Simulation and SAT Based ATPG for Compressed Test Generation

J. Balcárek, P. Fišer, and J. Schmidt
Dept. of Digital Design
Czech Technical University in Prague
Prague, Czech Republic
{jiri.balcarek, petr.fiser, jan.schmidt}@fit.cvut.cz

***Abstract*—This paper presents a novel ATPG algorithm directly producing compressed test patterns. It benefits both from the features of satisfiability-based techniques and symbolic simulation. The ATPG is targeted to architectures comprised of interconnected embedded cores, particularly to the RESPIN architecture. We show experimentally that the proposed ATPG significantly outperforms the state-of-the-art approaches in terms of the test compression ratio.**

Keywords-ATPG, satisfiability, symbolic simulation, test compression, embedded cores, RESPIN.

I. INTRODUCTION

As the complexity of integrated circuits and systems continually increases, their testing becomes more and more difficult. The test data volume increases with the circuit size, making the test storing and application unfeasibly memory and time consuming. Therefore, using some kind of test compression becomes inevitable. According to the ITRS roadmap [1], the required test data volume compression reaches tremendous ratios (2,000-times in 2015 and up to more than 100,000-times in 2024).

The compression (and subsequent decompression) can be accomplished by several means. The test compression is performed algorithmically, whereas the decompression always involves some additional hardware. Basically, there are three major approaches:

1. A non-compressed test is generated by a *conventional* Automatic Test Pattern Generation tool (ATPG) and then it is algorithmically compressed. The decompression is then performed by a *special non-intrusive hardware*, usually a kind of FSM. This approach comprises Huffman encoding based algorithms [2], Golomb codes [3], statistical (FDR) codes [4], but also the well-known LFSR reseeding [5], [6], and the Embedded Deterministic Test (EDT) technique [7], which is now the industrial state-of-the-art.
2. *Dedicated design-for-testability (DFT) architectures* are used for test decompression, while the test generation process still relies on a *conventional ATPG*. Random access scan [8], [9], Illinois scan [10] and RESPIN-based [11], [12], [13] architectures belong to this category, together with rather theoretical papers with no particular architecture proposed [14].
3. *Dedicated ATPGs* are used to generate test for *dedicated architectures*. Such an approach theoretically offers the highest possible flexibility. Methods presented in [15], [16], and [17] are typical representatives. Here the ATPG is constrained or modified, so that the compressed test stream for the RESPIN architecture is generated directly. This is also the approach we have adopted in this paper.

As for ATPGs, there are two major baselines: circuit-based ATPGs [18], [19], [20], [21] and approaches transforming the ATPG problem to the satisfiability (CNF-SAT) problem [22], [23]. Modern ATPGs then combine benefits of both, mostly by introducing structural information to help the SAT-solver compute the solution faster [24], [25], [26].

In this paper we extend the SAT-Compress algorithm [16], [17] by injection of "don't cares" into the generated test stream, in order to maximize the freedom of the search algorithm, and thereby produce more compact results.

The SAT-Compress ATPG algorithm generates the test stream by constraining a conventional SAT-based ATPG. Conventional SAT solvers [27], [28] used as the vital part of most of SAT-based ATPG tools produce completely specified solutions (all variables are assigned a value in the satisfying solution), which somewhat restricts the search for subsequent test patterns. Although there are many techniques allowing don't cares in the solution [29], [30], or there are even optimization SAT solvers maximizing the number of don't cares in the solution [31]-[35] (let us call them DC-SAT solvers), we will show that they are highly unsuitable for our application.

We propose a fault simulation-based technique to alleviate the constraints imposed by completely specified SAT solutions by sequentially trying to "unassign" particular variables.

The benefits of such an approach will be experimentally documented, showing significant improvement in both the test stream size and test compression time.

The paper is organized as follows: the target architecture and basic principles of test compression/decompression are sketched in Section II. Basics of SAT-based ATPGs and the SAT-Compress algorithm are given in Section III. The way

of injecting don't cares is described in Section IV, Section V presents experimental results, to illustrate the contribution of the proposed method. The presented don't care injection technique is justified and discussed in Section VI. Section VII concludes the paper.

II. Target Architecture

A. The RESPIN Architecture

The SAT-Compress algorithm [16], [17] and also its enhancement proposed in this paper are based on the RESPIN architecture [11], which is targeted to SoC designs compliant with the IEEE P1500 standard [36], [37]. Only a very small modification of P1500 (addition of one multiplexer) can accomplish the test decompression job.

The basic idea of RESPIN is illustrated by Figure 1. Multiple embedded cores are considered here. To test one core (CUT – Core under Test), the test decompression is performed by another core (ETC – Embedded Tester Core).

RESPIN uses two features of P1500 – the serial and parallel test access mode. The compressed test bitstream serially enters the ETC, which is configured as a shift-register. Then the decompressed data is applied to the CUT, which is tested in the parallel scan-chain mode.

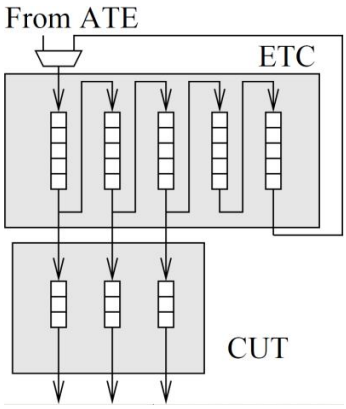

Figure 1. RESPIN architecture [11]

The ETC is provided by a multiplexer, enabling *rotation* of the pattern. Thereby, if no data come from the ATE, no information on the stored pattern is lost. This enables a simple way to compression: the deterministic non-compressed test patterns are *reordered* (see Subsection II.B), so that they *overlap* when rotated by one (optimally) clock cycle. Then the next test pattern to be applied to the CUT involves only one bit coming from the ATE. For details see [11].

B. Patterns Overlapping Based Approaches

An illustrative example of an overlapping-based compression is shown in Figure 2. Here the non-compressed test length equals to the number of patterns multiplied by the number of CUT scan-chain cells, 10×5 = 50 bits in the example case. When properly overlapped, the compressed test length is only 16 bits.

Note that by shifting the pattern only by one bit the overlap needs not be always achieved. Then two or more clock cycles (shifts) must be applied. Such a situation is in [11] referred to as a presence of *link patterns*. They do not increase the fault coverage, but may increase the defect coverage.

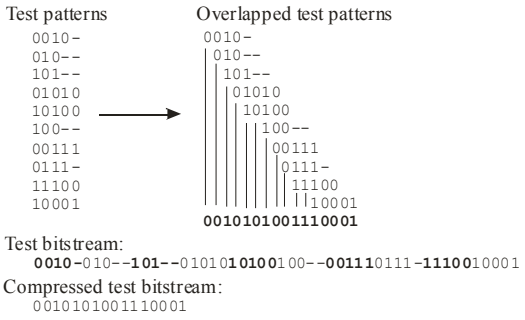

Figure 2. Patterns overlapping

Algorithms producing compressed test from test patterns obtained by ATPG rely on one property of scan-based designs: the order of patterns in which they are applied to the CUT is *insignificant*. Therefore, the patterns may be *reordered*, in order to reach maximum compression (i.e., maximum overlap).

Further, standard circuit-based ATPGs [21] are able to generate test patterns with a huge amount of don't care values (typically more than 90% in industrial designs [7]). Test don't cares are greatly beneficial for the compression, since they can be overlapped with any value. Thus, don't cares bring more freedom into the overlapping process.

The RESPIN compression algorithm [11] just sequentially tries all yet uncovered test vectors for overlap with the pattern stored in the ETC shifted by one bit, with the bit at the released position left unassigned (don't care). If such a vector is found, the pattern is constrained by this vector and a new test stream bit is generated. If not, the ETC pattern is shifted by one more bit, adding another don't care, and the procedure is repeated. The algorithm is greedy, in the first-improvement manner.

Let us note that other algorithms used for the purpose of finding an optimum overlapping exist. For example, in [14] the authors transform the problem to the Travelling Salesman Problem (TSP). However, TSP is NP-complete and therefore hard to apply to practical examples.

III. The SAT-Compress Algorithm

Unlike in the previously mentioned approaches ([7]-[14]), we do not rely on pre-generated test patterns. Even though numerous unspecified (don't care) bits are present in non-compacted tests produced by conventional ATPGs, there is still some information lost; all vectors detecting a fault usually cannot be described by a single pattern, even with don't cares.

Therefore, we use an *implicit* representation of *all* test patterns for a given fault as a SAT instance described in a conjunctive normal norm (CNF). Any satisfying solution of the related CNF-SAT problem represents a test vector for the fault [23] and vice versa. If the CNF is not satisfiable, the fault is undetectable (redundant).

The size of the SAT instance is linear with the circuit size, therefore such an approach imposes no computational and memory overhead.

For details of the circuit-CNF conversion, see [23], [24], [16], [17].

The compressed test is produced by constraining SAT instances by patterns stored in the ETC. Note that a similar approach was proposed in [15]. Here a conventional (commercial) ATPG was constrained in a similar way. However, SAT-based representation of test vectors offers much higher flexibility and possibilities of speed-up [17].

Basic principles of SAT-based ATPGs and the SAT-Compress algorithm will be briefly reviewed in this Section.

SAT-Compress tries to find the best overlap of test patterns by gradually building the compressed test bitstream, while each generated test pattern imposes constraints on the subsequent test pattern. The basic algorithm is shown in Figure 3.

First, a fault list for the circuit is generated (1). Redundant faults are detected by solving SAT for each fault and deleted from the fault list (2). Then the bitstream is initialized by the first test pattern TP_0 (3, 4). It can be an all-zero pattern representing the reset state of the scan-chain [11], [12], or any pattern the designer chooses. Then the pattern is submitted to the fault simulation, and faults detected by it are removed from the fault list (5).

The initial pattern forms constraints for the subsequent pattern. Indeed, the constraints are formed by shifting the pattern by one bit left and a don't care is put to the rightmost position, unconstraining the last bit of the pattern *TP*[*n*-1] (6, 7). The variable *n* represents the number of bits in the pattern which corresponds to the number of PIs of the circuit. Therefore, all but one bits of the subsequent pattern are constrained.

The compressed bitstream is gradually generated in the main loop of the algorithm (8-26). A CNF is generated for each fault (10), the constraints are applied to this CNF (11), and SAT is solved (12). If the constrained formula φ is satisfiable, a new test pattern is formed from the assignment of primary inputs in the SAT solution (14). Faults detected by this pattern are then removed from the fault list (15) and the inner loop is terminated (16).

If no fault can be detected by the pattern with the current constraints imposed (19), a link pattern (Subsection II.B, [11]) is generated by *randomly* assigning one bit (20-21).

Then a new bitstream bit and new constraints are formed (23-25). The test generation continues until the fault list is not empty (26).

SAT-Compress (*circuit*)

```
1   Generate FL for circuit
2   Remove redundant faults from the FL
3   TP = TP0
4   bitstream = TP
5   FL = FL - Detected_by_simulation(TP)
6   constraints[0...n-2] = TP[1 .. n-1]
7   constraints[n-1] = DC
8   do {
9       for each fault f∈FL {
10          φ = Create_CNF(circuit,f)
11          φ = Apply_constraints(constraints, φ)
12          S = SAT(φ)
13          if (S ≠ ∅) {
14          TP = Assignment_of_PIs(S)
15          FL = FL - Detected_by_simulation(TP)
16          break the for loop
17          }
18      }
19      if (S == ∅) {
20          TP = constraints
21          TP[n-1] = random_bit
22      }
23      bitstream += TP[0]
24      constraints[0...n-2] = TP[1 .. n-1]
25      constraints[n-1] = DC
26  } while (FL ≠ ∅)
27  return bitstream
```

Figure 3. The SAT-Compress algorithm

IV. INJECTING DON'T CARES

The proposed extension of the SAT-Compress algorithm by don't care injection, the *Coverage Preserving Don't Care Injection* (CPDCI) technique, will be described in this section. Basically, it involves an alleviation of constraints obtained by the SAT solver, which gives the SAT-Compress more freedom in test patterns overlapping. It can be assumed, that such a constraints reduction can accelerate the algorithm and increase the compression ratio, because less constrained patterns could be overlapped easier.

InjectDCs(*constraints*, *TP*, *FL*)

```
1   d1 = |detected_by_simulation(TP)|
2   TP_tmp = TP
3   for (i = 0; i < n; i++) {
4       if ( constraints[i] == DC ) {
5           TP_tmp[i] = DC
6           d2 = |detected_by_simulation(TP_tmp)|
7           if (d1 == d2) TP = TP_tmp
8       }
9   }
10  return TP
```

Figure 4. Coverage Preserving Don't Care Injection (CPDCI)

The DCs injection is performed by a procedure which is called for each test pattern obtained as the SAT solution of the CNF (Figure 3, step 14). Note that the SAT solution is completely specified, i.e., all bits of the pattern are assigned a value. However, the fault coverage of the pattern can remain intact, even when some of its bits are "unassigned". This is the main idea of the Coverage Preserving Don't Care Injection technique.

The procedure has three input parameters: the constraints, the test pattern (TP) and the fault list (FL). Its pseudo-code is shown in Figure 4.

First, the number of faults detected by the original completely specified pattern is found by simulation (1). A temporary test pattern is then formed as a copy of the test pattern (2). Next, each variable is sequentially tried for "unassignement", i.e., the don't care injection (3). Of course, the value of the variable cannot be changed if previous constraints were applied to it, since it would modify the already generated bitstream (4).

A don't care is then temporarily injected (5) and fault simulation [38] is performed (6). If the fault coverage remains intact, the injection is made permanent (7).

The result of the procedure is a test pattern covering all faults the original pattern covered, with some bits unassigned.

This procedure is greedy; its complexity is polynomial with the circuit size (depending on the fault simulation subroutine used). Therefore, it imposes no big run-time overhead.

When summarized, the CPDCI technique maximally alleviates the constraints, whereas the fault coverage of the pattern is preserved.

Restricting the don't care injection to fully preserve the fault coverage of the pattern may seem to be too strong. Actually, any pattern covering at least one fault can be used as a candidate. To accept such patterns, only the condition in step 7 has to be modified to "$d_2 > 0$". Even though maximum of don't cares are injected this way, such an approach did not lead to satisfactory solutions. This issue will be discussed in Section VI.

V. Experimental Results

A. Summary Comparison Results

A comparison of the basic SAT-Compress algorithm and its extension by *Coverage Preserving Don't Care Injection* technique (CPDCI) will be presented in this Subsection.

The measurements were performed on a CPU i5-2400 3.1GHz with 8GB RAM. Atalanta [21], [38] was used for the fault list generation and fault simulation purposes, MiniSAT v1.14 [27] as the SAT solver.

The experiments have been performed on a subset (170 benchmark circuits) of the ISCAS'85 [39], ISCAS'89 [40], ITC'99 [41] and LGSynth [42] benchmark circuits.

The summary results are shown in TABLE I.

The first column of the table (*"Circuit"*) represents the name of the benchmark circuit. The second column *"#Flts"* gives the number of faults in the circuit, which reflects its size. The next two columns *"#Bits"* and *"Time"* represent the number of bits of the compressed bitstream and the time spent by compression by the basic SAT-Compress algorithm. The next columns show results for the SAT-Compress algorithm with the CPDCI technique. The length of the compressed bitstream and the compression time is shown here too. The percentage test length and time improvements w.r.t. the basic SAT-Compress algorithm are shown in the "*Bits impr.*" and "*Time impr.*" columns.

Furthermore, the column *"#DCs tried"* shows the number of care bits tried for DCs injection and the "*DCs set*" column the number of successfully injected bits. The percentage of successfully injected don't cares is then shown in the "*Success*" column.

Finally, results of the test compression tool COMPAS [12], [13] are shown. This tool was chosen for comparison, because it represents the current state-of-the-art and is based on the same principles (the RESPIN architecture). However, it relies on a pre-computed test, instead of generating the compressed test sequence adaptively.

The compressed bitstream lengths are given in the "*#Bits*" column, the bitstream length differences w.r.t. the proposed CPDCI technique is shown in the last column. COMPAS runtimes are not present, since the experiments were conducted on different platforms, thus they are hardly comparable.

The last row of the table shows average values obtained from all benchmarks.

We can see that the CPDCI technique can significantly decrease the length of the compressed bitstream and accelerate the algorithm. The bitstream length is reduced by 46.31% on average and the compression time is reduced to 35.18% in comparison with the basic SAT-Compress algorithm. Even the successfulness of the CPDCI technique in don't care injection is remarkable; more than 65% of bits tried were successfully assigned a don't care.

The CPDCI technique increased the efficiency of the SAT-Compress algorithm, both in the compressed test length and test generation runtime. However, there are some cases where the extended SAT-Compress algorithm produces worse results, e.g. for the c499 circuit. We assume that such results are caused by some special properties of the benchmarks, like the percentage of random pattern resistant faults. This phenomenon deserves a more detailed exploration.

In comparison with COMPAS we reach a 6% improvement on average. There are benchmarks, for which SAT-Compress strikingly overcomes COMPASS (e.g., c1355, c2670). For some benchmarks COMPAS wins, however, the differences are not so big. This is probably due to a huge amount of randomness introduced into the ATPG process, as it will be shown in the following subsection.

B. *Influence of Randomness on the ATPG Process*

There are many random aspects that can influence the test generation process. First of all, it is the selection of the initial test pattern (TP_0 in Figure 3). It defines the initial constraints and therefore it influences the whole run of the greedy algorithm. The same holds for the ordering of the fault list. The fault list is traversed sequentially until a test vector detecting some fault is found (see Figure 3, steps 9, 16). Different orderings of the fault list will induce different runs of the test generation heuristic.

The influence of the initial test pattern is shown in Figure 5 and Figure 6 for two benchmark circuits (c432 and c880). The ATPG process was executed 5,000-times, each time with a randomly generated initial pattern. The frequencies of occurrence of the resulting bitstream of different lengths (the x-axis) are shown, both for the basic SAT-Compress and the SAT-Compress augmented with CPDCI.

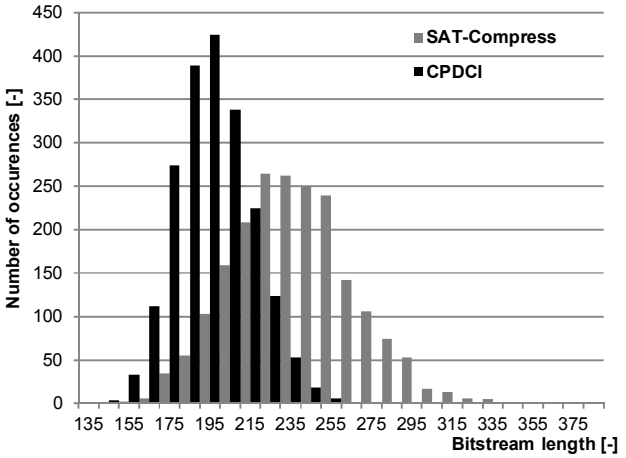

Figure 5. Frequency of bitstream length distribution (c432)

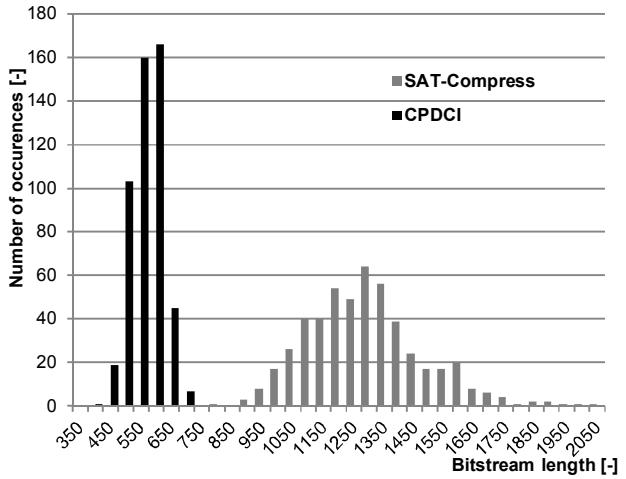

Figure 6. Frequency of bitstream length distribution (c880)

We can see that the histograms follow the Gaussian distribution, which is expectable. More importantly, the two distributions have different mean values, advantageously to the CPDCI. CPDCI also has smaller standard deviation. This gives us an experimental proof that CPDCI is systematically better than the original algorithm. Nevertheless, the influence of the randomness is crucial (even though reduced in the CPDCI case), and worse results can be obtained by CPDCI accidentally, see Figure 5, where the histograms overlap.

VI. JUSTIFICATION OF CPDCI AND DISCUSSION

As it was mentioned in Section IV, the requirement of completely preserving the fault coverage in CPDCI may be too strong. Theoretically, even more don't cares could be injected by the procedure in Figure 4, for a cost of losing the fault coverage of the processed pattern. Note that any pattern covering at least one fault is "useful" and may be returned as a result of the *InjectDCs* procedure. Faults that became undetected by this pattern are just not removed from the fault list, so as to be covered by latter patterns.

Therefore, we may ask whether the fault coverage, or the number of patterns don't cares, play more significant role in the compression process.

Now let us assume a slight modification of the *InjectDCs* algorithm from Figure 4, see Figure 7. Only the step 7 is modified, so that a single don't care injection is accepted, if the fault coverage of the resulting vector drops by the ratio *CL* at most, $CL \in \langle 0, 1)$. Therefore, there are two extreme cases:

1. $CL = 0$. This corresponds to the CPDCI technique; the fault coverage must remain the same. Therefore, the fault for which the pattern was generated (*f* in Figure 3, step 9) is covered, together with all the other faults the original pattern (*TP*) covered.
2. *CL* infinitely approaches 1. Here any vector covering *at least one* fault is accepted. Note that the fault *f* needs not be covered any more. However, a pattern covering any yet uncovered fault represents a "useful" and valid solution. Patterns having more don't cares are produced, thus the overlapping algorithm is offered yet more freedom.

Summarized, low values of *CL* represent cases, where the coverage is not lost by the pattern, however less don't cares are injected. High *CL* values induce injecting more don't cares, for a cost of losing fault coverage of the pattern.

InjectDCs_CL(*constraints*, *TP*, *FL*, *CL*)

```
1   d1 = |detected_by_simulation(TP)|
2   TP_tmp = TP
3   for (i = 0; i < n; i++) {
4       if ( constraints[i] == DC ) {
5           TP_tmp[i] = DC
6           d2 = |detected_by_simulation(TP_tmp)|
7           if ( (d1 - d2) / d1 ≤ CL ) TP = TP_tmp
8       }
9   }
10  return TP
```

Figure 7. The simulation based DCs injection

It is difficult to say intuitively, what *CL* values will produce best results. Low values preserve the fault coverage of the pattern, which may theoretically speed up the whole compression process. Since patterns covering more faults are generated, less patterns (and therefore SAT instances) will be needed to achieve complete fault coverage. However, these patterns will be rather constrained (low number of don't cares), and thus the chances of a successful overlap decrease.

Conversely, high *CL* values induce many don't cares, the vectors will more likely overlap, however, more vectors would be probably needed to achieve the complete fault coverage.

While the influence of *CL* on the numbers of generated SAT instances and injected don't cares is quite clear, it is discussable what effects will these two aspects have on the final bitstream length and the compression run-time. Therefore, we have evaluated the influence of the *CL* value on the algorithm execution experimentally.

The results for one representative ISCAS'85 [39] benchmark circuit c3540 are shown in TABLE II. The SAT-Compress algorithm was run with different values of the *CL* parameter and the absolute numbers of SAT instances solved (*"SATs"*), the absolute numbers of injected don't cares (*"DCs"*), the final bitstream length (*"Bits"*), and the compression run-time (*"Time* [s]*"*) were measured. The values were obtained from averaging values of 30 runs with random initial patterns, to diminish the influence of randomness.

TABLE II. INFLUENCE OF LOSING FAULT COVERAGE

CL [%]	*SATs*	*DCs*	*Bits*	*Time [s]*
0	770.33	3385.97	820.33	585.07
5	765.59	3442.73	815.59	467.85
10	773.15	3509.79	823.15	637.94
15	804.88	3686.73	854.88	606.44
20	799.12	3804.24	849.12	571.74
25	796.12	3899.79	846.12	548.21
30	822.09	4008.97	872.09	578.85
35	843.61	4305.64	893.61	590.09
40	861.39	4439.18	911.39	605.66
45	868.23	4492.18	918.23	342.38
50	946.50	5206.18	996.50	344.73
55	941.55	5216.45	991.55	492.73
60	950.91	5331.45	1000.91	514.31
65	985.50	5573.05	1035.50	844.77
70	1042.68	6130.23	1092.68	845.89
75	1087.73	6565.82	1137.73	775.16
80	1099.50	6756.91	1149.50	754.09
85	1126.05	7106.68	1176.05	884.67
90	1191.00	7839.77	1241.00	982.47
95	1226.36	8462.09	1276.36	1072.92
99.99…	1307.88	9924.30	1357.88	1105.55

We can see that the initial assumptions were confirmed: the number of solved SAT instances monotonously grows with increasing *CL*, while the number of injected don't cares increases too.

The most important observation concerns the final bitstream length: the bitstream length *monotonously increases* with *CL* (see Figure 8), with best results obtained for $CL = 0$, i.e., the CPDCI technique. The same holds for the runtime. Similar experiments were performed on many other benchmark circuits and the same behavior was observed in all cases.

This experiment has shown that no fault coverage of every single pattern can be sacrificed, even though more don't cares would be injected otherwise. Therefore, the usage of the CPDCI technique from Figure 4 is fully justified; there is no need for looking for a compromise between the number of injected don't cares and the fault coverage.

Consequently, this also compromises using DC-SAT solvers ([29]-[35]) to obtain don't cares; don't cares must be injected with care, and definitely their number in the SAT solution must not be the optimization criterion for the SAT-solver.

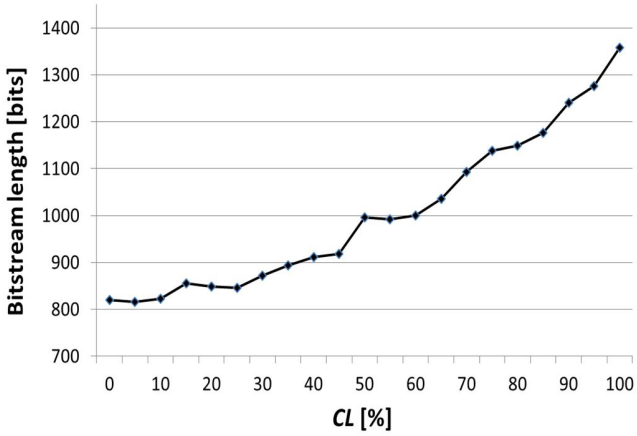

Figure 8. Influence of CL on the generated bitstream length

VII. CONCLUSIONS

We have presented an efficient enhancement of the SAT-Compress ATPG algorithm, the *Coverage Preserving Don't Care Injection* technique (CPDCI). Basically, the SAT-Compress algorithm gradually constructs compressed test patterns by repetitively solving the SAT problem for instances constrained by patterns generated in previous steps. The CPDCI technique significantly alleviates these constraints by substituting defined values by don't cares, without any loss of the fault coverage in each step. This is accomplished by a procedure based on a symbolic fault simulation. Less constrained SAT instances allow reaching better results, both in test bitstream size (by 46% on average) and test generation time (by 35% on average). We see that even though the fault simulation imposes some computational overhead, the resulting run-time is significantly reduced, because of shorter bitstreams generated.

ACKNOWLEDGMENT

This research has been supported by the grant of the Czech Technical University in Prague, SGS13/101/OHK3/1T/18.

REFERENCES

[1] Semiconductor Industry Association, "The International Technology Roadmap for Semiconductors (ITRS)", 2009. On-line: http://www.itrs.net/

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

[3] A. Chandra and K. Chakrabarty, "Efficient test data compression and decompression for system-on-a-chip using internal scan chains and Golomb coding," in Proc. of Design, Automation and Test in Europe, Conference 2001, pp.145-149.

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

[5] B. Koenemann, "LFSR – Coded Test Patterns for Scan Designs," in Proc. of Europian Test Conf., Munich, Germany, 1991, pp. 237-242.

[6] S. Hellebrand, et al., "Built-In Test for Circuits with Scan Based on Reseeding of Multiple-Polynomial Linear Feedback Shift Registers," in IEEE Trans. on Comp., vol. 44, No. 2, February 1995, pp. 223-233.

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

[8] D. H. Baik, K. K. Saluja, and S. Kajihara, "Random Access Scan: A Solution to Test Power, Test Data Volume and Test Time," in Proc. of 17th International Conf. on VLSI Design, Jan. 2004, pp. 883-888.

[9] Dong Hyun Baik and K.K Saluja, "Progressive random access scan: a simultaneous solution to test power, test data volume and test time," in Proc. of IEEE International Test Conference, Nov. 2005.

[10] I. Hamzaoglu and 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.

[11] R. Dorsch and H.-J. Wunderlich, "Reusing Scan Chains for Test Pattern Decompression," in Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 18, Issue 2, April 2002, pp. 231 – 240.

[12] O. Novák, J. Zahrádka, "COMPAS – Compressed Test Pattern Sequencer for Scan Based Circuits," in Proc. of EDCC, 2005, pp. 403-414.

[13] J. Jeníček and O. Novák, "COMPAS Advanced test compressor," in Proc. of IEEE East-West Design and Test Symposium 2010, pp. 543-548.

[14] C. Su and K. Hwang, "A Serial Scan Test Vector Compression Methodology," in Proc. of IEEE International Test Conference (ITC), 1993, pp. 981-988.

[15] R. Dorsch and H. J. Wunderlich, "Tailoring ATPG for embedded testing," in Proc. of IEEE International Test Conference (ITC), 2001, pp. 530–537.

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

[17] J. Balcárek, P. Fišer, and J. Schmidt, „Techniques for SAT-based Constrained Test Pattern Generation," in Microprocessors and Microsystems, Vol. 37, Issue 2, March 2013, Elsevier, pp. 185-195.

[18] J.P. Roth, "Diagnosis of automata failures: A calculus and a method," in IBM Journal of Research and Developmen, Vol. 10, Issue 4, July 1966, pp. 278-291.

[19] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," in IEEE transactions on Computers, Vol. C-30, no. 3, 1981, pp. 215-222.

[20] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," in IEEE Trans. Comput. Vol. 32, No. 12 (December 1983), pp. 1137-1144.

[21] 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, 1999.

[22] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co. New York, USA, 1990, p. 338.

[23] T. Larrabee, "Test pattern generation using boolean satisfiability," in IEEE Transactions on Computer-Aided Design, vol. 11, 1992, pp. 4-15.

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

[25] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver," in Proc. of the 39th annual Design Automation Conference (DAC '02), ACM, New York, NY, USA, pp. 747-750.

[26] S. Safarpour, A. Veneris, R. Drechsler, and J. Lee, "Managing don't cares in Boolean satisfiability," in Prof. of Design, Automation and Test in Europe Conference and Exhibition, 16-20 Feb. 2004, pp. 260-265.

[27] N. Éen, N. Sorensson, "An extensible SAT-solver," in Lecture Notes in Computer, Science 2919 - Theory and Applications of Satisability Testing. Springer Verlag, Berlin Heidelberg New York (2004) pp. 333-336.

[28] N. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an effcient SAT solver," in Proc. of 39th Design Automation Conference (DAC 2001), pp. 530-535.

[29] M. Elm, M. A. Kochte, H. Wunderlich, "On Determining the Real Output Xs by SAT-Based Reasoning," In 19th Asian Test Symposium, Shanghai, 2010, pp. 39-44.

[30] M. A. Kochte, M. Elm, H. Wunderlich, "Accurate X-Propagation for Test Applications by SAT-Based Reasoning," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 31, no. 12, pp. 1908-1919, 2012.

[31] R. Ben-Eliyahu, R. Dechter, "On Computing Minimal Models," Annals of Mathematics and Artificial Intelligence , vol. 18, pp. 2-8, 1993.

[32] C. Pizzuti, "Computing Prime Implicants by Integer Programming," in 8th International Conference on Tools with Artificial Intelligence, 1996, pp. 332.

[33] V. Manquinho et al. "Prime implicant computation using satisfiability algorithms," in 9th International Conference on Tools with Artificial Intelligence, Newport Beach, CA, 1997, pp. 232-239.

[34] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in TACAS'04, Barcelona, Spain, Mar.-Apr. 2004, pp. 31-45.

[35] I. Dillig, T. Dillig, K.L. McMillan, and A. Aiken, "Minimum satisfying assignments for SMT," in Proceedings of the 24th international conference on Computer Aided Verification, 2012, pp. 394-409.

[36] Y. Zorian, E.J. Marinissen, S. Dey, "Testing embedded-core-based system chips," in Computer, vol.32, no.6, pp.52,60, Jun 1999.

[37] E.J. Marinissen, R. Kapur, M. Lousberg, T. McLaurin, M. Ricchetti, Y. Zorian, "On IEEE P1500's Standard for Embedded Core Test," in Journal of Electronic Testing, August 2002, Vol.18, Issue 4-5, pp. 365-383.

[38] H. K. Lee and D. S. Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits," in IEEE

Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, pp. 1048-1058, September 1996.

[39] 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.

[40] 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.

[41] F. Corno, M.S. Reorda, G. Squillero, "RT-level ITC`99 benchmarks and first ATPG results," in: Proc. of the IEEE Design and Test of Computers (2000) 44-53.

[42] K. McElvain, "LGSynth93 benchmark set: Version 4.0", 1993.

TABLE I. EXPERIMENTAL RESULTS FOR THE BASIC ALGORITHM AND EXTENSION BY CPDCI

Circuit	*#Flts*	SAT-Compress		SAT-Compress with CPDCI							COMPAS	
		#Bits	*Time [s]*	*#Bits*	*Bits impr. [%]*	*Time [s]*	*Time impr. [%]*	*#DCs tried*	*#DCs set*	*Success [%]*	*#Bits*	*Diff. [%]*
alu4	6435	3349	1994.97	3048	8.99	1773.53	11.10	3380	349	10.33	-	-
b04_C	1666	5408	463.67	910	83.17	88.25	80.97	7659	6876	89.78	-	-
b05_C	1928	1091	90.95	631	42.16	55.57	38.90	1593	1018	63.90	-	-
b07_C	1084	997	9.89	706	29.19	7.88	20.32	1444	895	61.98	-	-
b11_C	1675	863	41.90	562	34.88	32.00	23.63	1557	1084	69.62	-	-
c1355	1566	330	13.40	334	-1.20	13.43	-0.22	312	19	6.09	1040	67.88
c1908	1869	607	44.66	495	18.45	36.71	17.80	542	82	15.13	1009	50.94
c2670	2629	3103	556.27	1806	41.80	387.30	30.38	11276	9791	86.83	6553	72.44
c3540	3291	3422	1618.65	833	75.66	323.90	79.99	4146	3415	82.37	747	-10.32
c432	520	209	1.39	156	25.36	1.28	7.91	368	256	69.57	195	20.00
c499	750	182	1.51	219	-16.89	1.67	-10.60	206	28	13.59	260	15.77
c5315	5291	1205	261.30	815	32.37	275.89	-5.58	2410	1812	75.19	1255	35.06
c7552	7419	6581	2739.73	3522	46.48	1902.73	30.55	9029	5998	66.43	6005	41.35
c880	942	1195	35.71	614	48.62	15.16	57.55	2250	1765	78.44	540	-12.05
duke2	1302	1486	56.80	986	33.65	35.13	38.15	1717	810	47.18	-	-
ex5p	5430	276	38.72	276	0	42.12	-8.78	268	0	0	-	-
intb	1893	2070	220.27	1653	20.14	171.01	22.36	2103	471	22.40	-	-
jbp	1132	2174	41.42	843	61.22	16.69	59.71	2281	1563	68.52	-	-
misex3	9251	3556	5240.01	3467	2.50	5220.65	0.37	3551	100	2.82	-	-
s1196	1242	2487	109.33	876	64.78	36.36	66.74	4292	3474	80.94	740	-15.53
s1238	1286	2705	141.46	876	67.62	40.06	71.68	4926	4105	83.33	741	-15.41
s13207	9664	114390	285075	5498	95.19	22678.30	92.04	206673	202598	98.03	4163	-32.07
s1423	1501	1179	46.38	628	46.73	39.53	14.77	2346	1871	79.75	596	-5.10
s15850	11336	77582	147342	5734	49.41	22686.3	84.60	179148	174836	97.59	8234	30.36
s344	342	161	0.59	95	40.99	0.47	20.34	280	210	75.00	85	-10.53
s35932	35110	3686	308382	4998	85.76	390677	-26.69	2971215	2969101	99.93	1860	-32.21
s382	399	255	0.61	131	48.63	0.39	36.07	258	161	62.40	123	-6.11
s420	430	526	2.81	370	29.66	1.62	42.35	748	463	61.90	352	-4.86
s526n	553	830	5.27	471	43.25	2.70	48.77	1197	785	65.58	344	-26.96
s5378	4511	19847	6765.48	1989	89.98	870.94	87.13	31022	29444	94.91	2148	7.40
s641	463	1335	13.35	469	64.87	5.62	57.90	2282	1919	84.09	397	-15.35
s713	543	1223	11.64	454	62.88	6.08	47.77	2199	1859	84.54	428	-5.73
s820	850	702	10.30	664	5.41	9.97	3.20	692	65	9.39	460	-30.72
s838	857	2078	32.20	955	54.04	19.27	40.16	2957	2242	75.82	920	-3.66
s9234	6475	24395	25844.19	5688	76.68	10238.03	60.39	53308	48599	91.17	11594	50.94
s953	1079	3131	95.99	771	75.38	20.23	78.92	4317	3693	85.55	723	-6.23
t481	2853	5541	1808.29	5147	7.11	1561.09	13.67	5433	304	5.60	-	-
table3	2487	2025	382.11	2085	-2.88	413.89	-8.32	2134	69	3.23	-	-
table5	2384	3191	703.46	2821	11.60	609.92	13.30	3301	547	16.57	-	-
term1	1314	6221	405.79	1418	77.21	102.96	74.63	5443	4089	75.12	-	-
vda	1970	680	31.95	594	12.65	27.24	14.74	652	103	15.80	-	-
vg2	1122	2507	59.94	1403	44.04	32.53	45.73	2430	1093	44.98	-	-
x1	2504	7583	886.07	2953	61.06	354.54	59.99	7689	5013	65.20	-	-
Avg.	3396	7649	23209.59	1585	46.31	13907.19	35.18	86341	85018	65.54	1981	6.08