Cellular automata based self-test for programmable data paths

Jos van Sas Francky Catthoor Hugo De Man [†]

IMEC Laboratory
Kapeldreef 75
B-3030 Leuven
Belgium

Abstract

In this paper, a new method will be presented for the generation of a set of predetermined test vectors on chip to be used as part of a BIST strategy for a class of data paths. Given a set of faults and a corresponding set of test vectors which cover these faults, automatically the corresponding self-test hardware will be generated. For this purpose, use has been made of a cellular automaton which can be toggled between two rules. The CAD program "CAST" accomplishes the synthesis of the cellular automaton, and the method has been applied to a data path used in the Cathedral-II/2nd silicon compilation environment. This new technique allows to obtain a BIST implementation with 100% stuck-at and stuck-open/close fault coverage for all detectable faults at the cost of an area overhead larger than conventional LFSR based methods, which is still acceptable though.

1 Introduction

The increase in VLSI circuit density has complicated the problem of chip and system testing considerably. In recent years, numerous design for testability techniques have been proposed, e.g. scan design [10], LSSD [27] and boundary scan [2]. Another important approach, which has attracted a lot of attention lately, is to incorporate self-test features in the design. To this purpose test pattern generators and response compressors are included on chip. This methodology is usually called BIST and aims at reducing the test generation time and overall test complexity. The stimulus generation is the topic of this paper.

Within the BIST domain several approaches for the stimulus generation exist.

The simplest approach is to *store all test patterns* in an on- or off-chip ROM and to provide a sequencer which controls the order in which they are applied to the scan-path. However, this solution is also the most costly one as the number of test bits can be tens of millions.

It is also possible to store a test pattern generation program in a ROM. This approach is only feasible for large repetitive structures e.g. a ROM or RAM [7,25].

Hence, most approaches for data paths apply some type of *"sequential machine"* which can produce appropriate test pattern sequences. Unfortunately, these test patterns will typically be totally different from the ones which have been derived with structured or other approaches. Therefore, the crucial question of fault coverage pops up. Exhaustive excitation with all possible input combinations should provide a reasonable coverage but is infeasible in most practical cases. Obviously a trade-off is involved here. The most popular approaches can be classified into pseudo-exhaustive and pseudo-random stimulus generation [18].

Random sequence generation is possible with a variety of sequential machines. The most popular ones are the *linear feedback shift registers* (LFSR's) [1,18]. A very recent approach is the use of a number of identical, small 1-bit finite state machines which exchange data only with their neighbours [11,12]. The use of a *"cellular automaton"* (CA) looks very promising as it can be used potentially to generate sequences which are optimized for a specific circuit. The CA properties will be discussed in section 2.

Each of these random-based test pattern generation approaches, can be combined with a number of "deterministic" test patterns which are stored in ROM to detect the "hard" faults.

We are of the opinion that in the future a high fault coverage will be of primordial importance. This implies that a 99% fault coverage of single stuck-at faults only will not be sufficient any more even for a data path. The idea to "store" a predetermined set of test patterns on chip with a CA has originally been proposed in [14]. There procedures are described to construct the CA. However, the methods presented are only applicable to simple hardwired data paths which are not parameterizable at all and where only 1-pattern tests are considered.

On the other hand, the methods we propose in section 4 will deal with programmable parameterizable data paths as used in silicon compilers. In addition, we will allow 2-pattern test sequences, required to cover CMOS stuck-open faults [26], to be mapped on a CA. The self test strategy for data paths will be discussed in section 3. Our method has been applied to data paths as used in the Cathedral-II/2nd silicon compilation environment [8] (section 5). Some preliminary results are given in section 6.

This research has been performed under the ESPRIT 2318 EVEREST contract, sponsored by the EC and the industrial partners Philips, Siemens, TID, Bull and Elektronikcentralen.

[†]Professor at the K.U.Leuven

2 Overview of CA properties

2.1 Definitions

A cellular automaton is defined as a uniform array of identical cells in a n-dimensional space. Each cell is capable of covering a finite discrete state space where, in general, the state space is fairly small (e.g. {0,1}). Moreover, the cell is restricted to local neighbourhood interaction only and as a result is incapable of immediate global communication. The algorithm computing the cell's successor state (based on information received from its neighbours) is commonly referred to as the computation rule [28]. The evolution of a CA occurs in a series of time steps (clock cycles). In this study, we have made use of 1-dimensional cellular automata, in which the neighbourhood is defined as the von Neuman neighbourhood [21] which includes the physically nearest neighbour cells only (Figure 1). Furthermore, each of the cells may be in only one of two possible states at any given time with no memory associated with the cell beyond the previous clock cycle. At the boundaries of the array, *constant or periodic excitation conditions* are possible. These boundary conditions (BC) describe the functions performed by the leftmost and the rightmost cell of a length *l* CA. The set of BC consists of four fixed values 00, 01, 10, 11 (Figure 1a) and the periodic BC as illustrated in Figure 1b. In case of constant BC, the values of x_l and x_r (see Figure 2a) of cells 0 and *l-1* are always constant (either 0 or 1).

In Figure 2a a CA is represented. The rule of operation is indicated with 'F'. On the top row of Figure 2b, all 8 possible values of the cell itself and its neighbourhood are given. Below each triplet, the value achieved by the central site on the next time step according to "rule 90" is shown. There can be 256 possible distinct CA rules. The rule presented above specified in binary terms 01011010 has as decimal equivalent 90 [16], hence its naming. It must be noted that each rule can be represented by a boolean function of the state of the cell and its neighbourhood. For example, the boolean equivalent of rule 90 is $x^1 + x^{-1}$ (modulo-2 addition of left and right neighbour).

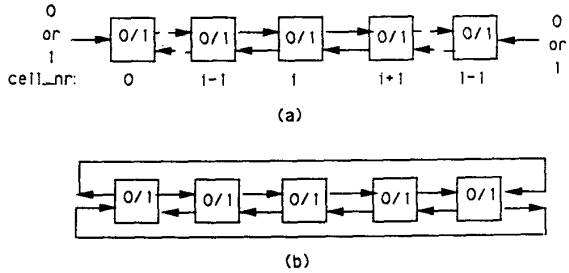

Figure 1: A 1-dimensional CA under (a) constant and (b) periodic boundary conditions

2.2 Groups in cellular automata

Two basic categories of groups are present in cellular automata[21].

The first is based on the effect of temporal multiplication (composition) of the rules of operation. A *rule group* is formed

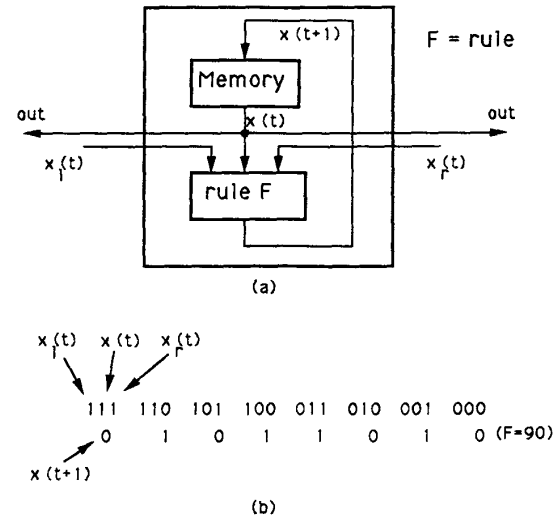

Figure 2: Representation of (a) CA cell and (b) "rule 90"

when the generator element is the rule of cellular operation, i.e., the mapping of any input state to its successor state, and the composition law is the composition of mappings, that is the successive operation of the CA rule.

If a state transition graph of a CA contains states, each of which has only one predecessor, then the corresponding rule forms a cyclic rule group (mapping is a permutation). This behaviour is for instance reflected in the state transition diagram of a length 4 CA with zero BC's and operating under rule 90, as illustrated in Figure 3a. However for periodic BC this rule does not form

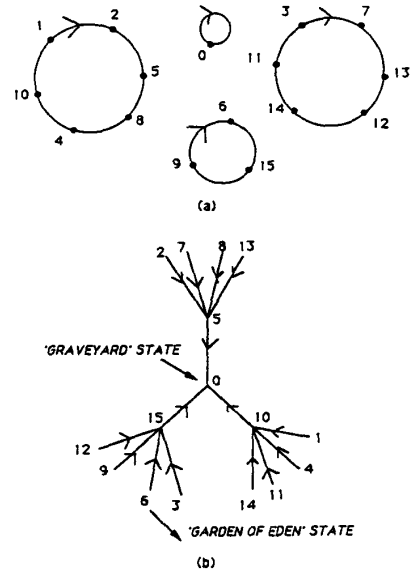

Figure 3: Transition graph for rule 90 under (a) zero and (b) periodic boundary conditions

a rule group (Figure 3b). In particular, the "graveyard" and "garden of eden" states should be noted.

The second type of groups involve mutations in the global state space and are referred to as state groups. A *state group* is formed if there exist a set of global states which form a cycle in a state transition graph. Consequently, rule groups imply state groups, but not vice versa.

It will be shown in section 3 that rule groups are better suited than state groups for our self-test methodology since we will make use of the fact that the state transition graph of rule groups consists solely of cycles. Therefore, in the remainder of this paper only cellular automata with cyclic rule group properties will be discussed, unless explicitly stated otherwise.

3 Self-test strategy for data paths with general fault model and high fault coverage

In this section we will outline the chosen self-test strategy. This strategy can be clarified on the basis of two problems.

Formulation of problem 1:
Determine the optimal hardware for a test pattern generator if a fault coverage of +/- 100% is required.
Selected solution:
For the generation of the test patterns for an n-bit wide data path without memory (combinational logic only), cellular automata or fully equivalent LFSR's can be used in conjunction with test vectors stored in a ROM to cover the hard-to-detect faults.
Motivation:
It must be noted that a CA offers the opportunity to cover most, if not all, of the test patterns of the list. If it is not possible to generate some test patterns in an efficient way by using a CA, these test patterns can be stored in a ROM.

A primitive LFSR contains only one large cycle, containing all states but the all-0 state [1]. This implies that it is very difficult to cover 2-pattern tests [26], without the use of many initialization vectors. Moreover, if the number of data inputs is larger than +/- 22, fault coverage will decrease since it is impossible to visit all states of the cycle (see also section 6).

In [23] it has been shown that linear cellular automata and LFSR's with the same irreducible characteristic polynomial are isomorphic. The important consequence of the isomorphism between the CA and the LFSR is that their cycle structure is the same. Only the sequencing of states is different between the CA and the LFSR. Therefore, the theory developed in this paper also applies to LFSR's that are isomorphic with the selected set of CA's (see section 4.2). In the rest of this paper only the CA case will be considered, even though the actual decision whether a CA or an equivalent LFSR will be used, depends on hardware considerations and will be reported in a future paper.

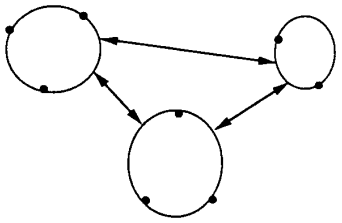

Figure 4: Use of only one rule group

Formulation of problem 2:
Given a set of faults and one or more test vectors for each fault covering it, determine the optimal composition of the test vectors, required to cover the complete set of faults on CA generators.
Selected solution:
In order to cope with problem 2 efficiently, two rule groups can be used which cover the required test vectors by traversing parts of some cycles of the state transition graph (this will be referred to as a **stretch** *of a cycle) of each rule group.*
Motivation:
First we will discuss the usage of one rule group for the generation of the test set. It is very important to note that for the detection of stuck-open faults, a sequence of test patterns must be applied. This implies that there are ordering constraints for some of the test vectors of the test set. Therefore, the usage of CA's generating only state groups is not opportune for our test methodology since it is possible that a sequence of two consecutive test patterns (tv1 followed by tv2) cannot be applied if for instance the state tv2 is a predecessor state of tv1 in the state transition graph and tv1 and tv2 are not members of the same state group. In this case vector tv2 must be stored in a ROM. Unfortunately, when using a rule group, it is also possible that some test vectors must be stored in a ROM. For instance, when not all test vectors appear in the same cycle of a rule group, which is usually the case, we have to store also test vectors in a ROM to switch from one cycle to the other cycle as indicated in Figure 4. However, for an arbitrary set of test vectors, it is more likely that when using state groups, more test vectors must be stored in a ROM than when using rule groups. Indeed, in the latter case all test vectors appear in cycles and no garden of eden or graveyard states are present.

Moreover, in order to save even more ROM space, we will use two rule groups instead of only one. The trick is here to toggle between these 2 rules, whenever a change in cycle has to occur, thus avoiding the storage of a new initial state in the ROM (Figure 5).

Consequently, our CA cell must be able to perform two rule groups. This can be accomplished by adding one extra "toggle" control signal to the CA cell.

The most important reason for using rule groups is that if the length of the CA is more than 22, it is much more difficult to find a path from one state in the first graph to another state in the second graph. In the transition graph of a CA generating only state groups, a state (not belonging to a state group) can have several predecessor states. So the different paths which

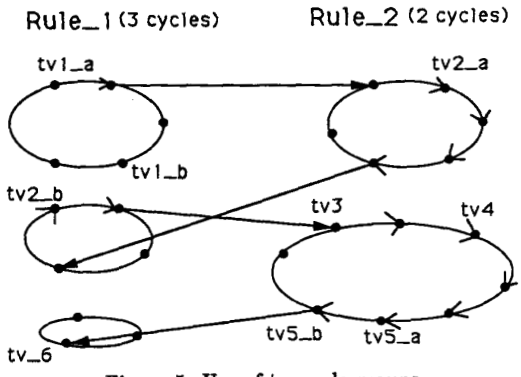

Figure 5: Use of two rule groups

lead to this state must be investigated in order to find a path from a state of rule_1 to a state of rule_2. Since it is impossible to compute the predecessor state of every state, the entire transition state graph should be computed. This is not feasible within a reasonable amount of time. Because the length of the CA can be much more than 20 (which coincides with a mere 10 bit-adder) we opted for the use of rule groups.
On the contrary, when we use rule groups, we only compute the cycles in which the test vectors appear. Since the number of test vectors is much smaller than the number of states in the graph, the number of cycles to be computed is very limited, as are the states in the cycles.

Finally, compared to the use of primitive LFSR's, which exhibit only a single cycle of maximal length, our novel approach allows also to avoid long stretches of useless vectors by switching between the rule groups.

4 CAST: CAD tool supporting Cellular Automata based Self-Test

In this section, we will give an overview of the program CAST which generates all control and switch information. This program is still under development, however some preliminary results are given in section 6.

The input for CAST is a set of faults F (which can belong to any fault model relevant for the building block under test) and for each fault $f_i \in F$ a set of 1 or 2-pattern tests which detect it.
In the remainder of this text we will use the term $PT_{1,2}$ for a 1- or a 2-pattern test. CAST has been split up into three subtasks of which the first and the second will be discussed extensively. The third subtask will be reported in a future paper.

1. Selection of the test vector list:
   In this task a test vector list, containing $PT_{1,2}$'s, covering all faults $f_i \in F$, is computed. This test vector list will be optimal with respect to the number of $PT_{1,2}$'s.
2. Find optimal (within restrictions that the problem is decoupled) combination of 2 cellular automata:
   This task has been further subdivided into 2 subtasks:

PHASE 1:
During this phase, a selection of a first *optimal rule* for the test vector list, computed in task 1, is performed. A rule is said to be *optimal* with respect to other rule groups if 1) the number of cycles in the state transition graph of the rule group, required to cover the test vector list is minimal and 2) the number of 2-pattern tests for which both test vectors appear in the same cycle is maximal. Consequently, the number of states that have to be traversed within a cycle is not of primordial importance. Currently, priority is given to criterion 1.
PHASE 2:
In this phase, a second rule group will be selected which enables us to generate the entire test set by switching between the 2 rules instead of storing all the initial states. The following data are generated:

- the second rule group
- the entire test sequence, i.e. the parts of the cycles (or stretches) of **both** state transition graphs of the rule groups which have to be visited.
- the test vectors to be stored in a ROM if it is not possible to get access to a certain part of a cycle by toggling between the states.

3. Generation of extra self-test hardware:
   From the entire test sequence as derived during phase 2, a schedule can be derived. From this schedule a local controller(FSM) to steer the cellular automata can be generated.

4.1 Selection of the test vector list

The input for CAST is a set of m faults F and for each fault $f_i \in F$ there exist a set of $PT_{1,2}$'s.

This formalism will give us the required flexibility in composing the final test set. The input description is thus not merely a sequence of test patterns which must be realized with self-test hardware. Indeed, it is not sure that a minimal set of test vectors, will also give rise to the most optimal self-test hardware solution. For example, one can imagine that adding 3 new test vectors (which cover 2 faults) to a test set, and deleting 2 test vectors (which cover also the 2 faults) can result in less transitions between the rules, because it is possible that the 2 old test vectors do not belong to the same cycle of a CA while the other 3 do belong to the same cycle. This is a topic of further research.
Three types of $PT_{1,2}$'s have been defined:

1. 2-tp_s: initialization vector followed immediately by an evaluation vector. These patterns are for instance used for the detection of stuck-open faults or other sequential faults.
2. 2-tp_ns: two test vectors are necessary for the detection of the fault. However, in contrast to the class of 2-tp_s, the 2 test vectors must not be applied successively. It is sufficient that both vectors occur in the final test set.

Some stuck-close faults can be detected with this class of test sequences [5,25].

3. 1-tp: Only one vector suffices for the detection of a fault. This is typically the case for most stuck-at faults.

It must be noted that the first class imposes the most stringent restrictions on the final test set. Therefore, this class will be called the most restrictive one. A 2-tp_s test sequence is more restrictive than a 2-tp_ns sequence which on its turn is more restrictive than a 1-tp test sequence. This means that a more restrictive $PT_{1,2}$ can cover also a less restrictive $PT_{1,2}$. The opposite in not true. The selection of the test vector list can be split up into 3 steps.

STEP 1: The list of $PT_{1,2}$'s is first reduced, using the information concerning this restrictiveness of the $PT_{1,2}$'s.

A bipartite graph represents the problem as illustrated in Figure 6b. Every vertex in the first partition class A represents a test pattern sequence. A vertex in the second partition represents a fault $f_i \in F$. If there is an edge between a vertex of A and B, this implies that the $PT_{1,2}$ covers the fault. The problem of finding a minimal set of vertices of A that cover all vertices of B can then be defined as a *bipartite dominating set problem.*

Our problem of finding a minimal set of $PT_{1,2}$'s, given a set of $PT_{1,2}$'s for each fault $f_i \in F$, belongs to the class of the NP-complete problems. This can be proven as follows:

Let us define an instance of our dominating set problem: Given a bipartite graph $G = (A,B,E)$ and a positive integer $K \leq |A|$. Is there a dominating set of size K or less for G, i.e. a subset $A' \subseteq A$ with $|A'| \leq K$ such that for all $u \in B$, there is a $v \in A'$ for which $\{u,v\} \in E$?

Theorem 1: Our dominating set problem is NP-complete.
Proof:
Our dominating set problem $\in NP$, since a nondeterministic algorithm need only guess a subset of vertices of A and check in polynomial time whether each vertex of B is joined to at least one member of that subset by an edge in E and whether the subset has the appropriate size.
Let us transform the vertex cover problem, which is one of the six well known NP-complete problems [19], into our dominating set problem.
Let graph G = (V,E) (Figure 6a) and a positive integer $K \leq |V|$ constitute an arbitrary instance of vertex cover. We define an instance of our bipartite dominating set problem as follows:
Let G'(V',E') (Figure 6b) be a graph with $V' = V \cup E$ and $(x,y) \in E'$ if $x \in V, y \in E$ and y is incident to x in G(V,E). Also let A = V, B = E and $|A'| = K$. It is not hard to see that this instance of our bipartite dominating set problem can be constructed in polynomial time from the vertex cover instance. Then G(V,E) has a vertex cover of size K or less if and only if G'(V',E') has a dominating set of size K or less.
Suppose that G'(V',E') contains a dominating set of size K or less. Suppose that $|A'| = K$. Consequently, each vertex of B is joined to at least one member of A' by an edge in E'. This implies that for each edge $\{u,v\} \in E$ of G(V,E) at least one u or v belongs to V', with $V' \subseteq V$ and $|V'| = K$. Therefore, V' is a vertex cover of size K for G.

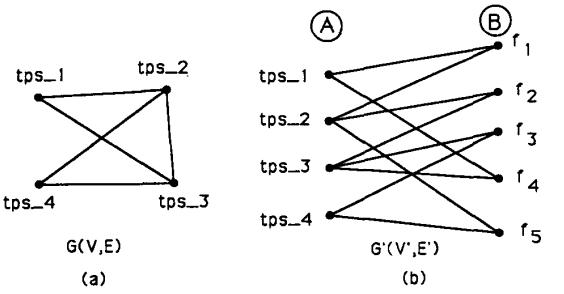

Figure 6: Example: Transformation from G(V,E) to G'(V',E').

Conversely, if G(V,E) contains a vertex cover of size K, then we can construct a dominating set as follows: Let A' contain the K vertices of the vertex cover. Then each vertex of B is joined to at least one member of A'. This is a dominating set for G'(V',E'). □.

It must be noted that our bipartite dominating set problem can be shown to be equivalent to the unate covering problem [22] along the following line of thought. A familiar form of the covering problem is solved as part of the Quine-McCluskey procedure for boolean function minimization [17]. Given the set of prime implicants, a subset of minimum cost is sought, such that all the vertices of the function are covered.

If the boolean formula derived from the matrix of a covering problem is (positive) unate, i.e., all variables appear in their uncomplemented forms only, this problem is referred to as the unate covering problem. For example, the graph G'(V',E') given in Figure 6b can be represented by the following matrix equation:

$$F = CT$$

In our problem F is the matrix containing m faults, and T is the matrix containing the n different $PT_{1,2}$'s, indicated as tps_i. The matrix $C = [C_{ij}]$ is the incidence matrix of the graph G' iff C is an $m \times n$ matrix. Then we want to minimize the number of $PT_{1,2}$'s (corresponding to the Boolean literals) required to satisfy the following equation:

$$f_1 \wedge f_2 \wedge ... f_i \wedge ... \wedge f_m = 1$$

The problem can thus be formulated as finding the minimum cost assignment that satisfies a boolean formula in conjunctive form. In this form it is known as Petrick's method [20].

STEP 2: An efficient solution to the unate covering problem can be found using the EQNTOTT software tool [6]. EQNTOTT generates a truth table suitable for PLA programming from a set of boolean equations which define the PLA outputs in terms of its inputs.
Since we want to search for the optimal set of $PT_{1,2}$'s that cover all faults it is easy to see that we have to choose the product term with the least number of literals in the minimized expression. The heuristic EQNTOTT procedure generates a

near optimal number of $PT_{1,2}$'s.
In practice it is advantageous to store a set of near optimal solutions, as the optimality also depends on factors which are also appearing in the later stages. However, for the moment we restrict ourselves to the most optimal solution due to CPU time reasons.

STEP 3: The resulting list of $PT_{1,2}$'s is then further reduced, using the information concerning the restrictiveness of the $PT_{1,2}$'s. For every selected $PT_{1,2}$, it is verified whether this $PT_{1,2}$ "covers" parts of other $PT_{1,2}$'s.

4.2 Selection of rules for CA

4.2.1 Phase 1: selection of first CA rule

As has been outlined in section 3, we will limit ourselves to cellular automata which are capable of rule group properties. However, there are 10 rule groups with zero BC and 16 rule groups with periodic BC. Only rule groups with constant BC will be considered. The first reason is that with periodic BC feedback tracks are required from the left cell to the right cell of the CA and vice versa. This will cause additional area overhead. Another important reason is that we want to make every possible combination between 2 rule groups. It turns out that a combination of any 2 rule groups with constant boundary conditions is easier to accomplish in hardware than a combination of a rule group with constant BC and one with periodic BC. This can be motivated when we consider the boolean equation of rule 89 which forms a rule group under periodic BC [21]: $(x^1 \vee \overline{x^0}) + x^{-1}$. To combine this rule with one of the rule groups of table 1, extra hardware is required to implement the "$\vee$" function.

We will not limit ourselves to rule groups with zero BC, because it has been shown experimentally that these rules are capable of rule group properties under all constant BC.

For this purpose, a program has been written that simulates the behaviour of one-dimensional, two-state CA's. This program is called "CATPG", and stands for Cellular Automata Test Pattern Generation.

Using a depth-first search and linear graph algorithm [24], all states within all cycles of the state transition graph can be computed. Computing cycles in the state transition diagram is equivalent to computing the strongly connected components, because the CA's behave deterministically.

The experiments which have been performed using this program revealed for instance that the lemma's and theorems derived in [21] for CA's with zero BC, also hold for 10, 01 and 11 BC.

Under constant BC, the following rules form rule groups for a CA of length l:

- rule 204, 51, 60, 102, 195 and 153 form rule groups for each length l.
- rule 90 and 165 form rule groups iff $l \times mod2 = 0$.
- rule 150 and 105 form rule groups iff $l \times mod3 \neq 2$ is satisfied.

Rule	Equation
204	x^0
51	$\overline{x^0}$
60	$x^1 + x^0$
195	$\overline{x^1 + x^0}$
102	$x^0 + x^{-1}$
153	$\overline{x^0 + x^{-1}}$
90	$x^1 + x^{-1}$
165	$\overline{x^1 + x^{-1}}$
150	$x^1 + x^0 + x^{-1}$
105	$\overline{x^1 + x^0 + x^{-1}}$

Table 1: 10 rules which lead to rule group properties for zero boundaries [21]

At first sight we would end up with 40 different state transition transition graphs (10 rule groups, with each 4 BC). This is however not the case, since there are rule groups which are not "sensitive" to all BC. In table 2 it has been shown which rules are sensitive to certain BC. For example, the state transition graph of rule 51 is independent of any BC.

For ease of understanding we will distinguish between a single rule group with different BC. Consequently, we obtain 26 valid rules which form rule groups under constant BC, but the identity rule (= rule 204) has been left out.

Now we will present the heuristic algorithm1 which has been used for the selection of a first CA rule. A rule is said to be *applicable* if it forms a rule group for the given length l of the CA, corresponding e.g. to the width of the data path to be tested times the number of data inputs.

Algorithm1: Selection of first CA rule

FOR every applicable rule:
{
  FOR every test vector of the test vector list:
  {
    COMPUTE the cycle and all the states within that cycle in which the test vector appears }
}
SORT all rules according to the following two criteria:
{
  a) *total number of different cycles required to cover a test set should be minimal*
  b) *the number of 2-pattern tests that appear in the same cycle should be maximal*
}

The first criterion will offer us the rule group with the least number of cycles to cover all test vectors. This fact will alleviate the switching between different cycles during the selection of a second CA rule. For pattern tests of type 2-tp_s the ordering of the test vectors is important. Therefore, the more 2-pattern tests which reside in the same cycle, the easier it is to cover the entire test set.

Rule	sensitive to BC
204	none
51	none
60	$00 \wedge (01 \vee 11)$
195	$00 \wedge (01 \vee 11)$
102	$00 \wedge (10 \vee 11)$
153	$00 \wedge (10 \vee 11)$
90	$00 \wedge 01 \wedge 10 \wedge 11$
165	$00 \wedge 01 \wedge 10 \wedge 11$
150	$00 \wedge 01 \wedge 10 \wedge 11$
105	$00 \wedge 01 \wedge 10 \wedge 11$

Table 2: Sensitiveness of rule groups to 00, 01, 10 and 11 boundary conditions

4.2.2 Phase 2: selection of a second CA rule

Phase 1 provides the first automaton rule, the location of each test vector in the corresponding cycle and all other states of that cycle. The next task is to select a complementary CA rule which deals with the remaining $PT_{1,2}$'s. Hence, the selection criteria, to be minimized, are listed below:

1. the total number of different "stretch-lengths". A stretch-length is the number of states taken by the stretch of a cycle.
2. the total number of test vectors to be stored in a ROM
3. the total number of transitions between the two CA rules

In order to satisfy the above mentioned optimisation criteria, two different algorithms have been derived to cover the 2-tp_s test sequences on the one hand (Algorithm2), and the 1-tp and 2-tp_ns test sequences on the other hand (Algorithm3). For the latter class of test sequences, the order in which the test patterns appear is of no importance. Therefore, different heuristic algorithms have been proposed to optimize the 3 criteria.

Exhaustively, each of the remaining applicable CA rules will be combined with the CA rule selected in phase 1 according to the two heuristic algorithms outlined below:

It can be noted that during step 2 of algorithm2, test patterns of the 2-tp_ns and 1-tp test sequences can also be covered.

The procedure to find a path from one test vector (tv_x) to a second different test vector (tv_y) is described below. Each of the four steps illustrated in Figure 7 is tried out in the indicated order.

Algorithm3 computes alternatingly a stretch of rule_1 and rule_2 and verifies whether one or more test vectors reside in this stretch.

In the future, we will concentrate on the reduction of the number of vectors to be stored in a ROM and the different number of stretch-lengths, by applying a more elaborate search.

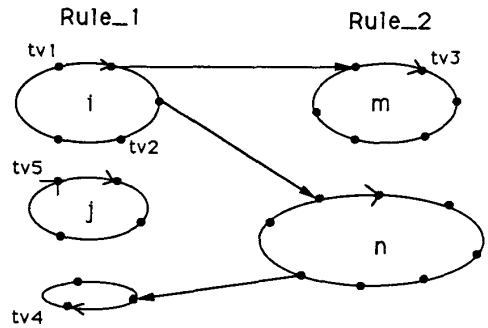

Figure 7: Illustration of the SearchPath procedure

Algorithm2: Covering 2-tp_s test sequences

WHILE there are still uncovered 2-tp_s of $PT_{1,2}$ list:
{

1. *SELECT an uncovered 2-tp_s of the list (= current $PT_{1,2}$);*
2. *SEARCH a path from tv2 of previously covered $PT_{1,2}$ to tv1 of selected $PT_{1,2}$, using the* SearchPath *procedure;*
3. *SEARCH a path from tv1 to tv2 of the selected $PT_{1,2}$, using the* SearchPath *procedure;*
4. *FOR every uncovered 2-tp_s of which both test vectors occur in the same cycle as the tv2 of the previous $PT_{1,2}$: GO TO step 2;*
5. *IF SELECTION of a 2-tp_s from the set of $PT_{1,2}$'s which are not covered by a single cycle for which: tv1 appears in the same cycle as tv2 of the previously covered $PT_{1,2}$, has been successful GO TO step 2;*
   ELSE GO TO step 1;

}

SearchPath procedure

1. *check whether tv_x and tv_y appear in the same cycle (tv_1 and tv_2 in Figure 7);*
2. *check whether it is possible to go from a state of cycle i (rule_1) in which tv_x appears to a cycle m (rule_2) in which tv_y appears (tv_1 and tv_3 in Figure 7) by simply toggling the CA;*
3. *check whether it is possible to go from a state of cycle i (rule_1) in which tv_x appears to a cycle n (rule_2) and from that cycle to cycle j (rule_1) in which tv_y appears (tv_1 and tv_4 in Figure 7). If necessary extra cycles are computed additional to the cycles containing a test vector, which have been computed during phase 1.*
4. *use a test vector to be stored in a ROM (tv_1 and tv_5 in Figure 7).*

Algorithm3: Covering 2-tp_ns and 1-tp test sequences
COMPUTE the initial stretch-length;
WHILE there are uncovered 2-tp_ns and t-tp of $PT_{1,2}$ *list:*
{
IF (number of iterations > ITER (=user specified value))
{ *SELECT uncovered test vector of* $PT_{1,2}$ *list.* }
IF ("dead-lock" situation occurs)
{ *UPDATE stretch-length;* }
IF (ITER is even) { *RULE = rule_1;* }
ELSE { *RULE = rule_2;* }
Compute_Stretch(RULE, stretch-length);
Find_Test-vectors_in_Stretch();
}

5 Complete self-test procedure for programmable data paths

5.1 The cathedral-II/2nd architecture and data path test strategy

At IMEC, a silicon compilation environment called Cathedral-II/2nd [8] is being developed. This system is targeted towards a dedicated but flexible multi-processor architecture which is suited to implement low- to medium speed Digital Signal Processing (DSP) algorithms.

The concepts of the underlying architectural methodology are illustrated in Figure 8. More details can be found in [4]. The basic operation units in every processor are called execution units (EXU's) (Figure 9) and are composed of functional building blocks (FBB's). Each of the EXU's or data paths provides foreground storage by means of register-files.

In order to make the synthesis process feasible, the processors in Figure 8 have to be working nearly independent from each other. For that purpose, the necessary data transfers are fully buffered by means of memory structures.

At present all of the FBB's which produce a data-word and not only status flags have been adapted to exhibit a mode in which their inputs are passed in a one to one way to the outputs. Due to the presence of this so called "pass-mode" (inverted or not), full controllability and observability is available for all components, so all FBB's of an EXU can be tackled sequentially [5]. For each of the FBB's in the library, a set of parameterizable test patterns has been developed for which the size is dependent on the actual parameter values only in a very limited way [5,25]. This property is a less restrictive form of C-testability [9].

The test overhead can be reduced by adapting techniques specific to a multi-processor architecture like Cathedral-II/2nd [5]. We propose to partition the chip based on the hierarchy which is evident in Figure 8.

5.2 Self-test procedure for a Cathedral-II data path

5.2.1 Test pattern generation

In this section the self-test for the data paths as used in the

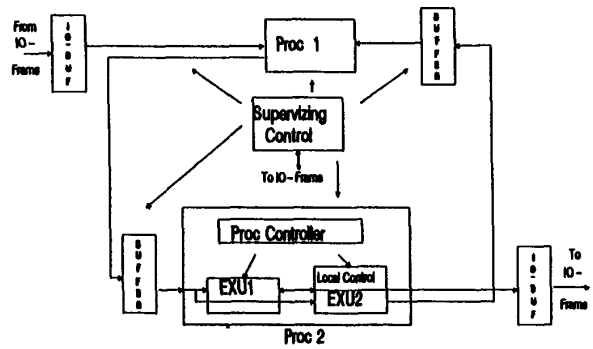

Figure 8: Definition of the multi-processor methodology

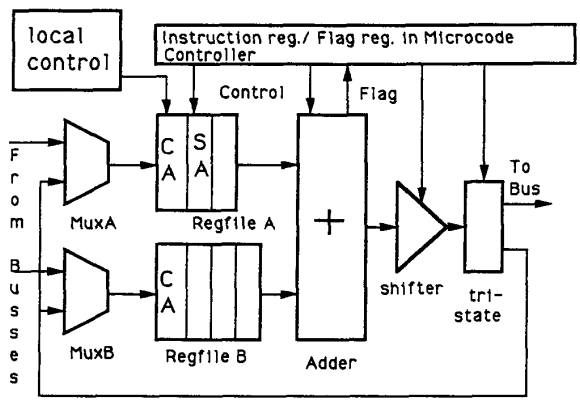

Figure 9: A typical execution unit with local self-test controller, CA and SA-register

Cathedral-II/2nd silicon compilers will be discussed. The techniques outlined in previous sections are general and applicable to any type of hierarchically structured design where the data paths are programmable and where all internal memory sites do contain a CA. Moreover the output should be observable [5]. The flexible programmability is an important requirement as the number of test patterns to be applied must be limited. This is the case with the EXU's used in the Cathedral-II/2nd library [5].

Since an EXU is composed of several FBB's we can again apply the "divide and conquer" strategy. When testing a specific FBB, the control signals of the other FBB's can be kept constant, since every FBB contains a pass-mode [5]. These control signals are generated by a special program coded into the microcoded controller. The control signals for the CA register, are generated using a local self-test controller as indicated in Figure 9.

The fact that the control signals of the EXU's will not be applied by cellular automata is crucial. Including control would be very hard to accomplish, because the CA, which generates the data signals for the EXU and the CA generating the control signals would have to be correlated with each other. The data

inputs of the data paths are generated with only 1 CA though, because correlation cannot be avoided. If there are two inputs data_A and data_B, the msb bit of data_A is connected to the lsb of data_B. Consequently, one register of each register file belongs to a CA, with the capability to scan data in and to detect sequential faults. Therefore, one extra control signal is required [25]. It must be noted, that to detect sequential faults, our CA cell has the capability that though it traverses several cycles of the state transition graph, these values are not fed to the combinational logic under test. This feature has been added to the cell in order to ensure that the initialization of the circuit nodes is retained.

5.2.2 Signature analysis

Since a feedback bus is always available for each of the EXU's (Figure 9), currently a second CA-register has been included in one of the register-files. In [13] it has been shown that the CA consisting of a sequence of alternating rule90-rule150 CA cells has better aliasing properties than the corresponding LFSR. However, the area penalty of a CA seems to be much larger than that of the corresponding LFSR. The choice of SA-register is a topic of future research.

6 Preliminary results

The program has been applied to an 8-bit and 16-bit data path ("AS8" and "AS16" respectively) consisting of a C^2MOS based adder which includes a carry bypass for speedup and a logarithmic shifter which consists of 3 shift-sections, shifting respectively over 1, 2 and 4 positions [3].

The fault model applied includes both stuck-at faults and transistor stuck-open/close [5]. Since the adder is C-testable and the number of test patterns for the shifter is only dependent on the number of shift-sections, the number of distinct test vectors is limited and equals 30 [5]. Consequently, in both circuits the same number of test vectors and $PT_{1,2}$ have been applied.

For the AS8-circuit, CAST has first selected rule 60 with BC=01. The number of different cycles required to cover all test vectors equals 9 and the number of 2-pattern tests residing in the same cycle is 6. Out of the 24 combinations the 4 most interesting have been selected. In table 3 the number of states to be visited (#state), the number of test vectors to be stored in a ROM (#init) and the number of different stretch-lengths (#sl) are indicated. The final choice depends on actual hardware overhead considerations.

For the AS16-circuit, the first rule selected is rule 195 with BC=01. Here, the number of required cycles is 10 and the number of 2-pattern tests that appear in the same cycle is 10. Out of the 16 combinations, again the 4 most interesting ones have been selected (table 3).

It is surprising that although the state diagram of the AS16 circuit contains 2^{32} states, the number of states to be visited has not been significantly increased compared to the AS8 circuit (2^{16} states). However, with the AS16, at least 9 test vectors have to be stored in a ROM. Note that the fault coverage for the CA based method is 97.4%. In contrast, the coverage for the same fault model in the AS8 circuit in the case of a complete random test sequence is only 69.5%. This would be even less for a shorter pseudo-random sequence, as needed in practical cases with larger word lengths. All these figures have been derived with the LOFSCATE fault simulator [15], which allows a model including stuck-open and stuck-close faults.

Circuit	Rule_2 (BC)	#state	#init	#sl
AS8	165 (11)	4762	2	37
	105 (11)	1966	3	46
	90 (00)	1655	2	36
	102 (10)	15869	4	34
AS16	60 (01)	1638	9	26
	153 (00)	1813	10	22
	102 (00)	1411	14	18
	195 (00)	1653	11	18

Table 3: Some results for AS8 and AS16 circuits

7 Conclusions

Recently, the importance of BIST has increased considerably, also for data paths. However, we believe the fault coverage should remain as high as in the ATE case for a realistic fault model. Since the pseudo-exhaustive and pseudo-random based methods typically do not allow a really high fault coverage for large data paths (more than +/- 22 inputs), a novel powerful technique has been presented in this paper. This technique makes use of cellular automata (CA's) and enables the user to apply a predetermined set of test patterns by means of a CA which can toggle between two rules. To come to an efficient solution, optimization techniques have been derived and implemented in the CAD-tool CAST. The potential of the CA-based method has been demonstrated on several practical examples. Results show that the outlined strategy enables to obtain a roughly 100% fault coverage. This new method is in particular suited for programmable data paths. In addition, a complete self-test strategy for data paths as used in the Cathedral-II/2nd silicon compilations environment has been proposed.

The self-test overhead consists of the self-test control circuitry to steer the CA for the stimulus generation and the CA-register and SA-register which must be incorporated in each EXU. The size of the control circuitry depends on the number of initializing test vectors to be stored and the number of different stretch-lengths. The size of the CA register depends on the combination of the 2 CA rules.

For a practical design example, we obtained an area overhead of about 50 % with regard to the data path only. This by applying some crude heuristic algorithms. In addition, since self-test overhead for RAM's of moderate size is only about 5 à 10 % [7], the total chip self-test overhead will decrease significantly. This will be reported in a future paper.

Acknowledgements
We acknowledge the interesting and stimulating cooperation with our partners within the EVEREST project. We also want to thank Peter De Worm, Jan Zegers, Paul Vanoostende, Maryse Wouters and Serge Vernalde for the interesting discussions.

References

[1] P.H. Bardell, W.Mc Anney, J. Savir, "Built-in test for VLSI: pseudo-random techniques", J.Wiley & Sons, New York, 1987.

[2] F. Beenker, K. van Eerdewijk, R. Gerritsen, F. Peacock, M. van der Star, "Macro Testing: unifying IC and board test", IEEE Design and Test, pp.26-32, Dec.1986.

[3] E. Blokken, H. De Keulenaer, F. Catthoor, H. De Man, "A flexible module library for custom DSP applications in a multi-processor environment", Proc. of ESSCIRC, Vienna, pp.64-67, Sept. 1989.

[4] F. Catthoor, J. Rabaey, G. Goossens, J. Van Meerbergen, R. Jain, H. De Man, J. Vandewalle, "Architectural Strategies for an Application-specific Synchronous Multiprocessor Environment", IEEE Trans. on ASSP, pp.265-284, Feb. 1988.

[5] F. Catthoor, J. van Sas, L. Inzé, H. De Man, "A Testability Strategy for Multiprocessor Architecture", IEEE Design and Test, pp.18-34, April 1989.

[6] B. Cmelik, J. Deutsch, EQNTOTT, Berkeley CAD Tools User's Manual, 1989.

[7] R. Dekker, F. Beenker, L. Thijssen, "Realistic built-in self-test for static RAM's", IEEE Design and Test, pp.26-34, Feb.1989.

[8] H. De Man, J. Rabaey, J. Vanhoof, G. Goossens, P. Six and L. Claesen, "Cathedral-II: a computer-aided synthesis system for digital signal processing VLSI systems ", in Comput. Aided Engineering Journal, pp.55-66, April 1988.

[9] A.D. Friedman, "Easily testable iterative systems", IEEE Trans. on Comput., Vol.C-22,pp.1061-1064, 1973.

[10] H. Fujiwara, "Logic testing and design for testability", MIT Press series in computer systems, 1985.

[11] C.S. Gloster, F. Brglez, "Boundary Scan with Built-in Self-test", IEEE Design and Test, pp.36-44, Feb. 1989.

[12] P.D. Hortensius, R.D. McLeod, W. Pries, D.M. Miller, H.C. Card, "Cellular-Automata-Based Pseudorandom Number Generators for Built-In Self-Test", IEEE Trans. on CAD/ICAS, vol.8, no.8, pp.842-859, Aug. 1989

[13] P.D. Hortensius and R.D. McLeod, "Cellular Automata-Based Analysis for Built-in Self-Test", Proc. of the Third Technical Workshop: New Directions for IC Testing, Halifax, Nova Scotia, pp.117-127, Oct.1988.

[14] M. Khare and A. Albicki, "Cellular Automata used for Test Pattern Generation", Proc. of ICCD'87, pp.56-59 1987.

[15] LOFSCATE users' manual version 7.1, LAMM, March 1989.

[16] O. Martin, A.M. Odlyzko, and S. Wolfram, "Algebraic Properties of Cellular Automata", Communications in Mathematical Physics 93, pp.219-258, Springer-Verlag 1984.

[17] E.J.Mc Cluskey, "Minimization of Boolean Functions", Bell Syst. Tech. Jour., Vol.35, pp.1417-1444, Nov. 1956.

[18] E.J.Mc Cluskey, "Built-in self-test techniques", IEEE Design and Test, Vol.2, Nr.2, pp.21-28, April 1985.

[19] C.H. Papadimitriou, K. Steiglitz "Combinatorial Optimization: Algorithms and Complexity", by Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1982.

[20] S. Petrick "A Direct Determination of the Irredundant Forms of a Boolean Function from the Set of Prime Implicants," Air Force Cambridge Res. Center, Bedford, MA, Techn. Rep., 1956.

[21] W. Pries, A. Thanailakis, and H. C. Card, "Group Properties of Cellular Automata and VLSI Applications", IEEE Trans. on Comput., vol.C-35, no.12, pp.1013-1024, Dec. 1986.

[22] R.L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization", Trans. on CAD/ICAS, vol.6, no.5, pp.727-750, Sept. 1987.

[23] M. Serra, D.M. Miller, J.C. Muzio, "Linear Cellular Automata and LFSR's are isomorphic", Proc. of the Third Technical Workshop: New Directions for IC Testing, Halifax, Nova Scotia, pp.213-223, Oct.1988.

[24] R. Tarjan, "Depth-first and linear graph algorithms", SIAM J. Comput., vol.1, no.2, June 1972.

[25] J. van Sas, F. Catthoor, L. Inzé, H. De Man, "Testability Strategy for Registers and Memories in a Multi-processor Architecture", Proc. of the 1th European Test Conference, Paris, pp.294-303, April 1989.

[26] R.L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits", The Bell Tech. J., Vol 57 (May-June 1978) pp.1449-1474.

[27] T. W. Williams, K. Parker, "Design for testability - A survey", IEEE Trans. on Comput., Vol.C-31, No.1, pp.2-15, Jan. 1982.

[28] S. Wolfram, "Statistical Mechanics of Cellular Automata", Reviews of Modern Physics, vol.55, no.3, July 1983.