On Don't Cares in Test Compression

Jiří Balcárek, Petr Fišer, and Jan Schmidt
Dept. of Digital Design
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
{jiri.balcarek, petr.fiser, jan.schmidt}@fit.cvut.cz

Corresponding author:
Petr Fišer
Dept. of Computer Science & Engineering
Czech Technical University in Prague, FIT
Thákurova 9, CZ-160 00, Prague 6
fax: +420 22435 9819, tel: +420 22435 9842
e-mail: fiserp@fit.cvut.cz

Abstract. **Both test compression tools and ATPGs directly producing compressed test greatly benefit from don't care values present in the test. Actually, presence of these don't cares is essential for success of the compression. Contemporary ATPGs produce tests having more than 97% of don't cares for large industrial circuits, thus high compression ratios can be expected. However, these don't cares are placed in the test in an "uninformed" way. There are many possibilities of constructing a complete test for a circuit, while the ATPG chooses just one particular, without respect to the subsequent compression process. Therefore, the don't cares cannot be fully exploited. In this paper we show how severe this issue is. A novel ATPG algorithm directly producing compressed test patterns for the RESPIN decompression architecture is presented. Test don't cares are placed in an informed way, so that they are maximally exploited by compression. We compare the results with several ways of uninformed don't care generation to show the benefits of the proposed method. Results for the ISCAS and ITC'99 benchmark circuits are shown and compared to state-of-the-art test compression techniques.**

Keywords-ATPG, test don't cares, satisfiability, symbolic simulation, test compression, embedded cores.

1. 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: 2,700-times in 2015 and almost 50,000-times in 2028.

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 dedicated 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] to some extent, and the Embedded Deterministic Test (EDT) technique [7], which is now the industrial state-of-the-art.
2. *Generic 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 *generic or dedicated architectures*. Such an approach theoretically offers the highest possible compression ratio. The algorithm has the highest flexibility, since the compressed test generation is not performed in two subsequent and separated phases. 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 target 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].

Current ATPGs are able to produce tests containing unspecified values – we call them *test don't cares*. The *test vectors* (*patterns*) that could be *incompletely specified* are then referred to as *test cubes*. Don't cares can be efficiently used in the test compression process, since they introduce a kind of flexibility, as any value can be assigned to them. However, there are many ways of forming a (complete) test. The following two aspects must be accounted for:

1. Usually every single fault can be detected by many different test vectors (test patterns). Moreover, a complete set of such vectors usually cannot be described by a single test cube.
2. Test compaction [27], [28] performed by ATPG tools, merging test cubes to reduce the test size, can be executed in different ways. Actually, it is an NP-hard process [22] and heuristics are used in practice.

As a result, the don't cares are placed in the test *randomly*, from the point of view of their subsequent usage. Even though the amount of test don't cares typically reaches very high values (more than 97% for industrial designs [7]), still even more flexibility could be exploited by compression. In other words, the test compression process is provided a single set of test cubes only, out of numerous possibilities. There is no guarantee that the compression would not perform better, when given a different test set.

In this paper we show how severe this issue is. We present a SAT-based ATPG producing a compressed test directly. The test cube generation is driven by the compression process, so that most suitable test cubes are used. Naturally, don't cares in test patterns are beneficial for the compression. To obtain these don't cares we propose a method of generating test don't cares in an *informed* way. Then we compare the results with methods where don't cares are obtained in an *uninformed* way, to show the benefits of the former one.

Note that the terms "informed" and "uninformed" used throughout this paper come from the concept of informed and uninformed local search heuristics. The informed methods use some additional information to properly guide the neighborhood exploration. Here, the neighborhood are incompletely specified test patterns, and the heuristic function is the number of faults covered by them.

We extend the SAT-Compress algorithm [16], [17] by injection of "don't cares" into test patterns. The SAT-Compress ATPG algorithm generates the compressed test stream by constraining a conventional SAT-based ATPG. Conventional SAT solvers [29], [30] used as the vital part of most of SAT-based ATPG tools produce completely specified solutions, where all variables have a specified value in the satisfying solution. There are several ways of introducing don't cares (unspecified variables) into the SAT solution. First, there are SAT-solvers producing incompletely specified solutions directly [31]-[34]. Here satisfying solutions comprised of minimum literals (minimal models, prime implicants) are generated. However, the optimization criterion is computed over *all* variables, which is unsuitable for our application, where values of only *some* variables are of interest.

Next, optimization version of SAT can be transformed to Integer Linear Programming (ILP) [35]. Here the optimization criterion can be modified for our purposes, so that only some variables are accounted in its computation. In this paper we propose a similar method, particularly the conversion of the SAT problem minimizing the number of specified variables in the satisfying solution to Pseudo-Boolean Optimization (PBO) [36].

Finally, don't cares can be injected into a completely specified vector obtained from a conventional SAT solver [29], [30], while the coverage is checked by symbolic fault simulation. When fault simulation is performed, we get additional information on the obtained test cube – its fault coverage [38]. Then we can, e.g., inject don't cares while respecting the fault coverage. This is the informed way of obtaining test don't cares proposed in this paper. We compare this simulation-based method with the uninformed ones and show its benefits in test compression, in terms of both the compressed test stream size and test compression time.

The paper is organized as follows: the target architecture and basic principles of the studied compression algorithms are shown in Section 2. Possibilities of obtaining don't cares in the SAT-Compress process are described in Section 3, Section 4 presents experimental results, to show the role of don't cares in test compression and to illustrate the benefits of informed don't care injection. Section 5 concludes the paper.

2. The decompression architecture and compression algorithms

2.1. The RESPIN architecture

The SAT-Compress algorithm [16], [17] and also its enhancements proposed in this paper and [38] are based on the RESPIN architecture [11], which is targeted to System-on-Chip (SoC) designs compliant with the IEEE Std. 1500 [39], [40]. Only a very small modification of IEEE Std. 1500 (addition of one multiplexer) can accomplish the test decompression job.

The basic idea of RESPIN is illustrated in 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 IEEE Std. 1500 – the serial and parallel test access modes. 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.

The ETC is provided with a multiplexer, enabling *rotation* of the pattern. Thereby, if no data come from the ATE, no information on the stored pattern is lost. This opens a simple way to compression: when the deterministic non-compressed test patterns *overlap* when rotated by p bits, each test pattern to be applied to the CUT involves only p bits coming from the ATE. Actually, rotation needs not be used in practice. Typically, one bit comes from ATE in each cycle ($p = 1$), while the remaining bits of the pattern are formed by shifting the previous pattern by one bit. This approach eliminates the need for any control data provided by ATE. For details see [11].

2.2. 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, $10 \times 5 = 50$ bits in the example case. When properly overlapped, the compressed test length is only 16 bits.

Note that by shifting the pattern by one bit only, the overlap needs not be always achieved. Then two or more clock cycles (shifts) must be applied. Such a case is in [11] referred to as a presence of *link patterns*. They do not increase the fault coverage, but may increase the defect coverage. In Figure 2 there are link patterns in the 6^{th} and 11^{th} clock cycle.

Since the RESPIN decompression architecture is based on full scan, only combinational logic is tested. Therefore, the order of patterns in which they are applied to the CUT is *insignificant*. The patterns may then be *reordered*, to reach maximum compression (i.e., maximum overlap).

Further, as mentioned above, standard circuit-based ATPGs [21] are able to generate test patterns with a huge amount of don't care values. Test don't cares are greatly beneficial for the compression, since they can be overlapped with any value (see Figure 2). Thus, don't cares bring more freedom into the overlapping process. These two principles are aimed to be fully exploited by RESPIN-based compression techniques [11], [12], [13], [15], [16], [17].

2.3. The RESPIN algorithm

In order to show the main differences between the original RESPIN compression algorithm [11] and SAT-Compress [16], [17], whose enhancement is proposed in this paper, we will briefly review both.

The RESPIN algorithm (see Figure 3) starts with running a conventional ATPG to produce a test T for the circuit (1). Then the constraints cube c is initialized with the initial test pattern (2). This can be the previous content of the ETC scan-chain, an all-zero vector, or it can be decided by the compression algorithm. For the latter case, c is initialized to an all-DC vector.

Then all test cubes from T are tried for merging with c (5). If it is possible (there is a non-empty intersection of the cubes), c is constrained by T (6) and T is removed from the test set (7).

When all test vectors have been tried for merging, a new bitstream bit is formed (10), the constraints cube is shifted by one bit (11), and the released position is assigned a don't care (12). The process repeats, until the test set is not empty (3). Then the bitstream is completed by the remaining bits in c (14).

Note that if the merging (4, 5) fails for all vectors from T, a new bitstream bit is formed anyway; the link pattern is thus generated (see Subsection 2.2).

Such an approach is apparently greedy, and its efficiency is crucially influenced both by the pre-generated test T and its ordering. Don't care values in the test enable easier cube merging (5), thus higher compression.

Let us note that other algorithms used to find an optimum overlapping of pre-generated test vectors exist. For example, in [14] the authors transform the problem to the Travelling Salesman Problem (TSP) [22].

In COMPAS [12], [13], there is a fault simulation employed, in order to efficiently remove dominated test patterns.

2.4. The SAT-Compress algorithm

Unlike in the previously mentioned approaches ([7]-[14]), SAT-Compress does not rely on pre-generated test patterns. Even though test patterns produced by conventional ATPGs contain many unspecified (don't care) values, there is still some information lost in the ATPG process, as discussed in the Introduction.

Therefore, SAT-Compress uses 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, benefiting the scalability of the approach. For details of the circuit-to-CNF conversion, see [23], [24], [16], [17].

The compressed test for RESPIN 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].

Similarly to RESPIN and COMPAS, 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 patterns. The basic algorithm is shown in Figure 4.

First, a fault list for the circuit is generated (1), from which redundant faults are removed by constructing and solving a SAT instance for each (2). The constraints cube is set to the initial test pattern, as in RESPIN (3). The compressed bitstream is gradually constructed in the main loop of the algorithm (4-18), fault by fault (5). A CNF is generated for the processed fault (6) [23], the constraints in form of unit clauses are applied to this CNF (7), and SAT is solved (8). If the constrained formula φ is satisfiable, the constraints are tightened by the assignment of primary inputs (PIs) in the SAT solution (10). Then the loop is terminated (11). Note that the new constraint cube is a subcube of the former one, since the original constraints imposed to PIs are included in the SAT instance (7).

Then a new bitstream bit is formed (14), faults detected by the pattern are removed from the fault-list (15), and new constraints are formed by a shift left (16-17). The test generation continues until the fault list is not empty (4). Finally, the bitstream is completed by the bits remaining in *c* (19).

The faults are processed in the order defined by the pre-generated fault-list (5). We have observed that this ordering, together with the initial pattern selection, crucially influences the result quality (see Subsection 4.1), similarly as does the ordering of test cubes in RESPIN [42]. Despite of thorough investigations, we were not able to find a way of determining a "good" ordering, e.g., by preferring hard-to-detect faults. Actually, similar influences occur in almost all local search heuristics [41], [42] and the problem is difficult to be fought. We reduce the influence of faults ordering by making the algorithm more robust and more efficient (see Subsection 4.1).

Note that compared to RESPIN and related algorithms, there is no concept of *test set* in SAT-Compress; the test is represented implicitly. On the other hand, RESPIN does not operate with concepts as *fault* and *circuit* – these are treated in the ATPG phase run prior to the compression algorithm.

3. Obtainig test don't cares

If a conventional SAT solver ([29], [30]) is used in step (8) of the algorithm in Figure 4, completely specified satisfying solutions are returned. As a result, the produced test patterns (which also impose constraints on the subsequent patterns) are completely specified too. Then the only don't care values occurring in the algorithm are obtained by the pattern shifting (17), thus they appear at the "tail" of the constraints cube only.

However, a less specified SAT solution can be obtained, yielding less constraints. Possibilities of doing so will be discussed in this section.

3.1. Special SAT solvers

There is a group of special optimization SAT solvers that minimize the number of specified variables in the solution. So called "minimal models" [31], [33], [34] or "prime implicants" [32] are computed here. There are also techniques converting the optimization SAT problem to Integer Linear Programming (ILP) [35].

These SAT solvers return a satisfying assignment of variables as a solution, while the total number of variables assigned a value is minimized.

The SAT instances used in SAT-Compress (and all SAT-based ATPGs) contain variables representing the circuit primary inputs (*n* in Figure 4), but also incomparably more other variables describing the circuit structure, fault propagation, etc. [23], [24]. However, the constraints are derived from values of *primary inputs only* (line 10 in Figure 4); values of the other variables are of no interest in the overall algorithm. Therefore, minimizing the number of specified primary input variables only makes sense. All the above mentioned techniques, except ILP, minimize the number of *all* specified variables. This renders those techniques almost useless for our purpose, since there is no guarantee (or at least a promise) that the number of specified *input* variables, which form a very small subset, will be near minimum. For this reason, these techniques will not be studied in this paper, and more suitable techniques will be devised instead.

3.2. Pseudo-Boolean optimization

Similarly to [35], the optimization SAT problem can be converted to Pseudo-Boolean Optimization (PBO) and solved by available efficient PBO solvers, like MiniSAT+ [37]. The resulting problem is an optimization one, in contrast to the original constructive problem. The principles of the conversion will be described in this subsection.

For our purposes, it is convenient to obtain a maximally unspecified test vector that is a solution with *maximum of unspecified values* at primary inputs. This criterion is often phrased as *maximum don't cares*. As we show here, such a use collides with don't cares (DC) in function description. To avoid confusion, we use the term *unspecified value* in this section, and use the symbol U for such value.

For better understanding, we will start with a conversion of a MIN-SAT problem, where the number of *variables assigned to "1"* is minimized, to PBO:

1) Let $x_1, \ldots, x_m$ be variables of the original MIN-SAT problem.
2) For each clause $(l_1 + l_2 + \cdots + l_j)$, where l_i are individual literals (variables or their negations) construct an inequality $l_1 + l_2 + \cdots + l_j \geq 1$.
3) If a literal $l_i = x_k$ (variable in its direct form), substitute $l_i = x_k$ in the inequality.
4) If a literal $l_i = \overline{x_k}$ (variable in its negated form), substitute $l_i = (1 - x_k)$.
5) Form the optimization criterion as $x_1 + x_2 + \cdots + x_m = min.$

Example:

Let us have a CNF formula of 3 variables:

$$(x_1 + x_2)(x_2 + x_3)(\overline{x_2} + \overline{x_3}) \quad (1)$$

It will be transformed into the following PBO formulation:

$$\begin{aligned} &x_1 + x_2 \geq 1 \\ &x_2 + x_3 \geq 1 \\ &(1 - x_2) + (1 - x_3) \geq 1 \Rightarrow -x_2 - x_3 \geq -1 \\ &x_1 + x_2 + x_3 = min. \end{aligned} \quad (2)$$

There are three satisfying solutions of the SAT instance:

$$\begin{aligned} &x_1 = 0, x_2 = 1, x_3 = 0, \\ &x_1 = 1, x_2 = 1, x_3 = 0, \\ &x_1 = 1, x_2 = 0, x_3 = 1. \end{aligned} \quad (3)$$

When solved as PBO, a single solution minimizing the number of variables assigned to "1" will be returned:

$$x_1 = 0, x_2 = 1, x_3 = 0. \quad (4)$$

■

Still, all the variables are in the Boolean domain, while we need to encode unspecified values. For this purpose, we must use two Boolean variables to encode each literal, for example, as shown in TABLE I.

In this encoding, x_i^A value indicates whether the particular variable is assigned a value, x_i^V is then the value.

The optimization criterion can be then formed as:

$$x_1^A + x_2^A + \cdots + x_{n-1}^A = min. \quad (5)$$

where $x_1 \ldots x_{n-1}$ are primary inputs.

Detecting a fault means to control *specified* values in the circuit, and to observe *specified* values at outputs. Hence, the propagation of unspecified values must be observed, and *every* original CNF variable must be doubled in PBO.

The CNF is then rewritten into a PBO instance in a straightforward way, as shown in the Example above. The solution of the PBO instance maximizes the number of unspecified PI values, i.e., a maximum of test don't cares is obtained.

During the circuit-to-CNF conversion, characteristic functions of all gates in the circuit in CNF form are added to the SAT instance. For a gate with inputs $x_1 \ldots x_m$ and output *y*, the signature of the characteristic function *F* is

$$F: \{0,1\}^{m+1} \rightarrow \{0,1\}$$

For our problem, we need to take undefined value U into account, and the function *F* becomes

$$F: \{0,1,\mathrm{U}\}^{m+1} \rightarrow \{0,1\}$$

The strategy is to calculate *F* in some form, then to encode it by TABLE I. into

$$F: \{0,1\}^{2m+2} \rightarrow \{0,1\}$$

or, alternatively, into two functions

$$F^V: \{0,1\}^{2m+1} \rightarrow \{0,1\}$$
$$F^A: \{0,1\}^{2m+1} \rightarrow \{0,1\}$$

which have y^V resp. y^A as the last argument, and to convert them to CNF form.

The main task is to find a concise and complete representation of *F*. By completeness we mean that all possible combinations at input and output are covered, so that the origin, propagation, and termination of undefined values can be calculated. For this purpose, we adapted D-intersection [43]. Because we represent *F* as a set of terms in tabular form, TABLE II. includes also the '-' symbol. Notice that incompatibility cannot occur here.

The complete algorithm for generation of a CNF for a given gate is shown in Figure 5. Let us assume that *gate* is a table describing the on-set of a completely specified Boolean function with one output, and that the columns of the table are labeled $x_1, \ldots, x_{m-1}, y$. Furthermore, if *t* is a term, let *t*[*j*] be the symbol of *t* in the column labelled *j*.

The algorithm has four main phases. The first one (lines 1 to 5) derives the characteristic function. Lines 6 to 16 are the main phase, which adds terms describing the behavior of undefined values to the function. Finally, the third phase (line 18) encodes the table and phase four (19 – 27) outputs the resulting CNF, using CNF and DNF duality. Before phase four, *F* can be split into F^V and F^A. In many cases, the resulting functions are smaller and easier to complement.

For simplification of the truth table and, more importantly, off-set computation (complementation), the Espresso minimizer is used [44].

The algorithm is as feasible as Espresso minimization [44] and off-set generation are. Large XOR gates are difficult and have large characteristic functions, but are manageable to 10 inputs. Specialized algorithms can be devised for even larger gates, this one, however, has the advantage of being universal.

A complete example of a 2-NAND gate to CNF conversion is shown in Figure 6.

For purposes of the test compression algorithm, a library of CNF descriptions for every supported gate is created using the procedure from Figure 5. Thus, the conversion is run only once.

3.3. Injection techniques

Don't cares can also be obtained by "*injection*" into a completely specified vector (completely specified SAT solution). The most straightforward injection method will be to try to inject don't cares ("unassign" variable values) into a completely specified SAT solution, while checking whether the incompletely specified solution is still a satisfying one, under all assignments of don't cares. However, this would require the number of SAT-solver calls exponential with the number of injected don't cares. This makes this approach impractical.

In SAT-Compress, we can benefit from the nature of the problem. Indeed, we primarily require a cube that covers a particular fault as a solution. Hence, the above exponential satisfiability checking job can be accomplished by symbolic fault simulation [45], which can be conducted in polynomial time with the circuit size. This idea can be extended further more – we even need not insist on covering the fault the CNF was constructed for; a test cube covering *any other* fault is a no less valuable solution.

Naturally, the more specified the test pattern is, the more faults it covers. On the other hand, unspecified bits (don't cares) alleviate the constraints and thus maximize the possible overlap of subsequent patterns. Thus, there are two extreme cases here: either we can try to inject as many don't cares as possible, sacrificing the fault coverage of the test cube (which can drop to covering one fault only), or we can try to inject only don't cares retaining the original coverage. Even in the latter case, some don't cares can be injected. Any compromise between these two extremes can be used, by driving the injection by some factor. This will be denoted as *CL* in this paper, the *Coverage Loss*.

The don't cares injection algorithm is shown in Figure 7. It is supposed to be run after a completely specified SAT solution in Algorithm from Figure 4 is obtained (step 8). The SAT solution (*s*), the constraints cube (*c*), and the *CL* parameter are the inputs to this procedure.

First, the number of faults detected by *s* is determined by symbolic fault simulation (1) [45]. Then the test cube (*s*) is tried for don't care injection in a greedy way (5). Don't cares can be injected into positions allowed by the constraints cube only (4). If the number of faults covered by the resulting cube does not sink below the *CL* factor, the injection is made permanent (7). The procedure returns an expanded test cube, while its fault coverage is no more than by the *CL* factor less than the coverage of the original one.

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.

Note the two extremes: for *CL* = 0, no fault coverage drop is allowed. This technique will be referred to as *Coverage Preserving Don't Care Injection* (CPDCI) [38]. For *CL* approaching 1, maximum don't cares are injected, while the fault coverage may drop to one fault only.

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, at expense of losing fault coverage of the pattern. The issue of *CL* choice will be further discussed in the following section.

Note that the *CL* values in the experimental section will be represented in per cent units (0-100%).

4. Experimental results

The experimental results will be presented in this section. All the measurements were performed on a CPU i5-2400 3.1GHz with 8GB RAM. Atalanta [21] with Hope [45] was used for the fault list generation and fault simulation purposes, MiniSAT v1.14 [29] as the SAT solver, and MiniSAT+ v1.0 [37] was used as the PBO solver.

The experiments have been performed on a subset (170 benchmark circuits) of the ISCAS'85 [46], ISCAS'89 [47], ITC'99 [48], and LGSynth [49] benchmark circuits.

The result quality of most EDA processes based on local heuristics depends on random aspects coming from the input description [41], [42]. This issue is discussed and documented in Subsection 4.1. To diminish the influence of randomness on evaluation, most experiments were conducted repeatedly, with random initial patterns and random faults ordering, and the results were averaged.

4.1. Influence of randomness on the ATPG process

As stated before, there are many random aspects that influence the test generation process. First of all, it is the selection of the initial test pattern (tp_0 in Figure 4). 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 4, step 5). Different orderings of the fault list will induce different runs of the test generation heuristic. Also the order in which don't cares are injected (see Figure 7, line 3) influences the final bitstream length.

The influence of the initial test pattern is shown in Figure 8 and Figure 9 for two illustrative benchmark circuits (*c432* and *c880*). The algorithm was executed 5,000-times, each time with a randomly generated initial pattern. The frequencies of occurrence of the resulting bitstreams of different lengths (the x-axis) are shown, both for the basic SAT-Compress (Figure 4) and the SAT-Compress augmented with CPDCI.

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, thus randomness plays a smaller role here, making CPDCI *more robust* than the standard SAT-Compress [41]. Nevertheless, the influence of randomness is still crucial (even though reduced in the CPDCI case), and worse results can be obtained by CPDCI accidentally, see Figure 8, where the histograms overlap. Similar frequency distributions were observed for different orderings of faults and don't care injections.

4.2. Choice of the CL parameter in the simulation-based don't care injection

It is difficult to say intuitively, what *CL* values will produce best results. Low values preserve the fault coverage of test patterns, 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 the complete fault coverage, and thus the main loop will be shorter. 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 patterns will more likely overlap, however, more patterns would be probably needed to achieve the complete fault coverage.

While the influence of *CL* on the number 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 [46] benchmark circuit *c3540* are shown in TABLE III. Here the SAT-Compress algorithm was run with different values of the *CL* parameter and the absolute numbers of injected don't cares (*"DC injected"*), the absolute numbers of SAT instances solved (*"SATs"*), 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 and obtain more precise results (see Subsection 4.1).

These results are also visualized in Figure 10 - Figure 13, more precisely. Here the minimum, maximum, and average values from the 30 runs are shown.

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

The most important observation concerns the final bitstream length: the average bitstream length *monotonously increases* with *CL* (see Figure 12), with best results obtained for *CL* = 0, i.e., the CPDCI technique. Also the run-time decreases for low values of *CL* (Figure 13).

Similar experiments were performed on many other benchmark circuits and exactly the same behavior was observed in all cases. Results for some other ISCAS benchmarks are shown in TABLE IV. The final bitstream length only was measured. The data was obtained by averaging 30 runs with random initial patterns, the values are rounded to integer values.

This experiment has shown that no fault coverage of every single pattern should be sacrificed, even though more don't cares would be injected otherwise. Therefore, the usage of the CPDCI technique from [38] is fully justified; there is no need for looking for a compromise between the number of injected don't cares and the fault coverage.

Note that the extreme two cases, *CL* = 0 and *CL* $\rightarrow$ 100% represent the completely informed and completely uninformed don't care injection techniques, respectively.

4.3. Comparison of CPDCI and PBO

As a consequence of the previous experiments, uninformed SAT solvers producing don't care values, including the PBO-based technique from Subsection 3.2 seem to be compromised. Don't cares must be injected with care (in our case, the fault coverage is of the major concern), and definitely not only their number in the SAT solution must be the optimization criterion.

To confirm this, we have made a comparison with the PBO-based technique. The PBO technique produces test patterns with the real maximum of don't cares, however in an uninformed way, too. The results for some benchmark circuits [48], [49] and the three processes (uninformed don't care injection, CPDCI, and PBO) are shown in TABLE V. The final bitstream lengths and the average percentages of test don't cares (out of the number of PIs) are shown. The values were obtained from 1,000 randomized measurements (random initial pattern) and averaged (see Subsection 4.1). The average values over all the circuits are computed in the last table row.

We can see that even though maximum of don't cares is obtained by PBO, the resulting bitstreams are typically longer than those obtained by CPDCI.

4.4. Comparison of SAT-Compress variants

In this subsection we present a more thorough comparison of the original SAT-Compress algorithm (Figure 4) and SAT-Compress extended by don't care injection. The experiments were conducted using a subset of the ISCAS benchmarks [46], [47].

The results are shown in TABLE VI. After the circuit name, the number of its faults is given, as the measure of the circuit complexity. Then results of three variants of SAT-Compress are shown: the original SAT-Compress, i.e., without don't care injection, the uninformed don't care injection (*CL* $\rightarrow$ 100%), and the informed one (*CL* = 0, CPDCI). The absolute numbers of injected don't cares (where applicable), the lengths of the final bitstreams, relative bitstream length improvements w.r.t. the original SAT-Compress, and the compression times are shown. Average values are shown in the last table row.

We can see that the don't care injection helps to reduce both the final bitstream length and the test compression time. Even the uninformed don't care injection significantly reduces both in some cases. On the other hand, in some cases the injected don't cares make the solution worse. This indicates that the uninformed injection can sometimes do more bad than good. The average bitstream size reduction is positive, by 15%.

Finally, the CPDCI technique reduces the bitstream length by 31% on average and the deteriorating cases are only rare and not significant. This experiment justifies the CPDCI one last time.

4.5. 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 results for some selected circuits are shown in TABLE VII.

The first column of the table (*"*Circuit*"*) represents the name of the benchmark circuit. The second column *"Faults"* 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 lengths of the compressed bitstreams and the compression times are shown there 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 absolute numbers of care bits tried for DCs injection and the "*DCs inj.*" 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 it is based on the same principles (the RESPIN architecture). However, as well as RESPIN, 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 measured 170 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 success 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 cases where the extended SAT-Compress algorithm produced worse results, e.g. for the *c499* circuit. We assume that this is caused by the random noise introduced by the algorithm, as shown in Subsection 4.1.

The scalability of the method is naturally given by the circuit size, particularly the number of faults, but also by its efficiency – the more faults are detected in each algorithm step, the faster is the overall algorithm. Therefore, the CPDCI technique maximizing the number of covered faults in each step also significantly reduces the runtime.

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 large (except of some extreme cases, like *s35932*). This is probably due to a huge amount of randomness introduced into the ATPG process, as shown in Subsection 4.1. We can conclude that these two techniques are competitive.

5. Conclusions

The role of don't cares in the compressed test generation was studied in this paper. Several techniques for obtaining don't cares in the test are proposed, both uninformed and informed ones. We have shown that don't cares obtained in an uninformed way cannot be efficiently exploited in test compression and sometimes they even have disturbing effects.

The observations resulted in 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.

REFERENCES

[1] Semiconductor Industry Association, "The International Technology Roadmap for Semiconductors (ITRS)", 2013. 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] D. H. 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 Proc. of Design, Automation and Test in Europe Conference and Exhibition, 16-20 Feb. 2004, pp. 260-265.

[27] M. Abramovici, M. A. Breuer, and A. D. Friedman, “*Digital Systems Testing and Testable Design*”, Computer Science Press, 1990.

[28] A. Czutro, I. Polian, P. Engelke, S. M. Reddy, and B. Becker, “Dynamic Compaction in SAT-Based ATPG,” in Proc. of the 2009 Asian Test Symposium (ATS'09). IEEE Computer Society, Washington, DC, USA, pp. 187-190.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] C. Pizzuti, “Computing Prime Implicants by Integer Programming,” in 8th International Conference on Tools with Artificial Intelligence, 1996, pp. 332-336.

[36] E. Boros and P. L. Hammer, “Pseudo-Boolean optimization, in” Discrete Applied Mathematics, Volume 123, Issues 1–3, 15 November 2002, pp. 155-225.

[37] N. Éen, N. Sorensson, “Translating Pseudo-Boolean Constraints into SAT,” in Journal on Satisfiability, Boolean Modeling and Computation, vol. 2 2006, pp. 1-25.

[38] J. Balcárek, P. Fišer, and J. Schmidt, “Simulation and SAT Based ATPG for Compressed Test Generation,” in Proc. of 16th Euromicro Conference on Digital Systems Design (DSD), Santander (Spain), September 4-6, 2013, pp. 445-452.

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

[40] 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.

[41] A. Puggelli, T. Welp, A. Kuehlmann, and A. Sangiovanni-Vincentelli, “Are Logic Synthesis Tools Robust?,” in Proc. of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC), 5-9 June 2011, pp. 633-638.

[42] P. Fišer, J. Schmidt, and J. Balcárek, "Sources of Bias in EDA Tools and Its Influence," in Proc. of 17th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS), Warsaw (Poland), April 23-25, 2014, pp. 258-261.

[43] J.P. Roth, “Diagnosis of automata failures: A calculus and a method”, IBM J. Res. Develop., vol. 10,1966, p. 278.

[44] R.K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Boston, MA, Kluwer Academic Publishers, 1984, 192 p.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] K. McElvain, “LGSynth93 benchmark set: Version 4.0“, 1993.

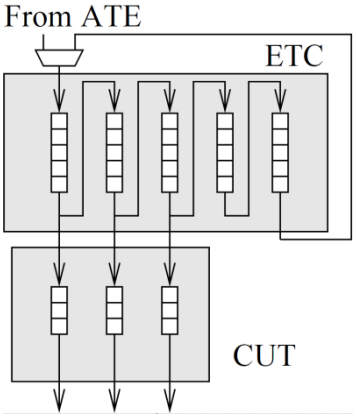

Figure 1. RESPIN architecture [11]

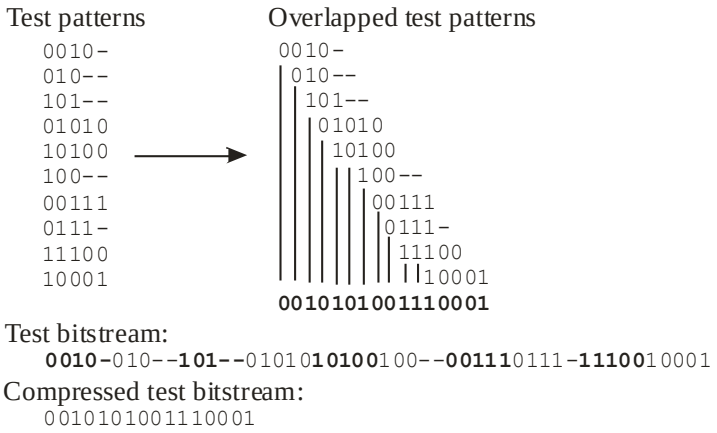

Figure 2. Patterns overlapping

```
RESPIN(circuit)
1   Generate test T for circuit
2   c = tp0
3   while (T ≠ ∅) {
4       for each t ∈ T {
5           if (c ∩ t  ≠ ∅) {
6               c = c ∩ t
7               T = T − t
8           }
9       }
10      bitstream += c[0]
11      c[0...n-2] = c[1...n-1]    // n is the number of primary inputs
12      c[n-1] = DC
13  }
14  bitstream += c[0...n-2]
15  return bitstream
```

Figure 3. The RESPIN algorithm

```
SAT-Compress(circuit)
  1   Generate FL for circuit
  2   FL = FL − Redundant_faults
  3   c = tp0
  4   while (FL ≠ ∅) {
  5       for each f ∈ FL {
  6           φ = Create_CNF(circuit, f)
  7           φ = Apply_constraints(c, φ)
  8           s = SAT(φ)
  9           if (s ≠ ∅) {
  10              c = Assignment_of_PIs(s)
  11              break
  12          }
  13      }
  14      bitstream += c[0]
  15      FL = FL − Detected_by_simulation(c)
  16      c[0...n-2] = c[1...n-1]        // n is the number of primary inputs
  17      c[n-1] = DC
  18  }
  19  bitstream += c[0...n-2]
  20  return bitstream
```

Figure 4. The SAT-Compress algorithm

```
CNFlib(gate)
1  // generate the characteristic function
2  minimize gate                // espresso
3  add the off-set to gate      // espresso
4  move the output column to input columns of gate
5  put all 1s into the output column of gate
6  // calculate intersections
7  do {
8     for each unordered pair (t1, t2) of terms from gate {
9       let s be the intersection of t1[y] and t2[y]
10      if (s == U) {
11      let t be the intersection of t1 and t2
12      if t is not in gate
13             insert t into gate with output symbol 1
14      }
15    }
16 } while new terms are added to gate
17 // encode and convert to Boolean domain
18 encode gate using TABLE II. giving F
19 // produce CNF
20 turn F into off-set description   // espresso
21 for each term t in F {
22    start a new clause
23    for each column label j {
24      if (t[j] == 0) output j
25      if (t[j] == 1) output ¬j
26    }
27 }
```

Figure 5. An algorithm generating the CNF characteristic function of a library gate for gate-to-CNF transformation with encoded undefined values

x_1	x_2	y
0	0	1
1	0	1
0	1	1

a)

x_1	x_2	y	
-	0	1	1
0	-	1	1
1	1	0	1

b)

x_1	x_2	y	
-	0	1	1
0	-	1	1
1	1	0	1
U	1	U	1
1	U	U	1
U	U	U	1

c)

x_1^V	x_1^A	x_2^V	x_2^A	y^A	
-	0	0	1	1	1
0	1	-	0	1	1
1	1	1	1	1	1
-	0	1	1	0	1
1	1	-	0	0	1
-	0	-	0	0	1

d)

$$\left(x_1^A + x_2^A + \overline{y^A}\right)\left(x_2^V + \overline{x_2^A} + y_A\right)\left(x_1^A + \overline{x_2^V} + \overline{y^A}\right)$$
$$\left(x_1^V + \overline{x_1^A} + \overline{x_2^A}\right)\left(x_1^V + \overline{x_1^A} + y^A\right)\left(\overline{x_1^V} + x_2^A + \overline{y^A}\right)$$
$$\left(\overline{x_1^A} + x_2^V + \overline{x_2^A}\right)\left(\overline{x_1^A} + \overline{x_2^A} + y^A\right)$$

e)

Figure 6. Example of a) *nand2* function b) its characteristic function, c) with undefined values propagation, d) encoded F^A, e) F^A in CNF

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

Figure 7. The don't cares injection algorithm

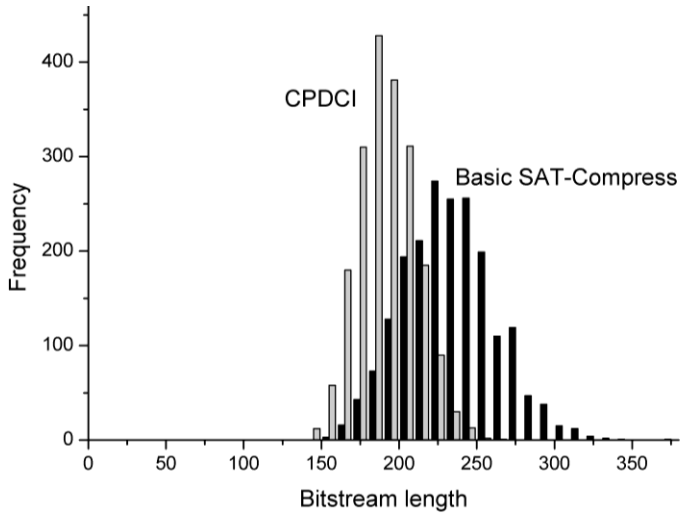

Figure 8. Frequency of bitstream lengths distribution (*c432*)

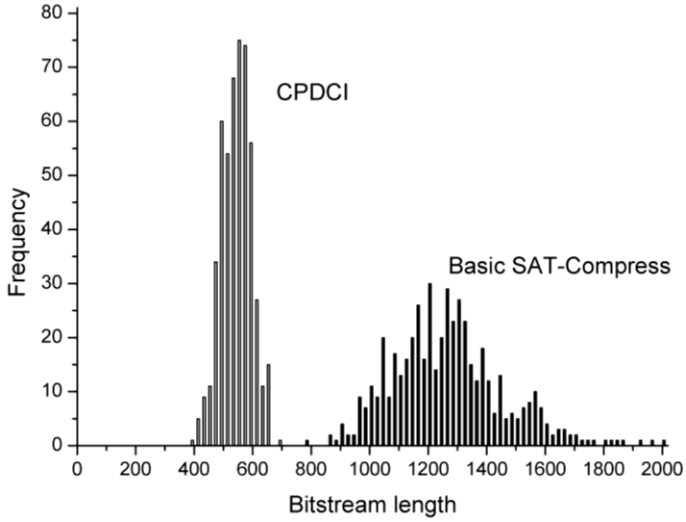

Figure 9. Frequency of bitstream lengths distribution (c880)

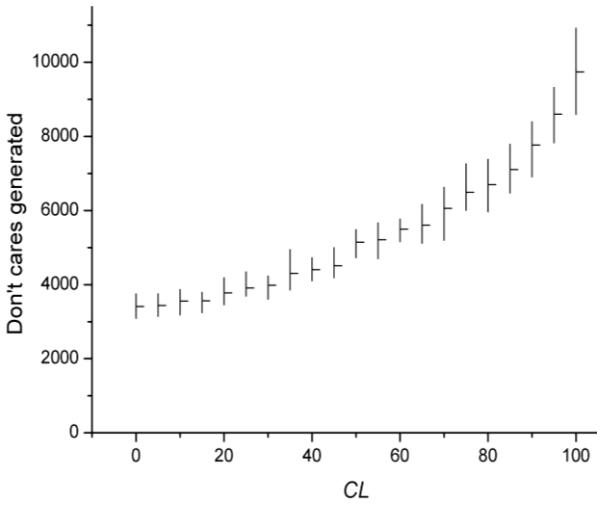

Figure 10. Influence of *CL* on the total number of injected don't cares

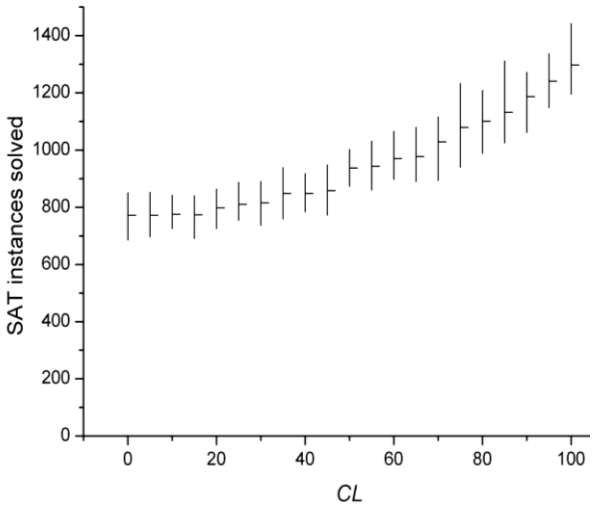

Figure 11. Influence of *CL* on the total number of SAT instances solved

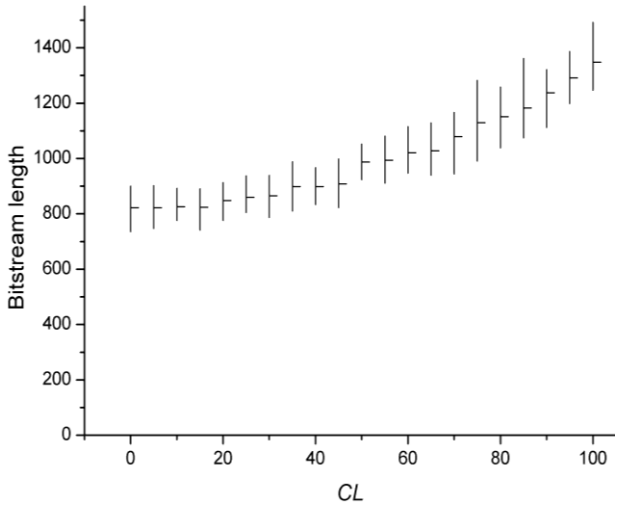

Figure 12. Influence of *CL* on the generated bitstream length

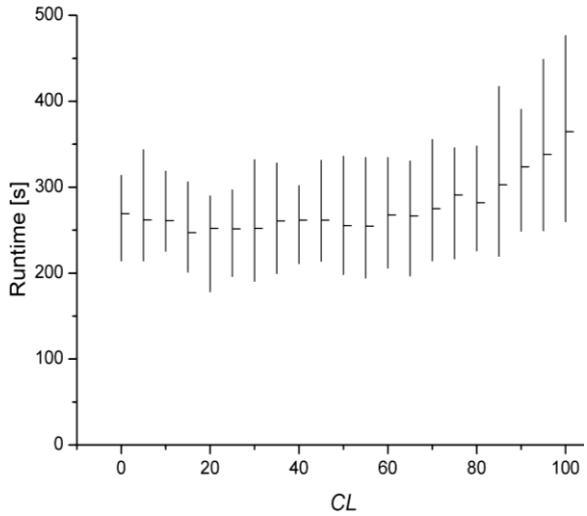

Figure 13. Influence of *CL* on the compression run-time

TABLE I. LITERALS TRANSCRIPTION

x_i	x_i^V	x_i^A
0	0	1
1	1	1
unspecified	any	0

TABLE II. SYMBOL INTERSECTION

	0	1	-	U
0	0	U	0	U
1	U	1	1	U
-	0	1	-	U
U	U	U	U	U

TABLE III. INFLUENCE OF LOSING FAULT COVERAGE

CL [%]	*DCs injected*	*SATs*	*Bits*	*Time* [s]
0	3410.7	772.0	822.0	269.3
5	3436.7	771.8	821.8	262.1
10	3555.6	775.5	825.5	261.0
15	3563.1	773.5	823.5	247.2
20	3774.9	798.2	848.2	252.0
25	3913.4	810.0	860.0	251.6
30	3985.1	815.4	865.4	252.1
35	4303.1	848.6	898.6	260.9
40	4406.7	848.7	898.7	261.8
45	4512.9	857.9	907.9	261.6
50	5145.0	937.4	987.4	255.3
55	5210.8	943.4	993.4	254.6
60	5497.9	970.5	1020.5	267.9
65	5604.0	977.7	1027.7	266.7
70	6061.7	1028.8	1078.8	275.1
75	6493.1	1079.3	1129.3	291.0
80	6701.7	1101.3	1151.3	282.0
85	7108.0	1132.6	1182.6	302.8
90	7768.5	1187.1	1237.1	323.7
95	8598.3	1241.0	1291.0	338.3
$\rightarrow$ 100	9743.6	1297.6	1347.6	364.5

TABLE IV. INFLUENCE OF LOSING FAULT COVERAGE – FINAL BITSTREAM LENGTHS FOR MORE CIRCUITS

Circuit / *CL* [%]	0	10	20	30	40	50	60	70	80	90	100
br1	503	509	518	550	575	623	649	685	716	746	754
c1908	545	538	540	542	551	551	551	560	563	577	609
c2670	1778	1800	1850	1869	1891	1927	1948	1927	1945	2021	2030
c3540	822	826	848	865	899	987	1021	1079	1151	1237	1348
c5315	773	772	774	764	760	783	795	823	866	921	1179
c7552	3608	3620	3630	3700	3706	3948	3934	3971	4080	4195	4398
chkn	3611	3636	3677	3798	3964	4526	4604	4921	5310	5558	5617
comp	559	555	563	562	585	613	615	624	655	650	655
duke2	1010	1018	1043	1072	1080	1146	1177	1201	1262	1310	1349
example2	630	622	633	643	647	685	691	709	728	758	778
frg1	2156	2171	2186	2202	2246	2349	2379	2459	2544	2584	2635
i2	5179	5173	5157	5185	5225	5723	5730	5735	5795	5747	5747
i4	898	899	904	908	911	918	930	949	959	950	962
in0	790	814	840	863	900	931	958	965	980	1008	1025
in2	731	733	730	735	742	776	764	787	822	875	923
in4	865	863	868	899	917	957	990	1044	1085	1145	1202
in5	701	709	731	751	816	930	978	1048	1135	1206	1235
in6	993	1004	1003	1017	1022	1060	1062	1079	1111	1137	1151
jbp	840	856	849	870	874	913	928	947	993	1033	1080
s1196	862	864	866	878	900	932	935	949	961	976	1011
s1238	886	886	887	902	925	942	935	963	982	1002	1000
s1423	608	626	635	645	684	719	718	746	781	804	814
s420.1	600	595	605	640	627	663	675	707	714	732	741
s5378	1920	1933	1954	1982	2030	2140	2151	2203	2275	2301	2344
s820	669	668	669	673	672	705	712	726	747	783	821
s832	665	651	661	673	660	687	694	722	746	771	807
s838	986	978	1022	1003	1036	1190	1238	1262	1302	1358	1337
s838.1	1620	1638	1668	1655	1657	1810	1848	1800	2002	1979	2022

TABLE V. COMPARISON OF CPDCI AND PBO

Circuit	*SAT-Compress, CL* → 100%		*SAT-Compress, CL* = 0 (CPDCI)		*PBO*	
	Bits	*DCs* [%]	*Bits*	*DCs* [%]	*Bits*	*DCs* [%]
5xp1	155.58	8.56	102.32	0.52	106.82	1.82
b03_C	139.29	12.7	113.12	2.31	111.71	5.80
b06_C	39.78	30.75	35.27	27.23	37.45	39.61
b9	271.23	9.11	215.10	4.14	217.76	9.94
c1355	292.18	1.80	265.57	0.29	264.09	0.79
c432	245.26	9.50	185.93	5.11	193.58	13.70
c499	222.66	2.18	198.28	0.29	200.74	0.90
c8	363.39	24.05	276.09	15.00	281.74	36.32
dc2	133.97	14.44	91.24	8.70	98.20	16.65
f51m	273.02	12.44	163.93	3.35	167.55	5.93
cht	156.09	6.76	133.98	4.94	138.03	6.77
i1	189.25	8.74	152.75	6.10	164.25	21.36
i3	392.22	8.55	349.81	4.83	361.34	13.15
average	221.07	11.51	175.65	6.37	180.25	13.29

TABLE VI. SAT-COMPRESS VARIANTS

Circuit	*Faults*	No don't cares		*CL* → 100%				*CL* = 0 (CPDCI)			
		Bits	*Time* [s]	*DCs*	*Bits*	*Impr.*	*Time* [s]	*DCs*	*Bits*	*Impr.*	*Time* [s]
c1355	1566	330	29.3	159	367	-11%	31.3	19	334	-1%	28.8
c1908	1869	660	121.6	371	583	12%	106.2	92	538	18%	110.5
c2670	2629	3758	1714.1	18934	1977	47%	738.3	9905	1874	50%	1050.0
c3540	3291	3188	3600.4	9711	1426	55%	1075.2	3358	774	76%	834.6
c432	520	209	3.0	460	185	11%	2.7	234	176	16%	2.9
c499	750	182	3.1	154	241	-32%	3.8	28	219	-20%	3.5
c5315	5291	1194	698.0	19137	1207	-1%	600.3	1750	795	33%	724.5
c7552	7419	6408	7856.4	21741	4314	33%	4149.1	6030	3819	40%	5878.1
c880	942	1134	107.1	3747	702	38%	45.2	1465	458	60%	25.9
s1196	1242	2430	262.1	5993	1066	56%	117.0	3553	864	64%	81.2
s1488	1486	502	61.0	170	662	-32%	96.2	11	557	-11%	63.3
s208	215	185	0.9	329	207	-12%	0.9	129	195	-5%	0.7
s298	308	243	1.2	613	166	32%	1.0	337	140	42%	0.8
s349	348	127	1.1	620	155	-22%	1.2	215	108	15%	1.1
s400	418	212	1.4	654	205	3%	1.2	230	144	32%	0.8
s510	564	180	1.7	126	232	-29%	2.2	2	165	8%	1.6
s641	463	1186	27.5	4223	613	48%	20.2	1972	493	58%	13.9
s713	543	1456	35.3	4440	598	59%	18.3	1894	464	68%	14.3
s820	850	700	25.7	142	855	-22%	38.8	65	664	5%	23.4
s953	1079	3360	322.9	5812	959	71%	78.6	3693	771	77%	48.4
average	1590	1382	743.7	4877	836	15%	356.4	1749	678	31%	445.4

TABLE VII. EXPERIMENTAL RESULTS FOR THE BASIC SAT-COMPRESS ALGORITHM AND CPDCI

Circuit	*Faults*	SAT-Compress		SAT-Compress with CPDCI							COMPAS	
		Bits	*Time* [s]	*Bits*	*Bits Impr.* [%]	*Time* [s]	*Time impr.* [%]	*DCs tried*	*DCs inj.*	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	-1.21%	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	-11.51%
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	-20.33%	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	-13.70%
duke2	1302	1486	56.80	986	33.65%	35.13	38.15%	1717	810	47.18%	-	-
ex5p	5430	276	38.72	276	0.00%	42.12	-8.78%	268	0	0.00%	-	-
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	-18.38%
s1238	1286	2705	141.46	876	67.62%	40.06	71.68%	4926	4105	83.33%	741	-18.22%
s13207	9664	114390	285075.00	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	-5.37%
s15850	11336	77582	147342.00	5734	92.61%	22686.30	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	-11.76%
s35932	35110	3686	308382.00	4998	-35.59%	390677.00	-26.69%	2971215	2969101	99.93%	1860	-168.71%
s382	399	255	0.61	131	48.63%	0.39	36.07%	258	161	62.40%	123	-6.50%
s420	430	526	2.81	370	29.66%	1.62	42.35%	748	463	61.90%	352	-5.11%
s526n	553	830	5.27	471	43.25%	2.70	48.77%	1197	785	65.58%	344	-36.92%
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	-18.14%
s713	543	1223	11.64	454	62.88%	6.08	47.77%	2199	1859	84.54%	428	-6.07%
s820	850	702	10.30	664	5.41%	9.97	3.20%	692	65	9.39%	460	-44.35%
s838	857	2078	32.20	955	54.04%	19.27	40.16%	2957	2242	75.82%	920	-3.80%
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	-6.64%
t481	2853	5541	1808.29	5147	7.11%	1561.09	13.67%	5433	304	5.60%	-	-
table3	2487	2025	382.11	2085	-2.96%	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%	-	-
average	3396	7649	23209.59	1585	46.31%	13907.19	35.18%	86341	85018	65.54	1981	6.14%