i}$, its output controlling value is $c \oplus i$.

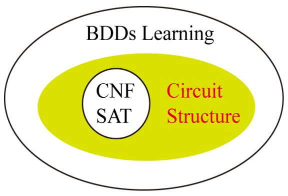


Figure 4. The layered structure of ATPG algorithm

5. Experimental Results

In order to validate the effectiveness of the proposed method, we implemented a prototype ATPG framework (see Figure 5) in the C programming language, which has integrated our BDD learning engine.

For our experiments, we used the circuit from ISCAS'85 benchmark set [14]. This set has 10 combinational circuits in all, which was intended to compare experimental results in the field of testing and testability of circuits.

We use the maximum clause length of 6, which has the better experimental results in comparison with among 5-10, and the learning nodes are all the TRLs selected, but we only create BDDs of those TRLs less than 10 levels. The results are shown in Table 1.

Table 1. Experimental results for ISCAS'85 circuits

circuit	#gate	#ft	T1	#trl	#cl	T2	redu
C432	160	524	0.49	1	30	0.43	12%
C499	202	758	0.33	0	0	0.34	-3%
C880	383	942	0.60	24	159	0.41	31%
C1355	546	1574	1.23	184	616	0.95	23%
C1908	880	1879	2.04	160	1215	1.24	39%
C2670	1193	2747	3.37	193	1447	1.73	49%
C3540	1669	3428	5.75	168	1332	2.16	62%
C5315	2307	5350	4.83	339	2403	2.19	55%
C6288	2416	7744	17.9	0	0	18.1	-1%
C7552	3512	7550	8.73	583	3068	3.98	54%

Column 2 and 3 respectively lists the number of gates and faults of the circuits. Column 4 reports the total time (T1, in seconds) taken for the test generation of all faults without the use of any BDD learning. The next three columns respectively represent the number of selected TRL, learned clauses, and total time (T2, in seconds) taken for the test generation of all faults of circuit with static BDD learning. Both didn’t use random patterns and fault simulation. The column 8 indicates the percentage reduction in time. In comparison to basic ATPG, our results show the total runtime reduction of up to 62% in the case of BDD learning. Also shown in Table 1, reduced runtimes vary with the circuit because ATPG is an NP-complete problem and used method of BDD learning is a heuristic. In the case of C499 and C6288, BDD learning adds extra efforts because both have no any TRL. Due to no random patterns and fault simulation in both cases, our algorithm detects all testable faults and proves all redundant faults to be redundant without aborting any faults. Therefore, it demonstrates that our algorithm is robust and effective.

```
BDD_learning_engine ( ) {
  while (for each stem stem_i in stem_list ) {
    ( $TRL_i$, level ) = find_TRL ( stem_i );
    if ( level < depth_threshold ) {
          $bdd_i$ = create_bdd ($TRL_i$, level );
          $j_{TRLi}$ = generate_learned_clauses( $bdd_i$);
}}}

TPG ( ) {
  while (for each fault $y$ in fault_list ) {
          create CNF formula $j_y$ for $C_y^{tfo}$ and $C_y^{sup}$;
          remove fault $y$ from fault_list;
          if ( stem_i $\in$ stem_list in $C_y^{tfo}$ )
           $j_y = j_y \bigcup j_{TRLi}$;
          workout = SAT_Solve ($j_y$, &test_pattern );
          switch (workout ) {
            case SATISFIED:
               augment $y$ to detected_fault_list;
               augment test_pattern to test_pattern_list;
               break;
            case UNSATISFIED:
               augment $y$ to redundant_fault_list;
               break;
            case ABORTION:
              augment $y$ to aborted_fault_list;
             break;
}}}
```

Figure 5. Pseudo-code for BDD learning & test generation

6. Conclusion

Learning techniques of automatic test pattern generation SAT-based is effective. Its performance is dominantly determined by the practical efficiency of the backend SAT solver. In this paper, we have described details of a lightweight and effective BDD learning technique, which adds learned clauses generated from BDDs to supplement other learning mechanisms and to restrict the overall search space in the underlying SAT-solver engine used in ATPG. We have demonstrated the effectiveness of our techniques on all ISCAS’85 benchmark circuits, which obtained up to 62% reduction in the total runtime.

We believe that the ATPG framework provides many opportunities for combining the relative benefits of SAT and BDDs and specific knowledge of circuit domain. Our work is a step in that direction. The robustness of SAT-based ATPG can be further increased by using the proposed circuit-based dynamic learning technique. A future direction of research work is to develop solving algorithms which are able to use knowledge about the ATPG problem to explore the solution space much faster.

References

[1] Larrabee T. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* 1992; 11(1): 4-15.
[2] Stephan P, Brayton RK, and Sangiovanni-Vincentelli AL. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1996; 5(9): 1167-1176.
[3] Gizdarski E, Fujiwara H. SPIRIT: A highly robust combinational test generation algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2002; 21(12): 1446-1458.
[4] Eggersglüß S, Tille D, Drechsler R. *Speeding up SAT-based ATPG using dynamic clause activation*. IEEE Asian Test Symposium. Taiwan. 2009; 177-182.
[5] Fey G, Warode T, Drechsler R. *Reusing learned information in SAT-based ATPG*. Int'l Conf. on VLSI Design. 2007; 69-76.
[6] Drechsler R, Eggersgluss S, Fey G. On Acceleration of SAT-Based ATPG for Industrial Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2008; 27(7): 1329-1333.
[7] Schulz MH, Trischler E, Sarfert TM. *SOCRATES: A highly efficient automatic test pattern generation system*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1988; 7(1): 126-137.
[8] W Kunz and DK Pradhan. Accelerated dynamic learning for test pattern generation in combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1993; 12(5): 684-694.
[9] Marques-Silva JP, Sakallah KA. GRASP: a search algorithm for Propsitional satisfiability. *IEEE Transactions on Computers.* 1999; 48(5): 506-521.
[10] Bryant RE. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers.* 1986; 35(8): 677-691.
[11] Kirkland T, Mercer MR. *A topological search algorithm for ATPG*. Design Automation Conference. New York. 1987; 502-508.
[12] Moskewicz M, Madigan C, Zhao Y, Zhang L, Malik S. *Chaff: engineering an efficient SAT solver*. Design Automation Conference (DAC'2001). Las Vegas. 2001; 530-535.
[13] Silva LG, Silveria LM, Silva JM. *Algorithms for solving boolean satisfiability in combinational circuits.* Proceedings of Design, Automation and Test in Europe Conference and Exhibition 1999. Munich. 1999; 209-214.
[14] Brglez, Fujiwara H. *A neutral netlist of 10 combinatorial benchmark circuits and a target translator in fortran*. 1985 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Kyoto. 1985; 895-698.