A Fast Untestability Proof 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*—Automatic Test Pattern Generation (ATPG) based on *Boolean satisfiability* (SAT) has been shown to be a beneficial complement to traditional ATPG techniques. SAT solvers work on instances given in *Conjunctive Normal Form* (CNF). The required transformation of the ATPG problem into CNF is one main part of SAT-based ATPG and needs a significant portion of the overall run time. Solving the SAT instance is the other main part. Here, the time needed is often negligible – especially for easy-to-classify untestable faults.**

This paper presents a preprocessing technique that speeds up the classification of untestable faults by accelerating the SAT instance generation. This increases the robustness of the entire ATPG process. The efficiency of the proposed method is shown by experiments on large industrial designs.

I. Introduction

The continuous growth of today's circuit designs requires a constant improvement of state-of-the-art *Electronic Design Automation* (EDA) tools. The post-production test is a vital 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, a fault model is usually used to abstract from the physical defects. To test the circuit for correctness with respect to the fault model applied, 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 is used. To generate a test pattern for a stuck-at fault, 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 [6] and FAN [5]. Further algorithms are Socrates [14] and Hannibal [8]. 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) [9], [16], [17], [3]. SAT-based methods proved to be highly advantageous in particular for hard-to-solve problem instances. Nowadays, SAT-based ATPG is a promising complement to the classical algorithms.

Since most modern SAT solvers (e.g. [11], [12], [7], [4]) work on an instance representation in *Conjunctive Normal Form* (CNF), a new SAT instance has to be generated for each fault[1]. In [18], [10], it was shown that the run time needed for instance generation is a significant part of the overall run time and often even dominates it. Especially CNFs of untestable faults can mostly be solved very easily, because in industrial circuits the reason for the conflict is often bounded locally. In those cases, building the entire SAT instance is a large overhead.

With the growth of the industrial designs, the number of untestable faults increases considerably. Today's circuits contain hundreds of thousands of untestable faults. On the one hand, the relative number of untestable faults with respect to the total number of faults, i.e. the fault coverage, have kept unchanged over the last years. On the other hand, all untestable fault (after fault collapsing) have to be addressed explicitly where testable faults could be classified by using a fault simulator. Therefore it may happen that more untestable faults than testable faults are addressed during ATPG although only a small number of all faults are untestable.

As a result, avoiding the above mentioned overhead for untestable faults can improve the robustness of the overall SAT-based ATPG process.

This paper presents a preprocessing method with the objective to accelerate the SAT instance generation by only building partial CNFs. During a detailed motivation it is shown that the technique is only useful for easy-to-classify untestable faults. Those faults occur frequently in industrial designs where restrictions to the primary inputs have to be applied.

However, due to the incremental manner of the technique, using the preprocess does not slow down the ATPG process for other circuits. As a result, the overall robustness can be increased.

This work is structured as follows: in the next section a brief overview on SAT-based ATPG is given. In Section III the motivation is presented in more detail. A preprocessing technique 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 specific fault is illustrated. Finally, a run time analysis provides further insight.

[1] In preliminary studies, we also experimented with a circuit SAT solver, but did not consistently observe improvements in run time or memory use for ATPG.

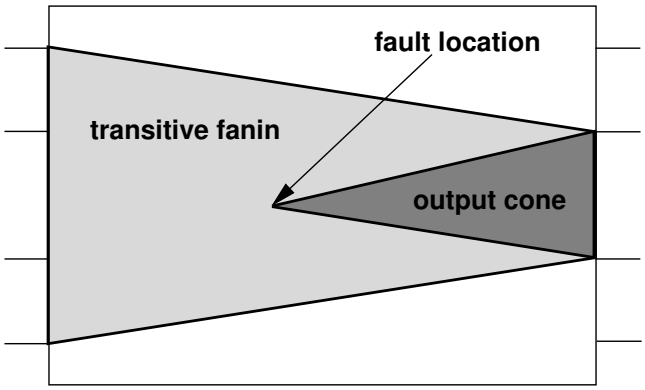


Fig. 1. Extraction of the influenced circuit parts

A. SAT-based ATPG

To create a test pattern for a stuck-at fault, 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. This formula 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 assignment.

Modern SAT solvers work on instances represented in *Conjunctive Normal Form* (CNF). 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 or a negative literal is satisfied if the respective variable is assigned positively or negatively, respectively.

How to transform an ATPG problem into a SAT instance is explained in the following.

B. Circuit-to-CNF Conversion

Consider the schematically depicted circuit in Figure 1. Here, a brief overview on the circuit-to-CNF conversion is given.

After the fault location has been 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 fanin of these POs influences the detection of the fault and must be marked, too. To generate the SAT instance for the given fault, this part of the circuit has to be considered.

As introduced in [16], two Boolean variables g_c and g_f are assigned to each gate g in order to represent the gate's value in the correct circuit and in the faulty circuit, respectively. A gate's CNF is generated by building its characteristic function [19]. The conjunction of all CNFs results in the CNF for the circuit.

To find a difference between the correct circuit and the faulty circuit, an additional Boolean variable g_d is assigned to each gate. If the variable g_d is true, the gate's values in both circuits differ. Therefore, the constraint

$$g_d = 1 \rightarrow g_c \neq g_f$$

is added to the CNF in form of the two clauses

$$(\overline{g}_d + g_c + g_f) \cdot (\overline{g}_d + \overline{g}_c + \overline{g}_f).$$

To compute a test pattern for a fault, there must be a path from the fault site to an output, where the assignment of each variable g_d is true. Following the notation in [16], 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 property – encoded by the constraint

$$g_d \rightarrow \bigvee_{i=1}^{n} h_d^i,$$

where the gates $h^1, \ldots, h^n$ denote the successors of gate g – is also added to the CNF. Moreover, the variable g_d^f, where the gate g^f represents the faulty gate, is set to true in order to inject a difference at the fault site.

As a result, the SAT instance generated this way 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.

Finally, as described earlier, this CNF is given to a SAT solver. After the classification, it is completely discarded. Therefore, the circuit-to-CNF conversion has to be done for each single target fault.

C. Run Time Analysis

To judge the effort of the repeatedly performed instance generation step with respect to the overall algorithm, we made a detailed run time analysis [18]. State-of-the-art SAT-based ATPG algorithms were applied to a set of industrial circuits.

Figure 2 gives a brief overview on this analysis on four circuits. The two basic steps – SAT instance generation and solving – are compared with respect to their run time. Moreover, the classification result was included into this comparison. It is distinguished between testable faults (denoted by '$+$') and untestable faults (denoted by '$\times$'). Moreover, the dashed lines indicate where both run times are equal. Thus it can easily be seen which part requires more run times.

It can be observed that:

- surprisingly, the generation time exceeds often the solving time.
- the solving time of testable instances exceeds mostly the solving time of untestable instances significantly.

Furthermore, it can be seen that there are two classes of untestable faults: *easy-to-classify* and *hard-to-classify* untestable faults.

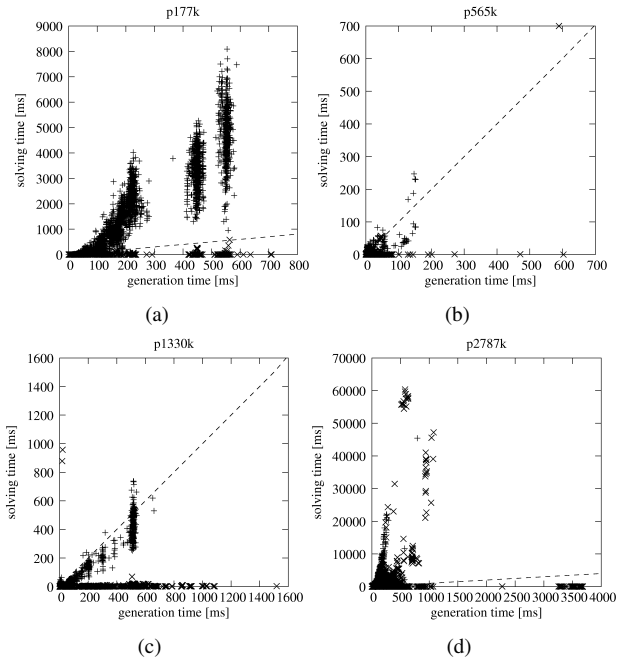


Fig. 2. Run time comparison for individual target

Easy-to-classify untestable faults can often be classified very quickly. They have their origin often in local redundancy or in restrictions to the primary inputs. Examples are shown in all four diagrams in Figure 2: the untestable faults placed exactly on the x-axis. It can be seen that once a SAT instance is built up, the SAT solver proofs unsatisfiability immediately. The instance generation is the bottleneck for those faults.

Hard-to-classify faults are contrary. Although the time needed to generate the instance is often quite small, the solving process is time-consuming. Examples can be seen on or near the y-axis in Figure 2(c) and in Figure 2(d).

For more details about this run time analysis we refer to [18].

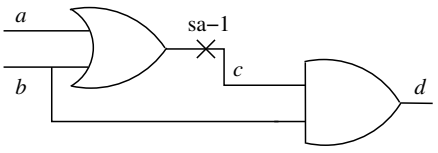
III. Motivation

This section gives the motivation for a preprocess, that is able to accelerate the SAT-based ATPG for untestable faults.

Following the observations in Section II-C, untestability is often proven in almost no time, i.e. unsatisfiability of the respective SAT instance is shown in a few propagation steps only. In this case, the conflict occurs due to a contradiction in the circuit. Figure 3 gives an example.

In the circuit C, depicted in Figure 3(a), a stuck-at 1 fault is modeled on signal line c. Obviously, the fault is untestable, since it is impossible to inject a difference (signal b has to be set to 0) and to propagate it (signal b has to be set to 1) at the same time.

Figure 3(b) shows the SAT instance ϕ_C, describing this particular fault. The notation follows Section II-B. The correct and the faulty circuit are modeled by clauses $\omega_1, \ldots, \omega_6$ and $\omega_7, \ldots, \omega_9$, respectively. The D-chain constraints are given by clauses $\omega_{10}, \ldots, \omega_{14}$. Clauses ω_{15} and ω_{16} contain exactly one



(a) Circuit C

$$
\begin{aligned}
\omega_1 &: \quad (\overline{c}_c + a_c + b_c) \\
\omega_2 &: \quad (c_c + \overline{a}_c) \\
\omega_3 &: \quad (c_c + \overline{b}_c) \\
\omega_4 &: \quad (d_c + \overline{b}_c + \overline{c}_c) \\
\omega_5 &: \quad (\overline{d}_c + b_c) \\
\omega_6 &: \quad (\overline{d}_c + c_c) \\
\omega_7 &: \quad (d_f + \overline{b}_c + \overline{c}_f) \\
\omega_8 &: \quad (\overline{d}_f + b_c) \\
\omega_9 &: \quad (\overline{d}_f + c_f) \\
\omega_{10} &: \quad (\overline{d}_d + d_c + d_f) \\
\omega_{11} &: \quad (\overline{d}_d + \overline{d}_c + \overline{d}_f) \\
\omega_{12} &: \quad (\overline{c}_d + c_c + c_f) \\
\omega_{13} &: \quad (\overline{c}_d + \overline{c}_c + \overline{c}_f) \\
\omega_{14} &: \quad (\overline{c}_d + d_d) \\
\omega_{15} &: \quad (c_f) \\
\omega_{16} &: \quad (c_d)
\end{aligned}
$$

(b) CNF ϕ_C

Fig. 3. Example for an untestable fault

literal. Such clauses are called *unit clauses* and they are used to set a variable to a certain value. In this example, they represent the injection of the fault and the injection of the difference at the fault site, respectively.

Due to the unit clauses ω_{15} and ω_{16}, a SAT solving algorithm can propagate within the CNF[2]. This propagation will lead to the two clauses $\omega'_3 = (\overline{b}_c)$ and $\omega'_5 = (b_c)$, i.e. the variable b_c has to be assigned positively as well as negatively. This is a conflict that makes the CNF unsatisfiable. Therefore, analog to the circuit-based algorithm, the fault is proven to be untestable by a directly implied contradiction.

In both classical ATPG and SAT-based ATPG, this contradiction is bounded locally. Even if the circuit C is a subcircuit of a large design, denoted by C', the conflict occurs immediately. Let $\phi_{C'}$ be the CNF that describes C'. Since $\phi_{C'}$ contains ϕ_C as *unsatisfiable core*[3], the SAT instance is proven to be unsatisfiable just by a few propagation steps.

Those unsatisfiable cores can be found frequently in SAT-based ATPG for industrial designs. With the growing design sizes, this number even increases. However, the reason for untestability may not occur as directly as shown in the example, but rather due to restrictions to the primary inputs. The fast classification can be made thus easily anyway.

In both cases, a SAT instance describing an untestable fault

[2]For details on how to effectively solve a SAT problem see e.g. [2].

[3]A CNF χ is called *unsatisfiable core* of an unsatisfiable CNF χ' if χ is a sub-CNF of χ' that is already unsatisfiable.

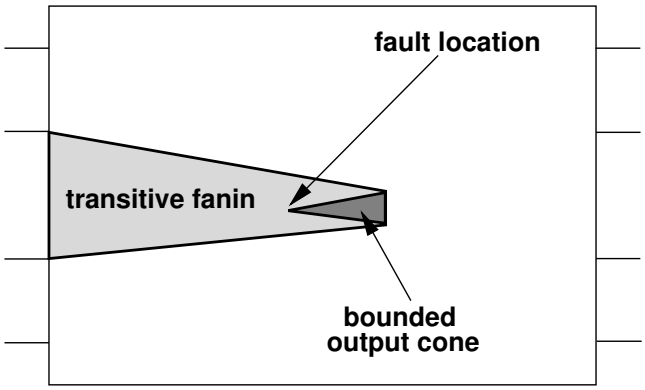


Fig. 4. Extraction of circuit parts in a preprocessing step

is generated completely, although only an instance describing an unsatisfiable core, i.e. describing the subcircuit, would be sufficient to classify the fault. As mentioned in Section II-C, the run time for generating an instance is a significant part of the overall run time. Building a complete instance where a partial one is sufficient is, therefore, an avoidable overhead.

A technique to overcome this drawback, is proposed in the next section.

IV. Preprocessing Method

As mentioned in the previous section, untestable faults in industrial designs are often easy to classify since the reason for the untestability is bounded locally. During preliminary studies it turned out that most untestable faults (about 90%, more detailed information is given in Section V) can be classified by considering a smaller part of the circuit during the circuit-to-CNF conversion than shown in Figure 1.

This part is determined as follows. As in the normal circuit-to-CNF conversion – described in Section II-B – the fault site's output cone is traversed by a depth first search towards the primary outputs. However, since the contradiction in easy-to-classify faults can often be found near the fault site, this traversal terminates after reaching a certain depth in the circuit. Evaluations show that considering all fanout gates with a depth of two is sufficient. That means, after traversing the *Fanout Free Region* (FFR) including the fault site and all directly successing FFRs, the search algorithm is stopped.

Since restrictions to the primary inputs often result in untestability, the transitive fanins of all fanout gates determined as described above are transformed into CNF. Finally, all restrictions to the primary inputs are added to the SAT instance.

Figure 4 presents an illustration. It can be seen that the circuit part, considered during circuit-to-CNF conversion, is significantly smaller than using the traditional method (illustrated in Figure 1). Even if the fault site is located near the primary outputs, mostly there is a considerable difference with respect to the size.

The observations made thus far offer the opportunity of applying a preprocessing step. Instead of building the entire SAT instance as shown Figure 1 only a subcircuit is transformed into CNF (as shown in Figure 4). This partial SAT instance is given to the SAT solver. If the considered fault is untestable, this CNF may be sufficient to classify the fault correctly.

Algorithm 1 Pseudo code of the SAT-based ATPG process using the preprocessing step

```
1:  list<gate> fanouts;
2:  list<gate> POs;
3:  CNF cnf;
4:  SAT_Solver solver;
5:  fanouts=bounded_dfs();
6:  for all g ∈ fanouts do
7:      cnf.add(g.transitive_fanin);
8:  end for
9:  solver.set_timeout(small);
10: state=solver.solve();
11: if state == UNSAT then
12:     return UNSAT;
13: else
14:     solver.clear_assignments();
15:     POs=dfs();
16:     for all g ∈ POs do
17:         cnf.add(g.transitive_fanin);
18:     end for
19:     solver.set_timeout(normal);
20:     state=solver.solve();
21:     return state;
22: end if
```

Since the objective is to accelerate the instance generation step of easy-to-classify untestable faults, the timeout limit of the SAT solver can be very small. If the CNF is not proven to be unsatisfiable immediately, it is unlikely that a classification can be given at all.

If the CNF is proven to be unsatisfiable, the fault is proven to be untestable. Due to the smaller size, the instance generation process can be accelerated. In this context, the partial SAT instance is an unsatisfiable core of the complete SAT instance, that is generated using the traditional method.

Otherwise, if this CNF is satisfiable or the SAT solver aborts with a timeout, no fault classification can be given. Then, the entire SAT instance has to be built up. All variable assignments in the SAT solver are cleared and the partial CNF is augmented by the missing part of the entire CNF. The already generated CNF as well as the learned information gathered during the first solving process are kept. Afterwards, the SAT solver is started with the normal timeout limit. This procedure is generally known as *Incremental SAT* [15].

Algorithm 1 summarizes the presented methodology. First, the fanout gates with a depth of two are calculated by a bounded depth first search algorithm (line 5). Then the transitive fanins of those gates are converted into CNF (lines 6-8). Afterwards, the SAT solver is started with a small timeout limit. If the SAT instance is unsatisfiable, the fault is classified

as untestable and the algorithm terminates (line 12).

Otherwise, all assignments to the SAT variables are cleared and the primary outputs structurally influenced by the fault site are computed by the depth first search algorithm (line 14 and 15, respectively). The fanin cones of those POs are added to the partial CNF (lines 16-18). This SAT instance is solved with normal timeout limit. The result (satisfiable, unsatisfiable or timeout) gives the classification result (testable, untestable or aborted, respectively).

V. Experimental Results

In this section, experimental results are given. The preprocessing approach, described in the last section, was implemented as a prototype into the ATPG tool of NXP Semiconductors. MiniSat [4] was used to solve the SAT instances. All experiments were carried out on an Intel Xeon System (3.4 GHz, 32 GByte, Linux).

Two benchmark sets have been considered: the publicly available ITC'99 benchmarks [1] and industrial circuits, provided by NXP Semiconductors Germany GmbH, Hamburg, Germany. The names of the NXP benchmarks indicate the number of elements contained in a circuit, e.g. the circuit p3852k consists of approximately 3.85 million elements.

Table I gives an overview on the overall run times of the ATPG process. The circuit's name is shown in the first column. The second column presents the number of targets, i.e. the number of faults after fault collapsing. The number of untestable targets is given in the third column.

Two configurations have been considered. Results of the traditional SAT-based ATPG, explained in [3], are given in column *Traditional*. Column *Improved* presents the results of the approach described in Section IV. For each method, the total run time for the ATPG process and the number of aborts are presented in column *Time* and column *Abort*, respectively. In the traditional method, an abort occurs after 10 MiniSat restarts. As explained in the last section, the preprocess needs only a small timeout limit. The solving process of the partial CNF is aborted after 2 MiniSat restarts. If a complete SAT instance has to be built up, the solving process on that CNF is aborted after 10 restart in total, i.e. the restart counter is not reset after the preprocess. This guarantees that the preprocess does not slow down the solving process significantly for testable faults or hard-to-classify untestable faults.

Column *Success rate* gives the portion of untestable faults that can be classified by the preprocess with respect to the total number of untestable faults. It can be seen that for most industrial circuits more that 90% of all untestable faults can be classified correctly using the proposed preprocess.

The run time can be reduced on almost all industrial designs. Only on circuits p77k and p99k a small slowdown can be observed. On p2787k, on the other hand, there is an acceleration by a factor of more than two.

As expected – since the proposed technique aims for accelerating the SAT instance generation time of large industrial designs – the influence on the ITC'99 benchmarks is quite small. For the same reason, the number of aborts changes only slightly. While it is increased for circuits p456k and p462k, it can be decreased for the designs named p2787k, p3327k and p3852k. The differences can be explained by the changed variable allocation process using the proposed technique. Since different test patterns may be calculated, different faults are targeted.

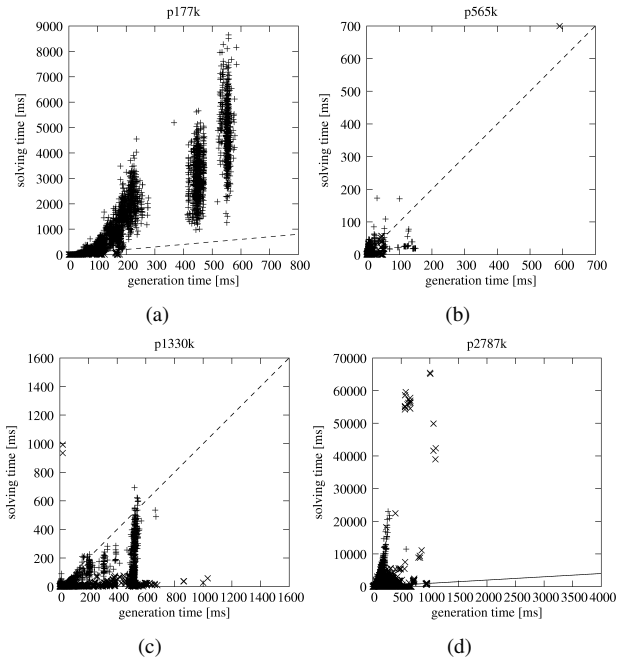


Fig. 5. Run time comparison for individual target using the preprocessing step

Figure 5 presents a run time analysis for ATPG runs employing the preprocessing technique. Again, testable faults are denoted by '+' and untestable faults are denoted by '×'. It can be seen clearly that the generation time of almost all easy-to-classify untestable faults can be reduced significantly. All clusters of untestable faults on the x-axis "disappeared".

VI. Conclusion and Future Work

The contribution of this paper is an incremental method to speed-up SAT-based test pattern generation for untestable faults in large industrial circuits. The bottleneck of classifying easy-to-classify untestable faults – the SAT instance generation – is accelerated. This is done by generating only a partial CNF during a preprocess.

The experimental results confirm that the robustness of SAT-based ATPG can be increased using the new technique. While the impact on small circuits is slight, the overall run time of the ATPG process for large industrial circuits, containing many untestable faults, can be significantly reduced by a factor of more than two.

It is focus of future work to combine the proposed method with other incremental instance generation schemes.

Acknowledgements

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

TABLE I
RUN TIMES FOR THE ATPG PROCESS

Circuit	Targets	Untest.	Traditional		Improved		
			Abort	Time	Success rate [%]	Abort	Time
b13	836	26	0	0:03m	76.9	0	0:03m
b14	22,700	156	0	0:56m	90.1	0	0:57m
b15	21,850	727	0	1:07m	41.9	0	1:04m
b17	76,493	1,958	0	2:54m	52.4	0	2:39m
b18	264,043	2,844	0	9:06m	85.4	0	9:03m
b20	45,461	319	0	2:14m	89.5	0	2:12m
b21	46,156	378	0	2:22m	90.9	0	2:17m
b22	67,540	344	0	2:48m	91.4	0	2:53m
p44k	64,105	2,385	0	49:21m	96.0	0	43:22m
p77k	163,310	9,181	0	0:27m	100.0	0	0:28m
p80k	197,834	124	0	6:33m	98.4	0	5:37m
p88k	147,742	2,640	0	2:14m	94.4	0	2:11m
p99k	162,019	2,141	0	1:35m	78.6	0	1:40m
p141k	267,948	13,815	0	3:02h	98.8	0	2:33h
p177k	268,176	13,840	0	2:37h	98.8	0	2:18h
p456k	740,660	35,396	14	47:23m	90.9	20	37:50m
p462k	673,465	132,249	0	1:10h	81.2	1	1:07h
p565k	1,025,273	28,287	0	6:25m	97.0	0	5:24m
p1330k	1,510,574	44,299	0	1:00h	95.9	0	42:14m
p2787k	2,395,388	651,868	15	15:15h	83.3	9	6:44h
p3327k	4,557,842	109,622	914	73:46h	89.1	903	71:01h
p3852k	5,507,779	164,988	849	39:01h	86.1	813	38:01h

Furthermore, the authors would like to thank Stephan Eggersglüß from the University of Bremen, Germany, and René Krenz-Bååth from Mentor Graphics, Hamburg, Germany, for helpful discussions.

REFERENCES

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

[2] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[3] R. Drechsler, S. Eggersglüß, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille. On acceleration of SAT-based ATPG for industrial designs. *IEEE Transactions on Computer-Aided Design for Circuits and Systems*, 27(7):1329–1333, 2008.

[4] N. Eén and N. Sörensson. An extensible SAT solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 2919, pages 502–518, 2004.

[5] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, 32(12):1137–1144, 1983.

[6] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic. *IEEE Transactions on Computers*, 30(3):215–222, 1981.

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

[8] W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *Proceedings of the International Conference on Computer-Aided Design*, pages 538–543, 1993.

[9] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design for Circuits and Systems*, 11(1):4–15, 1992.

[10] J. Marques-Silva and K. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 152–157, 1997.

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

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

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

[14] M. Schulz, E. Trischler, and T. Sarfert. SOCRATES: A highly efficient automatic test pattern generation system. *IEEE Transactions on Computer-Aided Design for Circuits and Systems*, 7(1):126–137, 1988.

[15] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of the Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 58–70, 2001.

[16] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design for Circuits and Systems*, 15:1167–1176, 1996.

[17] P. Tafertshofer, A. Ganz, and K. Antreich. IGRAINE - an implication graph based engine for fast implication, justification, and propagation. *IEEE Transactions on Computer-Aided Design for Circuits and Systems*, 19(8):907–927, 2000.

[18] D. Tille and R. Drechsler. Incremental SAT-instance generation for SAT-based ATPG. In *Proceedings of the IEEE Workshop on Design and Diagnosis of Electronic Circuits and Systems*, pages 68–73, 2008.

[19] G. Tseitin. On the complexity of derivation in the propositional calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968. English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.