Incremental SAT Instance Generation for SAT-based ATPG

Daniel Tille Rolf Drechsler
Institute of Computer Science, University of Bremen
28359 Bremen, Germany
{tille,drechsle}@informatik.uni-bremen.de

***Abstract*— Due to ever increasing design sizes more efficient tools for *Automatic Test Pattern Generation* (ATPG) are needed. Recently ATPG based on Boolean satisfiability (SAT) has been shown to be a beneficial complement to traditional ATPG techniques.**

This paper makes two contributions. Firstly, we analyze the two steps SAT-based ATPG consists of with respect to their run time on industrial benchmarks. Secondly, exploiting these analysis results, we propose an incremental solving technique with the objective to speed up the entire classification process. An experimental evaluation of the proposed method shows a significant reduction of the overall run time of the SAT-based ATPG process.

I. Introduction

The complexity of industrial circuits increases rapidly. The number of gates doubles every 18 months. As a result the size of problem instances that have to be handled by *Computer Aided Design* (CAD) tools also increases. The post production test is one particular step in the design flow. It ensures the functional correctness of a circuit. To guarantee high quality production, this step is very important. In practice this test is carried out by applying input vectors – so called *test patterns* – to the inputs and controlling the output response with respect to its correctness. The test patterns are computed during *Automatic Test Pattern Generation* (ATPG). ATPG tools also have to cope with the increasing size of problem instances.

In practice, usually a fault model is used to abstract from the physical defects. To test the circuit for correctness with respect to the fault model used, test patterns have to be computed. If there exists a test pattern for a particular fault F then F is called *testable*; otherwise F is called *untestable*.

In this work, the *Stuck-At Fault Model* (SAFM) is used. To generate a test pattern for a *Stuck-at Fault* (SAF), there exist many sophisticated algorithms. The D-algorithm [13] was the first algorithm that traversed the search space by backtracking. Improvements concerning decision strategies and propagation/justification were given in PODEM [7] and FAN [6]. Further algorithms are Socrates [14] and Hannibal [9]. All these algorithms have in common that they directly work on the circuit structure.

In contrast there also exist approaches based on Boolean satisfiability (SAT) [10], [15], [16]. These algorithms work on a representation of the problem instance in *Conjunctive Normal Form* (CNF). Since no efficient algorithms to solve large SAT instances were available in the past, these techniques were not applicable to large circuits. In the last decade, however, many improvements have been made for solving SAT [11], [12], [8], [5] and, so, the usage of SAT solving for ATPG became applicable. Because classical ATPG algorithms reach their limit, SAT-based ATPG is getting more important. Nowadays, SAT-based ATPG is a promising alternative to the classical algorithms (see e.g. [18], [4]).

In this paper we give a detailed run time analysis of state-of-the-art SAT-based ATPG algorithms. For large industrial circuits it is shown that often the time for the construction of the SAT instance dominates the overall run time. Based on this "surprising result" (since the generation of the instance runs in linear time, while SAT solving is NP-complete), we propose a technique to generate a partial CNF. If this SAT instance is satisfiable, a test pattern is derived. Otherwise, the SAT instance is enlarged and a new CNF is generated. For the next run, information from previous SAT computations can be reused. Experimental results on large industrial circuits with more than 3 million gates demonstrate the efficiency of the approach. By using the proposed *incremental* technique speed-ups of up to a

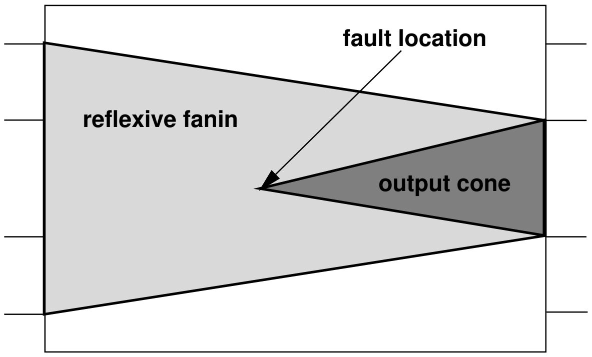

Fig. 1. Extraction of the influenced circuit parts

factor of eight can be observed.

This work is structured as follows: In the next section a short overview on SAT-based ATPG is given. In Section III the motivation is presented in more detail by an analysis on industrial circuits. The incremental solving approach is discussed in Section IV. Experimental results and conclusions are given in Section V and Section VI, respectively.

II. Previous Work

To make the paper self-contained this section presents a short overview on SAT-based ATPG. First a general explanation is given. Afterwards the generation of a CNF for a fault is illustrated.

A. SAT-based ATPG

To create a test pattern for a *Stuck-At Fault* (SAF), an assignment to the inputs has to be found that guarantees at least one different output value between the faulty circuit and the faultless circuit. While classical algorithms work directly on the circuit structure to find such an assignment, in SAT-based ATPG the question whether there exists a test pattern for a particular fault F is encoded into a Boolean formula which is satisfiable if and only if F is testable. Then, a SAT solver proves either satisfiability or unsatisfiability of the formula. A test pattern – if it exists – can be derived directly from the satisfying assignments.

Modern SAT solvers work on instances represented as CNFs. A CNF is a conjunction of clauses, a clause is a disjunction of literals, and a literal is the positive or negative occurrence of a Boolean variable. A SAT instance is satisfied if all clauses are satisfied; a clause is satisfied if at least one of its literals is satisfied; a positive and a negative literal is satisfied if the respective variable is assigned positively and negatively, respectively.

B. Circuit to CNF Conversion

In this section further insight on the generation of a SAT instance is given.

Consider the circuit in Figure 1. After the fault location is marked, the fault site's output cone is traversed by a depth first search. This determines all *Primary Outputs* (POs) that may be influenced by the fault, i.e. all POs where a difference between the faulty circuit and the faultless circuit could be observed. The transitive fan-in of these POs influences the detection of the fault and must be marked, too. To generate the SAT instance for the given fault, only this part of the circuit has to be considered.

As introduced in [15], two Boolean variables G_g and G_f are assigned to each gate to represent the faultless circuit and the faulty circuit, respectively. Both circuits are generated by building the characteristic function for each gate. To find a difference between both circuits, additionally a Boolean variable G_D is assigned to each gate. If the variable G_D is true, the gate's value in both circuits differ, i.e. the constraint $G_D = 1 \rightarrow G_g \neq G_f$ is added to the CNF.

To compute a test pattern for a fault, a path has to be found from the fault site to an output where each variable G_D is true. Following the notation in [15], this path is called a *D-chain*. Therefore, if a gate is on a D-chain, one successor must be on a D-chain as well. This constraint is also added to the SAT instance and, hence, the SAT instance is satisfiable if and only if a D-chain exists, i.e. the SAT instance is satisfiable if and only if the fault is testable.

III. Run Time Analysis

As mentioned in the previous section, SAT-based ATPG consists of two steps: building a SAT instance and solving it. In this section we give a detailed analysis of both steps.

For our analysis SAT-based ATPG was applied to several industrial circuits provided by NXP Semiconductors Germany GmbH. Figure 2 gives an overview on the needed run times for each SAT instance, i.e. each entry denotes one run for a dedicated fault. In the diagram separate run times (in CPU milliseconds) for generating the SAT instance and solving the SAT instance are given on the abscissa and on the ordinate, respectively. Moreover the entries are distinguished between their classification result, where '+' denotes a testable fault and '×' denotes an untestable fault.

Two general observations can be made:

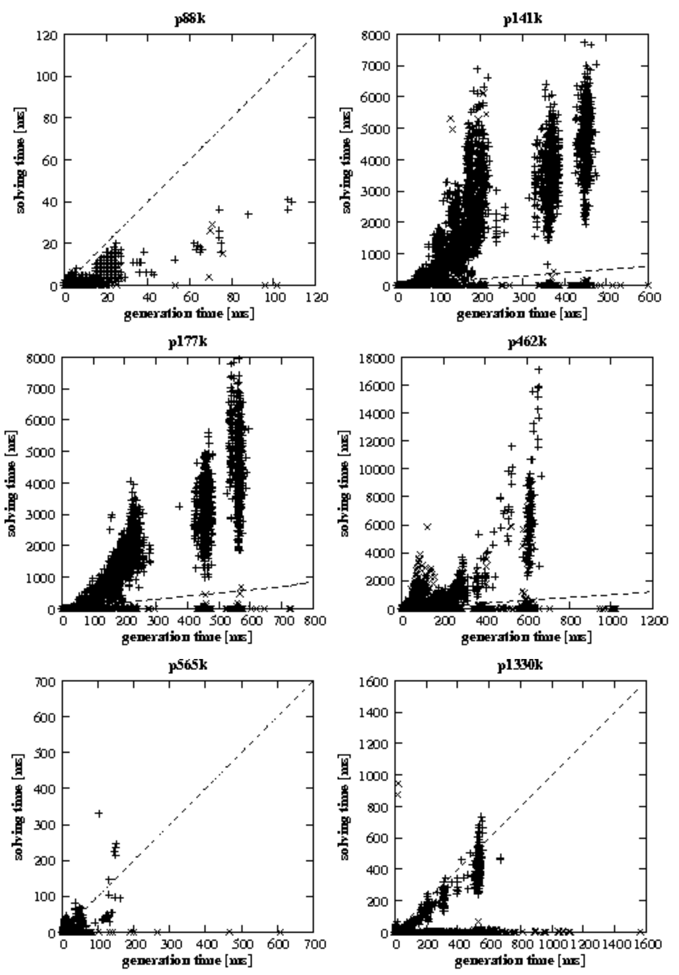

Fig. 2. Run time comparison for individual target

1) On many instances the generation time exceeds the solving time.
2) The solving time of testable instances exceeds the solving time of untestable instances significantly.

These "surprising" observations are discussed in the following.

A. Observation 1

From the theoretical point of view, the instance generation is only an algorithm on a *Directed Acyclic Graph* (DAG), i.e. its run time is linear with respect to the number of gates. Solving the SAT instance, however, is NP-complete [1]. Therefore it can be expected that the run time for solving an instance is significantly larger than generating it.

The observations made in Figure 2 can be explained as follows: Since the handled circuits are very large, the DAG algorithms are very expensive (e.g. with respect to main memory access[1]). Hence the instance generation is very costly.

Solving the SAT instance, however, is often "easy" because of its regular structure, i.e. there are many implications possible that accelerate the search (see also [11], [12], [8], [5]). Additionally, most CNFs are quite small, since the considered part of the circuit (cf. Figure 1) is also quite small. Moreover the data structures used in state-of-the-art SAT solvers are very efficient. For instance, they are tuned to reduce main memory access, so that they are able to handle even huge instances.

B. Observation 2

To compute a test pattern, it is sufficient to find only **one** D-chain. To prove untestability, however, it has to be shown that no D-chain exists at all, i.e. there is no path from the fault site to any output where a difference can be observed. It could be expected, that finding a test pattern is much easier than proving untestability.

When a SAT instance gets satisfied – i.e. a path from the fault location to an output is found that shows a difference between the faulty circuit and the faultless circuit – a SAT solver could stop solving. Each unassigned variable should become a don't care. Modern SAT solvers like MiniSat [5], however, prove satisfiability another way: Instead of checking every clause for satisfiability after each assignment, these solvers only check for conflicts. If each variable is assigned and no conflict occurred, the instance is satisfiable. Hence, after having found a D-chain, i.e. the fault is testable, each variable of the entire influenced circuit part has to be assigned without conflicts. Often, this is a very time consuming step.

If the fault, on the other hand, is untestable, often the conflict occurs quite fast. Due to the D-variables used to encode the difference between the good circuit and the faulty circuit, conflicts during propagation and justification of the fault effect occur early and often close to the fault site.

IV. Incremental Instance Generation

In this section, based on the observations made above, we propose an incremental solving technique to accelerate both steps instance generation and solving.

[1]These main memory accesses are one reason why classical ATPG algorithms – that work directly on the circuit structure – reach their limits.

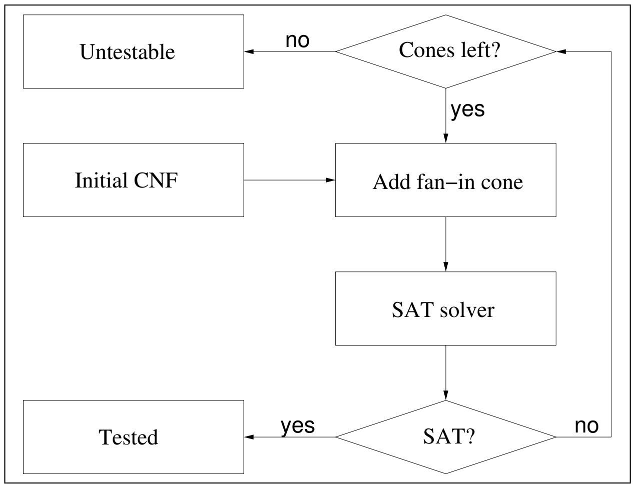

Fig. 3. Sketch of the proposed incremental solving technique

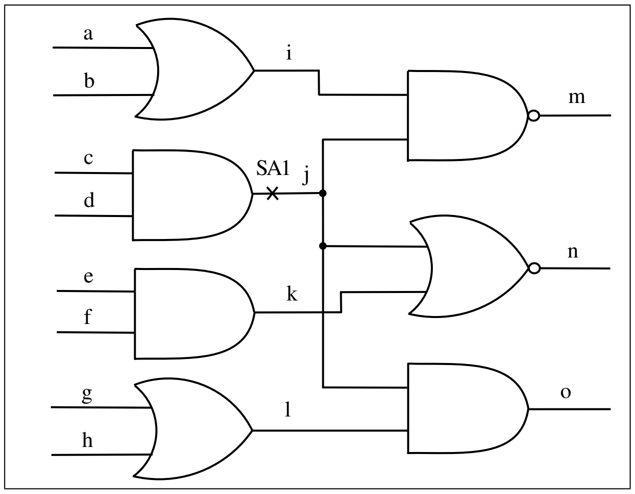

Fig. 4. Example circuit

A. Overview

Based on Observation 1, to speed up SAT-based ATPG it is insufficient to improve only the solving process. Therefore we propose a method that accelerates instance generation as well.

In our proposed methodology only a small portion of the circuit is converted into a CNF, i.e. only a partial SAT instance is generated. Therefore the instance generation time is reduced. Since generally satisfiability can be proven much faster on a smaller CNF (as be shown e.g. in [17]) the run time to classify testable faults can be reduced as well.

The proposed method has one major drawback: It is necessary to build the entire SAT instance to prove untestability. Therefore it cannot be expected to accelerate the classification of untestable faults. Since each incremental step creates some overhead (in form of additional variables and clauses) it is even possible to slow down the classification process if the number of incremental steps is too large. However in a typical industrial circuit the number of testable faults exceeds the number of untestable faults significantly. Therefore it is likely that the improvements thanks to incremental solving outweigh this drawback. This will be shown by our experiments later.

Solving a problem incrementally is a known approach in verification. For instance in [3] an incremental solving technique for SAT-based equivalence checking is given. Similar to our method the fan-in cones of the POs are taken into account incrementally. However the incremental approach is limited to the solving step; the SAT instance is always generated completely.

Figure 3 illustrates our methodology. The *initial CNF* only consists of clauses modeling the injection of the fault. Afterwards the first PO's fan-in cone is added and the resulting SAT instance is solved by the SAT solver. If it is satisfiable, the fault is tested. Otherwise no classification can be given since the fault could be observed on some other output. Therefore the initial CNF is augmented by the second PO's fan-in cone. This process is repeated until no PO is left. Then the fault is untestable.

Example 1: Figure 4 shows a circuit with three primary outputs (m, n, and o). A stuck-at fault is modeled on signal j. Building the complete SAT instance as introduced in Section II-B would result in a CNF for the entire circuit. This CNF consists of 23 variables (15 variables for the correct circuit, 4 variables for the faulty circuit, and 4 D-chain variables).

Using our approach to build the circuit, however, results in a much smaller CNF. The transitive fan-in of output m consists of gates i and j and inputs a, b, c, and d. The resulting CNF consists of only 11 variables (7 variables for the correct circuit, 2 variables for the faulty circuit and 2 D-chain variables).

Since the fault is observable at output m, the SAT instance is satisfiable. Therefore this smaller CNF with only half of the variables needed to build up the SAT instance for the entire circuit is sufficient to classify the fault.

B. Implementation Details

The approach discussed so far leaves the question open in which order to chose the outputs. Furthermore, to reduce the number of incremental steps, an option would be to not consider one after the other, but to build groups of outputs that are considered in parallel.

In the current implementation the POs are ordered with respect to their distance to the fault location, i.e. short paths are preferred. Moreover we use at most five incremental steps: The initial instance is always built up with only one fan-in cone. Each time a SAT instance has to be extended one forth of the remaining outputs are added.

For choosing the "most promising" POs early also testability measures – as used in classical ATPG approaches [6] – could be applied. However this has not been studied in further detail here, but can be considered as future work.

V. Experimental Results

In this section experimental results are given. The approach described in Section IV was implemented prototypically into the ATPG tool of NXP Semiconductors Germany GmbH. MiniSat [5] was used to solve the SAT instances. All experiments were carried out on an Intel Xeon System (3.4 GHz, 32 GByte, Linux).

In Figure 5 the run time analysis made in Section III is repeated using the proposed method. It can be seen that most of the testable faults (denoted by '+') can be classified with a significant speed-up on both instance generation and solving. As predicted in Section IV the proposed method has only small influence on untestable faults (denoted by '×').

Table I gives an overview on the overall run times using traditional SAT-based ATPG as introduced in [4] and the proposed method in column 'SAT' and column 'Inc. SAT', respectively. The circuit's name is shown in column 'Circuit'. We considered two benchmark sets: the publicly available ITC 99 benchmarks [2] and industrial circuits provided by NXP Semiconductors Germany GmbH, Hamburg, Germany. The names of the NXP benchmarks indicate the number of gates contained in a circuit, e.g. the circuit p3852k consists of approximately 3.85 million gates. In column 'Targets' and 'Untest.' the number of targets and the number of untestable targets are given, respectively. For each run the run time (columns 'Time') and the number of aborts (columns 'Ab.') is shown. An abort occurs after 7 MiniSat restarts.

It can be seen that using the proposed method results in a significant speed-up of the entire classification process of up to a factor of eight (circuit p141k). The number of aborted fault can be reduced as well.

In Table II the average CNF sizes, i.e. the number of variables (column 'Variables') and the number of clauses (column 'Clauses'), using traditional SAT-based ATPG and using the proposed incremental approach are given. In case of the proposed method the given numbers refer to

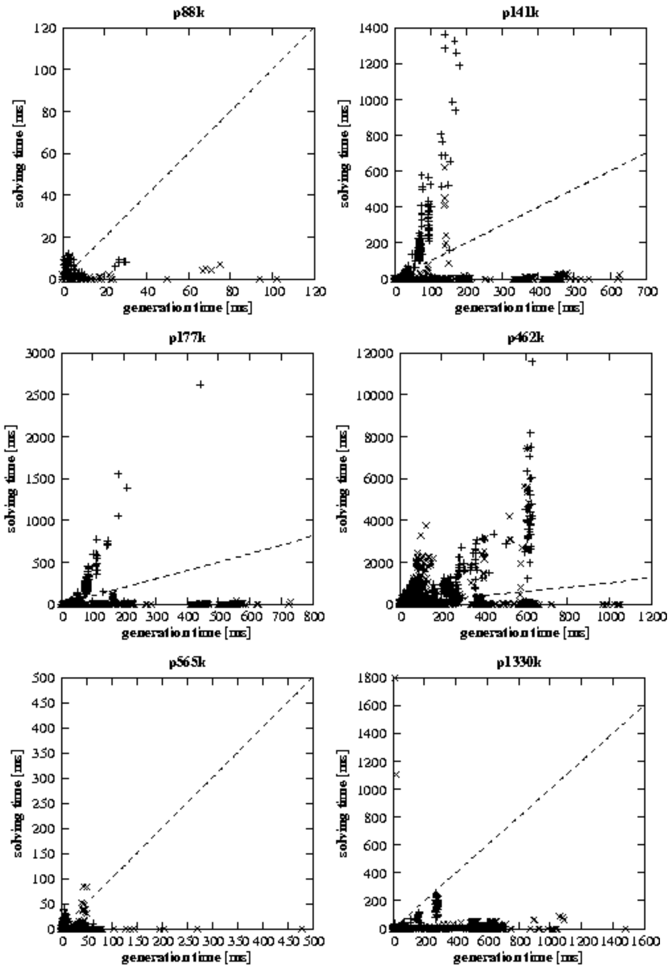

Fig. 5. Run time comparison for individual targets based on the incremental approach

TABLE I

Run times for the ATPG process

			SAT		Inc. SAT	
Circuit	Targets	Untest.	Ab.	Time	Ab.	Time
b17	76493	1958	0	2:51m	0	1:29m
b18	264043	2844	0	9:07m	0	4:12m
b20	45461	319	0	2:18m	0	0:46m
b21	46156	378	0	2:22m	0	0:49m
b22	67540	344	0	2:59m	0	0:57m
p44k	64105	2385	0	49:11m	0	15:18m
p77k	163310	9181	0	0:18m	0	0:12m
p80k	197834	124	0	6:30m	0	1:01m
p88k	147742	2640	0	2:19m	0	1:15m
p99k	162019	2141	2	1:35m	1	1:00m
p141k	267948	13815	1	3:02h	0	22:17m
p177k	268176	13840	0	2:35h	0	24:32m
p456k	740660	35396	194	39:03m	182	31:33m
p462k	673949	132249	11	1:09h	9	42:38m
p565k	1026851	28287	0	6:35m	0	5:42m
p1330k	1516144	44299	0	1:02h	0	54:22m
p2787k	2395388	651868	1628	14:55h	1433	12:37h
p3327k	4557842	109622	1833	48:38h	838	18:38h
p3852k	5507779	164988	1484	17:32h	604	8:25h

TABLE II

Average CNF sizes

	SAT		Inc. SAT	
Circuit	Variables	Clauses	Variables	Clauses
b17	6,424	16,693	3,613	9,046
b18	6,134	15,667	3,262	7,918
b20	7,383	19,433	2,854	7,028
b21	7,452	19,627	2,906	7,160
b22	7,420	19,533	2,667	6,511
p44k	29,819	72,767	21,011	49,269
p77k	544	1,374	378	934
p80k	4,312	9,930	1,369	2,848
p88k	2,366	5,570	1,244	2,744
p99k	2,589	5,955	1,367	2,992
p141k	33,521	95,672	18,782	53,249
p177k	37,775	109,659	21,386	61,807
p456k	6,727	18,611	5,772	16,257
p462k	4,365	12,530	3,790	10,779
p565k	1,681	4,316	1,326	3,445
p1330k	16,704	52,338	15,510	48,871
p2787k	16,911	56,483	16,679	56,609
p3327k	34,377	75,002	27,929	59,981
p3852k	20,622	47,205	14,557	33,253

the SAT instance size after the fault has been classified. In both approaches only the clauses added during the circuit to CNF conversion (cf. Section II-B) are given, i.e. no conflict clauses are considered.

It can be seen that using the proposed method results in smaller average CNF sizes than using the traditional approach. In one case (circuit p2787k) the number of clauses increases slightly. This can be explained with the unusual high number of untestable faults. On all other benchmarks significant reductions can be observed.

VI. Conclusion and Future Work

The contribution of this paper is a detailed analysis of state-of-the-art SAT-based ATPG algorithms with respect to their run time. It was shown that, firstly, instance generation often needs more run time than solving the instance and, secondly, it is often more complex to prove testability than proving untestability. Based on these observations we proposed an incremental SAT instance generation technique. The experimental results confirm that the overall run time of the ATPG computation can be significantly reduced using the new technique.

It is focus of future work to develop more sophisticated heuristics to determine an order in which to add the POs' fan-in cones incrementally.

Acknowledgements

This work was funded in part by DFG grant DR 287/15-1.

References

[1] S. Cook. The complexity of theorem proving procedures. In *3. ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[2] F. Corno, M. Reorda, and G. Squillero. RT-level ITC 99 benchmarks and first ATPG results. In *IEEE Design & Test of Comp.*, pages 44–53, 2000.

[3] S. Disch and C. Scholl. Combinational equivalence checking using incremental SAT solving, output ordering, and resets. In *ASP Design Automation Conf.*, pages 938–943, 2007.

[4] R. Drechsler, S. Eggersglüß, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille. On acceleration of SAT-based ATPG for industrial designs. In *IEEE Trans. on CAD*, 2008. to appear.

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

[6] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. on Comp.*, 32:1137–1144, 1983.

[7] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic. *IEEE Trans. on Comp.*, 30:215–222, 1981.

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

[9] W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *Int'l Conf. on CAD*, pages 538–543, 1993.

[10] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. on CAD*, 11:4–15, 1992.

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

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

[13] J. Roth. Diagnosis of automata failures: A calculus and a method. *IBM J. Res. Dev.*, 10:278–281, 1966.

[14] M. H. Schulz, E. Trischler, and T. M. Sarfert. SOCRATES: A highly efficient automatic test pattern generation system. *IEEE Trans. on CAD*, 7(1):126–137, 1988.

[15] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD*, 15:1167–1176, 1996.

[16] P. Tafertshofer, A. Ganz, and K. Antreich. Igraine - an implication graph based engine for fast implication, justification, and propagation. *IEEE Trans. on CAD*, 19(8):907–927, 2000.

[17] D. Tille, G. Fey, and R. Drechsler. Instance generation for SAT-based ATPG. In *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 153–156, 2007.

[18] K. Yang, K.-T. Cheng, and L.-C. Wang. Trangen: a SAT-based ATPG for path-oriented transition faults. In *ASP Design Automation Conf.*, pages 92–97, 2004.