Cellular Automata for Generating Deterministic Test Sequences*

Dimitrios Kagaris
Electrical Engineering Dept.
Southern Illinois University
Carbondale, IL 62901

Spyros Tragoudas
Computer Science Dept.
Southern Illinois University
Carbondale, IL 62901

Abstract— We propose an on–chip test pattern generator that uses an one–dimensional cellular automaton (CA) to generate either a precomputed sequence of test patterns or pairs of test patterns for path delay faults. To our knowledge, this is the first approach that guarantees successful on–chip generation of a given test pattern sequence (or a given test set for path delay faults) using a finite number of CA cells. Given a pair of columns (C_u, C_v) of the test matrix, the proposed method uses alternative "linking procedures" P_j that compute the number of extra CA cells to enable the generation of (C_u, C_v) by the CA. A systematic approach uses the linking procedures to minimize the total number of needed CA cells. Experimental results show that the hardware overhead is often reasonable. The performance of the scheme depends on an appropriate choice of linking procedures P_j.

Keywords: Built–In Self–Test, Deterministic Test Pattern Generation.

1 Introduction

Cellular automata (CA) as potential mechanisms for Test Pattern Generation have been studied by several researchers. (e.g., [8, 4, 10, 2]). In this paper, we concentrate on cellular automata (CA) for on–chip test pattern generation of a deterministic ordered test pattern sequence, generated by an ATPG tool. This is a very difficult task. Other type of built-in mechanisms, including Weighted–random LFSRs and binary counters which have been proposed as efficient mechanisms for unordered test patterns, require very hardware intensive mapping mechanisms to guarantee generation of the input test sequence T, and are impractical.

We consider binary one–dimensional CA with the communication between cells defined by the one–neighborhood relation. If the state of cell i, $1 \leq i \leq n$, where n is the total number of cells, at time t is x_i^t, then the next state x_i^{t+1} is computed by the function $x_i^{t+1} = g_i(x_{i-1}^t, x_i^t, x_{i+1}^t)$ [11]. Values x_0^t and x_1^t are constant (set to "1" or "0") for every t. Given a $p \times w$ test matrix T representing the test pattern sequence, the goal is to establish valid functions $g_i(x_{i-1}^t, x_i^t, x_{i+1}^t)$ so that for each column ($p \times 1$ vector) C_j of T, there is some cell i so that $C_j^t = x_i^t$, $1 \leq t \leq p$. For this purpose, additional cells may be needed. These cells generate columns, referred to as *link columns*, that do not correspond to test matrix columns. The structure of the link columns is chosen so that valid functions are formed. The functions can be implemented by 8:1 multiplexers (or their equivalents). Of course, the goal is to insert as few link columns as possible, since each link column requires one cell and one multiplexer.

A generic scheme has been proposed for this problem in [2]. The scheme tries to minimize the number of link columns based on a greedy algorithm. The main problem of a generic approach such as the one in [2] is that it cannot guarantee generation of T, as greedy choices can lead it into an infinite loop.

In this work, we propose a scheme based on systematic (programmable) procedures P_j for obtaining valid functions $g_i()$. These procedures are referred to as *link procedures*. Given two columns C_u and C_v of T, link procedure P_j computes a set of extra "link" columns to make possible the generation of C_u and C_v. This number is used as weight $w_j(u, v)$ on edge (u, v) of graph G. The problem is formulated using a graph G where each node corresponds to a column and the weight of each edge (u, v) is $\min_j\{w_j(u, v)\}$, where the minimum is taken over all considered link procedures P_j. The minimum number of CA cells is determined by applying a Traveling Salesman Algorithm as explained in Section 2. The proposed procedures not only guarantee valid sets of link columns between any two columns C_u and C_v but also that the CA generates the concatenated sequence of any two already generated sequences (C_u, C_v) and (C_v, C_w) without any additional link columns. In this way, the algorithm guarantees the minimum possible hardware overhead subject to the set of the considered link procedures $\Pi = \bigcup_j P_j$. Thus, the more efficient the link procedures (and the larger the pool of such candidates), the lower the hardware overhead will be. This important feature of the presented framework initiates an interesting research direction, that of determining an appropriate pool Π of link procedures so that the hardware overhead is

*Partially supported by NSF grant MIP-9409905

ED&TC '97 on CD-ROM

minimized. Section 3 gives experimental results on test sequences provided by Sunrise Test Systems, Inc. and test matrices for path delay faults generated by ATPD [7]. The results show that the approach may only be efficient (in terms of hardware overhead) on short test sequences but it tends to behave better on test matrices for detecting path delay faults. This is a somewhat expected result, due to the inherent difficulty of the studied problems. However, its performance may be improved if additional link procedures are proposed.

2 Our approach

2.1 The basic scheme

We consider a $p \times w$ test matrix T comprising p ordered test vectors. Let X, X_L,and X_R denote any column ($1 \times p$ vector). We define the following:

Given a sequence of 3 columns (X_L, X, X_R), we associate to each row i, $1 \leq i \leq p-1$, the *τ–template* $tau_i = [\begin{smallmatrix} x^i_L & x^i & x^i_R \\ & x^{i+1} & \end{smallmatrix}]$. (No τ–template is associated with the last row p). We will use $H(\tau_i)$ to denote the upper part $[x^i_L\ x^i\ x^i_R]$ of τ_i and $L(\tau_i)$ to denote the lower part, $[x^{i+1}]$.

Given a sequence of columns (X_L, X, X_R), we say that two τ–templates τ_i and τ_j, $1 \leq i, j \leq p-1$, are conflicting if and only if it happens that $H(\tau_i) = H(\tau_j)$ and $L(\tau_i) \neq L(\tau_j)$

Definition 1: A sequence of 3 columns (X_L, X, X_R) is a *valid triplet* if and only if there are no conflicting τ–templates.

Definition 2: A sequence of columns (X_1, X_2, ..., X_i) is a *valid sequence* if and only if all subsequences (X_1, X_2, X_3), (X_2, X_3, X_4), ..., (X_{i-3}, X_{i-2}, X_{i-1}), (X_{i-2}, X_{i-1}, X_i) are valid triplets.

Definition 3: Given any two columns A and B, a *link procedure* P is a program that returns a valid sequence of columns $(A, L_0, L_1, L_2, ..., L_j, B)$. Columns $L_i, 0 \leq i \leq j$ are called the *link columns* of the valid sequence.

Given an ordered test matrix T, we construct a directed weighted graph $G = (V, E)$. Graph G contains one node for each column of T and directed edges (u, v) and (v, u) for each pair of columns u, v of the test matrix. Graph G contains two additional nodes s and t that correspond to the starting and ending 0 columns, respectively. Node s points to every other node u through an edge (s, u) while node t is pointed to by every other node u. Given a set of linking procedures P_j, we assign a weight $w(u, v)$ on each directed edge (u, v) equal to $\min_j\{w_j(u, v)\}$, where $w_j(u, v)$ is the number of link columns in the valid sequence $(u, L_0, ...L_k, v)$, under procedure P_j ($k = w_j(u, v) - 1$).

Consider a procedure P_j which not only guarantees valid sequences between any two columns C_u and C_v but also that the CA generates the sequence of any two already generated valid subsequences (C_u, C_v) and (C_v, C_w) without any extra link columns. Using the above formulation, and such a procedure P, the problem of generating T by a Cellular Automaton synthesized using procedure P, amounts to finding a path from s to t that passes through each node and has minimum total weight. We assume that each node is visited only once. The problem then amounts to the directed version of the Traveling Salesman Problem (TSP) [5]. Although TSP is NP–hard, it has been extensively explored in the bibliography and very efficient algorithms in terms of both quality of solutions and running time have been developed. The algorithm that we use here is the one from [3]. If nodes are allowed to appear on the path more than once, the overall cost may be reduced more, but the problem can be formulated as a min–cost flow problem with lower capacity bounds, for which only general Integer Linear Programming solutions are known.

Once a path of minimum total weight is returned, the CA is synthesized by listing the nodes of the path (each node is a column of T or a zero column) and in between any two such nodes u and v , the $w(e)$ columns generated by procedure P. Let T' be the resulting matrix. Any sequence of 3 columns $(x^t_{i-1}, x^t_i, x^t_{i+1})$ of T' is a valid triplet which corresponds to a valid function $g_i()$ that can be simulated by appropriately designing cell i of the CA using standard techniques. We note that the proposed approach provides a general framework for minimization of the number of link columns, based on the selected set of linking procedures.

Theorem 2.1 *The proposed framework guarantees the minimum possible hardware overhead required for a 1–neighborhood CA to generate any test sequence T under a given set of linking procedures.*

From the above it becomes obvious that the proposed scheme opens an interesting research direction. That of finding an appropriate pool $\Pi(T)$ of link procedures P_j for a given matrix T so that the cost of generating T using a 1–neighborhood CA is the minimum possible. In this paper, we provide two such link procedures that are strong contributors to the minimization of the number of link columns. The proposed procedures are *generic*, i.e., can be applied to any input matrix T. In addition, they guarantee the CA generates the concatenated sequence of any two already generated valid subsequences (C_u, C_v) and (C_v, C_w) without any extra link columns. The proposed CA synthesis tool uses these procedures, but it can also accept any other new link procedures without any changes in the tool.

2.2 Two link procedures

2.2.1 Procedure Shifting_sequences

Definition 4: Given a column $X = (x^1, x^2, ..., x^p)^{tr}$, the *shift–up* column of X is the column $\hat{X} = (x^2, x^3, ..., x^p, d)^{tr}$, where d is a don't care.

Lemma 2.1 *Given a column X, the sequence of columns $(X_L, X, \hat{X})$ is a valid triplet for any column X_L.*

The advantage of Shifting_sequences is supported by the following lemma. This shows intuitively that the more shiftings, the easier to obtain a valid triplet having a matrix column as rightmost column.

Lemma 2.2 *Let X be a column that contains don't cares only in each entry with index i or higher for some given i, $1 \leq i \leq p$. Then, for any column X_R, the sequence $(X, \hat{X}, X_R)$ is a valid triplet if and only if there are no conflicting τ–templates containing no don't cares in their L parts.*

Given two columns A and B of the test matrix, a *shifting sequence* from A to B is a sequence of columns $(A, L_0, L_1, L_2, ..., L_j, B)$ such that $L_0 = \hat{A}$, $L_i = \hat{L}_{i-1}$, $1 \leq i \leq j$, and (L_{j-1}, L_j, B) is a valid triplet. We observe that because of Lemma 2.1, a shifting sequence is always a valid sequence.

The important property of a shifting sequence $(A, L_0, L_1, L_2, ..., L_j, B)$ is that column A can be preceded by any other column X with the resulting sequence $(X, A, L_0, L_1, L_2, ..., L_j, B)$ being still valid. That is, for any two columns A and B of the test matrix, column B can always be placed after column A with some intervening link columns *without regard to what column is placed before A.* More generally, we have the following lemma.

Lemma 2.3 *Let $(A, L_0^{AB}, L_1^{AB}, ..., L_{j_{AB}}^{AB}, B)$ be a shifting sequence from A to B and $(B, L_0^{BC}, L_1^{BC}, ..., L_{j_{BC}}^{BC}, C)$ be a shifting sequence from B to C. Then $(A, L_0^{AB}, L_1^{AB}, ..., L_{j_{AB}}^{AB}, B, L_0^{BC}, L_1^{BC}, ..., L_{j_{BC}}^{BC}, C)$ is a valid sequence.*

Given any two columns A and B of the test matrix, we are interested in finding a shifting sequence $(A, L_0, L_1, ..., L_{j_{AB}}, B)$ of minimum length. This means minimizing the number j_{AB} of link columns $L_1, L_2, ..., L_{j_{AB}}$. ($L_0$ must always be equal to $\hat{A}$.) This minimum number (denoted by m_{AB}) can be found by successive shift–ups of $L_0 = \hat{A}$ until a valid triplet ending with column B is formed. The weight $w_1(A, B)$ obtained by this linking procedure (referred to as P_1) for any two columns A, B is set to $w_1(A, B) = m_{AB} + 1$. The check for a valid triplet each time is done according to Lemma 2.2 by substituting column X_R in the Lemma with column B. Figure 1 gives an example. A procedure to find the required link columns is given below.

PROCEDURE MINSL(A, B)
$m_{AB} := 0$;
$X_L := A$; $X := \hat{A}$;
WHILE (X_L, X, B) is not valid according to Lemma 2.2
    $m_{AB} := m_{AB} + 1$;
    $X_L := X$; $X := \hat{X}_L$;
    END WHILE
RETURN m_{AB};
END MINSL

A	L_1	L_2	B
1	0	1	0
0	1	1	0
1	1	0	1
1	0	0	0
0	0	1	1
0	1	0	1
1	0	x_1	0
0	x_1	x_2	0

Fig. 1: Shifting sequence for column A to B. Column B can be inserted after the link column L_2. The don't care values are set to $x_1 = 0, x_2 = 1$.

2.2.2 Procedure Target_sequences

Given any column A of the test matrix, we define a *target sequence* for A to be a sequence $(C_1, A, L_1, L_2, ..., L_j, C_2)$ that starts with a constant column C_1 followed by A, contains some (possibly none) link columns, ends with a constant column C_2, and is valid. C_1 and C_2 can be independently any constant columns, but in the following discussion we assume that both C_1 and C_2 are constant 0. The introduction of the link columns is "targeted" to allow constant column C_2 be placed at some position after column A without creating any invalid triplets. The link columns are not necessarily generated by the shifting scenario, but are rather generated so as to increase the number of 0's in each successive link column. In particular, the generation of each new link column is done so as to "push" the last '1' entry appearing in the column as high (i.e., towards or past the first entry of the column) as possible. Given a column A, a target sequence $(0, A, L_1, L_2, ..., L_j, 0)$ can be generated as follows:

PROCEDURE TSEQ(A)
1. Initialize X to the all–zero column and Y to A;
2. Initialize the index n of the link columns to $n := 1$;

3. Find the maximum k
   so that if $L_n[p], L_n[p-1], ..., L_n[p-k+1]$ are set to 0,
   no conflicting τ–templates τ_i, $p-k+1 \leq i \leq p-1$, occur;
4. Assign $L_n[i] := 0$, $p-k+1 \leq i \leq p$;
5. IF $k < p$ THEN
6. FOR each j, $1 \leq j \leq p-k$ DO
7. IF $\exists i$, $p-k+1 \leq i \leq p-1$,
      so that $(X[j], Y[j]) = (X[i], Y[i])$ and $Y[j+1] \neq Y[i+1]$
8. THEN $L_n[j] := 1$
9. ELSE IF $\exists i$, $p-k+1 \leq i \leq p-1$,
    so that $(X[j], Y[j]) = (X[i], Y[i])$ and $Y[j+1] = Y[i+1]$
10. THEN $L_n[j] := 1$
11. ELSE $L_n[j] := Y[j+1]$
12. $X := Y$; $Y := L_n$; $n := n+1$;
13. GOTO 3;
14. ELSE RETURN(n);
END TSEQ

The maximum value k of the entries of a new link column that can be assigned to 0 is found in line 3. If $k = p$ then the algorithm terminates. Otherwise, all entries with index p up to $p-k+1$ are set to 0 (line 4), while all related entries among those with index $p-k$ up to 1 are set to consistent values, as required for valid triplet (lines 8 and 9). Any entries with index $p-k$ up to 1 that are not affected by the above assignments take values according to the shifting scenario (line 11). We note that for each new link column, the position of the last 1 in the column gets closer to the top by at least one position and that each new link column is not necessarily the shifted version of the previous column. An example is shown in Fig. 2.

	A	L_1	L_2	L_3
0	1	0	0	0
0	0	0	1	0
0	1	1	1	0
0	1	1	0	0
0	1	0	0	0
0	0	0	0	0
0	1	0	0	0
0	0	0	0	0

Fig. 2: Target sequence for a column A. The values of k for link columns L_1, L_2, L_3 are respectively $k = 4, k = 5, k = 7$.

The weight $w_2(A, B)$ obtained by this linking procedure (denoted by P_2) for any two columns A, B is the number of linking columns plus 1. The following lemma implies that the proposed procedure P_2 not only guarantees valid sequences between any two columns C_u and C_v but also that the CA generates the sequence of any two already generated valid subsequences (C_u, C_v) and (C_v, C_w) without any extra link columns.

Lemma 2.4 *Let* $(0, A, L_1^A, L_2^A, ..., L_j^A, 0)$ *and* $(0, B, L_1^B, L_2^B, ..., L_j^B, 0)$ *be target sequences for* A *and* B. *Then* $(0, A, L_1^A, L_2^A, ..., L_j^A, 0, B, L_1^B, L_2^B, ..., L_j^B, 0)$ *is a valid sequence.*

3 Experimental results

We applied our method in two contexts. The first is the generation of precomputed test pattern sequences for sequential circuit testing. Table 1 shows results on test sequences provided by Sunrise Test Systems Inc. for two different circuits. The first three entries correspond to the first circuit and the last three to the second. The matrices contained don't cares, and we used a procedure from [6] to assign values to the don't cares so that the number of distinct columns is minimized. The latter quantity is a lower bound to the number of required cells in the Cellular Automaton.

The first column in Table 1 gives the dimension (rows × columns) of the test matrices. Column two indicates the average inter–column distance (sum of the minimum distance between all pairs of columns using the proposed linking procedures, divided by the number of all column pairs). This can be used for a quick (over)estimation of the overall cost. The total number of cells (extra plus inputs) of our approach as obtained by using the TS procedure of [3], is shown in the third column. We observe that for large matrices (primarily due to the number of test patterns) the hardware overhead of the problem studied may become large due to its inherent difficulty. So on–chip test pattern generation is applicable to short test sequences. Additional linking procedures, however, may lead to improved results.

Table 1: The behavior of the proposed scheme on test sequences.

Test Matrix (rows × col)	Average interclmn. dist.	Total cost (by TS)
10×135	2.34	271
20×135	5.32	565
50×135	15.06	1723
10×362	2.65	794
20×362	4.19	1003
50×362	14.12	4693

The second part in our experimentation concerns the generation of test vector pairs for path delay faults. Although the order of the two vectors in each pair is fixed, the pairs may be ordered arbitrarily. We have used an iterative improvement heuristic to reorder the pairs using a constant number of passes. In each pass,

the reordering of the pairs yields a new test matrix to which our proposed algorithm is applied to compute only the average intercolumn distance with the specific linking procedures. The heuristic returns the ordering yielding the minimum average intercolumn distance over all passes. (The details of the implementation of a pass are omitted here.)

Table 2 shows results obtained on ISCAS'85 benchmarks. The pairs of test patterns were produced by ATPD [7]. We do not list results for c432 and c499 since those circuits contain XOR gates and the generator does not currently handle this type of gates. In addition, the generator is time consuming on larger circuits such as c6288 and c7552. The experimental results indicate the flexibility in the reordering of the pairs may have a significant effect on the required hardware overhead. The table also shows the effect of the number of test vectors (pairs of patterns) on the hardware overhead.

Table 2: The behavior of the proposed scheme on pairs of test vectors generated by ATPD [7].

circuit	inputs	pairs	aver. col. dist.	total cost
c880	61	27	5.4	317
c1908	33	25	5.02	145
c2670	233	10	2.3	268
c3540	50	8	1.73	79
c5315	178	25	4.85	796

References

[1] M. Abramovici, M. A. Breuer, A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, New York, 1990.

[2] S. Boubezari, B. Kaminska, "A Deterministic Built–In Self–Test Generator Based on Cellular Automata Structures," *IEEE Trans. Computers*, vol. 44, no. 6, pp. 805–814, 1995.

[3] G. Carpaneto, M. Dell'Amico, P. Toth, "Exact Solution of Large–Scale Asymmetric Traveling Salesman Problems," *ACM Trans. Mathematical Software*, vol. 21, no. 4, pp. 394–409, 1995.

[4] A. K. Das, M. Pandey, A. Gupta, "Built–In Self–Test Structures Around Cellular Automata and Counters," *IEE Proc., Part E: Computers and Digital Techniques*, vol. 137, no. 4, pp. 269–274, 1990.

[5] M. R. Garey, D. S. Johnson, *Computers and Intractability – A Guide to The Theory of NP–Completeness*, W. H. Freeman and Co., New York, 1979.

[6] D. Kagaris, S. Tragoudas, A. Majumdar, "Deterministic Test Pattern Reproduction by a Counter", *Proc. Eur. Design & Test Conf.*, 1996, pp. 37–41.

[7] D. Karayiannis, S. Tragoudas, "ATPD: An Automatic Test Pattern Generator for Path Delay Faults," *Proc. International Test Conference*, 1996, pp. 443–452.

[8] P. D. Hortensius, R. D. McLeod, B. W. Podaima, "Cellular Automata Circuits for Built–In Self–Test," *IBM J. of Res. Dev.*, vol. 34, no. 2, pp. 389–399, 1990.

[9] C.H. Papadimitriou, K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall Inc., New Jersey, 1982.

[10] J. Van Sas, F. Catthoor, H. De Man, "Cellular Automata Based Deterministic Self-Test Strategies for Programmable Data Paths," *IEEE Trans. CAD/ICAS*, vol. 13, no. 7, pp. 940–953, 1994.

[11] S. Wolfram, "Statistical mechanisms for Cellular Automata", *Rev. Modern Physics*, vol. 55, pp. 601–644, 1983.