Techniques for SAT-Based Constrained Test Pattern Generation

Jiří Balcárek, Petr Fišer, Jan Schmidt
Dept. of Computer Science & Engineering
Czech Technical University in Prague, FIT
Prague, Czech Republic
balcaji2@fit.cvut.cz, fiserp@fit.cvut.cz, schmidt@fit.cvut.cz

***Abstract*—Testing of digital circuits seems to be a completely mastered part of the design flow, but constrained test patterns generation is still a highly evolving branch of digital circuit testing. Our previous research on constrained test pattern generation proved that we can benefit from an implicit representation of test patterns set in CNF (Conjunctive Normal Form). Some techniques of speeding up the constrained SAT-based test patterns generation are described and closely analyzed in this paper. These techniques are experimentally evaluated on a real SAT-based algorithm performing a constrained test patterns compression based on overlapping of test patterns. Experiments are performed on a subset of ISCAS'85 and '89 benchmark circuits. Results of the experiments are discussed and recommendations for a further development of similar SAT-based tools for constrained test patterns generation are given.**

Keywords- testing; implicit representation; SAT; ATPG; constrained test

I. INTRODUCTION

As the number of analog and mixed signal parts in electronic devices continuously grows, more and more attention of researchers is focused on testing of these circuits. It could seem that testing of the digital circuits is a completely mastered part of the design flow. In fact, there are still some areas of digital circuit testing waiting to be resolved. There is a need to generate constrained test sets, e.g., to decrease heat dissipation and power consumption during the test [1] or test application time [2].

The basic idea of general constrained test pattern generation [3] is shown in Figure 1. There is a set of test patterns for each fault, by which it is detected. The most suitable (according to constraints) test pattern or a subset of the most suitable test patterns is chosen by application of specific additional constraints.

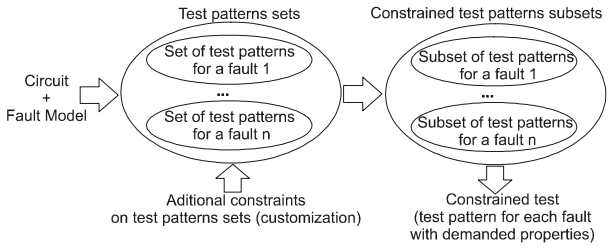

Figure 1. Basic concept of the constrained test generation

Conventional ATPGs (Automatic Test Pattern Generators) are based on PODEM [4] or FAN [5] algorithms. Here test patterns are generated by traversing the circuit structure, with backtracking employed. In contrast to this approach, SAT-based ATPGs [6] are searching for the test patterns by SAT (satisfiability) solving. The instance of the SAT problem in CNF (Conjunctive Normal Form) is generated for each fault. This CNF implicitly represents the whole set of the test patterns detecting a given fault. A test pattern for a fault is obtained by SAT solving of its CNF. A fault is untestable if there is not a satisfiable solution of its CNF.

Our research is focused on a class of algorithms where the same set of SAT instances is repeatedly processed with different constraints. Algorithms are forced to handle repeated processing of the same CNFs, which can cause a significant time overhead [7]. It is expected, that the majority of constrained SAT instances are classified as UNSAT (UNSATisfiable) [7, 8]. It means that they do not contribute by any sub-solution and must be repeatedly solved with other constraints.

Our experiments on CNF processing show the differences (time/memory consumption) between

- their repeated generation,
- storing,
- and storing of reduced CNFs.

Reduction of stored CNFs is made by solution set preserving SAT transformations [8], e.g., by resolution or propagation of 1-literal clauses. These reductions do not change the set of solutions.

Advantages and disadvantages of these approaches are discussed over the results and recommendations for further design of SAT-based constrained test patterns generation algorithms are proposed.

Next, possibilities of detecting unsatisfiable constrained CNFs were explored. The UNSAT instances are early detected by resolving conflicts between the demanded fixed values of the signals in the circuit and their values obtained by CNF implications. The SAT solving of these unsatisfiable instances can be skipped, which can significantly speed up the algorithm. Such a process is in the paper referred as *static* or *dynamic UNSAT filter* based on the performed type of implications.

II. PREVIOUS WORK ON CONSTRAINED TEST PATTERNS GENERATION

Generation of test patterns with some constraints is a common process in digital designs testing. Test patterns are constrained to be better compressed, tailored for a scan-

based designs, to speed up of the test generation process, to avoid illegal test patterns (on primary inputs, buses, tri-state elements), etc.

Constrained ATPG can be used for a broadside transition testing [9]. Conventional ATPG based on PODEM [4] algorithm produces a set of test patterns with a great number of the test patterns covering functionally untestable transition faults. These functionally untestable transition faults do not need to be tested because they do not affect the normal functionality of the chip (they cannot occur). Nevertheless, testing the chip for these faults may cause the test fail, and thus decrease the yield. Thus an ATPG is constrained by a set of illegal (unreachable) states (circuit's signals values) that enable detection of the untestable transition faults. These constraints are described as a Boolean formula in CNF. Constrained ATPG fixes variables in the generated test pattern and at the same time fixes corresponding variables in the CNF of constraints and checks the CNF for conflicts in variable settings. A conflict means that the illegal state occurs; an ATPG performs backtracking and searches for a different variable setting in the test pattern. Such a constrained set of test patterns reduces activation of the functionally untestable transitions, which increases the quality of the test set and reduces the yield loss due to testing of the functionally untestable faults.

In [10], an implicit representation of all test sequences for a fault is used to check for conflicts with rule matrices (of the cellular automata) during the test patterns generation constrained to cellular automata. This technique is used for testing of midsize controllers. The entire set of the test sequences of the controller under test is implicitly represented by a BDD (Binary Decision Diagram). The BDD is explored and only those sequences which can be reproduced by cellular automata are selected. A concrete algorithm and the way of application of the constraints were not described in the paper.

An ATPG for industrial circuits with restrictors [11] represents another application based on constrained test patterns generation. Industrial circuits contain a great number of buses, tri-state elements and other parts, where the set of permitted signal values is constrained. This structural information is stored as a set of restrictions, which are used by an ATPG to prune the search space and speed up the test patterns generation. This method was implemented as a conventional FAN algorithm [5] extended by a restrictor (constraint) concept.

Low power tests are mostly built from pre-generated test patterns [1] by their reordering. Constraints can be formed by a heuristic algorithm (test sequence is build incrementally). The best test patterns to be ordered into the test sequence are searched instead of the ordering of the pre-generated test patterns.

Test patterns compression based on a tailoring of test patterns for a scan-chain [2] is a compression technique based on the RESPIN architecture [2]. The test patterns generation process for a scan chain, where suitable test patterns to be overlapped are produced, is denoted as *tailoring of the test patterns*. Suitable test patterns are produced by a conventional ATPG tool performing dynamic compaction, while constraints to the circuit primary inputs are applied.

A similar approach to a compression of test patterns for RESPIN is the SAT-Compress algorithm [7], which is based on SAT. A set of test patterns for each fault is implicitly represented by a SAT instance in the CNF. These CNFs are not stored in memory, but are generated on-the-fly when requested by the SAT-Compress algorithm. A test pattern for each fault is obtained by SAT solving of the CNF instances with applied constraints (i.e., fixed variables values). The main difference between the tailoring of the test patterns for RESPIN and the SAT-Compress algorithm is the implicit representation of the test patterns set. The SAT-based approach is known to be much faster in test patterns generation for hard to be tested faults than standard ATPG tools, but easily tested faults can create a time overhead [6]. It can be assumed that similar behavior can be observed in the constrained test patterns generation process.

III. SAT-BASED TEST PATTERN GENERATION

A. *Circuit to SAT Transformation*

In SAT-based ATPGs, the ATPG problem is reduced to a SAT problem. Here the fault-free and faulty circuits are transformed into CNF, to obtain a SAT problem instance. The solutions of this instance are test vectors detecting the respective fault. Thus, the CNF implicitly represents the whole set of test patterns detecting a given fault.

A CNF Φ with m Boolean variables is a conjunction of n clauses, where each clause is a disjunction of literals. Each literal is a Boolean variable or its complement. The CNF Φ is satisfiable, if there is some assignment of variables, for which all clauses are satisfied.

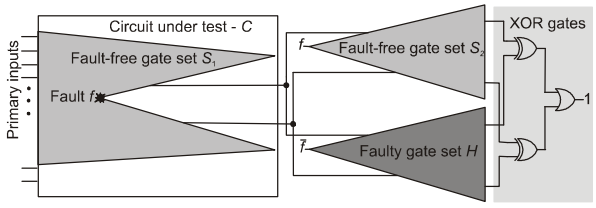

Figure 2. SAT instance generation for an ATPG

A Boolean variable is assigned to each signal in the circuit C (Circuit under test). Each gate in the output cone of the fault to primary outputs (POs) of the circuit is included in the set $H \subseteq C$ and all gates in the cone from the observable POs to the primary inputs (PIs) is included in the set $S \subseteq C$ ($S = S_1 \cup S_2$), see Figure 2. For each gate, the CNF Φ_g is derived from its characteristic function. The CNF Φ_c representing the fault free part of the circuit is constructed as a conjunction of all CNFs of gates $g_1,\ldots,g_n \in S$:

$$\Phi_c = \prod_{i=1}^{|S|} \Phi_{gi}$$

To generate the test for a fault F, the characteristic function Φ_f of the faulty circuit is generated as conjunction of all CNFs of gates $g_1,\ldots,g_n \in H$:

$$\Phi_f = \prod_{i=1}^{|H|} \Phi_{gi}$$

The SAT instance for a fault F is obtained as a product of Φ_c, Φ_f and a characteristic function of a XOR of the fault-free and faulty circuit POs Φ_{XOR}:

$$\Phi_{test}^{F} = \Phi_c \cdot \Phi_f \cdot \Phi_{XOR}$$

Finally, the variable of the faulty signal in Φ_f is fixed on the faulty value and its image in Φ_c is complemented. Output variable of the Φ_{XOR} is assigned by logic value 1 (Boolean difference must result in 1, if it is detectable fault).

The Φ_{test}^{F} solutions represents the whole set of test patterns detecting the fault F. If Φ_{test}^{F} is unsatisfiable, the fault F is undetectable. More information can be found in [6, 8].

B. SAT-based ATPG Algorithm

A conventional SAT-based ATPG algorithm [6] can be described in four steps:

1) Generate a fault list for a given circuit.
2) Pick one fault from the fault list and generate its CNF
3) Solve the SAT problem for this CNF. The solution represents a test pattern detecting the respective fault. If the CNF is unsatisfiable, the fault is removed from the fault list as an undetectable fault.
4) Simulate the test pattern obtained in step 3 and remove all detected faults from the fault list. Repeat steps 2-4 until the fault list is empty.

For details on the SAT-based ATPGs, see [6, 12].

IV. The SAT-Compress Algorithm

The presented speedup techniques are experimentally evaluated on the SAT-Compress algorithm [7]. It performs compression of the test patterns based on overlapping. The SAT-compress algorithm searches for the best overlap by gradually building the bitstream of compressed test patterns for the scan chain. The basic algorithm is shown in Fig. 3.

1) First, the complete fault list is generated (FL).
2) Redundant faults are removed from the FL.
3) A zero state (all-zero test pattern) or the test pattern covering any fault from the fault list is used as the initial test pattern.
4) The pattern is simulated and all detected faults are deleted from the fault list. The leftmost bit of the pattern goes to the resulting bitstream.
5) The pattern is shifted left and a DC bit is put to its rightmost position. This is the mask for the next pattern. The care bits of the mask constrain the values of primary inputs (PI) in subsequent SAT instances.
6) To generate the next test pattern having the highest overlap with the previously generated one, a CNF for each fault in the fault list is generated, while the primary input variables are set according the mask. The CNF for a fault is processed in the order given by its position in the fault list. If a CNF for a fault is satisfiable for a given assignment of primary variables, a new test pattern is obtained. If none of these CNFs is satisfiable, the pattern is shifted one more bit left, which generates a new mask of SAT constraints.
7) These operations (4-7) are performed while the fault list is not empty or until all care bits from the mask of primary inputs setting are shifted out while there is still no satisfiable CNF (the rest of the faults in FL is undetectable).

```
1. Generate FL                                  (FL = fault list)
2. Remove redundant fault from the FL
3. TP = 0                        (TP = test pattern = all-zero seed)
   mask = DC                                    (DC = don't care)
4. FL = FL - detected_by_simulation(TP)
   compressed_bitstream += TP[0]
5. mask = TP[1 .. n-1] + DC
6. do {
       for each fault in FL {
          Create CNF
          Apply mask to PIs in CNF
          if CNF is SAT {
             break the for loop
          }
       }
       TP = CNF_Solution
       FL = FL - detected_by_simulation(TP)
       compressed_bitstream += TP[0]
       mask = TP[1 .. n-1] + DC
   }
7. while (FL!=0)
```

Figure 3. The SAT-Compress algorithm

V. Techniques of Speeding up The Constrained Test Generation Process

A class of SAT-based algorithms for constrained test patterns generation deals with repeated processing of the same SAT instances in CNF with different constraints. It can cause constrained test patterns generation process to be time-consuming [7]. The constrained test patterns generation process is analyzed and additional techniques of its speedup based on the CNFs manipulation and its filtering based on the CNFs satisfiability are discussed.

A. On-the-Fly CNF Generation vs. CNF Storing

The CNFs generation takes a significant part of the constrained test patterns generation process. The SAT instances can be generated on-the-fly, or they can be pre-generated and stored in memory. Either original CNF are stored, or the CNFs are simplified by solution set preserving reductions, to reduce memory requirements.

In the CNF generation on-the-fly approach the CNFs are repeatedly generated and constrained in the test generation process. It is obvious that memory requirements are negligible. On the other hand, such a repeated generation of CNFs can increase the test generation time. The CNF generation time has been proved to take a significant part of the test patterns generation process for SAT ATPGs [6].

In the latter approach, CNFs for each fault are generated only once in the initial part of the algorithm and stored in memory. The time overhead incurred by repeated CNF generation is reduced. However, constraints change in the test generation process, thus original CNFs (unconstrained) must be repeatedly loaded into the SAT solver. Loading of the CNFs into SAT solver should create much less time overhead, but the number of literals stored in memory can be unfeasible for larger circuits. The number of stored literals can be further reduced by solution set preserving SAT reduction [8], e.g. resolution and propagation of 1-literal clauses.

B. Filering out the UNSAT CNFs

Constrained test patterns generation algorithms must solve a great number of constrained SAT instances repeatedly. It is expected, that the majority of these instances are unsatisfiable [2, 8] with given constraints (do not produce a test pattern). In SAT-Compress, 98% of generated CNFs are unsatisfiable with given constraints on average [8]. Generation and solving of these CNFs can cause a significant time overhead. That is why we focused on filtering of these UNSAT instances to speed-up the constrained test patterns generation.

Filtering out the UNSAT instances can be defined as an early detection of unsatisfiability, as a conflict between fixed faulty signal variable value (and its fault-free complement) in the CNF and implications of the constraints.

Two algorithms for filtering of the unsatisfiable instances incurred in the constrained test generation process are described in this subsection. An implication filter using static implications to detect unsatisfiability is used in the first approach, which is then extended by dynamic implications in the second.

1) Static implication filter is based on the observation that ATPG CNFs consist of 70% of 2-literal clauses and 24% of 3-literal clauses [8] on average. Each 2-literal clause can be substituted by two implication rules, e.g., a clause $x \vee y$ corresponds to implications $\neg x \Rightarrow y$ and $\neg y \Rightarrow x$. These implications can be further used to speed up the SAT solving [12] or to propagate constraints (fixed constraints are propagated by implications).

The static implication filter uses the implication rules to fix additional variables in the test pattern. Thus an essential part is the implication table. It consists of implication rules created from 2-literal clauses from the CNF of the fault free circuit. Each row of the implication table consists of the list of the variables and its values to be set if the row is referenced. The row is indexed by the name of the variable and its polarity (two rows for a literal). A variable fixed in the constrained test pattern selects a row in the implication table and cause the variables stored in the row to be also fixed in the constrained test pattern. The implication table is constructed once in the initial part of the constrained test generation algorithm.

The constrained test patterns generation algorithm extended by static implication filter checks unsatisfiability of the CNF instance to be generated, without need of its generation or solving. A variable corresponding to the faulty signal is in the constrained test pattern fixed as a complement of the faulty signal value. The constraints to be applied are also fixed in the constrained test pattern and further variables are fixed by implication rules stored in the implication table. It means that each fixed variable (by constraints or implications) addresses the row of the implication table and the variable or its complement stored in the list of variables is also fixed. The CNF to be generated is marked as unsatisfiable if some conflict between value of the variable to be fixed by implication rule and its value in the constrained test pattern occurs. The CNF instance is marked as unsatisfiable and it is skipped without generation and solving because it can not be satisfied with given constraints.

The static implication filter is a simple method for unsatisfiability checking of the constrained SAT instances. Its efficiency depends on the number of constrained variables. A high number of implication rules (2-literal clauses) gives us a solid ground for setting of the further variables in the constrained test pattern and increases our chances to detect unsatisfiability of the CNF to be generated. Construction of the implication table and unsatisfiability checking represents a negligible part of constrained test generation process and filtering of the UNSAT instances can cause significant speed-up.

2) Dynamic implication filter is an extention of the static implication filter. The static implication filter fixes variables values by implications stored in the implication table. Additional implications can be found in the CNF of the fault free circuit (fault free part is the same for CNFs of all faults). Each 2-literal clause is stored in the implication table (once in the initial part of the algorithm) as two implication rules and its removed from the CNF of the fault free circuit. Thus, the CNF of the fault free circuit consists of clauses with 3 and more literals only. The constraints to be applied are fixed in the constrained test pattern and the CNF of the fault free circuit is searched for clauses which produce additional implications with given variables values e.g., the clause $\neg x \vee y \vee z$ with fixed variable values x=1 and y=0 implies z=1. Searching for new implications continues, while there are some changes in fixed variables in the constrained test pattern.

It is obvious that further variable values can be fixed in the constrained test pattern. A higher number of fixed variables increase chances to find conflict and to identify more unsatisfiable instances than the static filter. The time consumption of the dynamic filter is much higher than of the static filter, because the set of the clauses from the fault free part of the circuit must be generated and searched for new implications for each constraints applied.

VI. EXPERIMENTAL RESULTS

In the following two subsections, we experimentally evaluate and compare presented techniques. The measurements were performed on a CPU Intel Core 2 Duo – 1,8GHz with 1GB RAM.

Experiments have been performed on a subset of smaller ISCAS'85 [13] and '89 [14] benchmark circuits, because memory requirements for CNFs storing were unfeasible for bigger circuits from those benchmark sets.

Properties of the proposed speedup techniques are demonstrated on the SAT-Compress algorithm [7] as a representative of the defined class of algorithms. MiniSat v1.14 [15] has been used as a SAT solver.

A. On-Fly CNF generation vs. CNF storing

A comparison of the three techniques of CNFs processing is presented in Table I. The first column of the table "*bench name*" represents the name of the benchmark circuit from ISCAS'85 or '89. Differences between processing of the CNFs on-the-fly, storing and storing of the reduced CNFs are shown in the three columns. These columns consist of columns "*CNF*" showing the time spent by a CNF generation. The next column "*SAT*" represents the time spent by a SAT solving of the processed CNFs. Column "*SIM*" shows the time spent by simulation and column "*SUM*" is the time consumption of the whole algorithm. Moreover, columns with storing CNFs and storing of the reduced CNFs consist of the column "*Lit. Count*" which represents a total number of literals in the stored CNFs, the column "Store" showing the time consumption of the CNFs storing or its reduction and storing. The last row of the table "Avg." represents average values of all columns.

First, let us focus on the time spent by the CNFs generation. Experimental measurements show that processing of the stored CNF instances is in all cases faster than their generation on-the-fly and generation of the reduced CNFs is even faster then generation of the stored CNFs, e.g., for benchmark circuit c3540 is the time of CNFs generation 2205 seconds while loading of the previously generated CNFs stored in the memory is made in 1112 seconds and for reduced CNFs it takes only 436.8 seconds. The time consumption of the storing and reductions of the CNFs seems to be negligible in comparison with CNFs generation and SAT solving. On the other hand, the number of stored literals grows significantly with the size of the circuit (number of gates), e.g., for test compression of the benchmark circuit c3540 having 1648 gates, there must be 3428 CNFs stored, which is 31,439,618 literals (in the reduced CNF). It is obvious that storing the CNFs is unfeasible for large circuits, because of memory consumption. Moreover, storing the reduced CNFs is also unfeasible for large circuits, because the reduction of the CNFs size is not as significant as we hoped [8].

The average values confirm previous observations. The time consumption of the CNFs processing can be dramatically decreased by storing the reduced CNFs in memory. An average total time for the CNFs processing show, that processing of the stored CNFs is in average 1.34 times faster than its processing on-the-fly. Thus it seems that storing of the CNFs is better than processing of the CNFs on-the-fly, but the memory consumption of the stored literals can be unfeasible. The storing of the reduced CNFs, does not decrease processing time of the CNFs, because solution set preserving reductions are time consuming for bigger instances. For example, the time of the reductions and storing of the CNFs for the benchmark circuit c3540 is 1322 seconds while solving of these CNFs takes only 1730 seconds.

It can be concluded that for small circuits it is better to store CNFs or reduced CNFs, but this is unfeasible for large circuits, because of high memory requirements. The SAT solving times indicate that generation of the CNFs on-the-fly can be the best way, because it is not bounded by the memory requirements.

B. Static and Dynamic implication filter

A comparison of filtering techniques is presented in Table II. The first column of the table "*bench name*" represents the name of a benchmark circuit from ISCAS'85 or '89. Differences between the basic algorithm (SAT-Compress [7]) and its modification with static and dynamic filtering are shown in the three columns. These columns consist of the column "*Gen.*", with a total number of the generated CNFs and "*Used.*", showing a total number of the CNFs giving a satisfiable solution. The next sub-column "*Red*" shows the percentage reduction of the number of processed CNFs referred to a basic algorithm. The sub-column "*Filter*" shows the time spent by filtering of the unsatisfiable CNF instances. The following sub-columns "*CNF*", "*SAT*", "*SIM*" and "*SUM*" have the same meaning as in Table I. The last row of the table "*Avg.*" represents an average value of the column.

Experimental results show that filtering of the unsatisfiable CNFs can speed up the process of the constrained test patterns generation more than 2 times, e.g., the total test patterns compression time of the basic algorithm for the benchmark circuit c3540 is 5784 seconds, while with a static filer it takes 2793 seconds and with dynamic filter the total test patterns compression time decreased to 2472 seconds. The static filter as a simple fast technique of detecting of the unsatisfiable instances is highly effective and detects 43% of all the processed unsatisfiable instances on average. Moreover, the dynamic filter is able to detect an additional 11% of the unsatisfiable CNF instances on average, but it is much more time-consuming than the static filter.

The implication filter seems to be a promising technique. The static filter can be used for any circuit and grants a significant speedup of the constrained test generation process by significantly decreasing the number of the unsatisfiable CNFs solved. The dynamic filter is better for small circuits, because searching for dynamic implications is much more time-consuming.

VII. CONCLUSION

A general class of the SAT-based constrained test patterns generation algorithms has been stated. Extending

techniques of speedup of these algorithms has been discussed and evaluated by experimental results.

The differences between the CNFs processing on-the-fly, processing of the stored CNFs or reduced CNFs have been discussed and shown on a set of ISCAS'85 and '89 benchmark circuits. It can be concluded that even if a generation of the CNFs on-the-fly can be time-consuming, it is still the best technique of CNF processing in a general case.

Techniques of the filtering of the unsatisfiable CNFs based on the static and dynamic implications have been presented. Our experimental evaluation proved that it can be a power-full technique for speedup of the SAT-based constrained test patterns generation.

ACKNOWLEDGMENT

This research has been supported by MSMT under research program MSM6840770014, by the grant of the Czech Grant Agency GA102/09/1668 and the grant of the Czech Technical University in Prague, SGS11/089/OHK3/1T/18.

REFERENCES

[1] Patrick Girard, Nicola Nicolici, Xiaoqing Wen, "Power-Aware Testing and Test Strategies for Low Power Devices," Publisher Springer Netherlands, ISBN: 1441909273, 2009, p.353.

[2] R. Dorsch and H.-J. Wunderlich, "Tailoring ATPG for embedded testing," in Proc. ITC, 2001, pp. 530–537.

[3] Balcárek, J., "Implicit Rrepresentations in Customized Testing of Digital Circuits," Počítačové architektury&diagnostika (PAD'2010), Češkovice (ČR), 13.-15.9.2010.

[4] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits", IEEE Trans. On Computers, 1981, pp. 221-222.

[5] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," IEEE Trans. Comput., C-32(12) , 1983, pp. 1137-1144.

[6] Drechsler, R., Eggersglüß, S., Fey, G., Tille, D., "Test Pattern Generation using Boolean Proof Engines," Publisher Springer Netherlands, ISBN 978-90-481-2360-5, 2009, XII, p. 192.

[7] Balcárek, J., Fišer, P., Schmidt, J.: Test Patterns Compression Technique Based on a Dedicated SAT-based ATPG, Proc. of 13th Euromicro Conference on Digital Systems Design (DSD'10), Lille (France), 1.-3.9.2010, pp. 805-808.

[8] Balcárek, J., Fišer, P., Schmidt, J.: On Properties of SAT Instances Produced by SAT-Based Test Pattern Generators, Proc. of Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09), Znojmo, ČR, 13.-15.11.2009, pp. 3-10.

[9] Xiao Liu, Michael S. Hsiao, "Constrained ATPG for Broadside Transition Testing," in Proc. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), 2003, pp.175.

[10] F. Fummi, D. Sciuto, "Implicit test pattern generation constrained to cellular automata embedding," Proc. of 15th IEEE VLSI Test Symposium (VTS'97), 1997, pp.54.

[11] Konijnenburg, M.H., van der Linden, J.Th., van de Goor, A.J. "Automatic test pattern generation for industrial circuits with restrictors" Microelectronics Journal, 26 (7), 1995, pp. 635-645.

[12] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability,"IEEE Transactions on Computer-Aided Design, 1992, pp. 4-15.

[13] Brglez, F., Fujiwara, H.: A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan. Proc. of International Symposium on Circuits and Systems, pp. 663-698, 1985.

[14] Brglez, F., Bryan, D., Kozminski, K.: Combinational Pro_les of Sequential Bench-mark Circuits. Proc. of International Symposium of Circuits and Systems, pp. 1929-1934, 1989.

[15] Eén, N., Sorensson, N.: An Extensible SAT-solver. Lecture Notes in Computer Science, Theory and Applications of Satis_ability Testing, vol. 2919/2004, pp. 333-336, 2004.

TABLE I. EXPERIMENTAL RESULTS FOR THE PROCESSING OF CNFS

Bench name	CNFs on-the-fly				Stored CNFs						Stored reduced CNFs					
	CNF	*SAT*	*SIM*	*SUM*	*Lit. Count*	*CNF*	*SAT*	*SIM*	*Store*	*SUM*	*Lit. Count*	*CNF*	*SAT*	*SIM*	*Store*	*SUM*
	[s]	*[s]*	*[s]*	*[s]*	*[-]*	*[s]*	*[s]*	*[s]*	*[s]*	*[s]*	*[-]*	*[s]*	*[s]*	*[s]*	*[s]*	*[s]*
c432	1.766	1.7969	0.06	3.625	896428	1.031	1.172	0.063	0.484	2.75	835842	0.5	1.15625	0.063	1.172	2.891
c499	1.047	1.8281	0.03	2.906	1657890	0.641	1.922	0.063	0.813	3.438	1608764	0.391	1.1875	0.078	1.594	3.25
c880	37.2	87.797	12.5	137.5	1121334	27.78	93.88	13.77	0.688	136.1	1053408	10.95	75.4063	9.547	1.734	97.64
c1355	9.656	20.5	0.38	30.53	6787138	7.516	25.17	0.469	4.734	37.89	6593756	4.109	23.9531	0.563	14.59	43.22
c1908	39.72	72.938	13.4	126	9743432	25.58	64.23	11.42	8.75	110	9384959	18.61	60.1719	11.34	46.67	136.8
c2670	212	972.55	3.98	1189	11631239	154.8	1047	4.922	8.703	1215	11151282	107.1	1213.89	6.422	59.88	1387
c3540	2205	3176.1	403	5784	33194406	1112	2201	260.7	42.53	3615	31439618	430.8	1730.28	220.8	1322	3704
c5315	27.19	153.59	17.7	198.5	22987852	16.31	169.6	16.67	15.52	218.1	22437695	16.77	162.172	20.69	50.47	250.1
s420	3.875	8.8438	0.08	12.8	208359	2.031	9.563	0.063	0.125	11.78	146410	1.063	8.51563	0.063	0.703	10.34
s510	0.453	1.25	0.38	2.078	346745	0.156	1.594	0.156	0.219	2.125	266822	0.063	1.26563	0.359	1.531	3.219
s526	1.172	7.25	0.14	8.563	173319	0.516	7.219	0.203	0.109	8.047	130328	0.313	7.0625	0.266	0.375	8.016
s526n	1.094	6.4531	0.17	7.719	173156	0.531	6.141	0.281	0.125	7.078	130075	0.313	6.17188	0.188	0.375	7.047
s641	6.953	21.203	1.17	29.33	539954	5.109	20.72	1.063	0.297	27.19	480537	2.719	19.8438	1.25	1.766	25.58
s713	6.625	18.969	1.27	26.86	674441	4.375	19.22	1.219	0.375	25.19	605613	2.703	18.2031	1.297	2.328	24.53
s820	8.078	34.234	1.97	44.28	451536	5.266	34.92	1.719	0.266	42.17	362319	2.594	33.4531	1.688	3.672	41.41
s832	9.609	37.609	2.81	50.03	462205	4.656	31.94	1.156	0.281	38.03	369538	2.875	30.3438	1.141	3.781	38.14
s838	46.52	126.58	0.2	173.3	705499	27.08	127.5	0.219	0.422	155.3	503436	12.98	109.047	0.391	6.203	128.6

Bench name	CNFs on-the-fly				Stored CNFs						Stored reduced CNFs					
	CNF	*SAT*	*SIM*	*SUM*	*Lit. Count*	*CNF*	*SAT*	*SIM*	*Store*	*SUM*	*Lit. Count*	*CNF*	*SAT*	*SIM*	*Store*	*SUM*
	[s]	*[s]*	*[s]*	*[s]*	*[-]*	*[s]*	*[s]*	*[s]*	*[s]*	*[s]*	*[-]*	*[s]*	*[s]*	*[s]*	*[s]*	*[s]*
s953	14.95	62.781	3.95	81.69	1037128	9.625	61.14	3.688	0.625	75.08	820193	3.781	58.8125	3.938	11.78	78.31
s1196	93.02	230.5	15.9	339.4	2134625	60.91	248.5	13.09	1.234	323.7	1884138	25.31	199.391	14.2	16.11	255
s1238	110.5	270.66	16.4	397.5	2350333	59.8	236.5	15.42	1.375	313	2049258	31.23	223.578	15.28	21.02	291.1
s1423	28.03	79.781	5.25	113.1	2635526	16.14	108.8	6.406	1.609	133	2538529	7.828	68.0781	5.125	5.703	86.73
s1488	3.344	27.234	2.06	32.64	1209448	2.234	27.73	1.656	0.734	32.36	1002073	1.016	27.75	1.984	25.58	56.33
s1494	3.906	33.141	2.02	39.06	1217152	2.609	34.84	2.094	0.75	40.3	1007773	1.453	35.5469	2.016	26.14	65.16
Avg.	124.84	237.11	21.94	383.9	4449528	67.22	199.11	15.5	3.94	285.78	4208798.5	29.8	178.92	13.85	70.65	293.24

TABLE II. EXPERIMENTAL RESULTS FOR THE UNSAT FILTERING

Bench name	Basic algorithm						Basic algorithm + static filter						Basic algorithm + static + dynamic filter					
	Gen.	*Used*	*CNF*	*SAT*	*SIM*	*SUM*	*Red*	*Filter*	*CNF*	*SAT*	*SIM*	*SUM*	*Red*	*Filter*	*CNF*	*SAT*	*SIM*	*SUM*
	[-]	*[-]*	*[s]*	*[s]*	*[s]*	*[s]*	*[%]*	*[s]*	*[s]*	*[s]*	*[s]*	*[s]*	*[%]*	*[s]*	*[s]*	*[s]*	*[s]*	*[s]*
c432	3517	74	1.77	1.8	0.06	3.63	25	0	1.17	1.36	0.08	2.61	64.5	0.3	0.66	0.58	0.11	1.65
c499	3210	73	1.05	1.83	0.03	2.91	9.1	0.02	0.63	1.77	0.05	2.47	49.6	0.94	0.38	0.86	0.08	2.26
c880	70395	181	37.2	87.8	12.5	137.5	46	0.19	22.3	48.8	11.7	82.99	54.6	5.22	18.4	43.5	12	79.12
c1355	13236	95	9.66	20.5	0.38	30.54	32	0.02	6.28	13.6	0.39	20.29	70.4	3.22	3.16	5.36	0.61	12.35
c1908	35443	162	39.7	72.9	13.4	126	19	0.13	31	59.3	13.5	103.93	47.3	8.81	21	34.7	13.5	78.01
c2670	277808	355	212	973	3.98	1188.98	33	1.45	132	635	3.91	772.36	52	58.2	91	471	4.3	624.5
c3540	717888	347	2205	3176	403	5784	53	4.14	1041	1352	396	2793.14	62.9	254	829	993	396	2472
c5315	26978	289	27.2	154	17.7	198.9	15	0.17	25.1	128	17.9	171.17	55.4	88.2	12.5	60.9	18.6	180.2
s298	2782	93	0.08	0.8	0.05	0.93	47	0	0.14	0.33	0.06	0.53	47.8	0.03	0.06	0.5	0.03	0.62
s382	1959	59	0.2	0.78	0.02	1	46	0	0.16	0.34	0.05	0.55	49	0.08	0.11	0.42	0	0.61
s400	4088	69	0.33	1.7	0.03	2.06	54	0	0.17	0.77	0.02	0.96	54.9	0.14	0.14	0.8	0.03	1.11
s420	17210	93	3.88	8.84	0.08	12.8	39	0.08	2.53	5.42	0.05	8.08	45	0.58	1.98	5.34	0.05	7.95
s444	2359	59	0.19	1	0.05	1.24	58	0	0.13	0.34	0.06	0.53	62.8	0.09	0.05	0.44	0.02	0.6
s510	2899	76	0.45	1.25	0.38	2.08	58	0.02	0.13	0.53	0.27	0.95	59.1	0.16	0.19	0.47	0.31	1.13
s526	15563	134	1.17	7.25	0.14	8.56	49	0.02	0.84	3.36	0.19	4.41	49.8	0.42	0.59	3.91	0.23	5.15
s526n	13901	134	1.09	6.45	0.17	7.71	48	0.06	0.56	3.28	0.16	4.06	49.6	0.38	0.53	3.06	0.3	4.27
s641	20397	136	6.95	21.2	1.17	29.32	37	0.02	3.83	12.8	1.14	17.79	50.2	1.08	3.77	10.2	1.02	16.07
s713	17928	130	6.63	19	1.27	26.9	35	0.02	4.06	11.9	1.31	17.29	46.2	1.59	3.14	10.2	1.38	16.31
s820	51351	206	8.08	34.2	1.97	44.25	38	0.13	5.08	20.3	2.22	27.73	38.6	1.09	5.86	19.4	1.73	28.08
s832	56590	203	9.61	37.6	2.81	50.02	39	0.13	5.66	22.5	2.36	30.65	39.5	0.86	5.91	21.9	2.31	30.98
s838	114070	187	46.5	127	0.2	173.7	40	0.13	29.5	77.9	0.16	107.69	43.6	7.55	27.8	80	0.22	115.57
s953	63076	198	15	62.8	3.95	81.75	59	0.06	6.23	23.5	4.03	33.82	67.7	3.73	5.47	18.2	3.5	30.9
s1196	180005	249	93	231	15.9	339.9	48	0.41	47.3	115	15.7	178.41	59.5	18.3	37.5	86	14.5	156.3
s1238	213134	247	111	271	16.4	398.4	49	0.55	57.9	132	16	206.45	56.3	20.4	48	107	17	192.4
s1423	44892	149	28	79.8	5.25	113.05	52	0.17	14.9	35.5	6.34	56.91	59.8	9.41	12.2	28.2	6.59	56.4
s1488	19053	204	3.34	27.2	2.06	32.6	48	0.08	1.81	13.4	2.11	17.4	50.5	4.27	1.64	12.6	2.09	20.6
s1494	22878	201	3.91	33.1	2.02	39.03	51	0.08	1.81	15.5	2	19.39	54	4.31	1.88	14.4	2.2	22.79
s5378	378447	502	210	2818	188	3216	50	4.25	104	1356	185	1649.25	63.3	274	79.9	951	191	1495.9
s9234	4654444	749	9233.7	65926	7016.4	82176.1	42	134	5535	35681	6986	48336	52.6	4014.7	4729.4	28495	7072.3	44311.4
s13207	10733919	1109	25209	228267	25035	278511	60	228.4	9869	76220	24511	110828.4	64.1	14130	9048.7	67857	24485	115520.7
s15850	11160862	980	36446	304751	30766	371963	52	315.1	17406	140654	30269	188644.1	59.5	25019	14212	114844	30138	184213
s35932	2000941	1419	5529.6	131884	290.4	137704	17	402.4	5154	113974	293	119823.4	20.7	10319	4935	113585	294.5	129133.5
s38584	11131264	2256	19658	478943	17419	516020	62	591.2	7392	175764	17696	201443.2	68.3	138870	6265	144431	17612	307178
Avg.	1274923.8	346	3004.8	36910.5	2461.2	42376.6	42.7	51.03	1421.2	16557	2437.5	20466.8	53.6	5852.1	1224.3	14309	2433.1	23818.5