4th IEEE International Symposium on Electronic Design, Test & Applications

Test Set Stripping Limiting the Maximum Number of Specified Bits

Michael A. Kochte, Christian G. Zoellin, Michael E. Imhof, Hans-Joachim Wunderlich
Institut fuer Technische Informatik
Universitaet Stuttgart, Germany
{kochte, zoellin, imhof, wu}@iti.uni-stuttgart.de

Abstract

This paper presents a technique that limits the maximum number of specified bits of any pattern in a given test set. The outlined method uses algorithms similar to ATPG, but exploits the information in the test set to quickly find test patterns with the desired properties. The resulting test sets show a significant reduction in the maximum number of specified bits in the test patterns. Furthermore, for commercial ATPG test sets even the overall number of specified bits is reduced substantially.

Keywords: Test relaxation, test generation, tailored ATPG

1 Introduction

With test data volume being a major concern in manufacturing tests, test compression has become a widely used technique. Most test compression techniques achieve better results if the patterns of the test set are partially specified. For built-in self test (BIST) techniques based on reseeding [1], [2], [3], [4] and test set embedding [5], [6] a partially specified test set is essential.

In reseeding, a linear equation is derived from each specified bit in a pattern. The solution of the equation system determines the seed. The higher the number of equations, the higher the probability that the equation system is inconsistent [7] and a seed cannot be computed. For acceptable probability of consistency the length of the LFSR should be chosen to be $s_{max}+20$, where s_{max} is the maximum number of specified bits in any pattern [2].

In test set embedding, a given LFSR stream is modified by bit-flipping [5] or bit-fixing [6]. In order to reduce the hardware overhead, a pattern in the LFSR stream is chosen which is similar and requires the least modifications. If the number of specified bits in a pattern is very high, the likelihood decreases that a similar pattern is found and as a result, the bit-flipping function grows considerably.

As a consequence, automatic test pattern generation (ATPG) methods tailored to a specific compression architecture have been proposed: To efficiently encode the patterns with a given LFSR, [8] presents an ATPG algorithm that generates patterns with a strictly limited number of specified bits. For test set embedding the correlation between individual patterns can be increased by constraining the ATPG process [9].

Even if the uncompacted test set has been optimised for the compression method, a highly compacted test set usually contains a significantly lower *overall* number of specified bits. An abundance of compaction methods has been proposed (e.g. [10], [11], [12]), and most commercial ATPG tools support sophisticated test pattern compaction. Consequently, these test sets are preferred in real-world applications with test compression [3], because they still result in the lowest overall test data volume. In this case, a highly compacted test set may be transformed in the desired ways. For example, [13] reduces the amount of specified bits in the test patterns by splitting excessively large patterns. However, the number of patterns in the test set and the test time may increase significantly.

Kajihara et. al. presented the XID test set stripping method [14], [15] that identifies a large amount of over-specified bits in a completely specified test set. The method relies on path tracing and justification with 3-valued logic. Another method which uses a similar approach has been presented in [16]. Here, a number of heuristics were evaluated to guide the path tracing and justification process. These methods do not take into account the maximum number of specified bits in a pattern and some of the resulting patterns still contain a very high amount of specified bits.

In this paper, we present a method for limiting the number of specified bits in highly compacted test sets. As a side-effect, the method uncovers a significant amount of over-specified bits in excess of the number of already unspecified bits in the original test set. The method extends the ideas outlined in [15] by the use of nine-valued logic and an implication graph, which provide higher accuracy when processing partially specified test sets. Furthermore, we work in the context of the decision tree that would be generated during ATPG. In contrast to the method in [13], the process does not increase the number of test patterns in the test set. Compared to the method in [15], the maximum number of specified bits in any pattern is reduced up to a factor of 4. The proposed method can be combined with any commercial ATPG or any constrained ATPG technique [17], [18], [9].

The remainder of the paper is organised as follows:

In the following section, a short formal definition of the problem and an algorithmic overview is given. In section 3, the data structures and algorithms to handle individual faults are explained in detail, which is followed by the heuristic that assigns target faults to patterns. Section 5 discusses the result for experiments with a number of test sets for ISCAS and ITC circuits.

2 Problem Definition and Algorithmic Overview

In this section, we provide a formalisation of the problem and outline the presented algorithm. Besides limiting the number of specified bits in a pattern, we define a number of conditions to be met. Given a set of completely or incompletely specified test patterns T for a combinational or full-scan circuit that targets a set of stuck-at faults F. Compute an incompletely specified set of patterns T' from T such that the following holds:

1) T' detects F
2) The number of patterns in T' is smaller or equal the number of patterns in T
3) The number of specified bits in each pattern of T' does not exceed a given limit.

In these bounds, we define the secondary goal that the overall amount of specified bits in T' is reduced as much as possible.

We accomplish this task in the following manner: To determine those bits in a pattern that are required to detect a fault, we use the method $mark\_bits()$ as described in section 3. If a fault $f \in F$ is detected by only one test pattern $t \in T$, t is essential with respect to f. For brevity, f shall be called an essential fault. For all essential faults, we directly use $mark\_bits()$ to find the bits to be specified. For faults that are detected by more than one pattern in T, we first have to choose a suitable pattern. For this, we present the heuristic top-level algorithm in section 4 which employs the $mark\_bits()$ method.

In addition to the structural approaches presented in [15], [16], $mark\_bits()$ makes use of knowledge that would be generated during an ATPG run for the circuit. During pattern generation, an ATPG algorithm builds up a decision tree to satisfy the detection of the target fault. Nodes in the tree denote decisions, usually assignments to inputs of the circuit. A leaf node of the tree denotes if a solution has been found (success) or not. Hence, a path along decisions from the root node to a leaf node represents a set of assignments or specified bits. A pattern known to detect the target fault determines one path in the decision tree to a success leaf node. In this context, reducing the number of specified bits in that pattern is equivalent to finding another path to a success leaf node that contains only a subset of the original decisions.

In order to generate the ATPG decision information, we formulate the problem of fault detection as an instance of the satisfiability problem [19]. We extract a set of sufficient requirements for fault detection, i.e. for fault excitation and propagation to an output. During justification of the requirements, we reach decision points. In contrast to regular ATPG, we do not branch but take the decision that complies with the given pattern. Note, that compared to ATPG the presented algorithms will never search for a solution in a non-solution area of the decision tree since we derive our decisions from a pattern known to detect the targeted fault. Thus, we never backtrack and avoid the exponential worst case effort of ATPG.

Since this algorithm uses heuristics when selecting a set of satisfying decisions, it may yield a set of assignments that is still over-specified. We extend this search by the $skip\_decisions()$ method, which looks for local shortcuts in the decision tree to skip redundant decisions and further reduce the number of specified bits.

3 Identification Of Care Bits

If the value of a bit in a test pattern cannot be flipped without losing fault coverage it is called a *care bit*. To implement the search for care bits in a pattern we use the implication graph (IG), a data structure for SAT problems on logic circuits first proposed by Tafertshofer et. al. [20]. In the following, we present the algorithms that implement the ideas outlined in the previous sections. The methods $imply()$, $justify()$ and $propagate()$ implement the ATPG that is restricted to the solution space in the given pattern. Using these methods, the algorithm $mark\_bits()$ is defined.

3.1 Circuit Representation by the Implication Graph

The IG can take advantage of structural properties of logic circuits when solving SAT-instances, and it has been successfully employed for a number of EDA problems, such as ATPG, equivalence checking and netlist optimization [21], [22].

An IG is defined as a directed graph $G = (V, E)$, where the set of nodes V is partitioned into *signal nodes* V_s and *conjunctive nodes* V_c. A node can take one of two states, either *set* or *unset*. Signal nodes represent the literals in the set of clauses of the SAT problem. Table 1 shows the encoding of a variable $x \in L_3$ into two signal nodes. A set of 3 conjunctive nodes is used to model a ternary clause in the SAT instance.

x	$\neg x$	value
0	0	X
0	1	0
1	0	1
1	1	conflict

Table 1: Signal encoding for L_3

Further, we distinguish forward and backward edges in the set of edges E in order to incorporate structural information. Forward edges represent all implications from input signals to output signals of each gate in the logic circuit. Inversely, backward edges model implications in the opposite direction. Figure 1 shows the IG for a two-input AND gate with inputs a, b and output c.

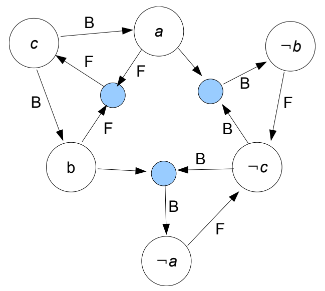

Fig. 1. Graph representation of an AND gate

The model in this paper uses a split circuit representation with two separate implication graphs IG and $IG^* = (V^*, E^*)$. The first one models the fault-free circuit and the latter one models the faulty circuit. This allows to implement the nine-valued logic using four signal nodes to encode a variable. Using the nine-valued logic we avoid the penalty of the extended implication method required in [15]. Furthermore, we may find a smaller set of input assignments in cases when the fault is propagated through a reconvergent fan-out. In figure 2, signal a requires a specific value in the five-valued logic, but in fact it does not need to be specified because of the known value in the faulty circuit.

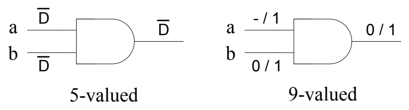

Fig. 2. Example for 5-valued and 9-valued logic

Let the subset $V_{PI} \subset V \cup V^*$ denote the signal nodes of both graphs that encode the primary inputs of the circuit.

We will use the following algorithms on the IG to find care bits in patterns:

Imply: The role of the $imply()$ method is to find those boolean implications in the IG that result from a value assignment of a signal in the circuit corresponding to the IG. In the IG, assigning a signal s a binary value corresponds to setting the proper signal node v_s or $\neg v_s$. The $imply()$ method is then a partial traversal of the IG along the implication edges starting at the signal node v_s to be set. The transitive successors v of the initial signal node v_s are set

- if $v \in V_S$ and it has a predecessor that is set or
- if $v \in V_C$ and all predecessors of v are set.

The implication stops at conjunctive nodes for which not all of the predecessors are set. The corresponding clause is called unjustified if that conjunctive node was reached from a signal node v along a backward edge. Hence, the unjustified clause in the IG represents the unjustified signal line encoded by v in the circuit.

Justify: Justification of unjustified signals in the circuit is performed by the $justify()$ method as outlined in Alg. 1. Justify relies on the $backtrace()$ method to determine a primary input that helps to justify a certain clause. The $backtrace()$ method takes a signal node v and finds an unset signal node $u \in V_{PI}$ which helps to justify the given node v and does not conflict with the current pattern under analysis. This is done by a depth-first traversal of the graph following only backward edges. In case $backtrace()$ finds multiple backward edges at a node, it orders them according to a controllability heuristic and follows the one easiest to control. Once $backtrace()$ reaches a node $v \in V_{PI}$ it returns this leaf node only if it encodes the value that is stored in the given pattern. In case that input is unspecified in the original pattern, $backtrace()$ also continues the search. Thus, $backtrace()$ finds only nodes that are known to justify the clause, so that no search for a non-conflicting solution and no backtracking is required.

While there are unjustified clauses, $justify()$ runs the $backtrace()$ method from the corresponding unjustified signal node to obtain a PI assignment, implies it and checks whether the objective signal has been justified. $justify()$ terminates if all clauses have been justified and hence the underlying SAT instance is satisfied.

Algorithm 1 justify
```
while ∃ unjustified clause $c$ with $v_c \in V_s$ do
  repeat
    $u \leftarrow backtrace(v_c)$
    $imply(u)$
  until $c$ is justified
end while
```

Propagate: Besides sensitisation of the fault site it is necessary to propagate the fault impact to an output of the circuit. It is straightforward to extract all propagating paths $\mathcal{P}_{sen,v}$ that detect the fault with the given pattern from fault simulation. The path with the lowest cost according to an observability heuristic is activated in both IG and IG^* by implying complementary values along the path. The implication process causes some clauses to be unjustified which are required for sensitisation of the path. By executing the $justify()$ method, these clauses are justified and all the necessary input assignments are determined.

3.2 Finding Care Bits

The method $mark\_bits()$ identifies the care bits in a given pattern targeting fault f. The method first implies all already marked bits in the pattern exploiting the current stripping state. Then it activates the fault

Algorithm 2 propagate(v)

```
select propagation path $P \in \mathcal{P}_{sen,v}$
for all $v_i \in P$ do
    $imply(v_i)$
    $imply(\neg v_i^*)$
    $justify$ in $IG$ and $IG^*$
end for
```

f : stuck at v_f by setting its complementary signal node $\neg v_f$ in the good circuit IG and the signal node of the stuck-at fault v_f^* in the faulty circuit IG^*. After executing the $propagate()$ method all nodes in the two implication graphs are set such that the fault is excited and propagated to one of the outputs. The method $skip\_decisions()$ checks all the decisions, and removes redundant decisions that can be undone while still satisfying all clauses. Finally, the primary inputs that are assigned a value in at least one of the implication graphs determine the specified bits in the pattern and the bits are marked as such.

Algorithm 3 mark_bits(f,t)

```
$pattern = t$
imply all marked bits in $t$
$imply(v_f)$
$imply(\neg v_f^*)$
$propagate(v_f)$
$skip\_decisions()$
for all $v_i \in V_{PI}$ do
    if $v_i$ is set then
        The bit corresponding to $v_i$ is marked in $t$
    end if
end for
reset all nodes
return number of marked bits in $t$
```

4 Test Set Stripping

To restrict the maximum number of specified bits per pattern according to section 2, the test set is processed in four passes which employ the $mark\_bits()$ method from the previous section. After processing the essential faults of each pattern, we use a heuristic approach to select for each remaining fault a pattern to detect it within the given restrictions. If we fail to find such a pattern, that fault will be dealt with in a final step.

We propose the top-level flow as in Alg. 4 to strip a test set T targeting all faults in F. The parameter $limit$ denotes the targeted upper bound on the number of specified bits in the patterns. Further, we assume a given ordering on the patterns in T.

Let μ_T be a function that assigns each pattern $t \in T$ a set of faults $f \in F$ detected by t for the first time w.r.t. the order of T. Then accordingly $\mu_{T_{Rev}}$ shall denote this mapping based on the reverse order of T.

The top-level algorithm processes the patterns in passes, reducing the number of undetected faults in each pass. For this purpose we define the method

Algorithm 4 top_level($T, F, limit$)

```
determine $\mu_T, \mu_{T_{Rev}}, F_{Ess}$ by fault simulation
$F' = \emptyset$
$strip\_test\_set(T, F_{Ess}, F', \mu_T, \infty)$
$strip\_test\_set(T_{Rev}, F, F', \mu_T, limit)$
$strip\_test\_set(T, F, F', \mu_{T_{Rev}}, limit)$
$strip\_all\_detect(T, F, F', limit)$
for all $f \in F \setminus F'$ do
    $mark\_bits(f, t)$ with $t$ having minimum $cost(t, f)$
    $F' = F' \cup$ fault simulation on marked bits of $t$
end for
return $T'$ containing only marked bits from $T$
```

$strip\_test\_set()$ as in Alg. 5. It uses $mark\_bits()$ to find the care bits in a pattern such that it detects the targeted faults. The limit imposed on the number of specified bits $\#specBits$ is taken into account as follows: For every pattern t, $mark\_bits()$ analyses the targeted faults f of t which are not covered yet. If $\#specBits$ does not exceed the limit, we mark f detected. Additional faults may be detected by subsequent fault simulation of the stripped pattern.

Algorithm 5 strip_test_set($T, F, F', \mu, limit$)

```
for all $t \in T$ ordered do
    $F_t = \{f \in (F \setminus F') \cap \mu(t)\}$
    for all $f \in F_t$ do
        $\#specBits_f = mark\_bits(f, t)$
        if $\#specBits_f \leq limit$ then
            $F' = F' \cup f$
        else
            undo marking of bits
        end if
    end for
    $F' = F' \cup$ fault simulation on marked bits of $t$
end for
```

For all essential faults F_{Ess} the care bits are marked without restricting $\#specBits$, since otherwise fault coverage would be impacted. In a second step, the patterns are processed in reverse order so that the hard faults get processed first. As only patterns with $\#specBits \leq limit$ are accepted, some faults may remain undetected. In the third pass, T is processed in forward order, again giving priority to hard faults. After this step every non-essential fault has been targeted by at most two patterns.

Now the search for a pattern is extended to all patterns detecting the fault (c.f. Alg. 6). For each failed attempt, Alg. 6 keeps track of the cost of detecting f using t. None or very few faults remain undetected after this step. For the remaining faults, the imposed limit may be violated since we give higher priority to fault coverage. The negative impact of these faults is limited, as they are targeted by the patterns that detect them at minimal cost.

5 Results

The proposed algorithm has been evaluated using the well-known ISCAS and ITC benchmark circuits.

Algorithm 6 strip_all_detect($T, F, F', limit$)

```
for all $t \in T$ do
    for all $f \in F \setminus F' \mid f$ detected by $t$ do
        $\#specBits_f = mark\_bits(f, t)$
        if $\#specBits_f \leq limit$ then
            $F' = F' \cup f$
        else
            undo marking of bits
            $cost(t, f) = \#specBits_f$
        end if
    end for
    $F' = F' \cup$ fault simulation on marked spec bits of $t$
end for
```

First, we show the influence of the parameter $limit$. Then we show results for various test sets for ITC and ISCAS circuits.

The experiments for the ISCAS '85 and '89 circuits have been performed using the highly compacted test sets which were used in [15] and have kindly been provided by the authors. These test sets are generated using the efficient compaction technique presented in [12].

To show the impact of the parameter $limit$ on the performance of the algorithm, we conduct the stripping process with three different limits. These results are presented in table 2. For each circuit we give the number of primary inputs (*#PIs*) and the number of patterns in the test set (*#tests*). Column *Ess.* gives the maximum number of specified bits in the patterns when only considering the essential faults. This is the practical optimum that can be achieved with the presented algorithm, independent of the limit that is targeted. For the different values of $limit$, column *#limit* contains the limit in terms of bits per pattern. For circuits where the maximum cannot be reduced to the given limit, column *#vio* shows the number of patterns that violate the limit. The overall number of specified bits in the stripped test set is given in column *#all*.

From table 2 it can be observed that if the targeted limit cannot be achieved, it is always due to the essential faults. In these cases the desired limit cannot be achieved by design since fault coverage should be retained for the resulting test set. Yet even though the test set is highly compacted, we are able to achieve limits as low as 20%. Remarkably, the overall number of specified bits is only marginally impacted by choosing a stricter limit.

For the ITC benchmark circuits, test sets have been generated using three commercial ATPG tools at their highest compaction effort. The test sets have been analysed in arbitrary order and in an anonymous manner. Here, the limit has been chosen as low as possible such that it is not violated by any pattern. The results are given in table 3. The columns *Test Set 1-3* contain the information for the source test set, where *#max* denotes the maximum number of specified bits. The column *Stripped* gives the results after test set stripping.

One of the ATPG tools did not allow to export partially specified test sets, but nevertheless the results after stripping closely match the results for the other test sets. The following remarks do not consider this test set. Regarding our primary goal, the method reduces the maximum number of specified bits in any pattern by at least 39% and up to 51%. The overall

Circuit	*#PIs*	*#tests*	*Ess.*	*limit=20% #PIs*			*limit=40% #PIs*			*limit=60% #PIs*		
			#max	*#limit*	*#vio*	*#all*	*#limit*	*#vio*	*#all*	*#limit*	*#vio*	*#all*
	[bits]	[pat]	[bits]	[bits]	[pat]	[bits]	[bits]	[pat]	[bits]	[bits]	[pat]	[bits]
c1908	33	106	31	7	106	2756	14	90	2750	20	75	2714
c2670	233	45	74	47	45	3088	94	0	3036	140	0	3004
c3540	50	93	30	10	90	2114	20	61	2114	30	0	2120
c5315	178	186	97	36	44	3173	72	21	3190	107	0	3165
c6288	32	14	32	7	14	448	13	14	448	20	14	448
c7552	207	75	139	42	72	6798	83	42	6794	125	9	6814
s5378	214	100	92	43	60	5746	86	16	5677	129	0	5356
s9234	247	111	98	50	102	8449	99	0	8457	149	0	8448
s13207	700	235	52	140	0	11255	280	0	11067	420	0	10977
s15850	611	97	187	123	42	11379	245	0	11081	367	0	10888
s35932	1763	12	1763	353	11	13497	706	9	13497	1058	7	13497
s38417	1664	87	442	333	75	36284	666	0	35332	999	0	34828
s38584	1464	114	404	293	58	32566	586	0	31752	879	0	31484

Table 2: Impact of $limit$

Circuit	*#PIs*	*Test Set 1*		*Stripped*		*Test Set 2*		*Stripped*		*Test Set 3*		*Stripped*	
		#max	*#all*	*#limit*	*#all*	*#max*	*#all*	*#limit*	*#all*	*#max*	*#all*	*#limit*	*#all*
			[10^3]		[10^3]		[10^3]		[10^3]		[10^3]		[10^3]
b14	277	277	75.3	164	50.1	265	51.6	154	44.7	277	233.2	111	49.4
b17	1452	1386	149.1	754	95.2	1452	1862.9	169	113.3	1399	124.3	681	100.7
b18	3357	3357	4934.8	547	450.5	2562	525.3	1467	412.8	3213	622.0	1873	398.1
b19	6666	3876	1134.9	2354	884.5	6342	1286.8	3672	806.7	6666	10865.6	1028	932.1
b20	522	522	154.6	321	99.5	491	110.0	286	89.9	522	530.4	224	112.0
b21	522	522	551.2	204	119.5	513	117.2	285	95.1	522	164.1	286	107.3
b22	767	752	148.4	423	120.0	767	190.3	378	128.0	767	738.6	262	155.1

Table 3: Three compacted ATPG test sets for ITC benchmarks

number of bits is reduced for all circuits by up to 35% and no less than 13%. This strongly hints at the necessity to apply test set stripping as a post-processing step, since even sophisticated ATPG techniques result in over-specified test sets.

For some applications like low power ATPG and embedded deterministic test, the overall number of specified bits in a test set is more relevant than keeping a limit for each pattern. The proposed method can also be used in this direction, and outperforms the most efficient method published so far (XID [15]) in many cases or reaches comparable results at least. Table 4 shows the results of the XID method as well as the presented technique when choosing a limit as low as possible. For the circuits s13207 and s15850 the proposed stripping method reduces the overall number of specified bits by 14% resp. 16%.

Circuit	*#PIs*	*#tests*	*XID* [15]		*Stripped*	
			#max	*#all*	*#limit*	*#all*
c1908	33	106	32	2903	31	2714
c2670	233	45	181	3072	82	3040
c3540	50	93	35	2172	30	2114
c5315	178	186	115	3161	97	3203
c6288	32	14	32	448	32	448
c7552	207	75	189	7064	139	6794
s5378	214	100	170	5714	93	5746
s9234	247	111	143	8499	98	8449
s13207	700	235	489	13160	110	11374
s15850	611	97	394	13454	192	11308
s35932	1763	12	1763	13498	1763	13497
s38417	1664	87	1073	36482	470	36284
s38584	1464	114	952	31543	475	32290

Table 4: XID (no limit) vs. Limiting Maximum

6 Conclusion

The presented algorithm is able to limit the number of specified bits in the patterns of a given test set. The experiments have shown a significant reduction in the maximum number of specified bits for a wide range of test sets. Notably, the experiments with test sets generated using commercial ATPG tools show not only a substantial improvement in the maximum number of bits per pattern, but even in the overall number of specified bits in all of the test sets.

7 Acknowledgment

The authors gratefully acknowledge Seiji Kajihara and Kohei Miyase for providing the test sets used for the experiments with the ISCAS benchmark circuits.

This work has been supported by the Deutsche Forschungsgesellschaft (DFG) under grants Wu245/3-3 and Wu245/5-1.

8 References

[1] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Trans. Computers*, 44 (2), pp 223–233 (1995).

[2] B. Koenemann, "LFSR-coded test patterns for scan designs," *Proceedings of the European Test Conference*, Munich, Germany, pp 237–242 (1991).

[3] E. H. Volkerink and S. Mitra, "Efficient seed utilization for reseeding based compression," *Proceedings of the 21st IEEE VLSI Test Symposium*, Napa Valley, CA, USA, pp 232–240 (2003).

[4] A.-W. Hakmi, H.-J. Wunderlich, C. G. Zoellin, A. Glowatz, F. Hapke, J. Schloeffel, and L. Souef, "Programmable deterministic built-in self-test," *Proceedings of the IEEE International Test Conference*, Santa Clara, CA, USA, (2007).

[5] H.-J. Wunderlich and G. Kiefer, "Bit-flipping BIST," *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, USA, pp 337–343 (1996).

[6] N. A. Touba and E. J. McCluskey, "Altering a pseudo-random bit sequence for scan-based BIST," *Proceedings of the IEEE International Test Conference*, Washington, DC, USA, pp 167–175 (1996).

[7] C. L. Chen, "Linear dependencies in linear feedback shift registers," *IEEE Trans. Computers*, 35 (12), pp 1086–1092 (1986).

[8] S. Hellebrand, B. Reeb, S. Tarnick, and H.-J. Wunderlich, "Pattern generation for a deterministic bist scheme," *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, USA, pp 88–94 (1995).

[9] M. Karkala, N. A. Touba, and H.-J. Wunderlich, "Special ATPG to correlate test patterns for low-overhead mixed-mode BIST," *Proceedings of the 7th Asian Test Symposium*, Singapore, pp 492–499 (1998).

[10] P. Goel and B. C. Rosales, "Test generation and dynamic compaction of tests," *Proceedings of the IEEE International Test Conference*, Cherry Hill, NJ, USA, pp 189–192 (1979).

[11] I. Hamzaoglu and J. H. Patel, "Test set compaction algorithms for combinational circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19 (8), pp 957–963 (2000).

[12] S. Kajihara, I. Pomeranz, K. Kinoshita, and S. M. Reddy, "Cost-effective generation of minimal test sets for stuck-at faults in combinational logic circuits." *IEEE Trans. on CAD of Integrated Circuits and Systems*, 14 (12), pp 1496–1504 (1995).

[13] I. Pomeranz and S. M. Reddy, "Reducing the number of specified values per test vector by increasing the test set size," *IEE Proceedings - Computers and Digital Techniques*, 153 (1), pp 39–46 (2006).

[14] S. Kajihara and K. Miyase, "On identifying don't care inputs of test patterns for combinational circuits." *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, USA, pp 364–369 (2001).

[15] K. Miyase and S. Kajihara, "Xid: Don't care identification of test patterns for combinational circuits." *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23 (2), pp 321–326 (2004).

[16] A. El-Maleh and A. Al-Suwaiyan, "An efficient test relaxation technique for combinational & full-scan sequential circuits," *Proceedings of the 20th IEEE VLSI Test Symposium*, Monterey, CA, USA, pp 53–59 (2002).

[17] A. R. Pandey and J. H. Patel, "An incremental algorithm for test generation in illinois scan architecture based designs," *Proceedings of the Design, Automation and Test in Europe Conference*, Paris, France, pp 368–375 (2002).

[18] R. Dorsch and H.-J. Wunderlich, "Tailoring ATPG for embedded testing," *Proceedings of the IEEE International Test Conference*, Baltimore, MD, USA, pp 530–537 (2001).

[19] T. Larrabee, "Test pattern generation using boolean satisfiability." *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11 (1), pp 4–15 (1992).

[20] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists." *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, USA, pp 648–655 (1997).

[21] P. Tafertshofer, A. Ganz, and K. Antreich, "Igraine-an implication graph-based engine for fast implication, justification, and propagation." *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19 (8), pp 907–927 (2000).

[22] E. Gizdarski and H. Fujiwara, "Spirit: a highly robust combinational test generation algorithm," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21 (12), pp 1446–1458 (2002).