Techniques for SAT-Based Constrained Test Pattern Generation

Jiri Balcarek, Petr Fiser, Jan Schmidt
Dept. of Computer Science & Engineering
Czech Technical University in Prague, FIT
Thakurova 9, CZ-160 00, Prague 6
fax: +420 22435 9819, tel: +420 22435 9848
balcaji2@fit.cvut.cz, fiserp@fit.cvut.cz, schmidt@fit.cvut.cz

Corresponding author:
Jiri Balcarek
Dept. of Computer Science & Engineering
Czech Technical University in Prague, FIT
Thakurova 9, CZ-160 00, Prague 6
fax: +420 22435 9819, tel: +420 22435 9848
balcaji2@fit.cvut.cz

Abstract: Testing of digital circuits seems to be a completely mastered part of the design flow, but Constrained Test Patterns Generation (CTPG) is still a highly evolving branch of digital circuits testing. Our previous research on CTPG proved that we can benefit from an implicit representation of test patterns set. The set of test patterns is implicitly represented as a Boolean formula satisfiability problem in CNF, like in common SAT-based ATPGs. However, the CTPG process can be much more memory or time consuming than common TPG, thus some techniques of speeding up the constrained SAT-based test patterns generation are described and analyzed into detail in this paper. These techniques are experimentally evaluated on a real SAT-based algorithm performing a test compression based on overlapping of test patterns. Experiments are performed on ISCAS'85, '89 and ITC'99 benchmark circuits. Results of the experiments are discussed and recommendations for further development of similar SAT-based tools for CTPG are given.

Keywords: testing, implicit representation, SAT, ATPG, constrained test.

1 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 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 the test application time [2].

The basic idea of general Constrained Test Patterns Generation (CTPG) [3] is shown in Figure 1. For each fault there is a set of test patterns by which it is detected. A test pattern or a set of patterns suitable for compression or other processes is selected from the complete set of test patterns by application of specific additional constraints.

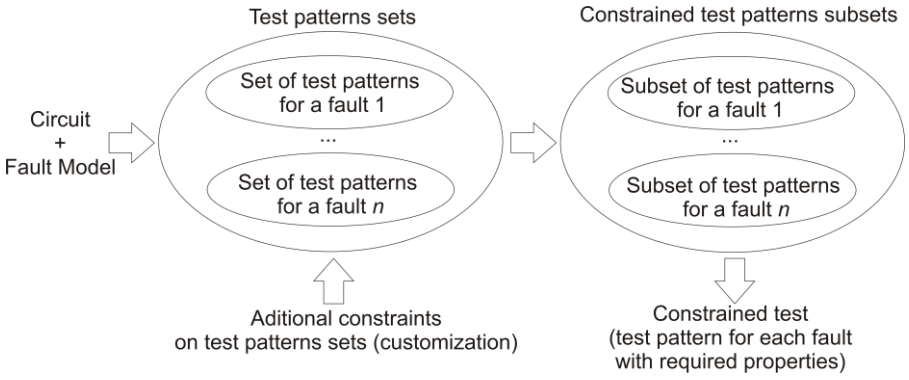

Figure 1. Basic concept of the constrained test generation

Conventional ATPGs (Automatic Test Pattern Generators) are based on PODEM [4] or FAN [5] algorithms. Here the test patterns are generated by traversing the circuit structure, with backtracking employed. In contrast to this approach, SAT-based ATPGs [11], [6] search for the test patterns by SAT (satisfiability) solving. An instance of the Boolean satisfiability problem in CNF (Conjunctive Normal Form) is generated for each fault. This CNF implicitly represents the whole set of test patterns detecting a given fault. A test pattern for a fault is obtained as a satisfiable solution of the CNF formula. A fault is classified as untestable if there is no satisfiable solution.

Our research is focused on a class of algorithms where a set of SAT instances is repeatedly processed with different constraints. This processing of CNFs can cause a significant time overhead [7]. It has been observed, that the majority of SAT instances are classified as UNSAT (unsatisfiable) with given constraints [7, 8] imposed.

Our experiments on CNF processing show the differences in time and memory consumption between

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

during computations, where a large number of related instances is solved.

Stored CNFs are reduced using solution set preserving SAT transformations [8], based e.g., on resolution or propagation of unit (1-literal) clauses [9, 10]. Such transformations do not alter the CNFs, so that any possible SAT solution is lost.

Advantages and disadvantages of the above approaches are discussed over the results and recommendations for further design of SAT-based CTPG algorithms are proposed.

Next, ways of an early detection of unsatisfiable constrained CNFs were explored. UNSAT instances are detected by the inability to resolve conflicts between the required fixed values of variables given by constraints and their values obtained by CNF implications. The CNF generation and SAT solver runs on these unsatisfiable instances can be skipped, which can significantly speed up the algorithm. Such a process is in the paper referred to as *static* or *dynamic fault filters*, based on the employed type of implications.

The proposed techniques of CNF processing and UNSAT filtering are evaluated in the framework of our test patterns compression algorithm *SAT-Compress* [7] which is a dedicated SAT-ATPG [6, 7] algorithm. Therefore, a more detailed description of the SAT-based ATPG and SAT-Compress algorithm is given in Sections 3 and 4.

2 Related Work

Generation of test patterns with some constraints imposed is a common process in digital circuits testing. Test patterns can be constrained for various purposes:

- to be better compressed [2, 7],
- to limit the SAT solver search space,
- to exclude invalid input combinations, etc [1, 11, 12, 13, 14].

One example application of a constrained ATPG is a *broadside transition testing* [12]. A conventional ATPG based on PODEM [4] algorithm produces a set of test patterns with a significant number of 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 (errors caused by these faults 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 forbidden variable assignments that enable detection of the functionally untestable transition faults. These constraints are described by a Boolean formula in CNF. The constrained ATPG fixes variables in the generated test pattern and at the same time it fixes the corresponding variables in the CNF of constraints and checks the CNF for conflicts in the variable assignment. When a conflict occurs, the ATPG backtracks and searches for a different variable assignment in the test pattern. The test set generated this way does not activate functionally untestable transitions, which increases the quality of the test and reduces the yield loss caused by testing of the functionally untestable faults.

In [13], constraining of test patterns to generate them by cellular automata is presented. All test sequences for a fault are checked for conflicts with rule matrices of cellular automata. The entire set of the test sequences for the circuit under test is implicitly represented by a BDD (Binary Decision Diagram) [15]. The BDD is used to select only those sequences which can be reproduced by cellular automata.

An ATPG for industrial circuits with restrictors [14] represents another application of 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 restricted. 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 concept of restrictors (constraints).

Low power tests are mostly built from pre-generated test patterns [1] by their reordering, to decrease the dynamic power consumption (i.e., the switching activity) during the test. However, constraints can also be formed to guide the test patterns generation process, in order to generate low power tests directly [1].

The constrained test patterns generation principle may be efficiently employed to compress test patterns. In [2], test patterns are customized for testing the circuit using the RESPIN (REusing Scan chains for test Pattern decompression) architecture [16]. This architecture is targeted to systems on chip (SoCs). To update values stored in the scan chain of the core under test, scan chains belonging to other cores are used. Test patterns are compressed by overlapping [18]. Suitable test patterns are produced by a conventional ATPG tool performing dynamic compaction [19]. Constraints to the circuit's primary inputs are applied, in order to reach a locally optimum overlap with the vector already present in the scan chain from the previous test cycle.

A similar approach to compression of test patterns for RESPIN is used in the SAT-Compress algorithm [7]. A set of test patterns for each fault is implicitly represented by a SAT instance in the CNF. A test pattern for each fault is obtained by SAT-problem solving of CNF instances with constraints applied (i.e., fixed variables values), which again represent the patterns already present in the scan chain. The main difference from the previously mentioned approach [2] is the implicit representation of the test patterns set. SAT-based ATPGs are known to be much faster in test patterns generation for hard to be tested faults than structural ATPG tools, however easily testable faults can cause an unnecessary time overhead [6]. Similar behavior can be observed in the CTPG process, but Boolean constraints propagation (BCP) [9, 10] in the SAT-based approach can quickly recognize that a test pattern with given constraints does not exist, thus accelerate the algorithm.

The latest approach to transition delay faults (TDF) [20] test compression is based on a constrained SAT solving, too [21]. TDF can be detected by a pair of test patterns applied in two subsequent clock cycles. Test compression is performed by test patterns overlapping as in the SAT-Compress algorithm, but pairs of test patterns are overlapped instead.

The SAT-based CTPG algorithm can also be enhanced by techniques used in common SAT-based ATPGs. These techniques can be divided into two groups. The first group of techniques deals with SAT solver acceleration by, e.g., variable ordering for the SAT solver heuristic [22], circuit-based dynamic learning [23] or different clause learning techniques [24]. The second group of techniques deals with the reduction of the time overhead caused by the CNF generation e.g. dynamic clause activation [23].

3 SAT-Based Test Patterns Generation

Recent research on test patterns generation for digital circuits proved SAT-based techniques to be very efficient even for large industrial circuits [6], [22], [23].

3.1 Circuit-to-CNF Transformation

In SAT-based ATPGs, the ATPG problem is reduced to the SAT problem. Here the fault-free and faulty circuits are combined and transformed into one CNF [6, 11], to obtain a SAT problem instance. Satisfiable assignments of variables of this CNF are test vectors detecting the respective 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 exists an assignment of variables, for which all clauses are satisfied.

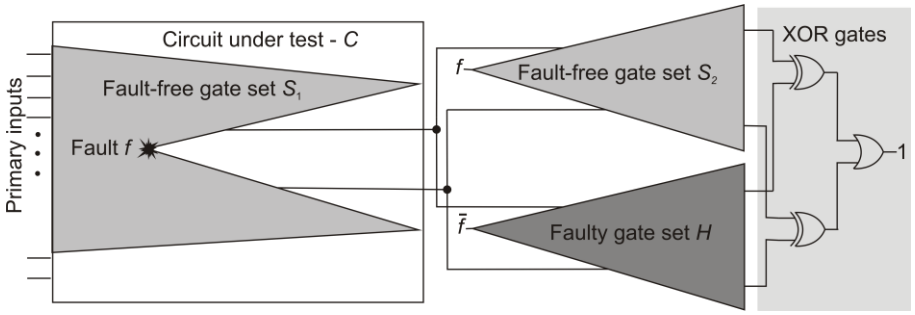

Figure 2. SAT instance generation for an ATPG

A Boolean variable is assigned to each signal in the circuit. Each gate in the subcircuit S_1, which corresponds to the output cone of the fault to primary outputs (POs) of the circuit is copied and forms the set of gates H (faulty part of the circuit). All gates in the cone from the primary inputs (PIs) to POs where the fault is observable are 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 $g_{s1},\ldots,g_{sn} \in S$:

$$\Phi_c = \prod \Phi_{gsi} \qquad 1 \leq i \leq |S|$$

To generate a test for a fault F, the characteristic function Φ_f of the faulty circuit is generated as a conjunction of all CNFs of gates $g_{h1},\ldots,g_{hn} \in H$:

$$\Phi f = \prod \Phi_{ghi} \qquad 1 \leq i \leq |H|$$

Outputs of the fault-free and faulty circuit are coupled by XOR gates whose outputs are further connected to an OR gate. The function Φ_{xor} is generated as a conjunction of characteristic functions of these gates. The SAT instance for a fault F is obtained as a product of Φ_c, Φ_f and Φ_{xor}:

$$\Phi_{test\_F} = \Phi_c \cdot \Phi_f \cdot \Phi_{xor}$$

Finally, unit clauses are added to inject the faulty value in Φ_f, to set its image in Φ_c as a complement of the faulty value, and to set the output of the OR gate in Φ_{xor} to 1 (the Boolean difference must result in 1, to be the fault F detectable).

The satisfying solutions of Φ_{test_F} represent 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, 11].

3.2 SAT-based ATPG Algorithm

A conventional SAT-based ATPG algorithm [11] can be described in four steps:

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

For details on the SAT-based ATPGs, see [6, 11].

4 The SAT-Compress Algorithm Principles

We demonstrate the proposed CNF processing and UNSAT filtering techniques on our SAT-Compress algorithm [7]. Therefore, we describe it in following sections.

The compression of the test patterns is performed by their overlapping [18]. State-of-the-art tools such as COMPAS [25] try to overlap test patterns *pre-generated* by a conventional ATPG. The compression efficiency strictly depends on properties of these patterns, particularly on the number of unspecified (don't care) bits in the test. The SAT-Compress algorithm tries to eliminate this drawback. The main idea is to generate the most suitable test patterns on the fly, to reach a locally optimum overlap. Each fault has its set of test patterns it detects. If the algorithm is able to pick the right pattern for each fault in the right order, it will reach the best possible compression of the test patterns.

It is obvious that explicit enumeration and storing of all test patterns for all faults in the circuit is inefficient (and mostly even infeasible), thus SAT-Compress takes advantage of principles of SAT-based ATPGs and efficiently represents all test patterns for one fault *implicitly*, by one SAT problem instance in a CNF.

The decompression of compressed test patterns is performed by the RESPIN [16] architecture. RESPIN is based on the IEEE P1500 standard for testing of embedded cores [26]. Each core is provided with a wrapper which forms an interface between the embedded core (core terminals) and its system on chip environment (the rest of the integrated circuit and the test access mechanism). The core wrapper defines the normal (non-test) mode and core test modes. The hardware overhead of the RESPIN architecture is minimal, since the decompression is

made by *reusing* of the scan chains (SCs) of non-tested cores and no additional decompression hardware is required. The SC of the non-tested core preserves useful bits of the test patterns by looping through the feedback, while the SC of the core under test (CUT) obtains responses of previous test patterns and shifts them to the signature analyzer, to be evaluated. The compressed bitstream is serially shifted into the SC of the non-tested core in each loop. At the same time a new test pattern is shifted from the non-tested SC into the SC of the CUT.

4.1 The SAT-Compress Algorithm Description

The SAT-Compress tries to find the best overlap of test patterns by gradually building the compressed test bitstream. Each generated test pattern imposes constraints on the subsequent test pattern to be generated. Only care bits generate the PI constraints. The basic algorithm is shown in Figure 4.

1. Generate a complete fault list (FL) for the given circuit.
2. For each fault in FL, generate and solve its CNF. If the fault is undetectable, remove it from FL. Therefore, there cannot be any aborted faults in the subsequent ATPG process.
3. Choose a zero state (all-zero test pattern T_0) or a test pattern covering any fault from FL is used as the initial test pattern. Set the selected initial test pattern as the current pattern.
4. Simulate the current pattern and remove all detected faults from FL.
5. Insert the leftmost bit of the test pattern into the resulting bitstream (Figure 3).
6. Shift the current pattern one bit left and put a DC bit to its rightmost position. The resulting pattern is used as the set of constraints for the next pattern (Figure 3).
7. Repeat the following until FL is empty (all faults are covered by test patterns):
    a. Find a fault that can be tested under current constraints as follows. For each fault in FL, generate the corresponding CNF, constrain it by the current pattern and solve it. If the CNF is solvable under the given constraints, stop the search.
    b. If the previous search has been successful, simulate the pattern resulting from the search, remove all detected faults from FL and use the pattern as the new current pattern.
    c. Insert the leftmost bit of the current test pattern into the resulting bitstream.
    d. Shift the current pattern one bit left and put a DC bit to its rightmost position.
8. Output the entire current pattern into the bitstream**.**

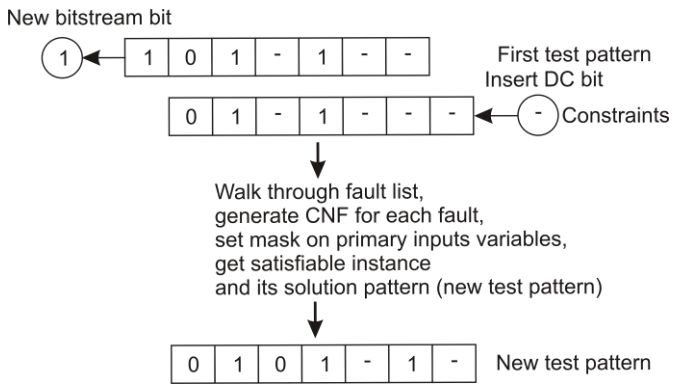

Figure 3. Next pattern generation example

```
FL.generate(circuit);                          // FL … fault list
FL.remove_untestable_faults();
TP = T0;                                       // TP … current test pattern
                                               // pick an initial pattern
FL = FL - circuit.fault_simulate(TP);          // remove faults detected by TP
output(TP[0]);                                 // put the leftmost bit to output
TP.DC_shift_left();                            // the rightmost TP bit becomes DC
while (!FL.empty()) {                          // loop until all faults detected
    for each F in FL {                         // find a fault which is detectable
                                               // under current constraints
```

```
        CNF.generate(circuit, F);
        CNF.constraint(TP);                    // use shifted pattern as constraints
        Y = CNF.solve();
        if (Y.exists()) break;                 // find the first testable fault
    }
    if (Y.exists()) {                          // successful search
        TP = Y;                                // new test pattern
       FL = FL - circuit.fault_simulate(TP);   // remove faults detected by TP
    }
      output (TP[0]);                          // put the leftmost bit to output
    TP.DC_shift_left();
}
output (TP);                                   // output all remaining bits
```

Figure 4. The SAT-Compress algorithm

The `DC_shift_left()` method shifts a bit vector one place to the left and fills the rightmost bit with the don't care value. `Output()` appends one or more bits to the output bitstream.

4.2 Experimental Evaluation of the SAT-Compress Algorithm

A comparison of SAT-Compress with other state-of-the-art test compression techniques is presented in Table 1. The first column "Circ." represents the name of the benchmark circuit. A comparison for only the seven biggest ISCAS'89 circuits [36] is shown, since no more relevant data on the other methods were available to us. The compressed test lengths in bits, for nine different competitive methods, are shown next. The last column shows the compressed test data size in bits for the SAT-Compress compression tool. An all-zero initial test pattern for both COMPAS and SAT-Compress is used, thus the results are not influenced by different initial states.

It can be concluded from Table 1, that the SAT-Compress algorithm can reach similar and often even better compression of test patterns than most of the presented state-of-the-art compression methods. The COMPAS algorithm reaches similar compression ratio as SAT-Compress, but its efficiency is influenced by the number of DCs in pre-generated test patterns, which is not the case of SAT-Compress. Thus, SAT-Compress can theoretically reach better results, if a more efficient fault-processing heuristic was used. Currently, the faults are processed in a greedy, first-only way. This will be the topic of our further research.

The time consumed by the presented tools could not be compared because neither their source codes, nor executables were available. Nevertheless, the time-consumption of the SAT-Compress algorithm is shown in the following sections which discuss techniques of its acceleration.

Circ.	MinTest [27]	Stat. Coding [28]	LFSR Reseeding [29]	Illinois Scan [30]	FDR Codes [31, 32]	EDT [33]	RESPIN++ [17]	COMPAS [25,41]	SAT-Compress
s5378	20,758	15,417	6,180	14,572	12,346	-	17,332	2,148	2,407
s9234	25,935	19,912	12,112	27,111	22,152	-	17,198	11,594	9,928
s13207	163,100	52,741	11,285	109,772	30,880	10,585	26,004	3,819	10,457
s15850	58,656	49,163	12,438	32,758	26,000	9,805	32,226	6,930	12,987
s35932	21,156	-	-	-	22,744	-	-	1,860	5,096
s38417	113,152	172,216	34,767	96,269	93,466	31,458	89,132	19,597	19,291
s38584	161,040	128,046	29,397	96,056	77,812	18,568	63,232	5,778	14,271

Table 1. Comparison of the test data amount for different compression techniques

Further experiments performed on ISCAS'85 [35] and '89 [36] benchmark circuits showed that the SAT-Compress algorithm can reach the compression ratio 86% on average (compared with a compacted test generated by a conventional ATPG [37]) [7].

Figure 5 shows an example of distribution of bitstream lengths obtained using different starting initial test patterns and permutations of PIs. Every starting pattern has been chosen as a test pattern covering one particular fault. Thus, the number of runs is equal to the number of faults. The permutations of the PIs in the test pattern (corresponding cells in the scan-chain) have been evaluated for 1,000 random permutations of PIs. An all-zero seed has been used as the starting test pattern for all PIs permutations.

As can be seen, the compressed bitstream length depends both on the starting test pattern (initial seed) and the permutation of PIs. Similar behavior of the SAT-Compress algorithm can be observed for all tested benchmarks. It indicates that the selection of the initial test pattern and the order of PIs have a crucial impact on the resulting compressed test length.

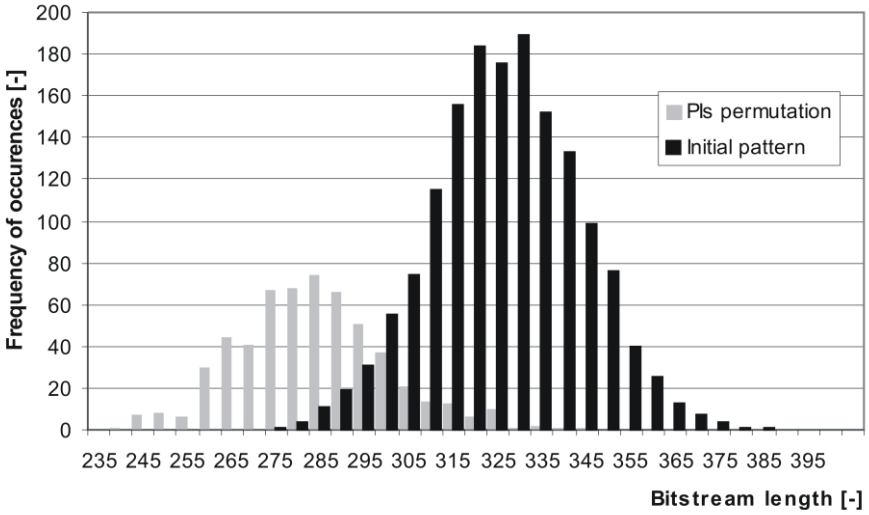

Figure 5. Frequency of bitstream length distribution (c1355)

5 Techniques of CTPG Acceleration

The SAT-Compress has been chosen to show properties of proposed acceleration techniques as a representative of SAT-based algorithms for CTPG. This algorithm and similar algorithms [21] deal with repeated processing of the same SAT instances in CNF with different constraints. The processed CNFs are often unsatisfiable with given constraints. As they are repeatedly generated and solved with different constraints, the CTPG process becomes time-consuming [7, 21]. Thus, the CTPG process itself and possible techniques of its acceleration have been analyzed. We discuss the CNFs manipulation and their filtering based on satisfiability, analyze the efficiency of the proposed techniques and evaluate their usability in SAT-based CTPG algorithms.

All measurements were performed on a CPU Intel Core 2 Duo – 1,8GHz with 1GB RAM. MiniSat v1.14 [34] has been used as the SAT solver. Experiments have been performed on a subset of smaller ISCAS'85 [35], '89 [36] and ITC'99 [41] benchmark circuits, because the memory requirements for CNFs storing were unfeasible for bigger circuits.

5.1 On-the-fly vs. Storing

Generating SAT instances in CNF takes on average 80% of test generation time in SAT-based ATPGs [6, 11, 38, 39] and can also cause a significant time overhead in the CTPG process. When used repeatedly, the SAT instances can be generated on-the-fly, or they can be pre-generated and stored in memory. Either original CNFs are stored, or the CNFs are simplified by solution set preserving reductions [8], to reduce memory requirements.

Under common fault models, a fault causes the value of a signal to differ between the faulty and fault-free copy of the circuit in the SAT ATPG process. The values are determined by the fault model. It suffices to inject them into the corresponding CNFs using unit clauses, however, this opens up a way to simplify the CNFs. Specifically, values of other variables common to all solutions can be discovered.

Repeated application of unit clause elimination identifies most of constant variables. The rest of them are detected by fixing the variable to a constant value and solving the SAT problem, see [40]. The number of clauses can be reduced by removing duplicities, clause absorption, and by creating resolution terms [8]. All these reductions preserve all SAT solutions.

Our experimental results show that on the average 60% of variables and 65% of clauses can be removed by solution-set-preserving reductions [8]. Thus it can be possible to store all CNF instances or their subset in memory, decrease the number of repeatedly generated CNF instances, and accelerate the CTPG process.

In the on-the-fly approach, the CNFs are repeatedly generated while they are differently 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 significantly [7].

In the first 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 (unconstrained) CNFs must be repeatedly loaded into the SAT solver.

Loading of the CNFs into the 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 the described solution set preserving SAT reductions [8, 9].

5.1.1 Experimental Evaluation of CNFs Processing Techniques

A comparison of the three techniques of CNFs processing is presented in Table 2. The first column of the table "*Circ. name*" represents the name of the benchmark circuit from ISCAS'85 [35] or '89 [36]. Differences between processing of the CNFs on-the-fly and storing of non-reduced/reduced CNFs are shown in the three columns. The "*CNF*" columns indicate the time spent by a CNF generation or loading of stored CNFs from the memory. The "*SAT*" columns represent the time spent by a SAT solving of the processed CNFs. Columns "*SIM*" show the time spent by simulation and columns "*SUM*" indicate the total time consumptions of the algorithm. For methods, where storing of CNFs was employed, the stored CNF literals count ("*Lit. count*") and the time spent for the CNFs generation eventual reduction and storing is shown ("*Store*"). The last row of the table ("*Avg*.") represents average values of all columns.

Circ. 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
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 2. Experimental results for the processing of CNFs

First, let us focus on the time spent by CNFs generation. Experimental measurements show that loading of the stored CNF instances is in all cases faster than their generation on-the-fly. In the case of the benchmark circuit c3540, the time of CNFs generation is 2205 seconds, whereas 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 CNFs storing seems to be negligible in comparison with CNFs generation and SAT solving. Similar behavior is observed for storing of the reduced CNFs on majority of processed benchmark circuits. However, in some cases, the CNFs reduction increases the time consumption of CNFs storing, thus it takes comparable time with the SAT solving. For example, the time of the reductions and storing of the CNFs for the benchmark circuit s1494 is 26.14 seconds, while solving of these CNFs takes 35.54 seconds.

On the other hand, the number of stored literals grows linearly with the size of the circuit (number of gates). For example, the benchmark circuit c3540 consists of 1648 gates and its fault list has 3428 faults. It means that 3428 CNFs must be stored, which is 31,439,618 literals (after reduction). It is obvious that storing the CNFs is unfeasible for large circuits, because of memory consumption. CNF reduction does not improve the situation, because the reduction of the CNFs size is not as significant as we hoped for [8].

The average values confirm the previous observations. The time consumption of the CNFs processing can be dramatically decreased by storing the reduced CNFs in memory. The average total time for the CNFs processing indicates, that processing of the stored CNFs is on 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 the processing time of the CNFs, because the solution set preserving reductions are time consuming for bigger instances.

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 to choose, because it is not limited by the high memory consumption.

5.2 Fault filtering

Constrained test patterns generation algorithms must solve a great number of constrained SAT instances repeatedly. As mentioned above, it has been observed, that the majority of these instances are unsatisfiable 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 faults which lead to such UNSAT instances, in order to accelerate the constrained test patterns generation.

To detect a fault, a set of signals must be set to required values. For a stuck-at fault, it is a single value of a single signal. Given some constraints as a set of fixed values for the primary inputs of the circuit, we can propagate these constant values to other signals in the circuit. If the signal values required to detect a fault conflict with values implied by the input constants, that fault cannot be *excited* and hence tested.

The advantage of such filtering technique is that constant propagation is done once for a set of constraints, and used repeatedly for all faults. In the circuit domain, it is equivalent to Boolean Constraint Propagation [10], [43] in the SAT domain.

Two fault filtering algorithms are described and experimentally evaluated in following subsections. 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.

5.2.1 Static Fault Filtering

The 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., the clause $c \vee e$ corresponds to implications $\neg c \Rightarrow e$ and $\neg e \Rightarrow c$ (see example in Figure 6). When a constraint is applied, a 2-literal clause may become unit clause, and then the implications serve as an efficient way for constraint propagation [9, 11, 43].

The static fault filter uses a data structure called *implication table* to store implications. The key of the table is a signal identification and polarity; the value is a list of signals and their polarities to be set.

The table is constructed once for a given circuit (hence the terms *static filtering* and *static implication*). For each gate in the circuit (Figure 6a), any pair of signals which would form a 2-literal clause in the description of the gate (Figure 6b) is selected and the corresponding two implications are entered into the table (Figure 6c).

When any signal is fixed to a constant value, the implication table is used to fix implied values of other signals (Figure 6d). This is done repeatedly until there are no more signal values to fix.

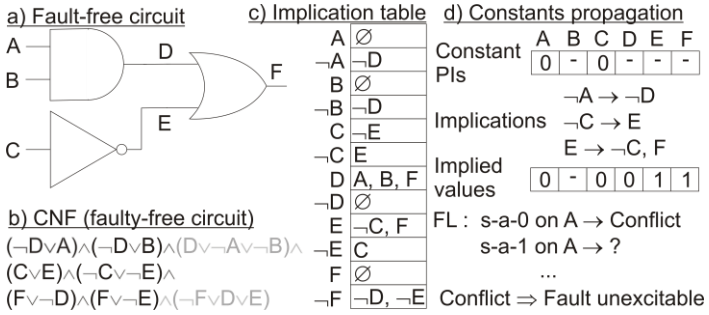

Figure 6. Example of the static implication filter

Constants propagation cannot produce conflicting values for any signal, because initially only the values of primary inputs are fixed and the circuit is assumed to be correct (e.g. any signal driven by one gate).

The implied fixed signal values are used in the way outlined above. In Figure 6, we have constant primary inputs A=0 and C=0. To detect a Stuck-At-0 on D, we need to set D=1. As can be observed in Figure 6d, this conflicts with the implied value D=0, and hence Stuck-At-0 on D cannot be detected under current constraints and shall be filtered out. On the other hand, Stuck-At-1 on D is compatible and can be processed further.

The static implication filter is a simple method to check faults for excitability. Its efficiency depends on the number of fixed input signal values and the structure of implications. A high number of applicable implications rules gives us a solid ground for fixing internal signal values and increases our chances to discard a fault before its CNF is generated and proved unsatisfiable. Static fault filtering takes a negligible part of the overall running time, so it can cause a significant acceleration of the algorithm.

5.2.2 Dynamic Fault Filtering

Static filtering used only implications between pairs of signals, which would be modeled by 2-literal clauses in the corresponding CNF. Not only a 2-literal clause in a CNF may become a unit clause under given constraints, but, with a sufficient number of constraints, a clause with any number of literals may become so. 4-clauses are relatively rare, but clauses with 3 literals do occur during SAT ATPG [8, 21]. To catch faults whose unexcitability follows from 3-literal clauses is the task of the dynamic filter.

Unlike static filter, dynamic filter cannot rely on a precomputed data structure (hence the term *dynamic*). Instead, it scans the circuit for each set of fixed signal values. Assuming that static filtering has been already done, it searches for triples of signals, which would form a 3-literal clause in the CNF description of a gate. If the values of two signals from the triple are already fixed, it propagates them and fixes the third. This is again done repeatedly until there are no more signal values to fix.

Let us assume constants A=1, B=1 in the circuit in Figure 6, Static implications do not bring any new fixed value (Figure 7a). The AND gate at the inputs A and B, which would be described by the clause D∨¬A∨¬B, allows the dynamic filter to also set D=1 (Figure 7b),

It is obvious that further signal values can be fixed. A higher number of fixed values increases the chances to find conflicts and to identify more unexcitable faults than the static filter. However, the time consumption of the dynamic filter is much higher than of the static filter, because the circuit must be searched for new implications for each constant value applied.

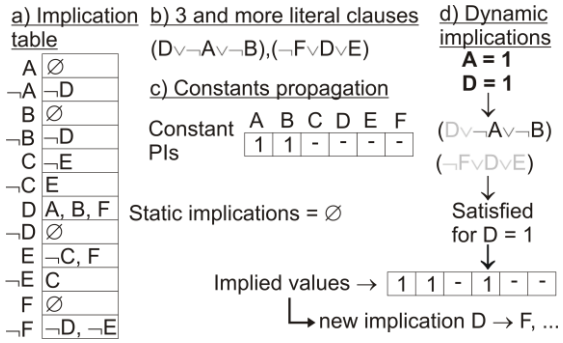

Figure 7. Example of the dynamic implication filter

The pseudocode of the SAT Compress algorithm equipped with a static and a dynamic filter is outlined in Figure 8. The effort spent in constant propagation is not wasted even in the case of excitable faults; the constants are used as additional constraints to the CNF. Technically, the 2-literal clauses contained in the implication table are not even generated to the CNF.

```
FL.generate(circuit);                          // FL … fault list
FL.remove_untestable_faults();
TP = T0;                                       // TP … current test pattern
                                               // pick an initial pattern
FL = FL - circuit.fault_simulate (TP);         // remove faults detected by TP
IMTAB.generate (circuit);                      // create implication table
output (TP[0]};                                // put the leftmost bit to output
TP.DC_shift_left();                            // the rightmost TP bit becomes DC
while (!FL.empty()) {                          // loop until all faults detected
    IM = TP;                                   // implied signal values
```

```
    IM.static_implications (IMTAB);           // fast static constant propagation
    IM.dynamic_implications (circuit);        // more thorough const propagation
    for each F in FL {                        // find a fault which is detectable
                                              // under current constraints
        if (F.conflict (IM)) {                // not excitable with current PIs
             continue;                        // skip (filter) unexcitable fault
        }
        CNF.generate (circuit, F);
        CNF.constraint (IM);                  // use propagate consts as constraints
        Y = CNF.solve();
        if (Y.exists()) break;                // find first testable fault
    }
    if (Y.exists()) {                         // successful search
        TP = Y;
        FL = FL - circuit.fault_simulate (TP);
    }
    output (TP[0]};
    TP.DC_shift_left();
}
output (TP);                                  // output all remaining bits
```

Figure 8. The SAT-Compress algorithm

5.2.3 Experimental Evaluation of Fault Filters

A comparison of filtering techniques is presented in Table 3. The first column of the table "Circ. name" represents the name of the benchmark circuit from ISCAS'85 [35], '89 [36] or ITC'99 [41]. Differences between the basic algorithm (SAT-Compress [7]) and its modification with static and dynamic filtering are shown in the three columns. The "*Gen.*" column represents the total number of generated CNFs and "*Used*" shows the total number of satisfiable CNFs. For algorithms, where static and dynamic filter was employed, the percentage reduction ("*Red.*") of the number of processed CNFs referred to the basic algorithm and the time spent by filtering ("*Filter*") of the unexcitale faults is shown. The last column "*SUM*" has the same meaning as in Table 2. The last row of the table "*Avg*." represents an average value of the column.

Circ. name	Basic algorithm			Basic algorithm + static filter			Basic algorithm + static + dynamic filter		
	Gen.	*Used*	*SUM*	*Red*	*Filter*	*SUM*	*Red*	*Filter*	*SUM*
	[-]	*[-]*	*[s]*	*[%]*	*[s]*	*[s]*	*[%]*	*[s]*	*[s]*
c432	3517	74	3.63	25	0	2.61	64.5	0.3	1.65
c499	3210	73	2.91	9.1	0.02	2.47	49.6	0.94	2.26
c880	70395	181	137.5	46	0.19	82.99	54.6	5.22	79.12
c1355	13236	95	30.54	32	0.02	20.29	70.4	3.22	12.35
c1908	35443	162	126	19	0.13	103.93	47.3	8.81	78.01
c2670	277808	355	1188.98	33	1.45	772.36	52	58.2	624.5
c3540	717888	347	5784	53	4.14	2793.14	62.9	254	2472
c5315	26978	289	198.9	15	0.17	171.17	55.4	88.2	180.2
s298	2782	93	0.93	47	0	0.53	47.8	0.03	0.62
s382	1959	59	1	46	0	0.55	49	0.08	0.61
s400	4088	69	2.06	54	0	0.96	54.9	0.14	1.11
s420	17210	93	12.8	39	0.08	8.08	45	0.58	7.95
s444	2359	59	1.24	58	0	0.53	62.8	0.09	0.6
s510	2899	76	2.08	58	0.02	0.95	59.1	0.16	1.13
s526	15563	134	8.56	49	0.02	4.41	49.8	0.42	5.15
s526n	13901	134	7.71	48	0.06	4.06	49.6	0.38	4.27
s641	20397	136	29.32	37	0.02	17.79	50.2	1.08	16.07
s713	17928	130	26.9	35	0.02	17.29	46.2	1.59	16.31
s820	51351	206	44.25	38	0.13	27.73	38.6	1.09	28.08
s832	56590	203	50.02	39	0.13	30.65	39.5	0.86	30.98
s838	114070	187	173.7	40	0.13	107.69	43.6	7.55	115.57
s953	63076	198	81.75	59	0.06	33.82	67.7	3.73	30.9
s1196	180005	249	339.9	48	0.41	178.41	59.5	18.3	156.3

Circ. name	Basic algorithm			Basic algorithm + static filter			Basic algorithm + static + dynamic filter		
	Gen.	*Used*	*SUM*	*Red*	*Filter*	*SUM*	*Red*	*Filter*	*SUM*
	[-]	*[-]*	*[s]*	*[%]*	*[s]*	*[s]*	*[%]*	*[s]*	*[s]*
s1238	213134	247	398.4	49	0.55	206.45	56.3	20.4	192.4
s1423	44892	149	113.05	52	0.17	56.91	59.8	9.41	56.4
s1488	19053	204	32.6	48	0.08	17.4	50.5	4.27	20.6
s1494	22878	201	39.03	51	0.08	19.39	54	4.31	22.79
s5378	378447	502	3216	50	4.25	1649.25	63.3	274	1495.9
s9234	4654444	749	82176.1	42	134	48336	52.6	4014.7	44311.4
s13207	10733919	1109	278511	60	228.4	110828.4	64.1	14130	115520.7
s15850	11160862	980	371963	52	315.1	188644.1	59.5	25019	184213
s35932	2000941	1419	137704	17	402.4	119823.4	20.7	10319	129133.5
s38584	11131264	2256	516020	62	591.2	201443.2	68.3	138870	307178
b03_C	604	48	0.27	42.2	0	0.2	42.5	0.063	0.234
b04_C	265730	198	667	22.6	1.2	519	39.7	25.16	435.5
b05_C	104484	171	319	24.5	0.64	253	41.7	24.39	214.7
b06_C	63	28	0.02	19	0	0.02	22.2	0	0.016
b07_C	36186	116	51.3	49.9	0.13	25.4	65.6	2.813	21.48
b08_C	19145	77	12.4	41.1	0.03	7.5	64.1	0.531	5.172
b09_C	3367	54	2.22	53.8	0.02	1.08	56	0.188	1.172
b10_C	3410	90	1.83	48.6	0.03	1.02	48.9	0.266	1.156
b11_C	56671	143	128	32.8	0.19	87.3	49.9	8.547	64.47
b12_C	157888	337	448	53.1	0.64	222	61.9	34.77	222
b13_C	3661	81	3.53	51.3	0	1.84	53.1	0.422	2.266
Avg.	928778	278.15	30436.12	42	36.65	14707.1	52.4	4200.37	17108.2

Table 3. Experimental results for the UNSAT filtering

Experimental results show that fault filtering can accelerate the process of the constrained test patterns generation more than 2-times. For example, the total test patterns compression time of the basic algorithm for the benchmark circuit c3540 is 5,784 seconds, while with the static filer it takes 2,793 seconds and with the dynamic filter the total test patterns compression time decreased to 2,472 seconds. The static filter, as a simple fast technique of detecting of the unsatisfiable instances, is highly effective and saves 43% of all the processed unsatisfiable instances on average. Moreover, the dynamic filter is able to detect and save additional 10% of the unsatisfiable CNF instances on average, but it is much more time-consuming than the static filter.

Circuit name		c3540	s13207	s15850	s35932	s38584
Basic algorithm	*Gen. [-]*	717888	10733919	11160862	2000941	11131264
	Used. [-]	347	1109	980	1419	2256
	CNF [s]	2205	25209	36446	5529.6	19658
	SAT [s]	3176	228267	304751	131884	478943
	SIM [s]	403	25035	30766	290.4	17419
	SUM [s]	5784	278511	371963	137704	516020
Basic algorithm +static	*Red. [%]*	53	60	52	17	62
	Filter [s]	4.41	228.4	315.1	402.4	591.2
	CNF [s]	1041	9869	17406	5154	7392
	SAT [s]	1352	76220	140654	113974	175764
	SIM [s]	396	24511	30269	293	17696
	SUM [s]	2793.41	110828.4	188644.1	119823.4	201443.2
Basic algorithm + static + dynamic	*Red. [%]*	62.9	64.1	59.5	20.7	68.3
	Filter [s]	254	14130	25019	10319	138870
	CNF [s]	829	9048.7	14212	4935	6265
	SAT [s]	993	67857	114844	113585	144431
	SIM [s]	396	24485	30138	294.5	17612
	SUM [s]	2472	115520.7	184213	129133.5	307178

Table 4. Detailed results for the UNSAT filtering

More detailed comparison of filtering techniques for five benchmark circuits is presented in Table 4. The rows in the Table 4 have the same meaning as the corresponding columns in Table 2 and Table 3. This table shows

detailed distribution of the CTPG time over all steps of the SAT-Compress algorithm. The results for presented circuits except of the s35932 show that the filters can decrease the time of CNFs generation and their solving in half.

Next, properties of our static and dynamic fault filters have been analyzed for the SAT-Compress algorithm. First, we have measured the number of constraints (fixed PI values) set during the CTPG process. Figure 9 shows an example of constant PIs for the ISCAS'89 benchmark circuit s13207. This circuit has 700 PIs and our measurement shows that on average 497 of them are fixed during the compression. Figure 10 shows the number of implied signal values for the same benchmark circuit (s13207). Each constant PI produces on average 9.3 implied signal values The dependence between the number of constant PIs and the number implied signal values appears to be linear. Similar behavior has been observed in all measured circuits.

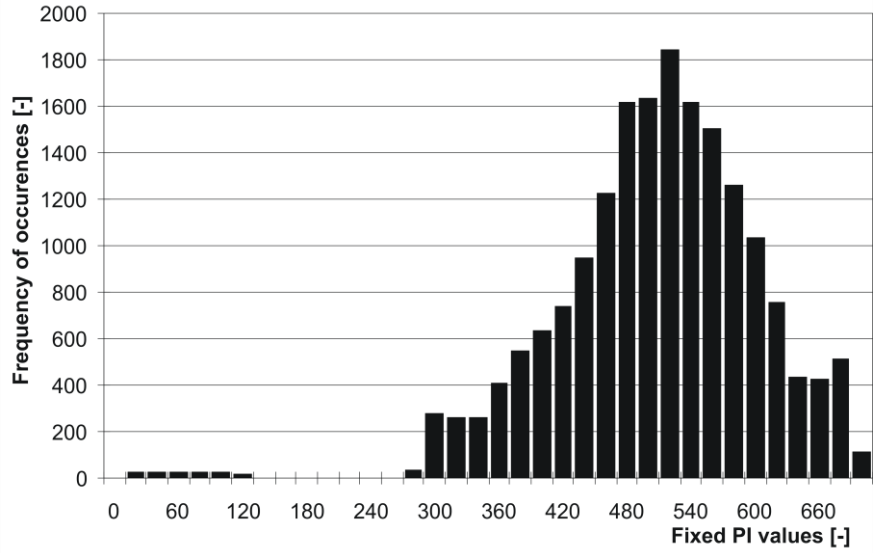

Figure 9. Frequency of fixed PI values during CTPG (s13207)

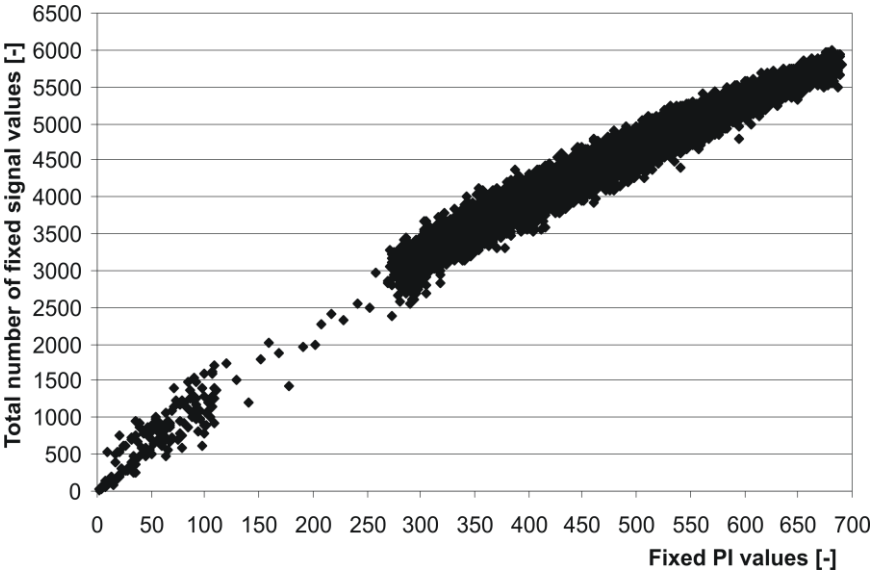

Figure 10.The number of values fixed by constraints implications (s13207)

Finally, we have compared the static and dynamic filters by the number of implied signal values. The example for ISCAS'89 benchmark circuit s13027 is shown in Figure 11. It seems that the distribution of the number of implied values is similar for both static and dynamic filter. The only difference is their offset. The results from Table 3 show that dynamic filter detects on average 10% more UNSAT CNF instances in comparison with static filter. These observations confirm the assumption that more fixed signal values can detect more conflicts and increase the efficiency of the filter. Similar behavior has been observed in all measured circuits.

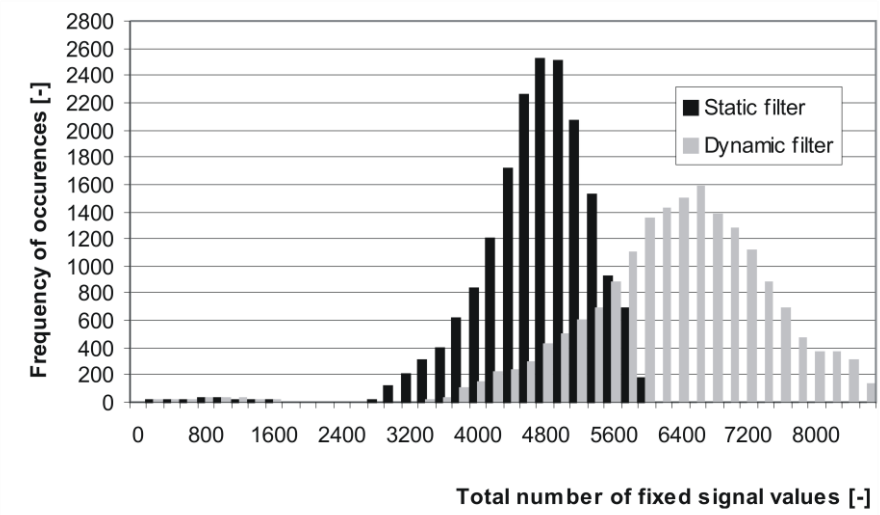

Figure 11. Frequency of values fixed by implications during CTPG (s13207)

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 generated and solved. The dynamic filter is better for small circuits, because searching for dynamic implications is much more time-consuming.

6 Conclusions

The SAT-based constrained test patterns generation problem has been stated to target a class of algorithms which should benefit proposed techniques. Some of these algorithms have been discussed.

The SAT-Compress algorithm has been chosen as a representative of CTPG algorithms. It is shown that implicit representation of the test set can be beneficial in CTPG and even a simple greedy algorithm as the SAT-Compress can reach results competitive with the state-of-the-art tools. However, it also implies that the time-consumption can grow significantly with the size of the circuit (number of gates). Thus some techniques of CTPG acceleration are proposed.

First, 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. The CNF storing seems to be beneficial for large circuits, but the size of stored CNFs grows considerably with a size of the circuit even for reduced CNFs. This observation implies that storing of the CNFs brings no significant improvement. Thus, the CNFs processing on-the-fly is still the best technique in a general case.

The second part of our research deals with the filtering of the unexcitable faults. Their CNFs would cause a significant time overhead in the CTPG process. We proposed two techniques of filtration based on the static and dynamic implications. Their properties have been shown on SAT-Compress algorithm. Our experimental evaluation proved that these simple techniques can save more than 50% of unsatisfiable instances and decrease the time consumption significantly.

Acknowledgement

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] P. Girard, N. Nicolici, X. Wen, Power-Aware Testing and Test Strategies for Low Power Devices, Publisher Springer Netherlands, ISBN: 1441909273, 2009, p. 353.

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

[3] J. Balcarek, Implicit Rrepresentations in Customized Testing of Digital Circuits, Proc. of Počítačové architektury&diagnostika, Češkovice (ČR), 2010, pp. 15-20.

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

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

[7] J. Balcarek, P. Fiser, J. Schmidt, Test Patterns Compression Technique Based on a Dedicated SAT-based ATPG, Proc. of 13th Euromicro Conference on Digital Systems Design (DSD'10), Lille (France), 2010, pp. 805-808.

[8] J. Balcarek, P. Fiser, J. Schmidt, 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), 2009, pp. 3-10.

[9] H. Zhang, M. Stickel, An efficient algorithm for unit propagation, in Proc. of the 4th International Symposium on Artificial Intelligence and Mathematics, 1996.

[10] S. Malik et al. et al. Efficient conflict driven learning in a boolean satisfiability solver.; In IEEE/ACM International Conference on Computer-Aided Design (San Jose, California), 2001, URL , ISBN 0-7803-7249-2, s. 279-285.

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

[12] X. Liu, M. S. Hsiao, Constrained ATPG for Broadside Transition Testing, in Proc. of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), 2003, pp.175.

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

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

[15] S. B. Akers, Binary Decision Diagrams, IEEE Transactions on Computers, C-27(6), 1978, pp. 509–516.

[16] R. Dorsch and H.-J. Wunderlich. Reusing scan chains for test pattern decompression. In Proceedings of the IEEE European Test Workshop (ETW), pages 124–132, Stockholm, Sweden, May 2001. IEEE Computer Society Press.

[17] L. Schafer , R. Dorsch, H. J. Wunderlich, RESPIN++- Deterministic Embedded Test, Proc. of the European Test Workshop, 2002, pp.37-42.

[18] W. Daehn, J. Mucha, Hardware Test Pattern Generation for Built-in Testing, Proc. of the IEEE Test Conference, 1981, pp. 110-113.

[19] B. Ayari, B. Kaminska, A new dynamic test vector compaction for automatic test pattern generation, IEEE Trans. on CAD of Integrated Circuits and Systems, 1994, pp.353-358.

[20] M. Balaz, R. Dobai, E. Gramatova, Delay faults testing, in Proc. of the Design and Test Technology for Dependable Systems-on-Chip, R. Ubar, J. Raik, and H. T. Vierhaus, Eds. Information Science Publishing, 2011, doi: 10.4018/978-1-60960-212-3.

[21] R. Dobai, M. Balaz, SAT-Based Generation of Compressed Skewed-Load Tests for Transition Delay Faults, in Proc. of the 14th Euromicro Conference on the Digital System Design (DSD), Oulu (Finland), 2011, pp. 191-196.

[22] D. Tille, S. Eggersglüß, H.M. Le, R. Drechsler; Structural heuristics for SAT-based ATPG; 17th IFIP International Conference on Very Large Scale Integration, VLSI-SoC 2009; Florianopolis; 12 October 2009 through 14 October 2009.

[23] S. Eggersglüß, R. Drechsler; Robust algorithms for high quality Test Pattern Generation using Boolean Satisfiability; 41st International Test Conference, ITC 2010; Austin, TX; 31 October 2010 through 5 November 2010.

[24] Xin Liu, Improving generation method for test pattern based on BDD learning; IEEE 2011 10th International Conference on Electronic Measurement and Instruments, ICEMI 2011; Chengdu; 16 August 2011 - 18 August 2011.

[25] O. Novak, J. Zahradka, COMPAS – Compressed Test Pattern Sequencer for Scan Based Circuits, in Proc. of the EDCC, 2005, pp. 403-414.

[26] E. J. Marinissen, Y. Zorian, R. Kapur, T. Taylor, L. Whetsel, Towards a Standard for Embedded Core Test: An Example, in Proc. of the IEEE International Test Conference (ITC), IEEE, 1999, pp. 616–627.

[27] J. H. Patel, I. Hamzaoglu, Test Set Compaction Algorithms for Combinational Circuits, in Proc. of the International Conference on Computer-Aided Design (ICCAD '98), 1998, pp.283-289.

[28] A. Jas, J. Ghosh-Dastidar, N. A. Touba, Scan vector compression/decompression using statistical coding, in Proc. of the VLSI Test Symp., 1999, pp. 114–120.

[29] C.V. Krishna, N.A. Touba, Reducing Test Data Volume Using LFSR Reseeding with Seed Compression, in Proc. of the International Test Conference, 2002, pp. 321-330.

[30] I. Hamzaoglu, J. H. Patel, Reducing Test Application Time for Full Scan Embedded Cores, in Proc. of the International. Symposium on Fault Tolerant Computing, 1999, pp. 260-267.

[31] A. Chandra, K. Chakrabarty, Frequency-Directed Run-Length (FDR) Codes with Application to System-on-a-Chip Test Data compression, in Proc. of the VLSI Test Symposium, 2001, pp. 42–47.

[32] A. Chandra, K. Chakrabarty, Test Data Compression and Test Resource Partitioning for System-on-a-Chip Using Frequency-Directed Run-Length (FDR) Codes, IEEE Transactions on Computers, vol. 52, No. 8, 2003, pp. 1076-1088.

[33] J. Rajski, Embedded Deterministic Test, IEEE Trans. on CAD, vol. 23, No. 5, 2004, pp. 776-792.

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

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

[36] F. Brglez, D. Bryan, K. Kozminski, Combinational Profiles of Sequential Benchmark Circuits, in Proc. of the International Symposium of Circuits and Systems, 1989, pp. 1929-1934.

[37] H.K. Lee and D.S. Ha, Atalanta: an Efficient ATPG for Combinational Circuits, Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993.

[38] P. R. Stephan, R. K. Brayton, A. L. Sangiovanni-Vincentelli, Combinational Test Generation Using Satisfiability, IEEE Transactions on Computer-Aided Design, vol. 15, No. 9, 1996, pp. 1167-1176.

[39] J.P.M. Silva, K.A. Sakallah, Robust Search Algorithms for Test Pattern Generation, in Proc. of the Fault-Tolerant Computing Symposium, 1997, pp. 152–161.

[40] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, L. Troyansky, 2+p-SAT: relation of typicalcase Complexity to the nature of the phase transition, In Random Structures & Algorithms, Vol. 15, Issue 3-4, 1999, pp. 414 – 435.

[41] F. Corno, M. S. Reorda, G. Squillero, RT-Level ITC 99 Benchmarks and First ATPG Results, IEEE Design & Test of Computers, Vol. 7, Issue 3, 2000, pp. 44-53.

[42] J. Jenicek, O. Novak, COMPAS Advanced test compressor, Proceedings of IEEE East-West Design and Test Symposium, EWDTS'10 2010, Pages 543-548.

[43] S. Malik et al., Chaff: engineering an efficient SAT solver, in Proc. of 38th ACM/IEEE Design Automation Conference, Las Vegas, Nevada, USA, 2001, pp. 530-535.