Genetic Evolution of Nonlinear Cellular Automata for Built-In Self-Test of Combinational Circuits

By

Jesse David Bingham
B.Sc., University of Victoria, 1998

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

mputer Science)

ient of Computer Science)

nt of Computer Science)

Mathematics and Statistics)

rtment of Mathematics and

University of Victoria

0-612-58526-3

Supervisor: Dr. J. C. Muzio

ABSTRACT

Maximum length linear Cellular Automata (CA) have been proposed as good pseudorandom test generators for built-in self-test (BIST) of combinational circuits. However, linear CA account for but a small fraction of possible CA and it is hypothesized that appropriate introduction of nonlinear rules may result in a shorter test length. Beginning with an initial population of maximum length CA, a genetic algorithm (GA) is used to evolve CA that achieve some target fault coverage with fewer vectors. Two simple mutation operators and two approaches to dealing with the initial vector are compared. It is shown that nonlinear CA can be found that reduce test length significantly. Two advanced mutation operators are considered, one which completely preserves the state-transition structure, and one which maximizes the similarity between sequences generated by the mutated and original CA. A technique called *good neighbor enforcement* (GNE) is used to improve the crossover operator by discouraging degenerate behavior in offspring. Both periodic and null boundary CA are used and it is shown that GNE and a simplified version called *sink avoidance* are beneficial; a fast sink avoidance algorithm is given. Finally, a new approach to obtaining the BIST stopping condition is proposed.

Examiners:

mputer Science)

nent of Computer Science)

ent of Computer Science)

ıf Mathematics and Statistics)

artment of Mathematics and

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I'd like to thank the elements of the following set... remember that sets are unordered and may contain other sets which are not necessarily disjoint: {the Big Guy in the Sky, Mom, Dad, my brothers, my friends, Dr. J. Muzio, Dr. M. Serra, Dr. D. J. Miller, Dr. F. Ruskey, Claudio Costi, Chris Norris, Eugen Toaxen, Gord Brown, Jackie Rice, Ken Kent, Laurence Yang, the VLSI group members, Claire Barnett}.

1. Introduction

FOR the purpose of this thesis, a *cellular automaton* (CA) is a 1-dimensional array of flip flops, each being updated synchronously by some function of a local neighborhood of cells; this function is called the cell's *rule*. CA have been proposed as an alternative to linear feedback shift registers (LFSR) for pseudorandom pattern generators in circuit testing [1, 2] and other applications [3].

Autonomous linear finite state machines have been studied extensively using the tools of linear algebra and matrix theory [4, 5, 3, 6, 7, 8, 2, 9, 10, 11, 12]. Much of this work has been geared towards a priori prediction of a machine's functional digraph structure and randomness properties, with applications in testing, coding theory, and cryptography. Little has been done along the same lines with respect to nonlinear machines, as these mathematical tools assume linearity[1] . Since linear machines represent a diminishing fraction of possible machines, we hypothesize that nonlinear CA exist that are better than the traditionally used linear CA at testing digital circuits. In this case *better* refers to being able to cover a target percentage of faults with a shorter test sequence length. The question remains: how does one acquire such CA?

The main idea of the thesis is the employment of a *genetic algorithm* (GA) to evolve CA that test a prescribed percentage of stuck-at faults in a combinational circuit using a minimal number of vectors. A GA is a probabilistic search technique that models the biological phenomenon of survival of the fittest and natural selection. The operation of GA has been described as "remarkably straightforward" [16]. GA evolve a population of candidate solutions through many generations, creating new solutions from old by performing the three genetic operations of reproduction, crossover, and mutation. The algorithm central to this thesis is named the *cellular automata genetic algorithm* (CAGA).

---

[1] [3] looks at a very restricted class of nonlinear CA, [13, 14, 15] analyzes uniform nonlinear CA, [7] analyzes nonlinear shift registers

Working with the nonredundant ISCAS '85 benchmarks circuits [17], it is shown that nonlinear CA can be evolved that significantly reduce test lengths over primitive CA. The initial vector of the test sequence can either be fixed or evolved along with the CA rule vector; experiments are performed comparing the two approaches. Also, two simple mutation operators are contrasted. We find that on average CAGA effectiveness does not vary extensively with these choices. The crossover probability "standard setting" [18] is verified for the CA testing problem.

Several attempts to improve CAGA via advanced genetic operators are investigated with varying levels of success. A mutation operator called *bit role flipping* (BRF) is defined. BRF has the appealing property of completely preserving the state-transition space of a CA. *Last possible mutation* (LPM) is another proposed advanced mutation operator. LPM maximizes the number of equivalent vectors produced by the mutated CA before diverging in state transition space and is found to be powerful when it is allowed to introduce *radius 2* rules. A technique called *good neighbor enforcement* (GNE) is used to improve the crossover operator by discouraging degenerate behavior in offspring. Two variants of GNE are shown to be effective against evolution of both null boundary and periodic boundary CA.

Also addressed is the problem of halting built-in self-test (BIST) testing. A new technique is described in which a *minimal stopping condition* (MSC) is sought in the test generating FSM. A MSC is a set of generator cells of minimal cardinality that uniquely distinguishes the final test vector from all others. GA and brute force search are discussed as means of finding a MSC.

The thesis is organized as follows. Chapter 2 gives an introduction to the theory of digital circuit testing including the concepts of BIST and pseudorandom pattern generation. Chapter 3 formally defines cellular automata and summarizes previous studies involving CA. In chapter 4, genetic algorithms are explained and exemplified. All discussion and results regarding CAGA and the advanced genetic operators are presented in chapter 5. The auxiliary chapter 6 proposes the new means of obtaining the BIST stopping condition. Finally, the thesis is concluded with chapter 7 in which many avenues of future work are suggested.

2. Digital Integrated Circuit Testing

2.1 Introduction

In the context of the design, production, and use of digital integrated circuits (ICs), *testing* (also called *structural testing*) refers to a process whose goal is the demonstration that a given IC is free from defects [4, 19]. Here, a *defect* is a physical anomaly in the chip that results in incorrect behavior and stems from either manufacturing or deterioration over time. Testing is separate from *verification*, during which it is checked that a prototype adheres to its logical functional specifications and electrical parameters. Testing is normally used to assess whether or not a chip should be discarded and replaced; the process of actually pinpointing the location and nature of a defect falls under *fault diagnosis* and is not relevant to this thesis.

Two distinct testing philosophies exist: *functional testing* and *structural testing* [4, 6]. The standard approach prior to the influential paper by Eldred [20] was to test a device based on knowledge of its functional specification. For example, testing an adder would involve checking that it could indeed add two binary numbers. Structural testing considers the circuit topology in terms of connections between primitive elements such as AND, OR, and NOT gates. Test vectors are applied to the circuit that reveal (via primary outputs) the presence of defects in the primitive elements and/or interconnections. For the remainder of this thesis, the term *testing* refers to structural testing.

The set of possible logical effects a defect can have on a primitive element or the interconnections must be defined a priori and is called a *fault model*, which is the focus of the next section. Section 2.3 discusses testing under the most common fault model, the *stuck-at model*. In section 2.4 *built-in self-test* is described, followed by a brief discussion of the testing of sequential logic in 2.5.

2.2 Fault Models

A physical defect on a chip may take on many forms. Open interconnections, shorts between conductors, excess leakage current, and burnout from electrical overload are but a few examples [4]. The commonality that these imperfections share, however, is that they each have the potential to affect the logical behavior of the circuit on which they reside. The term *fault* refers to this logical effect of a defect, and is dependent on the fault model being used.

A *fault model* provides a means of mathematically abstracting the physical defect to a logical effect. An example of a fault model is that of the *delay* fault, in which the defect causes signals in a circuit to be *slow-to-rise* or *slow-to-fall* and adversely affect the operating speed of a chip. Other fault models include the *bridging* fault, *short-circuit* fault, *inverse* fault, and *mix-up* fault.

In this thesis only *permanent* faults are considered. Permanent faults are contrasted by *intermittent* faults which are sometimes active while sometimes not, and *transient* faults which only occur for a brief period of time.

The most widely used fault model is the classic *stuck-at* fault model, proposed in 1959 by Eldred [20], in which the fault effect is that a wire or gate output is "stuck-at" logic 1 or logic 0. If a fault causes such behavior it is dubbed a *stuck-at fault*; the focus of this thesis is testing for stuck-at faults. The stuck-at fault model is essentially the foundation of test theory [6].

2.3 Testing for Stuck-At Faults

Let f be a stuck-at fault in some combinational circuit. A *test* for f is an input vector v such that given v, the output of the fault-free circuit differs from that of the circuit when f is present. The *test set* for f is the set of all tests for f. A fault with a relatively small test set is referred to as *hard to test*. Given a set of faults F, a *minimal test set* for F is a set of input vectors with minimal cardinality that contains a test for each fault in F. If x is a label assigned to a site in a circuit, the notation x/α represents the fault where x is stuck-at logic α (where α is 0 or 1).

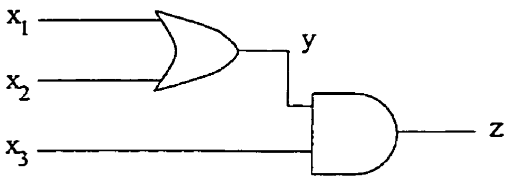


Figure 2.1: A circuit implementing z = $(x_1 + x_2)x_3$

input				Output in Presence of Faults									
x_1	x_2	x_3	z	$x_1/0$	$x_2/0$	$x_3/0$	y/0	z/0	$x_1/1$	$x_2/1$	$x_3/1$	y/1	z/1
0	0	0	0	0	0	0	0	0	0	0	0	0	$\boxed{1}$
0	0	1	0	0	0	0	0	0	$\boxed{1}$	$\boxed{1}$	0	$\boxed{1}$	$\boxed{1}$
0	1	0	0	0	0	0	0	0	0	0	$\boxed{1}$	0	$\boxed{1}$
0	1	1	1	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	$\boxed{1}$	0	$\boxed{1}$
1	0	1	1	$\boxed{0}$	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	1	1	1	1	1
1	1	0	0	0	0	0	0	0	0	0	$\boxed{1}$	0	$\boxed{1}$
1	1	1	1	1	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	1	1	1	1	1

Table 2.1: Fault effects for all stuck-at faults in the circuit of figure 2.1

In this thesis we adopt the *single fault assumption* which states that at most one fault in the given model can be present. Most of the time a *multiple fault* (the existence of more than one fault) can be detected by the tests for the involved single faults [21].

By analyzing the structure of a circuit, we can create a complete list of possible stuck-at faults and determine tests for each. In the circuit of figure 2.1 we see that there are 5 fault *sites* (labeled x_1, x_2, x_3, y, and z), and since each may be stuck-at-0 or stuck-at-1, there are 10 possible faults in total. The fourth column of table 2.1 gives the output z of the circuit in the absence of faults. The rightmost ten columns give the output of the circuit when each of the ten possible faults are active. When a value in one of these columns is enclosed in a box, this indicates that the output differs from that of the fault-free circuit for the corresponding input vector. In other words, the test set for each fault consists of the input vectors corresponding to the rows in which a box is present under the fault. For example, the test set for $x_3/1$ is {010,100,110}, in which the bitstrings represent the respective values of x_1, x_2, and

x_3. Notice that faults such as $x_2/0$ have only one test. Clearly, any minimal test set for these ten faults must include such tests, else at least one fault would go untested. Thus any minimal test set must contain 101 (test for $x_1/0$), 011 (test for $x_2/0$), and 001 (test for $x_1/1$, $x_2/1$, and y/1). But these three vectors also include tests for $x_3/0$, y/0, z/0, and z/1. The only fault left untested is $x_3/1$, for which we arbitrarily choose 010, giving us the minimal test set {001,010,011,101}.

An important fact illustrated by this example is that we can usually cover all possible stuck-at faults in an n-input combinational circuit with *far fewer* input vectors than the 2^n vectors applied in exhaustive testing. Our example circuit has $2^3 = 8$ possible input vectors, yet we only need 4 of them to test for all possible stuck-at faults.

. Observe that in table 2.1 the columns for $x_1/1$, $x_2/1$ and $y/1$ are identical. Such faults are called *functionally equivalent*, which is an equivalence relation that partitions the set of all stuck-at faults into equivalence classes. The equivalence classes for the example circuit are $\{x_1/0\}$, $\{x_2/0\}$, $\{x_3/0, y/0, z/0\}$, $\{x_1/1, x_2/1, y/1\}$, $\{x_3/1\}$, and $\{z/1\}$. Fault equivalence classes are important in testing because once a test has been generated for one member of a class, all other faults in the class are necessarily covered.

Given a single stuck-at fault f, the process of algorithmically finding a test vector for f is called *test generation*. The classic solution to test generation is Roth's d-algorithm [22]. PODEM [23, 24] uses heuristics to reduce the amount of backtracking done by the d-algorithm and is more effective on larger circuits.

Fault simulation refers to the converse problem: given a set of faults F and some test vector v, which faults of F does v detect? There are many fault simulation algorithms, for a good survey see [21]. The fault simulator used in this thesis was written by a group at Virginia Polytechnic Institute (Electrical Engineering Dept.) and based on the algorithm presented in [25].

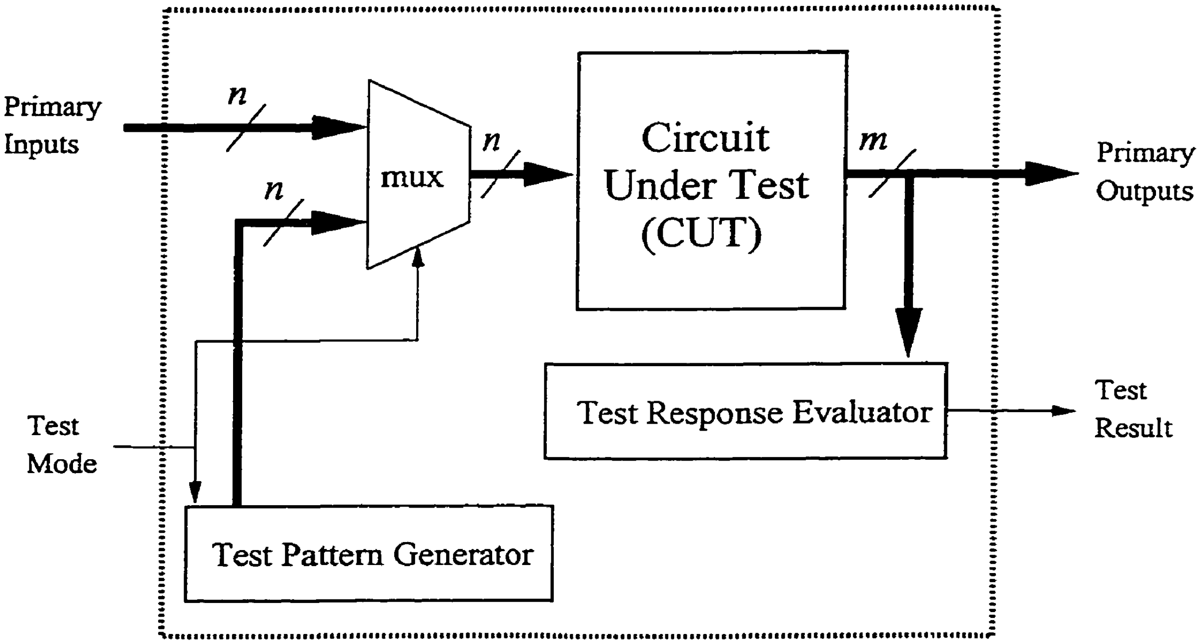


Figure 2.2: Basic BIST architecture

2.4 Built-In Self-Test (BIST)

BIST refers to the capability of a circuit or system to test itself and indicate a pass or fail status using extra test circuitry at the chip or board level [19]. BIST can be classified as *On-line* or *Off-line* [6, 4, 19].

In On-line BIST, the circuit contains logical redundancy that allows it to test itself while simultaneously performing the intended function. When a fault is present in a circuit endowed with on-line BIST, it either manages to still function correctly or it indicates that it cannot do so.

Off-line BIST means that the circuit must be put in *test mode* in order for testing to occur. When in test mode the circuit does not perform its normal function; it is essentially taken off-line. Some time (preferably a short time) after being placed in test mode, a circuit with off-line BIST will indicate a pass or fail status via one or two pins. In most of the literature and the remainder of this thesis, off-line BIST is referred to simply as BIST.

Figure 2.2 depicts the basic BIST architecture. Naturally, there is an overhead associated with BIST arising from the auxiliary circuitry needed, namely the *test* *pattern generator* (TPG) and the *test response evaluator* (TRE). Not explicitly shown in figure 2.2 is the *BIST controller*, which controls the CUT, TPG, and TRE when in test mode. Ideally, the TPG would be a ROM storing the vectors of some minimal or near minimal test set and would feed these vectors in some deterministic order during testing; such an approach would keep test time to a near minimum. Similarly, the ideal response evaluator would store the expected output vectors of the circuit under the TPG's test sequence (i.e. the response of the fault-free circuit), and perform a comparison against each observed output. If all actual outputs match the expected outputs, the chip passes, else it fails. However, such a scheme consumes a great deal of chip area. For example, if the CUT has n inputs and m outputs, and the test set has cardinality T, then the TPG ROM requires nT memory elements while the TRE ROM uses mT.

Two technologies have been developed to reduce this overhead significantly. On the TPG side, *pseudorandom pattern generation* (PRPG) is used, which typically requires $\mathcal{O}(n)$ memory elements. The cost of PRPG is an increased test sequence length. For test response evaluation, a technique based on data compaction called *signature analysis* is used. Signature analysis can use as many or as few memory elements as desired, though as the number decreases the probability of *aliasing* increases. Aliasing occurs when a faulty output stream is passed off as being fault free.

We now take a closer look at these BIST concepts.

2.4.1 Pseudorandom Pattern Generation

Recall that the problem of finding a vector that tests a specific fault is called *test generation*. When seeking a test set for a set of faults F, the standard approach is given in figure 2.3. The crux of this technique is that once a test vector v is generated for a specific fault, all other faults in F that are "accidentally" tested by v can be marked as tested. PRPG was first proposed when test engineers noticed that the number of faults tested in this accidental manner was large early on in

```
findTestSet($F$)
   $S \leftarrow \emptyset$
   while $F \neq \emptyset$ do
      chose some arbitrary $f \in F$
      $F \leftarrow F - \{f\}$
      generate a test $v$ for $f$
      $S \leftarrow S \cup \{v\}$
      foreach $f' \in F$ do
         if $v$ tests $f'$
            $F \leftarrow F - \{f'\}$
      end foreach
   end while
   return $S$
end
```

Figure 2.3: Test set generation

the algorithm [4]. The conclusion reached was that a randomly chosen vector will typically test many faults. Thus a random sequence of test vectors can be expected to test a large proportion of faults early on in the sequence. Given a set of faults and a combinational circuit, the length of a random test sequence that can be expected to detect a given ratio of the faults can be statistically determined [4, 6, 21]. In the preceding discussion, the term "random" indicates that each bit in the test sequence is 0 or 1 with equal probability [21].

There are sequence generating finite state machines (FSMs) that lend themselves to hardware implementation and exhibit many properties of true random sequences [7, 21, 4]. Of course, by definition, a FSM is totally deterministic and is therefore the antithesis of a random pattern generator; this fact is reflected by the use of the term *pseudorandom*.

In this context, a FSM can be thought of as a one dimensional array of n single bit memory elements, often called *cells*. The n-vector consisting of the bits stored in the cells constitute the machine's *state*. A FSM is a synchronous device, meaning that the state changes on the active edge of a clocking signal according to the machine's *next state function*. The cells themselves are typically thought of as

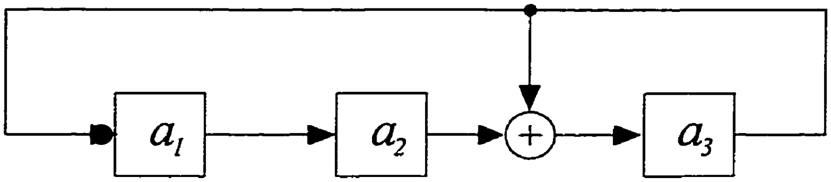


Figure 2.4: An example maximal length type I LFSR with $n = 3$

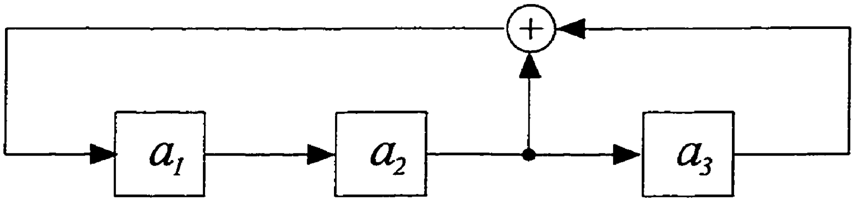


Figure 2.5: An example maximal length type II LFSR with $n = 3$

living on a horizontal line, and are referred to from left to right as $a_1, a_2, \ldots, a_n$; the next states of the cells are respectively $a_1^+, a_2^+, \ldots, a_n^+$.

The most commonly used class of PRPG in BIST are *linear feedback shift registers* (LFSRs) [4, 6, 19]. LFSRs are a type of *feedback shift register* with a linear next state function, and fall under the umbrella of *linear finite state machines* (LFSMs). Because of linearity, the next state function of a LFSM can be expressed as a transition matrix A, allowing for analysis using the tools of linear algebra [7, 9]. If s and s^+ are respectively the column vectors representing the current and next state of a LFSM, then $s^+ = As$, where arithmetic is performed over the Galois field of 2 elements, $GF(2)$.

LFSRs fall into two categories: type I and type II; both types place restrictions on communication between cells (i.e. the next state function). A type I LFSR performs polynomial division, and each cell's next state function is a linear function of its left neighbor and a_n. In a type II LFSR , each cell receives its next state directly from its left neighbor, i.e. the state vector is shifted right. The feedback in a type II LFSR only comes into play with a_1, whose next state is a linear function of all cells. Figure 2.4 and 2.5 respectively show a type I and type II LFSR, both with $n = 3$.

Let A_I and A_{II} be the respective transition matrices of the LFSRs of figures 2.4

and 2.5. For the LFSR of figure 2.4, the next state equations are given by $a_1^+ = a_3$, $a_2^+ = a_1$, and $a_3^+ = a_2 \oplus a_3$; for figure 2.5 these equations are $a_1^+ = a_2 \oplus a_3$, $a_2^+ = a_1$, and $a_3^+ = a_2$. From these equations we find:

$$A_I = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

and

$$A_{II} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Starting from initial state $s = [100]^T$ both example LFSRs see all $2^n - 1 = 7$ nonzero states before returning to the initial state. The LFSR of figure 2.4 follows the state sequence (100, 010, 001, 101, 111, 110, 011, 100), while the LFSR of figure 2.5 generates (100, 010, 101, 110, 111, 011, 001, 100). LFSMs with this property are called *maximal length* or *primitive.*

Another class of FSM are *cellular automata* (CA), which are a focus of this thesis and will be formally introduced in chapter 3. The next state of each cell in a CA is restricted to being a function of the cells in a local neighborhood, typically the cell itself and its two immediate neighbors. Thus all nonzero entries in the transition matrix of a linear CA (LCA) lie on the main, super, and sub diagonals. Primitive LCA were first proposed as PRPGs for BIST in [1]. Evidence that LCA make *better* BIST test generators than LFSRs is revealed in [11, 12], specifically LCA are better detectors of delay faults. Thus targeting stuck-at faults using LCA has the amiable side effect of providing tests for many delay faults. CA are discussed in further detail in chapter 3 and are used extensively in chapter 5.

The state sequences generated by primitive LFSRs and CAs have been shown to have many characteristics of random sequences. As such, they work well as PRPGs for BIST.

There are many techniques to transform the state vector of a FSM into an input vector for a CUT, especially if the CUT is sequential. In this thesis, we take the simplest approach in which a CUT with n primary inputs is tested using a FSM on n cells and the state of cell a_i drives the i^{th} input during testing.

2.4.2 Signature Analysis

Several techniques to perform data compaction on the responses (output vectors) of a circuit to a test sequence [4, 21] exist. The most popular is called *signature analysis* (SA), a term coined in [26]. For a single output CUT, SA is performed typically by a LFSR that takes a single input; specifically the CUT's output. This output signal is added modulo-2 to the feedback signal driving the leftmost cell of a FSM. For example, the LFSR depicted in figure 2.5 could be used as a signature analyzer by using a 3-input EXOR gate in place of the 2-input gate, with the additional input wired to the output of the CUT. Once the test sequence is complete, the LFSR stops accepting input and its final state is the output stream's *signature.* The signature is then compared to the known signature of the fault-free circuit, and the test fails if there is a discrepancy.

Aliasing occurs when the signature of a faulty circuit is that of the fault free circuit, due to error cancellation occurring in the LFSR. When the SA LFSR has a degree k *characteristic polynomial,*[2] the probability of aliasing approaches 2^{-k}. This result works under the assumption that all output streams are equally probable, which is seldom valid. However, by selecting the characteristic polynomial of the SA LFSR intelligently, some types of errors can be guaranteed to be nonaliasing. For instance, if the characteristic polynomial has at least 2 non-zero coefficients, all single bit error streams are nonaliasing [21].

When the CUT has more than one output, the hardware overhead incurred by using a LFSR on each of the m outputs is usually too high. In such circumstances a device called a *multiple input shift register* (MISR) is the common choice. MISRs

---

[2] this term will be defined in section 3.5

have at least as many cells as CUT primary outputs. Rather than obtaining its next state from its left neighbor as in a LFSR, the next state of a MISR cell is the modulo 2 addition of its left neighbor and one of the CUT outputs. The feedback structure driving the leftmost cell remains unchanged.

CA have been shown as effective for signature analysis as LFSRs in [27].

2.5 Testing of Sequential Circuits

As this thesis targets combinational logic networks, we only give a brief overview of sequential testing techniques.

Testing of faults in sequential circuitry is a harder problem than that of the purely combinational case. Generation of a test set for a sequential circuit is algorithmically more complex. The situation is even bleaker when dealing with asynchronous circuitry [21].

One source of this increase in difficulty is that many faults in sequential circuits need a test sequence rather than a single vector to be detected. For example, consider testing for the output of a flip-flop stuck-at 1. A sequence of at least one vector will be needed to store 0 in the flip-flop. Beginning in the succeeding clock period, a sequence of at least one vector must be applied to propagate the error to a primary output.

To alleviate this predicament, the practice is to adopt a *design for testability* (DFT) methodology. *Scan* design is a popular DFT technique in which the sequential network is essentially transformed into a combinational (or at least "less sequential") network when in test mode. This is accomplished through the use of *scan registers*. Scan registers include all or some of the design's memory elements. When in normal operation, these memory elements function as required by the design, though possibly with an increased delay. In test mode, the memory elements in a scan register are transformed into a shift register. The input and output of the shift register are connected to chip I/O pins (if external testing) or wired to BIST circuitry. To apply a test to the CUT, test data is shifted into the scan register(s). Next, the CUT's response to the test is latched into the scan registers. Finally, the response is shifted out to the external tester or SA circuitry. The costs of using scan-based design are the increased delay and area of the scan memory elements and the increased test time incurred in the process of shifting test and responses in and out of the scan register(s).

There are many different scan-based methodologies proposed by researchers and in use in industry. These include IBM's *Level Sensitive Scan Design* [28], *Scan Path* [29], *Random-Access Scan* [30], and *Scan/Set Logic* [31, 32].

2.6 Conclusion

This chapter has provided a brief introduction to a broad discipline that has been the focus of much research in both academia and industry. We have described fundamental concepts such as fault models, stuck-at faults, fault simulation, BIST, PRPG, signature analysis, and design for testability. The following chapter gives a formal definition of cellular automaton and an overview of CA research.

3. Cellular Automata

3.1 Introduction

Cellular Automata (CA) (singular: Cellular Automaton) were introduced in section 2.4 as pseudorandom pattern generators and signatures analyzers for BIST, but we have yet to give a formal definition.

A CA can be defined as a d-dimensional lattice of *cells*, also called *sites*, each of which exist in one of a finite set of *states*. The state at each cell is updated synchronously in discrete time steps according to some function of the states of a local neighborhood of cells. The next state function at a cell is called the cell's *rule*. A *uniform* CA has the same rule at every cell, while a *hybrid* CA need not adhere to this constraint. CA are also characterized by the neighborhood over which the next state rules are defined. For 1 dimensional CA, the neighborhood is characterized in terms of *radius*; a radius r 1 dimensional CA has each cell's rule depending on the cell itself, the r closest cells to the right, and the r closest cells to the left (giving a total of $2r + 1$ cells affecting the next state of each cell). Radius 1 CA are also called *nearest neighbor*. The number of cells in a CA may be finite or infinite; as the CA of this thesis are to be implemented in silicon they are all finite. For a finite CA, the terms *size* or *length* (in the 1-dimensional case) refer to the number of cells, and is usually denoted by the variable n. Unless otherwise noted, for the remainder of this thesis *CA* will refer to 1 dimensional nearest neighbor finite CA. When considering finite CA there are various ways of defining the neighborhood of the boundary cells, i.e. the cells that live on the perimeter of the CA. (In a 1 dimensional CA, these are the leftmost and rightmost cells.) Three common approaches are:

- *null boundary*: the missing cells in the neighborhood assume the constant state 0
- *periodic boundary*: the d-dimensional CA is embedded on the d-dimensional

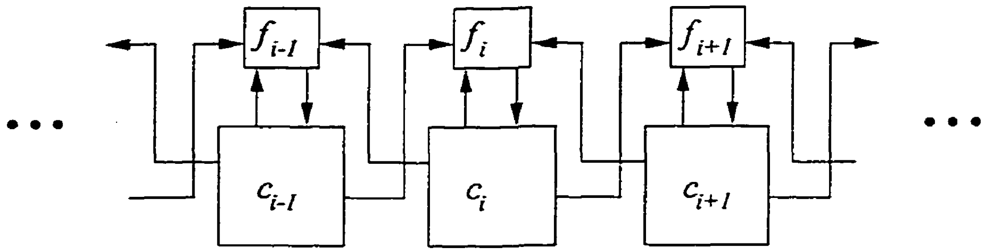


Figure 3.1: Abstract view of a 1 dimensional, radius 1 CA

torus rather than d-dimensional Euclidean space. For a 1-dimensional radius 1 CA with cells $c_0, \ldots, c_{n-1}$, the next state of cell c_0 is a function of c_{n-1}, c_0, and c_1 while cell c_{n-1} depends on c_0, c_{n-1}, and c_{n-2}.

- *intermediate boundary*: for a 1 dimensional CA, cell c_0 depends on c_0, c_1, and c_2, cell c_{n-1} depends on c_{n-1}, c_{n-2}, and c_{n-3}.

Figure 3.1 depicts a segment of a 1 dimensional hybrid nearest neighbor CA. When implemented in hardware, each cell c_i is typically a flipflop which is updated synchronously by the combinational logic $f_i(c_{i-1}, c_i, c_{i+1})$. Clocking lines are omitted for clarity.

There are various notations in the literature used when discussing CA. In this thesis the following notations are adopted. The actual cells are denoted by $c_0, c_1, \ldots, c_{n-1}$, left to right, where n is the number of cells. The states of these cells at time t are $v_0^{(t)}, v_1^{(t)}, \ldots, v_{n-1}^{(t)}$, representing the stored values, and the aggregate of these is the *state vector*[3] $v^{(t)} = (v_0, v_1, \ldots, v_{n-1})^{(t)}$; if t is understood from the context or is arbitrary the exponent is omitted. The rule at cell c_i is $f_i(c_{i-1}, c_i, c_{i+1})$. If time is arbitrary, the next state of cell c_i is c_i^+, thus $c_i^+ = f_i(c_{i-1}, c_i, c_{i+1})$.

A widely used convention used to identify one of the 256 nearest neighbor rules is the *rule number*, an integer r with $0 \leq r \leq 255$. The rule number of a rule f is the decimal equivalent of the binary number obtained when the values of the rule's truth table are listed, i.e. $[f(1,1,1)f(1,1,0)\ldots f(0,0,0)]_2$.

Note that the behavior of a CA throughout time is often referred to as *evolution*.

---

[3] often *state*, *CA state*, *vector*, or *configuration* are used in place of *state vector*

$$\begin{aligned}
f_0(c_{-1}, c_0, c_1) &= c_{-1} \oplus c_0 \oplus c_1 \\
f_1(c_0, c_1, c_2) &= (c_0 + c_1) \oplus c_2 \\
f_2(c_1, c_2, c_3) &= c_1 \oplus c_3 \\
f_3(c_2, c_3, c_4) &= c_2 \oplus (c_3 + c_4) \\
f_4(c_3, c_4, c_5) &= c_3 \oplus c_5
\end{aligned}$$

Table 3.1: The rules of an example CA

To avoid confusion with the concept of *genetic evolution* central to this thesis and introduced in chapter 4, this use of the word is neglected here.

Here we give an example of a null boundary CA with $n = 5$ cells. The rules of this CA are given in table 3.1; note that as this is a null boundary CA the nonexistent cells c_{-1} and c_5 are taken as constant 0. The rule numbers associated with the rules $f_0, \ldots, f_4$ are, respectively, 150, 86, 90, 30, and 90. The *functional digraph* of a finite CA (or any autonomous FSM) is a digraph in which each of the 2^n nodes is a state, and a single arc leaving each node follows the transition function. Figure 3.2 depicts the functional digraph for the null boundary CA of table 3.1. In this digraph, each state is represented by an integer obtained by interpreting the state vector as a 5-bit binary number with c_4 being the low order bit.

The remainder of this chapter consists of four sections, each summarizing a distinct vein of research on cellular automata. The sections are presented in the chronological order of when the research was instigated. John von Neumann coined the term *cellular automaton* and studied them as a medium for computation; his work is discussed in section 3.2. Perhaps the most famous family of CA are those that constitute Conway's *game of life* presented in section 3.3. CA are hailed as the discrete analogs of the physicist's dynamical systems and have been studied extensively by renowned physicist Steven Wolfram. Wolfram's work is summarized in section 3.4. Finally, section 3.5 gives some of the results pertaining to linear CA obtained using the tools of matrix algebra. Precluded from this chapter is previous work with evolutionary techniques targeting CA, which is reviewed in section 5.2, and the VLSI testing applications reviewed in section 2.4.

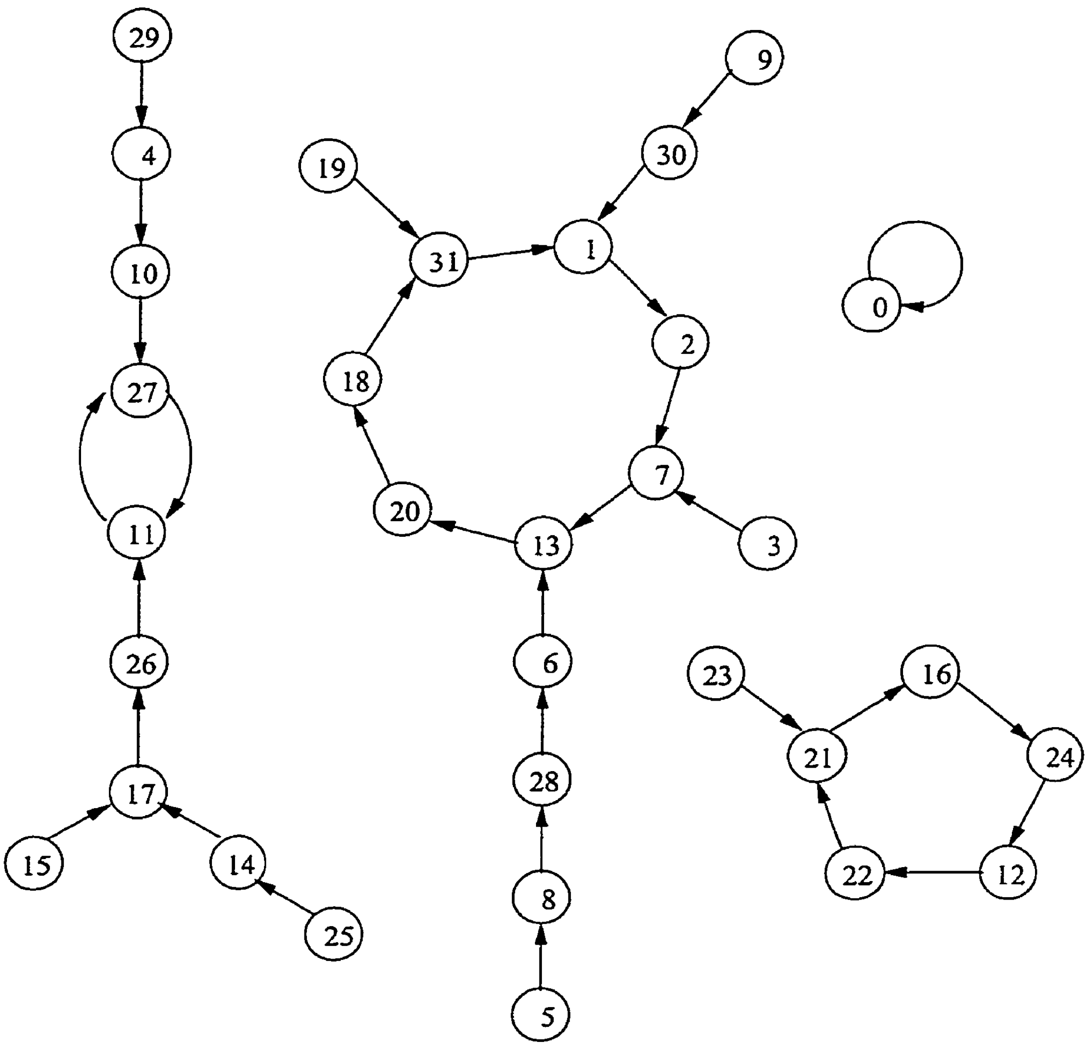


Figure 3.2: Functional digraph of the CA of table 3.1

3.2 Von Neumann's Work

The computer scientist and mathematician John von Neumann performed extensive pioneering work with CA. However, he did not investigate CA for mathematical properties, but rather as a means of facilitating his search for universal and self reproducing computational machines [33]. Von Neumann worked with 2-dimensional infinite *cellular space* in which each cell's neighborhood consists of itself and four immediate neighbors, i.e. those above, below, left, and right of the cell. He defined a cellular space with 29 states that is both *computation-universal* and *construction-universal.* In the set of 29 states there is a special blank state.

Computation-universal means that the CA is capable of simulating any Turing machine. To achieve this, he describes how, given any (deterministic) finite automaton, to construct an initial state assignment in the cellular space (with only a finite number being non-blank) that simulates the finite automaton. Also, he explains how to create physical cellular loops to read and write to the tape, thus emulating the tape mechanism of a Turing machine. Finally von Neumann explains the architecture that melds these two devices, thus allowing for implementation of any Turing machine in the cellular space.

Inspired by early work by Post and Turing, the primary goal of von Neumann's work was the investigation of self-reproductive systems [34]. Construction-universality requires the existence of a Turing machine M_c called the *universal constructor* (embedded in cellular space) that, given the description of an arbitrary Turing machine M on its tape, M_c will construct M in a blank region of the cellular space and begin simulation of M against its input (also specified on M_c's tape). von Neumann successfully built a universal constructor in cellular space, and since the Turing machine to be constructed by M_c is arbitrary, it could be M_c itself, thus self-reproduction was obtained.

As an interesting note, this work of von Neumann's was performed analytically. Later, with the aid of the computer, Codd reduced von Neumann's cellular space to one with the same universality properties but with a mere eight states [35].

3.3 Conway's Game of Life

One of the many "games" theoretically analyzed in [36] is called the *Game of Life*. The Game of Life (or simply *Life*) is an infinite 2-dimensional CA with binary states. Similar to von Neumann's cellular space, Life is a uniform CA. However in Life, the neighborhoods consist of 9 cells: the cell itself, and the eight neighbors directly or diagonally adjacent. The two states are dubbed *live* and *dead*, and the rule is expressed in table 3.2, in which s and s^+ are the current and next state, respectively, and C is the number of *live* cells of the 8 adjacent neighbors.

s	C	s^+	interpretation
live	≥ 4	*dead*	death by overcrowding
live	≤ 1	*dead*	death by exposure
live	$\in \{2, 3\}$	*live*	survival
dead	$= 3$	*live*	birth
dead	$\neq 3$	*dead*	remain dead

Table 3.2: Life's transition function

Conway and other researchers were interested in the limiting behavior of Life from various initial state assignments with a small number of adjacent *live* cells. Many such configurations received colorful names that reflect the induced behavior, some examples are the *tumblers*, *barbers pole*, *spaceships*, and *glider*. Some configurations (or *patterns*) are *stable* in that Life does not change over time. Others are *oscillatory*, i.e. they cycle through various configurations eventually returning to the initial configuration. *Translatory* patterns are oscillatory, but when a state cycle is complete, the pattern has physically moved in the Life space, the glider is a simple translatory pattern that consists of but 5 *live* cells.

Originally Conway hypothesized that Life would always yield a bounded number of *live* cells from finite initial conditions. However this was shown to be false when in 1971 a group at MIT discovered the *glider gun*, which emits a glider every 30 time units. Throughout the early 1970s much work was done finding interesting behavior in Life, most of which went unpublished [34].

Using the existence of glider guns and other constructs, Conway was able to create the basic logic gates NOT, AND, and OR, and also memory arrays in Life. Thus this simple CA could implement an arbitrary digital computer, in which the "wires" were glider paths. The conclusion is that Life is computation-universal; it was also shown that Life is construction-universal.

Though these results are theoretically very significant, more relevant to this thesis is the *unpredictability* of the patterns emerging in Life which Conway and many others studied. This is exemplified by Conway's enumeration of Life's reaction to each of a straight line of n *live* cells, $1 \leq n \leq 20$, and all 12 possible *pentominoes*

[36]. The resulting behaviors clearly do not adhere to any simple means of prediction, which suggests a sort of randomness.

3.4 A Physicist's Perspective

As a physicist, Stephen Wolfram saw CA as a means of (discretely) modeling complex physical, chemical, and biological systems. The former child prodigy sought a new model for the complexity found in the universe, which traditional mathematics had failed to capture. Familiar with the complexity found in Life, Wolfram turned to cellular automata for this purpose and during the 1980s embarked on an extensive study of CA, publishing around 20 papers on the topic. During the 1990s, he has ceased publication of his research, amassing his results for the populace in a 1200 page text boldly named *A New Kind of Science*[4].

Where the vast majority of prior CA studies focussed on 2 dimensional CA, Wolfram considered *elementary* CA to be the logical starting point for his investigations. Elementary CA are 1 dimensional, uniform, binary CA with nearest neighbor communication. Hence an infinite elementary CA can be fully described by the uniform rule, of which there are $2^8 = 256$. In his work, Wolfram presents many visual CA *growths* in which CA states are represented by a row of pixels (white for 0, black for 1), consecutive states in time being depicted on successive lines[5]. Examination of these growths give the observer an intuitive feel for the level of complexity inherent in the associated CA.

In [13, 14] CA with *legal* rules are considered. A legal rule $f(c_{i-1}, c_i, c_{1+i})$ is symmetric in c_{i-1} and c_{i+1} and has $f(0,0,0) = 0$; there are 32 such rules. Wolfram examines the behavior of all 32 infinite elementary CA with legal rules, starting from an initial state with exactly one nonzero site. Many of these are found to demonstrate fractal patterns with time; the fractal dimensions are calculated. Indeed, the growths of these CA hauntingly resemble the Sierpiński arrowhead, one of the many fractal

---

[4] to be released sometime in 2001

[5] see appendix B for examples of CA growths

sets presented in the classic text on the subject [37].

For an alternate approach, periodic boundary elementary CA of relatively small size n are examined on a *global* level, where all 2^n possible states are considered. Some states are more likely to appear than others; i.e. arise from multiple possible previous states. This phenomenon is called *irreversibility* and accounts for the self-organization prevalent in certain rules. Irreversibility implies the existence of functional digraph nodes with 0 indegree and others with greater than 1 indegree, this phenomenon is present in figure 3.2. Another characterization is the differences in the long term effect of commencing CA simulation with initial states of small Hamming distance. Wolfram briefly extends his analysis to elementary CA with $k > 2$ possible states per cell [14].

In [15], Wolfram qualitatively and quantitatively examines all 32 legal rules, and partitions them into four classifications depending on the limiting behavior as CA time becomes large:

1. Almost all initial states lead ultimately to a unique homogeneous state. In time all information regarding the initial state is lost.

2. Limiting behavior is characterized by simple separated structures. Changing the value at a cell in the initial state only affects a finite range of cells in successive states.

3. Attracting structures are chaotic patterns. Independent of the initial state, these patterns tend to exhibit the same statistical properties. A small change in the initial state affects an unbounded number of cells in future configurations.

4. The most complicated behavior, complex localized structures emerge. The temporal behavior can only be determined via explicit simulation. Wolfram hypothesizes that these CA are capable of universal computation.

Wolfram recognizes the fact that CA can be employed as random sequence generators. He investigates the use of an elementary CA with rule 30 $c_i^+ = c_{i-i} \oplus$

$(c_i + c_{i+1})$ as a generator of random sequences extensively [38]. Twenty unsolved problems in the field of CA are proposed in [39].

The preceding discussion has summarized the relevant cross-section of Wolfram's (published) work in the field. Omitted is his work applying CA to thermodynamics, fluid theory, and cryptography, and analyzing 2 dimensional CA.

3.5 Algebraic Results

Section 2.4.1 gave a brief introduction to the theory of linear finite state machines (LFSMs). This section elaborates and discusses the problem of *CA synthesis*, the solution to which is utilized in this thesis.

LFSM lend themselves to algebraic analysis because of the fact that the transition function of a LFSM can be expressed as a transition matrix A. Because the transition function of a LFSM can be expressed as a transition matrix over a finite field, many properties of these machines can be determined using algebraic techniques. Because of linearity, the all 0 state vector (or, simply, *the 0 state*) is mapped to itself under the next state function.

A CA is a LFSM if and only if every rule is itself linear; table 3.3 lists the eight possible linear rules. Of these eight rules, only the two rules 90 and 150 are nondegenerate in that they preserve two-way communication. For example a cell with rule 102 only depends on itself and its right neighbor, thus the cell and all cells to its right are never affected by the cells to the left.

Associated with a LFSM with transition matrix A is the *characteristic polynomial* $\phi(x) = det(xI - A)$. Much can be deduced regarding the structure of the functional digraph of a LFSM from its characteristic polynomial and transition matrix.

The transition function of a *group* LFSM forms a cyclic group. In a group LFSM, all states have predecessors. A LFSM is a group LFSM if and only if its transition matrix A has $det(A) = 1$ [3]. If $\phi(x)$ is irreducible, then all nonzero states lie on cycles of length k, where k is the least integer such that $\phi(x)$ divides $x^k - 1$ [9].

rule number	expression
0	$c_i^+ = 0$
170	$c_i^+ = c_{i+1}$
204	$c_i^+ = c_i$
240	$c_i^+ = c_{i-1}$
102	$c_i^+ = c_i \oplus c_{i+1}$
90	$c_i^+ = c_{i-1} \oplus c_{i+1}$
60	$c_i^+ = c_{i-1} \oplus c_i$
150	$c_i^+ = c_{i-1} \oplus c_i \oplus c_{i+1}$

Table 3.3: The eight linear nearest neighbor CA rules

An irreducible polynomial $p(x)$ of degree n is called *primitive* if the smallest such k is $2^n - 1$. It follows that a LFSM with a primitive characteristic polynomial has all $2^n - 1$ nonzero states on a single cycle; such LFSMs are called *maximum length* or *primitive.*

Nongroup LFSM have also been studied. Such machines have *transient* states that are not part of a cycle. The functional digraph always consists of one or more disjoint cycles called *attractors* with inverted trees of transients rooted on the *cyclic* nodes. States with no predecessors (i.e. the leaves of these inverted trees) have been called *garden of Eden* states; others are called *reachable.* It has been shown that in a LFSM's functional digraph, the number of predecessors of any reachable state is equal to that of the 0 state [3]. Furthermore, the trees rooted on cyclic states in the transition diagram of such CA are all isomorphic [3, 10].

LFSR and linear CA can defined by restrictions placed on the entries of the transition matrix. A (type I) LFSR has a transition matrix of the form in figure 3.3. The sub diagonal consists of all 1s, all other entries are 0 except the rightmost column, the entries $\{a_0, a_1, \ldots, a_{n-1}\}$ may be either 0 or 1. When a_i is nonzero this indicates that the next state of cell c_i is the modulo-2 addition $c_{i-1} \oplus c_{n-1}$, and thus the rightmost column dictates the placement of the EXOR gates in the LFSR (see figure 2.4). The characteristic polynomial is then given by

$$\phi(x) = x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} \cdots + a_1 x + a_0$$

$$\begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & a_0 \\ 1 & 0 & 0 & \cdots & 0 & a_1 \\ 0 & 1 & 0 & \cdots & 0 & a_2 \\ 0 & 0 & 1 & \cdots & 0 & a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & a_{n-2} \\ 0 & 0 & 0 & \cdots & 1 & a_{n-1} \end{bmatrix}$$

Figure 3.3: General form of a type I LFSR transition matrix

$$\begin{bmatrix} d_0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & d_1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & d_2 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & d_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & d_{n-2} & 1 \\ 0 & 0 & 0 & 0 & \cdots & 1 & d_{n-1} \end{bmatrix}$$

Figure 3.4: General form of a hybrid rule 90/150 CA transition matrix

Thus, given a polynomial $p(x)$, we can immediately construct a LFSR with characteristic polynomial $p(x)$ by filling in the rightmost column as per the coefficients in the polynomial. Primitive polynomials exist for all degrees n and can be generated in polynomial time [40]. These two facts in conjunct allow us to create a maximal length LFSR with any number of cells. As mentioned in section 2.4, maximum length FSM make good pseudorandom pattern generators for BIST.

In contrast, given a polynomial $p(x)$, the problem of finding a CA with characteristic polynomial $p(x)$ is difficult. In fact, the existence of a CA for any irreducible polynomial was not proven until 1995 [5]. A hybrid CA with rules 90 and 150 has transition matrix of the form of figure 3.4. The sub and super diagonals are all 1s, all other entries are 0 except for the main diagonal, which has $d_i = 0$ ($d_i = 1$) if cell i uses rule 90 (150). Unlike the entries of the rightmost column of figure 3.3, the values of $d_0, d_1, \ldots, d_{n-1}$ hold no simple relationship to the coefficients of the characteristic polynomial. Given an irreducible polynomial $p(x)$, the problem of finding a CA with characteristic polynomial $p(x)$ is called the *CA synthesis problem.* Of course given a CA, the characteristic polynomial can easily be obtained by computing the determinant $det(xI - A)$.

A near-brute force search algorithm is given in [2], but is prohibitively slow for large n. In his PhD dissertation [5], Cattell derives an efficient algorithm for CA synthesis. This algorithm uses the CA recurrence relation proved in [2]. Given a CA with n cells, let Δ_k denote the characteristic polynomial of the CA formed by removing cells $k, k+1, \ldots, n-1$, thus the characteristic polynomial of the original CA is Δ_n. Then the CA recurrence relation may be stated:

$$\begin{aligned} \Delta_{-1} &= 0 \\ \Delta_0 &= 1 \\ \Delta_k &= (x + d_k)\Delta_{k-1} + \Delta_{k-2}, 1 \leq k \leq n \end{aligned}$$

Cattell discovers that the CA recurrence relation essentially describes Euclid's greatest common divisor algorithm, with Δ_k the dividend, Δ_{k-1} the divisor, $(x + d_k)$ the quotient, and Δ_{k-2} the remainder. From this it is determined that a CA may be synthesized from a irreducible polynomial by solving a quadratic equation in $GF(2^n)$. The relevant quadratic has a known solution, and the result is an efficient algorithm to solve the synthesis problem. Other contributions of the dissertation are the demonstration that any irreducible polynomial has two CA realizations, and a proof that no periodic boundary CA has an irreducible characteristic polynomial. Cattell's efficient synthesis algorithm facilitates the production of a large number of primitive CA needed for the main work of this thesis.

4. Genetic Algorithms

4.1 Introduction

Genetic Algorithms (GA) are a class of search technique based on the natural phenomenon of natural selection and genetic evolution. They have been used to tackle hard optimization problems, i.e. those characterized by properties such as nonlinearity, discontinuity, high-dimensionality, and noisiness. This chapter does not attempt to summarize the history of research into the mechanics of GA, nor provide an exhaustive enumeration of the abundant areas in which GA have been successfully applied. It does aim to provide the background necessary for understanding of the basics of GA, and outline other evolutionary computing techniques.

Section 4.2 describes the *simple genetic algorithm*, which most GA closely resemble or expand on; an example of the simple GA in action is provided in section 4.3. The *building block hypothesis* is explained in section 4.4, while section 4.5 discusses *fitness scaling*, a mechanism employed in this thesis. Finally some other members of the family of *Evolutionary Algorithms* are summarized in section 4.6.

4.2 The Simple Genetic Algorithm

This section explains the inner workings of a basic genetic algorithm. More advanced genetic algorithms involve sophisticated genetic operators, parallel processors, and more accurate imitation of natural genetics.

The operation of a GA is "remarkably straightforward" [16]. Given a specific computational problem, a GA works against a *population* of fixed length strings of symbols called *individuals* or *chromosomes*. Each individual I encodes a candidate solution to the problem at hand, and there exists a *fitness function* $f(I)$ that maps I to a real number called the *fitness*. The fitness function is chosen by the GA user, and should be defined such that better solutions in the underlying problem domain receive higher fitness. In other words, $f(I_1) > f(I_2)$ should always indicates that the solution represented by the chromosome I_1 is superior to that of I_2. The GA evolves the population through many *generations*, starting from an initial population $P(0)$, which typically consists of completely random individuals. Generation $j+1$ is created from generation j by the application of three *genetic operators*, called *crossover*, *reproduction*, and *mutation*. Each time a genetic operator is applied, the individuals involved are selected randomly with probability proportional to the fitness. The genetic operators are repeatedly applied until enough new individuals have been created to constitute generation $j+1$ (in the simplest GA, the size of the population is fixed). The fitness of the individuals of the next generation are then evaluated, and the process iterates until some stopping condition is reached.

The three genetic operators are now described. Crossover takes two individuals I_1 and I_2 and randomly selects a *crossover point*, which is an integer between 1 and the length of the strings. The offspring I'_1 (I'_2) is produced by concatenating the prefix of I_1 (I_2) before the crossover point with the suffix of I_2 (I_1) beginning at the crossover point. Mutation replaces a symbol in a chromosome with a randomly chosen symbol from the alphabet; each symbol of an individual I is subject to mutation with a typically very small probability. The simplest genetic operator is reproduction, in which the selected individual is copied unchanged into the next generation.

The pseudocode for a genetic algorithm is given in figure 4.2. This algorithm is basically the *simple genetic algorithm* (SGA) of [16], and is used as the basis for all GA in this thesis. Note that P and P_{new} in figure 4.2 are multisets, thus the inner while loop always completes. Also, the routine `randomIndividual()` returns an individual selected randomly from P with probability proportional to the fitness.

4.3 An example

Here we follow a simple genetic algorithm that attempts to maximize the real-valued function $p(x)$ plotted in figure 4.2 ($p(x)$ is a quartic polynomial multiplied by $sin(x)$). Individuals are encoded as bitstrings of length 20; each encodes a real number by interpreting the string as a binary number with the decimal point after

```
create initial population $P(0)$
$P \leftarrow P(0)$
while halting condition not reached  do
   evaluate fitness of each individual in $P$
   $P_{new} \leftarrow \emptyset$
   while $|P_{new}| < |P|$ do
      $I_1 \leftarrow$ randomIndividual()
      $I_2 \leftarrow$ randomIndividual()
      if rand() < $p_c$ then
         $\{I_1', I_2'\} \leftarrow$ crossover($I_1, I_2$)
      else
         $I_1' \leftarrow I_1$
         $I_2' \leftarrow I_2$
      endif
      for $i \leftarrow 1 \ldots n$ do
         if rand() < $p_m$ then
            mutate($I_1', i$)
         end if
         if rand() < $p_m$ then
            mutate($I_2', i$)
         end if
      end for
      $P_{new} \leftarrow P_{new} \cup \{I_1', I_2'\}$
   end while
end while
```

Figure 4.1: The simple genetic algorithm

the 7^{th} bit from the left. Since the goal is function maximization, we take the fitness of an individual as simply the value of $p(x)$. Also, we wish to find the maximum on the interval $[0, 100]$, so values of x in $(100, 127)$ are assigned a fitness of 0. In this example, the population size used is 10.

Table 4.1 gives the initial and second generations of an actual run of the GA. The individuals of the first generation are chosen completely randomly, with each bit being 0 or 1 with equal probability. The top half of table 4.1 lists the individuals of the initial population under the column "chromosome", each is assigned an ID under the leftmost column. To the right of the chromosome is the value of the individual when decoded as binary number with the most significant bit corresponding to 2^6.

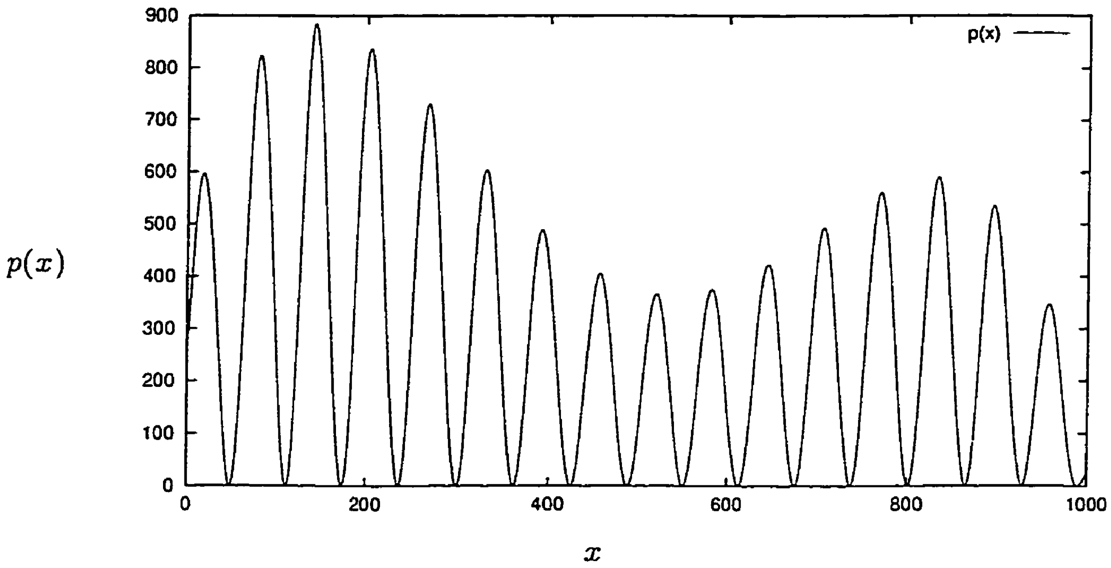


Figure 4.2: The target function in the example GA

The value of $p(x)$ is given in the next to leftmost column. We see that the fittest individual of this initial population has fitness 535, while a visual inspection of figure 4.2 reveals that the greatest possible fitness is close to 900.

The bottom half of table 4.1 shows the individuals of the second generation, produced via applications of the genetic operators of crossover, mutation, and reproduction against individuals of the initial generation. The rightmost column labeled "genetic history" indicates what operators and parents from the initial population bestowed each individual. $crossover(\alpha, \beta, c)$ indicates that the individual was formed by the crossover operator concatenating the leftmost c bits of individual with ID α with the rightmost $20 - c$ bits of individual β. When individual α was simply reproduced without crossover the genetic history states $reproduce(\alpha)$. Finally, an additional $mut(d)$ means that the individual arising from the primary operator experienced a mutation at bit d, i.e. bit d was flipped. We see that the fittest individual of the new generation enjoys an improvement over that of the prior. In fact even the second fittest is more fit than the initial best. Another important observation is that the average fitness across the population has increased drastically, from 191 to 326.

ID	chromosome	x	$p(x)$	genetic history
0	10100111101110110000	83.865	535.443	
1	11000000101000100100	96.317	297.995	
2	00100100010101000001	18.164	158.613	
3	10010101011100110110	74.725	101.640	
4	10111011000110001001	93.548	76.608	
5	11000111000100111011	99.539	10.927	
6	01010101011100010000	42.721	10.375	
7	11100101000010110011	114.529	0	
8	11100001011001110101	112.702	0	
9	11110101111110000001	122.984	0	
0	10100101011100110110	82.725	550.091	$crossover(0,3,4)$
1	10100111101000100100	83.817	543.443	$crossover(0,1,8)$
2	10100111110101000001	83.914	526.767	$crossover(0,2,7)$
3	10010111101110110000	75.865	399.596	$crossover(3,0,4)$
4	11000000101000100100	96.317	297.988	$reproduce(1)$
5	11000000101110110000	96.365	292.020	$crossover(1,0,8)$
6	00100000010100110110	16.163	246.549	$crossover(2,3,11), mut(5)$
7	00100100010101000001	18.164	158.633	$reproduce(2)$
8	00100100001110110000	18.115	142.648	$crossover(2,0,7)$
9	10010101011101000001	74.727	101.926	$crossover(3,2,11)$

Table 4.1: Initial and second generations of the example GA

4.4 the Building Block Hypothesis

Given the computational problem and solution encoding, the *building block hypothesis* is a theory that predicts how well a GA will perform [16]. In the study of GA, the term *schema* refers to special sets of individuals, defined as follows. Suppose the GA is working against strings of length n over an alphabet Σ. A schema is a string of length n over the alphabet $(\Sigma \cup \{*\})$, where $*$ represents the don't care value, i.e. *any* member of Σ. A position i in a schema $H = h_0 \ldots h_{n-1}$ is called *fixed* if $h_i \in \Sigma$. The set of strings represented by a schema H consists of all strings $X = x_0 \ldots x_{n-1}$ such that $x_i = h_i$ if i is fixed; a member of this set is said to *match* H (and vice versa). The *defining length* $\delta(H)$ of a schema H is the number of positions between the rightmost and leftmost fixed positions in H (inclusive), while the *order* $O(H)$ of H is simply the number of fixed positions. For example the schema $H = **01**1$ has $\delta(H) = 5$ and $O(H) = 3$. Given a population of individuals, the *observed performance* of a schema H is the average fitness over all individuals matching H.

Equations can be derived predicting the expected number of individuals in the next generation matching a particular schema. When reproduction is the only operator taken into account, the resulting equation predicts that schema with observed performance consistently higher (lower) than the average fitness receive exponentially increasing (decreasing) copies in successive generations. When crossover is included in the model, we find that schema with good observed performance *and* short defining length are most likely to propagate. This stems from the intuition that schema with long defining length are more likely to be "broken" by crossover. Finally, when mutation enters the picture, one finds that another factor in schema survival is the order; those with low order are less likely to be destroyed when mutation is applied.

The implication is that short, low order schema with high observed performance enjoy a representation that increases exponentially throughout GA evolution; such schema are called *building blocks*. This result is known as the *Fundamental Theorem of Genetic Algorithms*, which is formally stated:

$$m(H, t+1) \geq \frac{m(H,t)f_t(H)}{\bar{f}_t}\left[1 - p_c\frac{\delta(H)}{n-1} - O(H)p_m\right]$$

where $m(H,t)$ is the expected number of individuals matching schema H at generation t, $f_t(H)$ is the observed performance of H in generation t, $\bar{f}_t$ is the average fitness over all individuals in generation t, and p_c and p_m are the respective probabilities of crossover and mutation.

Now consider the optimal or near-optimal solutions to a problem, and the strings they map to under the GA encoding. If the building blocks that match one or more of these optimal individuals realize high observed performance, then we can expect the GA to converge to a good solution. However, if many of the building blocks also match a large number of weak individuals, then they are likely to be killed off. This concept is referred to as the *building block hypothesis*. A predicament faced when applying a GA is determining if the building blocks present in good solutions will usually receive above average fitness in other individuals that they represent. This can be a complex problem, since a single building block matches a large proportion of the entire search space.

4.5 Fitness Scaling

This section describes the *linear fitness scaling* (LFS) of Goldberg's simple GA [16]. Fitness scaling maps the raw fitness f returned by the user-defined fitness function to a *scaled fitness* f'. Under LFS, this mapping takes the form $f' = af+b$, the coefficients a and b are recomputed each generation such that the expected number of times the fittest individual is selected for participation in a genetic operation is equal to the user-specified parameter C_{mult}, and the average individuals are (each) expected to be selected once. Let f_{max}, f_{avg}, and f_{min} respectively denote the maximum, average, and minimum raw fitnesses in a specific generation. Then we require $f'_{max} = C_{mult}f_{avg}$ and $f'_{avg} = f_{avg}$, i.e. the scaled fitness of the fittest individual is C_{mult} that of the average fitness. These points yield:

$$\begin{aligned} a &= \frac{f_{avg}(C_{mult}-1)}{f_{max}-f_{avg}} \\ b &= f_{avg}\frac{f_{max}-C_{mult}f_{avg}}{f_{max}-f_{avg}} \end{aligned} \tag{4.1}$$

A slight problem with this function is that after evolution has occurred for some time and f_{avg} and f_{max} become close, the slope of the function can become sufficiently steep as to map f_{min} to a negative value. This is undesirable, since it complicates the process of computing selection probabilities. The remedy is to test if equation 4.1 would result with $f'_{min} < 0$, and if so, define a and b such that $f'_{min} = 0$, keeping the constraint $f'_{avg} = f_{avg}$.

In practice, the values of C_{mult} employed are typically in the range $[1.2, 2]$; in this thesis, $C_{mult} = 1.5$ [16].

4.6 Related Techniques

GA fall under the umbrella of *Evolutionary Algorithms*, along with its cousins *Evolutionary Programming*, *Classifier Systems*, and *Genetic Programming*. Also in this group are *Evolution Strategies*, not described here.

Evolutionary Programming (EP) [41] is a technique similar to GA, with two notable exceptions. EP does not use a crossover operator, the new population is created by applying mutations of various degrees of severity to the previous population. Also, the population size need not be constant. EP was originally used (with limited success) to evolve small FSM that predict the output sequence of a Markov process; such FSMs are of use in artificial intelligence.

Classifier Systems are a type of *genetics-based machine learning* system [16], and are briefly described here. Embedded in an environment, classifier systems receive input messages via *detectors*, which trigger internal if-then rules called *classifiers*. The triggered classifiers compete in a credit system, in which those with higher *strength* are more likely to win the privilege of transmitting to other classifiers or produce output. The population of classifiers are subject to a GA, in which they are subject to the usual genetic operators. Classifier Systems have been used to teach machines a variety skills, including maze navigation [42], gas pipeline control [43], and Norwegian verb forms [44].

Genetic Programming (GP) refers to an extension of the GA paradigm in which the individuals under evolution are actual computer programs. Rather then being encoded as fixed length strings, the programs are processed as rooted, labeled, edge-ordered trees. This tree representation is essentially the parse tree a compiler would build for the program. All functions used in defining the programs are defined a priori in a *function set*. Crossover is performed between two parent programs by randomly selecting a node in each tree, and then swapping the subtrees rooted at these nodes. Besides mutation, two other secondary genetic operators are utilized in GP. *Permutation* selects a node, and then permutes the children, in effect permuting the arguments to the function. *Editing* requires a list of editing rules, which are used to simplify programs. For example, the occurrence of $(NOT(NOT(X)))$ might be replaced with (X). Koza [45] provides a comprehensive survey of GP.

5. CAGA and Advanced Operators

5.1 Overview

This chapter presents the central work of the thesis. As described in chapter 3, GAs are an approach to complex optimization problems with search spaces too large for exhaustive evaluation. The problem we tackle is certainly such, and can be formally stated as follows:

Given a combinational network C with n primary inputs, find an n-cell cellular automaton A and a binary n-vector v such that when started in state v, A generates tests for all single stuck-at faults in C in a minimal number of clock ticks.

The size of the search space here is an impressive 2^{9n} (for each primary input we have a CA cell defined by an 8-bit truth table as well as 1 bit for the initial state). The encoding used is either a 256-ary or 512-ary string of length n, depending on whether the initial CA state is included in the chromosome. To what degree this problem and choice of encoding satisfies the building block hypothesis is clearly intractable. Thus evaluation is performed empirically over the nonredundant ISCAS '85 benchmark combinational circuits [17]. Details pertaining to these ten benchmarks can be found in appendix A.

The remainder of this chapter is partitioned as follows. Section 5.2 gives a brief survey of previous research relating to the creation of "good" CA-based pattern generators. In 5.3, the cellular automata genetic algorithm (CAGA) is presented and issues regarding handling of the initial vector and approach to mutation are addressed. Also experiments with differing crossover probabilities are performed. Section 5.4 defines and investigates the use of two advanced "application specific" genetic mutation operators as means of improving the power of CAGA. The possibility of improving the crossover operator is discussed in 5.5. A simple means of further shortening test length is given in section 5.6. The chapter is summarized and concluded in section 5.7.

5.2 Related Work

Genetic evolution of CA is not a new concept. Here we summarize previous research on finding good CA BIST generators and on the more general problem of finding a CA that satisfies certain randomness properties, such as some subset of Knuth's randomness tests [46]. Note that the history of "non-evolutionary" CA work is outlined in chapter 3.

In [45], Koza demonstrates how a random number generating 1-dimensional uniform CA on 32 cells is evolved. The fitness function is based on the *entropy* (a measure of randomness) of the bit sequence produced by a fixed cell over 4096 clock ticks. Somewhat surprisingly, the evolution tends to converge to a rule-30 CA, which has previously been shown by Wolfram to satisfy several common randomness tests [38]. Koza also extends his techniques to the evolution of a 2-dimensional CA.

Cellular Programming refers to the local evolution of non-uniform CA to perform computational tasks [47]. Cellular programming has been investigated when the target computational task is random sequence generation. Instead of a population of CA, cellular programming involves a single non-uniform CA. In each generation, a fitness is assigned to each cell based on the entropy of the bit sequence produced by the cell. The reproduction and crossover operators are only performed between the rules of adjacent cells; mutation is also applied. It is demonstrated that as this evolution proceeds, the average entropy of all cells in the CA approachs the maximum possible value [48].

There has been extensive research on applying GAs to various VLSI problems by the Electronic CAD & Reliability Group at Politecnico di Torino in Italy; the sequel elaborates.

The most relevant to the work of this thesis is that of Chiusano et al. who implement a GA working on a population of non-uniform nonlinear CA with fitness based on fault coverage of a specific CUT [49]. A restricted 2-dimensional CA is the theme structure; this variation has a width of n cells, where the CUT has n primary inputs, and a vertical depth of 2 cells. Chiusano et al. target the stuck-at faults of sequential circuits. As outlined in section 2.5, testing sequential circuits is a harder problem then the testing of purely combinational networks. The motivation behind the use of this $2 \times n$ CA structure is to provide the generator with the resources to produce the *ordered* sequences necessary when testing sequential CUTs.

Low power BIST adds another dimension of complication to the standard testing problem in that the power consumption of the chip when in test mode is sought to be minimized (or at least constrained). In [50], Corno et al. describe a technique for synthesis of nonlinear CA when the objective is *low power* BIST. They do not use a GA, but rather another probabilistic algorithm: *Random Mutation Hill Climber*. This technique involves evaluation of randomly chosen mutations on a CA until an improvement is found, then feeding back the new CA into this process. Of course the fitness function here is not only based on fault coverage and test length, but also an estimation of power consumption. Though the search algorithms differ, an important similarity between this thesis and the work in [50] is that both use primitive CA as starting points.

In [51], the group use a GA for automatic test pattern generation for sequential logic. A significant contribution is the concept of improving a GA by using advanced genetic operators that are specific to the underlying problem. Development and evaluation of such operators is a primary effort of this thesis.

The Politecnico di Torino group has also investigated the genetic evolution of a CA for what is dubbed *circular CA BIST*, in which the same CA is used as the TPG and output compactor [52, 53]. Other areas of VLSI in which this group has explored the use of GAs include: floorplan area optimization [54], automatic test pattern generation [55, 56, 57], selection of flip-flops for partial scan [58], non-aliasing output compaction [59], and equivalence verification [60].

5.3 The Cellular Automata Genetic Algorithm

5.3.1 CAGA Described

In this section the central idea of the thesis is presented: a genetic algorithm that evolves a nonlinear CA that is "good" at testing some target combinational circuit for stuck-at faults. Here, "good" translates to "able to detect some target percentage of all stuck-at faults in as few clock ticks as possible". This subsection describes the algorithm, its parameters, and which parameters will remain fixed throughout the thesis. Section 5.3.2 considers two different mutation strategies and inclusion of the initial CA state in the chromosome. Section 5.3.3 presents a set of experiments that justifies using a crossover probability of 0.6.

The CAGA algorithm is essentially the simple genetic algorithm (SGA) given by Goldberg [16] applied to an initial population of distinct primitive CA. These CA are all of length n, where n is the number of primary inputs to the target circuit (CUT). The test vector seen by the circuit is simply the state of the CA, i.e. no additional gating or cells are used. The raw fitness f_{raw} of a CA is determined as follows. Fault simulation determines the test length t required for the target coverage, F_{cov}. The parameter T_{max} controls the maximum number of vectors to generate before fault simulation halts, i.e. if a CA has not achieved the target coverage after T_{max} vectors, a value of $t = T_{max}$ is returned. f_{raw} is then computed using the simple formula $f_{raw} = T_{max} - t$. The choice of T_{max} itself is such that most of the initial population has positive raw fitness. The actual value of T_{max} for each benchmark is given in appendix A.

In CAGA, the chromosomes have length n and the value at position i holds the rule number at cell i of the associated CA. If Goldberg's SGA paradigm were to be followed precisely, the initial population would consist of completely random individuals, i.e. the rule of each cell of each individual would be selected uniformly from the set $\{0,1,\ldots,255\}$. One does not intuitively suspect that a CA created in such a manner is very likely to be even a mediocre test generator; this suspicion stems from both preliminary experimentation and the fact that randomly generated CA do not normally generate sequences with good randomness properties.

If such an initial population was employed, T_{max} would necessarily be very large, thus impeding run times significantly. To remedy this situation, CAGA is initially populated with (usually distinct) primitive CA. This initial population of N primitive CA is obtained by using Cattell's efficient CA synthesis algorithm, derived in [5], to each of a population of degree n primitive polynomials. In turn, the list of primitive polynomials is generated efficiently using the algorithm outlined in [40]. Note that when n is sufficiently small compared to N, i.e. when there is less than N primitive CA of length n, the initial population will have duplicates.

Determining the optimal population size for a GA is nontrivial. Too small of a population causes premature convergence, while using a very large population incurs long computation times due to processing of redundant individuals. However, the focus of this thesis is obtaining good results in some reasonable amount of time, rather than doing so as efficiently as possible. In the absence of GA runtime constraint, the use of a conservative (i.e. large) population size is "good practice" [61]. As such, a population size of 300 is used.

According to De Jong's landmark dissertation [62], crossover and mutation probabilities of $p_c = 0.6$ and $p_m = 0.001$, respectively, gave consistently good performance over his benchmark functions. These parameter values have since been incorporated so often that they have become known as the "standard settings" [18]. We present a set of experiments that verify that the standard setting for p_c is a good choice for the problem at hand. Mutation on the other hand is specified on a "per individual" basis, rather than per gene or bit. Given a value for p_m, typically in the range [0.10,0.15], each gene is mutated with probability $1-(1-p_m)^{1/n}$, the effect of which is that a CA undergoes *at least one mutation* with probability p_m.

The scaling coefficient C_{mult} gives the expected number of times the fittest CA from generation g will be selected for reproduction in generation $g+1$, see section 4.5. Values between 1.2 and 2.0 have been successful in practice [16]; here the value of 1.5 is used.

An auxiliary mechanism used in CAGA not present in Goldberg's SGA is that the fittest k_{rep} individuals of generation g are reproduced unaltered into generation $g+1$. This allows for use of a higher than normal mutation rate, since the elite individuals are never lost in the evolution. Of course, use of a higher mutation rate further encourages the introduction of nonlinearities, thus allowing for CA with large numbers of nonlinear rules to be visited in the search process. A value of 10 is used for k_{rep}; the fittest k_{rep} *distinct* CA are reproduced each generation in this manner, but still included in the mating pool for creation of the remaining $N-k$ individuals.

Regarding the fault simulator itself, the parallel pattern fault simulator *fsim* [25] has been integrated into the genetic algorithm code, thus preventing the overhead of making calls to an external executable and transferring test data via temporary files. The fsim algorithm simulates packets of consecutive vectors in parallel, the size of these packets being dictated by the parameter b, which is typically a power of 2. Typically the larger b is, the faster fsim runs, however, the test length returned by fsim is a multiple of b. Thus the granularity of fitness levels is adversely affected by large b. Given these tradeoffs, it makes sense to set b proportional to the expected test length of the CUT.

Finally, CAGA needs to have a halting condition. The choice made here is to run *all* experiments such that after given number Δg of generations without seeing an improvement, CAGA halts. Obviously the higher Δg is set, the more likely CAGA is to make an improvement and buy another Δg generations to work with. Unless otherwise noted, a value of 200 is used in this thesis, as this tends to keep the algorithm's runtime down to about 3 or 4 days for the "hardest" circuit.

The parameters of CAGA are now summarized:

- the n-input combinational CUT, given in the ISCAS '85 netlist format.
- crossover probability $p_c = 0.6$.
- mutation probability p_m.
- population size $N = 300$.

- $P(0)$: the initial population of N primitive CA.

- $C_{mult} = 1.5$: the coefficient used in the linear scaling function; determines the expected number of selections of the fittest individual of a generation.

- $k_{rep} = 10$: the number of CA that are reproduced unchanged in the next population.

- T_{max}, the maximum number of vectors used to evaluate the raw fitness of a CA.

- $b \in \{1, 2, 4, 8, 16, 32\}$, the number of inputs vectors processed by the fault simulator in parallel.

- F_{cov}, the target fault coverage (expressed as a percentage).

- $\Delta g = 200$, the number of generations without improvement before CAGA halts.

In all experiments, $F_{cov} = 100\%$, with the exception of experiments in which either of c2670 or c7552 are the CUT. For these two circuits, test lengths required from typical primitive CA for 100% coverage are at least an order of magnitude greater than the other benchmarks. As such, the coverage used for c2670 and c7552 is always 99%.

5.3.2 Choices Regarding Mutation and Initial Vectors

There are two issues left unresolved regarding the use of GA techniques to evolve a CA for BIST. One pertains to how mutation is performed; the other deals with the handling of the initial vector of the CA being processed by the GA. We discuss two possible choices for each and then give results from a set of experiments conducted to evaluate all four possible combinations of approaches.

5.3.2.1 Slight or Full Rule Mutation

If we view the chromosomes as length n strings over a 256-ary alphabet, then the mutation operator, when applied, fully replaces a cell's rule with a rule drawn uniformly from {0,1,...,255}. Alternatively, once a gene has been singled out for mutation, the operation could be performed by randomly flipping one of the eight bits defining the truth table of the rule of the corresponding cell. The effect of this "slight" mutation is either the addition or deletion of one minterm from the rule. It is unobvious which of the two operators is likely to result in a more productive GA. On one hand, slight mutation would intuitively seem to have a less drastic effect on the structure of the functional digraph of a CA, and, indeed, its ability to test. On the other hand, full mutation encourages a more thorough exploration of the search space, allowing the more unbalanced rules to be quickly introduced into the population.

For instance, slight mutation causes a linear rule to become a rule with either 3 or 5 ones in its truth table. Suppose the hard to test faults in the CUT are mostly tested by vectors with 0 on input i. Then it would seem that shorter test lengths would be promoted by placing a rule with only one or two minterms at cell i. For CAGA to arrive at such a rule under slight mutation, two or three mutations at the appropriate positions must occur – an unlikely event. Full mutation facilitates direct introduction of such rules, but also is more likely to bring about trivial rules or rules that break two-way communication in the CA, though this breakage might not necessarily be detrimental. Clearly, experimentation is necessary to determine the better mutation operator.

5.3.2.2 Inclusion of Initial Vector in the Chromosome

In this section we address the question as to whether the initial vector should be included in the chromosomes of the GA. Suppose we choose some arbitrary initial vector $v^{(0)}$ that all CA in all generations are seeded with before fault simulation. This somewhat limits the GA's ability to find the best CA/initial vector pair in that the initial vector is fixed. In effect the problem being addressed is that of finding the best CA for testing the CUT with the prescribed initial vector.

The alternative is to include the initial vector in the chromosome of the individuals, and allow the initial vector to be unique to each individual and undergo

	BOIP		Slight Mutation		Full Mutation	
circuit	FIV	VIV	FIV	VIV	FIV	VIV
c1355	1240	1208	984	968	1000	**952**
c1908	4320	4400	2064	1992	2056	**1576**
c2670	29024	37024	**544**	1216	1248	576
c3540	5568	7888	3008	**1488**	3072	4336
c432	180	208	176	**164**	180	176
c499	508	404	320	304	**280**	348
c5315	1216	1320	**728**	1048	992	824
c6288	55	50	35	**31**	34	37
c7552	5184	6752	1696	2048	2144	**1632**
c880	3136	3088	400	368	**352**	656

Table 5.1: Results of mutation and initial vector approach experiments

crossover and mutation in conjunction with the rule array. Note that inclusion of the initial state increases the alphabet size from 256 to 512 (each CA cell is now defined by 9 bits). Under this approach, each primitive CA in $P(0)$ is assigned a randomly selected initial vector. These vectors are constructed by setting the ith bit to 0 or 1 with equal probability. The crossover operator proceeds as expected; given individuals A and B, a crossover point k is randomly selected, and the two offspring consist, respectively, of the concatenation of the first k rules and initial state bits of A (B) with the last $n - k$ rules and initial state bits of B (A). When a gene is selected for mutation, slight mutation is performed by uniformly choosing and flipping one of the 9 bits. Thus the initial bit of the selected cell is flipped with probability 1/9, while the rule is modified with probability 8/9. Full mutation uniformly selects a new rule and initial bit, i.e. a new symbol from the 512-ary alphabet is chosen to replace the current symbol.

We call these two approaches *fixed initial vector* (FIV) and *variable initial vectors* (VIV), respectively. As in the question of slight verses full mutation, it is unclear whether FIV or VIV will result in the superior CAGA.

5.3.2.3 Results

Experiments were run for each of the four possible choices: slight mutation with FIV, slight mutation with VIV, full mutation with FIV, and full mutation with VIV. For the FIV experiments, the initial vector used was always $v^{(0)} = 101010\ldots$. The results are summarized in table 5.1. The first column gives the name of the CUT (see appendix A for details). The next two columns give the *best of initial population* (BOIP) for the experiments with fixed and variable initial vectors, respectively. The BOIP is the shortest test length required over all primitive CA in $P(0)$, i.e. the best of 300 primitive CA. Of course this only varies between FIV and VIV runs as the type of mutation employed has no affect on $P(0)$. BOIP is included here to demonstrate the power of CAGA regardless of which of its four variants is used. One will observe that the effectiveness of CAGA varies from profound (against c2670) to almost negligible (c432).

The rightmost four columns give the test length required by the best CA found by each of the four CAGA variants. The lowest test length on each row is indicated in **bold font**. We observe that these "winning" CA are quite evenly distributed across the four columns.

As the running time of CAGA against these particular benchmarks and parameter values is on the order of several days, time did not allow execution of multiple runs with differing `drand48()` seeds. Of course doing so would strengthen the repeatability of these experiments. The results of the four approaches in table 5.1 along with results forthcoming in this thesis verify that CAGA always finds CA with test lengths in the same "ballpark" for a given benchmark[6]. Also omitted from this thesis is the exact distribution of fitnesses in the final generation of CAGA. Typically these populations have many very fit individuals that are closely related, i.e. of small or zero Hamming distance, as well as weak individuals that were created by "bad" genetic operations against the previous generation.

To provide another view of the data, table 5.2 expresses the test lengths as

---

[6] a notable exception are two experiments for c2670 in table 5.4

circuit	Slight Mutation		Full Mutation	
	FIV	VIV	FIV	VIV
c1355	18.5	19.9	17.2	21.19
c1908	52.2	53.9	52.4	63.52
c2670	98.1	95.8	95.7	98.02
c3540	46.0	73.3	44.8	22.13
c432	2.2	8.9	0.0	2.22
c499	20.8	24.8	30.7	13.86
c5315	40.1	13.8	18.4	32.24
c6288	30.0	38.0	32.0	26.00
c7552	67.3	60.5	58.6	68.52
c880	87.0	88.1	88.6	78.76
average	46.2	47.7	43.8	42.6

Table 5.2: Results of mutation and FIV/VIV experiments, as percent improvement

percentage improvement over the best of the two BOIPs. It appears that on average slight mutation achieves a better improvement ratio. The best combination of options occurs when slight mutation is used with VIV, providing a mean improvement of 47.7%.

These experiments do not strongly point at one approach as the most powerful. We will later find this beneficial, as two advanced genetic operators are introduced. One of these requires VIV, while the other can be thought of as a *strategically applied slight mutation*. Due to the results of this section, we can be confident that these advanced operators will not inherently weaken CAGA.

Appendix B contains CA growths for the best of initial population and best evolved CA for each benchmark in the experiments for full mutation with FIV[7]. Note that some best evolved nonlinear CA growths have distinctive features not seen in any of the primitive CA, i.e. those for c2670, c499, c6288, and c7552.

---

[7] though these experiments were not the most successful of the four combinations, full mutation typically results in a more visually pronounced difference between the two growths.

circuit	$p_c = 0$	$p_c = 0.2$	$p_c = 0.4$	$p_c = 0.6$	$p_c = 0.8$	best
c1355	1024	1072	1016	1000	888	0.8
c1908	2432	2488	2832	2056	1424	0.8
c2670	576	2720	5472	1248	1184	0.0
c3540	4592	4448	2944	3072	3440	0.4
c432	180	180	176	180	180	0.4
c499	352	368	284	280	276	0.8
c5315	1112	1080	888	992	800	0.8
c6288	47	42	32	34	35	0.4
c7552	4256	3616	2944	2144	3808	0.6
c880	448	432	752	352	448	0.6
					average	0.56

Table 5.3: Experiments with differing values of p_c

5.3.3 Crossover Probability Experiments

As stated in section 5.3.1, previous work in the study of GAs has determined that using $p_c = 0.6$ seems to provide good results in a wide variety of GA applications. In this section we verify that this is indeed true for the problem at hand. Experiments were run with five different values of p_c, 0, 0.2, 0.4, 0.6, and 0.8. This was the only parameter varied across these runs. FIV with $v^{(0)} = 1010\ldots$ was used; full mutation was applied with probability $p_m = 0.15$.

Table 5.3 lists the results for each value of p_c. The final column indicates the value that caused CAGA to arrive at the best CA for the given CUT. When these best values of p_c were averaged, the result was 0.56, which indicates that $p_c = 0.6$ is a reasonable choice for this paramenter. Typical of GA research, there is a deviant instance in which the algorithm was "lucky", namely the experiment for benchmark c2670 with $p_c = 0$; of course without the crossover operator, the algorithm cannot be considered truly genetic.

5.4 Advanced Mutation Operators

5.4.1 Bit Role Flipping

5.4.1.1 Discussion

Bit role flipping (BRF) is the name given to a mutation like mechanism that completely preserves the state-transition structure of a CA. Similar to full and slight mutation, BRF is done with respect to a CA cell, say cell c_i. Then the sole difference between the state-transition diagram of a CA and its BRF-mutated clone is that every vector has the bit at position i flipped. BRF not only changes the rule at cell c_i, but also the rules at the neighboring cells c_{i-1} and c_{i+1}.

Given the rule f_i at cell c_i, define functions f_i^R, f_i^C, and f_i^L as follows.

$$\begin{aligned} f_i^L(v_{i-1}, v_i, v_{i+1}) &= f_i(\overline{v_{i-1}}, v_i, v_{i+1}) \\ f_i^C(v_{i-1}, v_i, v_{i+1}) &= f_i(v_{i-1}, \overline{v_i}, v_{i+1}) \\ f_i^R(v_{i-1}, v_i, v_{i+1}) &= f_i(v_{i-1}, v_i, \overline{v_{i+1}}) \end{aligned}$$

We may now define the effect of BRF at cell c_i : f_i is replaced with $\overline{f_i^C}$, and the neighboring rules f_{i-1} and f_{i+1} are replaced with f_{i-1}^R and f_{i+1}^L, respectively. Additionally, the bit at position i in the initial vector $v^{(0)}$ is flipped.

The motivation behind the BRF operator is twofold. First, preservation of the state-transition structure guarantees that if a CA-initial-vector pair $I = (A, v^{(0)})$ achieves the target fault coverage in k vectors, then the individual $\hat{I} = (B, u^{(0)})$ resulting from an application of BRF *will not visit a repeated state in the first k clock ticks.* The same assertion cannot be made with regard to slight or full mutation and hence we may confidently apply BRF with higher probability than slight or full mutation. Second, the vectors generated by the two individuals are *very similar*, namely for all $t \geq 0$, $v^{(t)}$ and $u^{(t)}$ are the same with the exception $u_i^{(t)} = \overline{v_i^{(t)}}$ for some i.

circuit	best of 5.1	$p_m = 0.15$		$p_m = 0$	
		$p_{BRF} = 0.2$	$p_{BRF} = 0.4$	$p_{BRF} = 0.15$	$p_{BRF} = 0.5$
c1355	952	**920**	**936**	1008	**944**
c1908	1576	**1480**	2072	2912	2600
c2670	544	768	576	10368	14368
c3540	1488	3008	4224	4768	3184
c432	164	164	**152**	168	164
c499	280	332	316	352	304
c5315	728	880	952	760	**712**
c6288	31	34	33	43	44
c7552	1632	1632	1760	2112	**1568**
c880	352	672	640	1184	976

Table 5.4: BRF results for various values of p_{BRF} and p_m

5.4.1.2 Results

BRF was incorporated into CAGA by performing the operation at the same stage as full mutation with probability p_{BRF}, independent of application of full mutation. Four experiments were performed, with varying values of p_{BRF} and p_m; table 5.4 summarizes. The third and fourth columns give results for p_{BRF} set to 0.2 and 0.4 respectively, both using $p_m = 0.15$. In the fifth and sixth columns the full mutation operator was never applied, with respective values of p_{BRF} 0.15 and 0.5.

Note that application of the BRF operator on a linear CA simply negates the rules at the cells neighboring the point of mutation and the bit at the point of mutation in the initial vector. Since $P(0)$ consists entirely of the linear rules 90 and 150 only, it follows that all individuals in the experiments with $p_m = 0$ only have these two rules and their nonlinear negations, 165 and 105.

In table 5.4 the second column reproduces the *best* test lengths over all four columns of table 5.1. Bold values in table 5.4 indicate test lengths that are better than the second column. BRF seems to work well against some benchmarks while the operator impedes evolution for others; there does not appear to be any predictable pattern in table 5.4. For example, the rightmost column in which only BRF mutation is used (with the very high probability $p_{BRF} = 0.5$), the GA outperformed all experiments of section 5.3.2.3 for three benchmarks. However the same parameter set failed miserably for others, the most prominent being c2670. For exactly half of the benchmarks, at least one of the paramenter sets of table 5.4 obtained a better CA. The most influenced benchmark was c1355, for which three of the four BRF experiments achieved better results then the previous experiments. In the following section we discuss another advanced mutation operator which provides more consistent improvement.

5.4.2 Last Possible Mutation

5.4.2.1 Discussion

Very little can be said about the vector sequence generated by a mutated individual when compared to that of the original. *Last Possible Mutation* (LPM) is a mutation operator that seeks to remedy this situation.

The following discussion requires the definition of $\hat{t}$, which is used in the context of an individual I and a mutated clone $\hat{I}$. If I and $\hat{I}$ generate sequences $v^{(0)}, v^{(1)}, \ldots$ and $u^{(0)}, u^{(1)}, \ldots$ respectively, let $\hat{t}$ denote the largest time such that $v^{(0)} = u^{(0)}, v^{(1)} = u^{(1)}, \ldots, v^{(\hat{t})} = u^{(\hat{t})}$. Then LPM can be described as a slight mutation in which the point of mutation, rather than being selected at random, is chosen such that $\hat{t}$ is maximized.

The motivation behind LPM is this. Recall that when pseudorandom testing, the majority of the faults are detected early on in the test sequence. Thus a fit individual I would be expected to cover a large number of faults during the onset of its sequence. The test length required by I is determined by how soon I reaches the desired coverage following this massive and early "fault consumption". If a mutated clone $\hat{I}$ of I is created such that $\hat{t}$ is large, we would expect the probability of $\hat{I}$ being fitter to be higher than if $\hat{I}$ has few or no vectors in common with I.

LPM is best thought of as a *strategically applied slight mutation.* Recall that slight mutation involves randomly flipping one of the 8 defining bits of a rule. Let $f_i(v_{i-1}, v_i, v_{i+1})$ be the rule of cell i, and let $f_i^{abc}(v_{i-1}, v_i, v_{i+1})$ be the rule formed when f_i undergoes a slight mutation at row (a, b, c). More specifically, for some

3-tuple (a, b, c), we have

$$f_i^{abc}(v_{i-1}, v_i, v_{i+1}) = \begin{cases} \overline{f(v_{i-1}, v_i, v_{i+1})} & \text{if } (v_{i-1}, v_i, v_{i+1}) = (a, b, c) \\ f(v_{i-1}, v_i, v_{i+1}) & \text{otherwise} \end{cases}$$

Now consider an individual I defined by the CA initial vector pair $(A, v^{(0)})$. Let $\hat{I}$ be the individual formed when rule f_i of I undergoes slight mutation and is replaced with f_i^{abc}. Then it follows that the vector sequence generated by $\hat{I}$ will be identical to that of I up to and including the first occurrence of a vector $v^{(\hat{t})}$ such that $(v_{i-1}, v_i, v_{i+1})^{(\hat{t})} = (a, b, c)$. This arises from the fact that the next state function of both CA only differ for such vectors.

As previously explained, the correlation between the test lengths required by I and $\hat{I}$ would intuitively seem dependent on $\hat{t}$, as both individuals' vector sequences have at least $\hat{t}$ vectors in common.

But how can we force $\hat{t}$ to be large? For each CA cell c_i, the CA object associates 8 bits of memory $m_i(0), m_i(1), \ldots, m_i(7)$, initially zeroed. Before the transition from CA time t to $t+1$, $m_i((v_{i-1}v_iv_{i+1})^{(t)})$ is set for each cell c_i, where $(v_{i-1}v_iv_{i+1})^{(t)}$ is interpreted as a 3-bit binary number. Any time a bit is set, the cell and bit number are stored as a pair in a list called the LPM list. The LPM list is overwritten each time step, unless no bits are set. At the end of CA simulation, i.e. when the target fault coverage is realized or T_{max} vectors have been generated, the LPM list will necessarily contain at least one cell/bit number pair. Execution of the LPM operator is performed by simply flipping the rule bit corresponding to an LPM list element (chosen randomly if there is more than one).

For an example, the vector sequence generated by a (primitive) CA with rule vector (150, 90, 90, 150, 150) is given in table 5.5. The values stored in $m_1(j)$, $m_2(j)$, and $m_3(j)$, $0 \leq j \leq 7$, over time are given in the 24 columns with numerical headings (the headings are the values of j). One observes that at time $t = 0$, the neighborhoods of c_1, c_2, and c_3 are respectively 011, 110, and 100. The decimal equivalents of these binary numbers are 3, 6, and 4, respectively, thus the only bits that are set in the first row of table 5.5 are $m_1(3)$, $m_2(6)$, and $m_3(4)$. As time progresses, more and more of these bits become set, the last being $m_3(0)$ at time $t = 18$. We see that at times such as $7, 10, 11, 14, 15, 18, \ldots$, no bit is set, and therefore the LPM list retains its previous contents. At the end of simulation the LPM list contains the sole cell/bit number pair (3,0). The LPM operator can now be applied by replacing f_3 with f_3^{000}, resulting in the rule vector (150, 90, 90, 151, 150). Note that the new CA has $v^{(19)} = 10110 = v^{(14)}$, thus, in this case, LPM causes the CA to fall into a short cycle. However, when 2^n is large compared to $\hat{t}$, the probability that the LPM-mutated CA will fall into a short cycle is very small. The crux of this example is the fact that when started in the state 01100, the two CA (150, 90, 90, 150, 150) and (150, 90, 90, 151, 150) generate 19 equal vectors before differing.

In the previous example we saw that the *time of last possible mutation* (TOLPM) was 18, i.e. the vector at CA time 19 is the first differing vector generated by the LPM-mutated clone. Given n, the width of a CA, we wish to determine the expected number of vectors generated before TOLPM, call this value $E_{LPM}(n)$. Considering any three adjacent CA cells (c_{i-1}, c_i, c_{i+1}), we view their combined state as a symbol over the 8-ary alphabet {000,001,010,011,100,101,110,111} and refer to this symbol as *the symbol at cell* c_i. As CA time progresses, we may observe the number of unique symbols that have been encounterred. This is equal to the number of bits set in m_i. We assume that at any point in time the next symbol is chosen from a uniform distribution independent of the sequence of symbols seen before hand. This assumption is clearly not valid, especially with regards to rules with unbalanced truth tables. Another assumption taken in this analysis that does not accurately reflect reality is the independence of neighboring symbols. For example, if the symbol at cell c_i is 001, than the only two possible symbols at cell c_{i+1} are 010 and 011. However, we will see that the approximated values of $E_{LPM}(n)$ and observed TOLPMs for primitive CA correspond.

Suppose at some time t we have encountered d of the 8 symbols. Then the probability of the next symbol being a new symbol is $(8 - d)/8$, while the probability

t	$v^{(t)}$	$m_1(j)$								$m_2(j)$								$m_3(j)$								LPM List (i,j)
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	
0	01100	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	(1,3),(2,6),(3,4)
1	11110	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	(1,7),(2,7),(3,6)
2	00001	1	0	0	1	0	0	0	1	1	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0	(1,0),(2,0),(3,1)
3	00011	1	0	0	1	0	0	0	1	1	1	0	0	0	0	1	1	0	1	0	1	1	0	1	0	(2,1),(3,3)
4	00100	1	1	0	1	0	0	0	1	1	1	1	0	0	0	1	1	0	1	0	1	1	0	1	0	(1,1),(2,2)
5	01010	1	1	1	1	0	0	0	1	1	1	1	0	0	1	1	1	0	1	1	1	1	0	1	0	(1,2),(2,5),(3,2)
6	10011	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	1	1	0	1	0	(1,4)
7	11100	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	1	1	0	1	0	(1,4)
8	00110	1	1	1	1	1	0	0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	0	1	0	(2,3)
9	01101	1	1	1	1	1	0	0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	0	(3,5)
10	11101	1	1	1	1	1	0	0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	0	(3,5)
11	00101	1	1	1	1	1	0	0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	0	(3,5)
12	01001	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	(2,4)
13	10111	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	(1,5),(3,7)
14	10110	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	(1,5),(3,7)
15	10101	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	(1,5),(3,7)
16	10001	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	(1,5),(3,7)
17	11011	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	(1,6)
18	01000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	(3,0)
19	10100	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	(3,0)
20	10010	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	(3,0)
21	11111	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	(3,0)

Table 5.5: LPM example for CA with rule vector $(150, 90, 90, 150, 150)$

that the symbol has already been seen is $d/8$. This defines an *absorbing Markov chain* of 8 states [63] with the following transition matrix.

$$M = \begin{bmatrix} 1/8 & 7/8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2/8 & 6/8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3/8 & 5/8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4/8 & 4/8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5/8 & 3/8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6/8 & 2/8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7/8 & 1/8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Using this model, we may quite easily compute the mean number of steps before absorption, i.e. the number of time steps before all 8 symbols have appeared [63]. However, this is not the solution to our original problem. Under the assumption that the sequence of neighborhood configurations for each of the n cells of a CA are independent, we conclude that $E_{LPM}(n)$ is the expected *maximum* time to absorption (TTA) *over n independent runs*[8] of the Markov chain with transition matrix M. The following solution to this problem is due to [64].

Let m be the column of the single absorbing state of the transition matrix M of a Markov chain with n states[9], and suppose column 1 represents the unique start state. Let $\{T_1, \ldots, T_n\}$ be the set of n random variables denoting the TTA of n runs of the Markov chain. Then the probability that the TTA for run i is no more than some integer t is $P(T_i \leq t) = (M^t)_{1,m}$. Over all n runs, the probability that none of the n TTAs exceed t is

$$\begin{aligned} P(T_1, \ldots, T_n \leq t) &= [P(T_i \leq t)]^n \\ &= [(M^t)_{1,m}]^n \end{aligned}$$

---

[8] A slightly more accurate analysis might reduce this number to $n-2$ as the two boundary cells have neighborhoods that are restricted to 4 symbols.

[9] in our case $m = 8$ and $n = 8$

Now defining $T_{max}(n) = max(T_1, \ldots, T_n)$, the probability that T_{max}(n) is equal to t is

$$\begin{aligned} P(T_{max}(n) = t) &= P(T_1, \ldots, T_n \leq t) - P(T_1, \ldots, T_n \leq t - 1) \\ &= [(M^t)_{1,m}]^n - [(M^{t-1})_{1,m}]^n \end{aligned}$$

Hence the expected value of T_{max}(n) is

$$E_{LPM}(n) = E(T_{max}(n)) = \sum_{t=1}^{\infty} t \left[[(M^t)_{1,m}]^n - [(M^{t-1})_{1,m}]^n\right] \tag{5.1}$$

Figure 5.1 shows this value[10] (labeled "rad 1 computed") for a range of values of n that spans the input cardinalities of the benchmark set and indeed the majority of practical combinational circuits. Thus, for circuits with at least 20 primary inputs, we would expect that the first 50 vectors generated by some individual I to be the same as those generated by the individual obtained by performing LPM against I.

Now suppose we repeat the preceding theory with respect to radius 2 CA, in which the next state of cell c_i depends on the five cells $c_{i-2}, \ldots, c_{i+2}$. Then the neighborhood state of an internal cell is a 5-vector corresponding to one of $2^5 = 32$ possible symbols. The implied Markov chain now has 32 states and its transition matrix, M_2 is defined by $(M_2)_{i,i} = i/32$ for $1 \leq i \leq 32$ and $(M_2)_{i,i+1} = (32 - i)/32$ for $1 \leq i \leq 31$, with all other entries 0. The expected point of LPM, $E_{LPM2}(n)$ can be computed by replacing the matrix M with M_2 in equation 5.1 and taking $m = 32$; this function is also plotted in figure 5.1 as the function labeled "rad 2 computed". One can observe that for radius 2 CA, the LPM operator is expected to preserve the first two to three hundred vectors.

In general, radius 2 CA incur a substantial increase in hardware resources over their radius 1 cousins, since the next state rules depend on 5 cells rather than 3. The beauty of the radius 1 CA for implementation in silicon is the simple and local logic driving its flip-flops; increasing the radius of a CA decreases both the simplicity and locality of this logic. As such, the focus of this thesis has been restricted to radius 1

---

[10] computed using a C program, using 500 for ∞

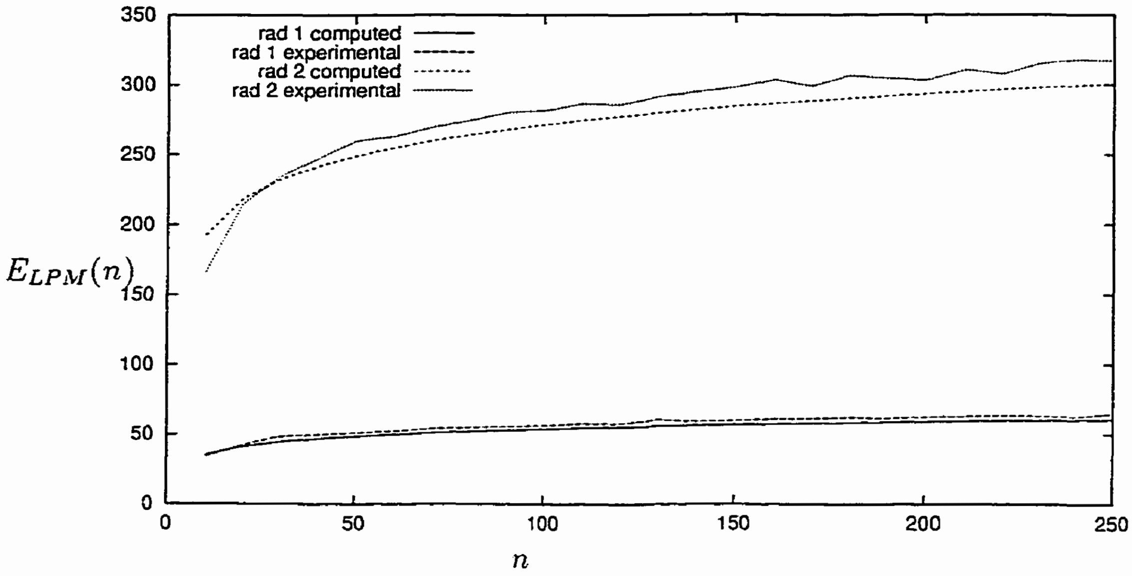


Figure 5.1: Computed and observed radii 1 & 2 TOLPM (primitive CA)

CA. However, if we track the 5-neighborhood states of a radius-1 CA and apply LPM *as if the CA was of radius 2*, the result will be a CA in which all cells except the mutated one still have 3-variable next state rules. The mutated cell's rule is altered to depend on is "next-to-next-door" neighbors, but all other rules keep the simple and local properties. With this very small (and constant) hardware price comes a major increase in the number of identical states visited by a CA and its mutated clone. For the remainder of the thesis, LPM2 refers to the aforementioned version of LPM that slightly breaks the definition of the radius 1 CA.

5.4.2.2 Experiments and Results

Experments pertaining to the LPM operator were performed in an attempt to answer the following three questions:

1. How well does equation 5.1 predict the expected point of LPM?
2. How often does application of LPM give an improvement?
3. Is CAGA more powerful when augmented with LPM?

Item 1 was addressed by selecting 300 arbitrary primitive CA and simulating from a random initial state. The TOLPM for both radius 1 and radius 2 neighborhoods were recorded; this was done for CA widths $n \in \{10, 20, \ldots, 250\}$. The results are compared to the expected TOLPM (computed via equation 5.1) in figure 5.1. We see that the equation predicts quite well, though seems to be slightly lower than than the empirical results. This discrepancy is most pronounced for the radius 2 LPM.

To resolve item 2, 300 primitive CA of appropriate width were obtained for each of the benchmarks. Since we wish to determine if applying LPM is more beneficial than simple slight mutation, for each CA the test length of the original CA, a slightly mutated clone, and a LPM-mutation clone were computed. Extending the experimentation to LPM2, the control mutation operator is the radius 2 equivalent of radius 1 slight mutation, SM2. SM2 is executed by considering the radius 1 rule as a radius 2 rule that happens to be independent of both "outer" neighbors, and flipping one randomly selected bit in the 32 bit truth table. Again, the initial vectors were chosen randomly.

Table 5.6 gives the results. The numbers in table 5.6 indicate the number of CA (of the 300 total) for which the mutation operator resulted in a CA with test length strictly less than the original CA. To facilitate accurate results, the fsim packet size used was 1. Some of the values of T_{max} were increased for these experiments, so that at most 1 of the 300 "CA families" all failed.

These numbers indicate that radius 1 LPM does not provide a significant improvement over SM. However, LPM2 sees far more improvements on test length than radius 2 slight mutation. This can be accounted for by the fact that the number of vectors a LPM2 mutated clone has in common with the original CA is typically 4 to 6 times that of a LPM mutated clone.

Note that the only CUT for which LPM2 was not more successful than SM2 was c6288. The LPM2-CAGA code only keeps track of which minterms of each cell's rule are used up until the target coverage is reached. The result of this is that when the typical CA in the population is LPM2-mutated, only the last few (probably 1 or

circuit	SM	LPM	SM2	LPM2
c1355	124	141	70	157
c1908	101	114	69	142
c2670	149	128	135	149
c3540	112	112	42	118
c432	148	149	78	149
c499	141	144	71	160
c5315	142	160	58	154
c6288	118	138	61	13
c7552	142	140	118	161
c880	139	145	70	159
total	1316	1371	772	1362

Table 5.6: Effectiveness of various mutations on primitive CA

```
reproduce $k_{rep}$ best individuals
$i \leftarrow k_{rep}$
while $i < N$ do
   mate_1 ← randomIndividual()
   if drand48() < $p_{lpm}$
      then add lpm-mutated clone of mate_1 to new population
      $i \leftarrow i+1$
   else
      mate_2 ← randomIndividual()
      recombine mate_1 and mate_2 as per CAGA
      $i \leftarrow i+2$
   end if
end while
```

Figure 5.2: Incorporation of the LPM operator into CAGA

2) vectors are changed. For the new individual to achieve a better test length than its unmutated clone, the new vectors it generates after diverging must cover the untested faults in less time. Since these last faults are typically hard to test, it is unlikely that these vectors will fulfill this requirement.

Figure 5.2 gives the evolve algorithm used to integrate LPM or LPM2 with CAGA in order to address item 3. Since applying the operator only makes sense if the target is among the fitter individuals, the LPM operator is done as an alternative to the normal recombination operators of crossover and normal mutation. When

circuit	SM	LPM	$\%100\frac{SM-LPM}{SM}$
c1355	1000	880	12.0
c1908	1536	2120	-38.0
c2670	640	1536	-140.0
c3540	2592	3392	-30.9
c432	136	168	-23.5
c499	324	264	18.5
c5315	896	664	25.9
c6288	38	34	10.5
c7552	1504	3968	-163.8
c880	272	432	-58.8
		average	-38.8

Table 5.7: LPM results

an individual is selected, its LPM-mutated clone is added to the new population with probability p_{lpm}. Otherwise, the individual is processed exactly as in CAGA. Specifically, a mate is chosen and the two are crossed over with probability p_c, and fully mutated with probability p_m.

For these experiments, a value of 0.2 was used for p_{lpm2}. Note that in LPM-CAGA, the actual probability that a selected individual will be involved in crossover is reduced from p_c to $p_c(1 - p_{lpm})$. In an attempt to maintain consistency and offset this reduction, p_c was increased from 0.6 to 0.7. The probability of mutation used was $p_m = 0.1$, the initial vector used was fixed at $101010\ldots$.

Table 5.7 gives the results when CAGA is augmented with LPM. As a control, a batch of experiments was performed in which the LPM operator was replaced with slight mutation. In other words, the algorithm of figure 5.2, the SM-mutated clone is added instead of the LPM-mutated clone, in line 6. The resulting test lengths for LPM-CAGA and the control are in the columns labeled LPM and SM, respectively. The rightmost column give the percentage improvement that LPM gave. The results are less than encouraging, on average the LPM-CAGA runs saw a -38.8% "improvement" over the control experiments. We hypothesize that the number (35-60) vectors that are shared between an individual and its LPM-mutated clone is not large enough to incur a significant correlation between their fitnesses.

circuit	SM2	LPM2	$\%100\frac{S2-LPM2}{S2}$	LPM2 5-rules	n
c1355	1040	1016	2.3	2.0	41
c1908	2168	1296	40.2	4.0	33
c2670	1536	640	58.0	8.3	233
c3540	2752	2304	16.3	15.5	50
c432	176	164	6.8	5.0	36
c499	332	364	-9.6	2.5	41
c5315	936	888	5.1	4.2	178
c6288	36	44	-22.2	2.7	32
c7552	4128	4640	-12.4	2.0	207
c880	368	304	17.4	11.6	60
average[a]			13.8	6.1	97.7

---

[a] c6288 omitted

Table 5.8: LPM2 results

Table 5.8 illustrates the effect LPM2 has on CAGA. The column labeled LPM2 gives the minimal test length found by CAGA augmented with the LPM2 operator. As for radius-1 LPM, control experiments were executed. In these, the LPM2 operator (line 6 of figure 5.2 was replaced with the radius-2 equivalent of slight-mutation, SM2. Under SM2, a single rule is picked at random, converted to an equivalent radius-2 rule if necessary, and then one randomly selected bit in the 32 bit truth table is negated. This, in effect, is exactly what LPM2 does, with the exception that the choice of rule and bit is done wisely.

The columns labeled LPM2 and SM2 gave the LPM2 and control results, respectively. One will observe that LPM2 outperforms the control, demonstrating that it indeed is a useful approach to mutation. The increase in expected shared vectors between an individual and its mutated clone when LPM2 is used seems to be substantial enough to have a significant effect on the correlation between the two's ability to test.

Because of our previous discussion on the affect LPM2 on c6288, we conclude that it is justified to leave out the results for this benchmark from the computation of the average, and that it only makes sense to use LPM2 in CAGA when the "ball park" test length of good testers is somewhat greater than the expected time of LPM2.

This said, the average improvement of LPM2 over the radius-2 slight mutation is an impressive 13.8%. This suggests that, for circuits with similar test length requirements as those in the benchmark set, LPM2 is a powerful augmentation to CAGA. If hardware restrictions are somewhat relaxed, one would expect LPM stretched over high radii to be useful when the test length magnitudes are greater. The column labeled "LPM2 5-rules" in table 5.8 gives the average number of radius 2 rules found in the best CA; when compared to the number of CA cells given in the rightmost column we see that on average only 6% of the cells use the larger neighborhood. This is a reasonable trade off for the decrease in test length.

In the following section we consider advanced techniques for the crossover operator.

5.5 Can we Perform "Intelligent" Crossover?

In this section we consider the interactions between neighboring rules in a CA, and attempt to discourage degenerate behavior. In section 5.5.1, the concepts of *sinks*, *abandons* , and *good neighbors* are defined. A fast algorithm to detect sinks is given in section 5.5.2. In section 5.5.3 a technique that reduces CAGA run time is described. Section 5.5.4 gives experimental results when good neighbor techniques are used against null boundary CA. In section 5.5.5 a phenomenon called *boundary subcycles* is described, the frequency of which is arguably lessened by using the periodic boundary condition; pertaining experimental results are presented in section 5.5.6.

5.5.1 Good Neighborhoods

In section 5.4, we looked at two different variations on the mutation operators that attempt to improve CAGA. Given a relatively fit individual, these operators increase the probability that the mutated-clone would also be fit, or, equivalently, they decrease the probability that the clone would be highly unfit. In this section we attempt to improve the crossover operator using the same philosophy. Under our GA parameters, the crossover operator is applied much more frequently than mutation, hence we expect that improvement of crossover might have a more drastic effect on CAGA.

Inspection of a sample of the worst individuals in various generations of CAGA revealed two undesirable phenomenon in these CA. To assist in our discussion of these traits, we introduce the following definitions:

Definition 1 *Given a CA, suppose there exists a k-vector w ($k < n$) and index i ($0 \leq i \leq n-k$) with the property that* $(c_i, \ldots, c_{i+k-1})^{(\hat{t})} = w$ *always implies* $(c_i, \ldots, c_{i+k-1})^{(t)} = w$ *for all* $t \geq \hat{t}$. *Then the cells* $\{c_i, \ldots, c_{i+k-1}\}$ *are called a k-*sink *and the CA is said to be k-*sinking. *A CA with no k-sink is called k-*nonsinking. *We say that* $\{c_i, \ldots, c_{i+k-1}\}$ sink into w.

Definition 2 *Given a CA, suppose there exists a k-vector w ($k < n$) and index i ($0 \leq i \leq n-k$) with the property that* $(c_i, \ldots, c_{i+k-1})^{(\hat{t})} \neq w$ *always implies* $(c_i, \ldots, c_{i+k-1})^{(t)} \neq w$ *for all* $t \geq \hat{t}$. *Then the cells* $\{c_i, \ldots, c_{i+k-1}\}$ *are called a k-*abandon *and the CA is said to be k-*abandoning. *A CA with no k-abandon is called k-*nonabandoning. *The vector w is called the* abandoned *vector.*

Clearly, the presence of sinks or abandons degrades pseudorandomness. However, do either phenomenon impede testing? If a k-sink exists, then all vectors produced after time $\hat{t}$ will have certain cells at fixed values, thus any faults that require vectors with different values in these cells can never be detected after time $\hat{t}$. In the presence of a k-abandon, once a value other than w is seen at the involved cells, no vector that includes w at these positions is ever generated again. When k is small, the effect is a reduction in the set of cubes the CA generates vectors from. Note that a 1-abandon and a 1-sink are equivalent.

In terms of CA growths, sinks manifest themselves as vertical stripes in the bitmap as is visible in the right half of some of the figures in appendix B. Figure B.3 has a wide sink near the left side and a single cell that sinks to 0 on the right. The CA of figures B.8 and B.9 are also noticeably sinking. Abandons do not generally have a simple visual manifestation.

We now give examples of neighboring CA rules that result in a 2-abandon and a 2-sink.

The neighboring rules (149, 76) result in a 2-abandon, the abandoned vector being (11). Consider four adjacent CA cells (a, b, c, d), with cells b and c having rules 149 and 76, respectively. Expressing these rules algebraically yields $b^+ = \bar{b}\bar{c} + \bar{a}\bar{c} + abc$, and $c^+ = \bar{b}c + c\bar{d}$. For the next value of (b, c) to be (11), we need:

$$\begin{aligned}
(b^+, c^+) = (11) \quad &\Rightarrow b^+c^+ = 1 \\
&\Rightarrow (\bar{b}\bar{c} + \bar{a}\bar{c} + abc)(\bar{b}c + c\bar{d}) = 1 \\
&\Rightarrow abc\bar{d} = 1 \\
&\Rightarrow (a, b, c, d) = (1110)
\end{aligned}$$

Thus the only way (b, c) can reach the state (11) is when the previous state of (a, b, c, d) is (1110). But in this previous state, we have $(b, c) = (11)$. Therefore once (b, c) enter a state other than (11), the two cells will never return to this state.

Now we examine the neighboring rules (106, 232); these cause a 2-sink into the vector (00). Let (a, b, c, d) be four neighboring CA cells, and suppose b and c have respective rules 106 and 232. Then the next state functions for b and c are $b^+ = \bar{a}c + \bar{b}c + ab\bar{c}$ and $c^+ = cd + bd + bc$. To show that b and c sink to (00), note that the cofactors $b^+|_{b=0,c=0}$ and $c^+|_{b=0,c=0}$ are both the constant function 0. Thus once (b, c) equals the vector (00) these two cells will remain in this state for all future times, regardless of the bits stored in a and d, i.e. (b, c) sinks to (00).

An observation[11] that has an important impact on CAGA is that primitive CA are k-nonsinking and k-nonabandoning for all $k < n$. Therefore, starting from an initial population of primitive CA, if we are cautious when applying mutation and crossover, we can ensure that no k-sinking or k-abandoning CA are born. Given a k-vector $r = (f_0, \ldots, f_{k-1})$ of rules, we can check if the presence of r in a CA's rule vector would imply a k-sink or a k-abandon; if neither occurs we call r a *good*

---

[11] for $k = n$, the all zero vector $\mathbf{0}$ is both sunk into and abandoned

neighborhood. For the problem of checking for an abandon, exponential time brute-force search may be the only option. However in section 5.5.2 we describe a linear time algorithm for detecting sinks.

When two CA are cut and joined in a crossover operation, there are (at most) $2(k-1)$ new k-rule vectors created. Suppose CA A and B are to be involved in a crossover operation, and the rule vectors of the respective CA are $(a_0, \ldots, a_{n-1})$ and $(b_0, \ldots, b_{n-1})$. Given the random crossover point c, the two offspring A' and B' have respective rule vectors $(a_0, \ldots, a_c, b_{c+1}, \ldots, b_{n-1})$ and $(b_0, \ldots, b_c, a_{c+1}, \ldots, a_{n-1})$. Then the $2(k-1)$ new k-rule vectors are:

$$\left.\begin{array}{c}
a_c, b_{c+1}, b_{c+2}, b_{c+3}, \ldots, b_{c+k-2}, b_{c+k-1} \\
a_{c-1}, a_c, b_{c+1}, b_{c+2}, \ldots, b_{c+k-3}, b_{c+k-2} \\
a_{c-2}, a_{c-1}, a_c, b_{c+1}, \ldots, b_{c+k-4}, b_{c+k-3} \\
\vdots \\
a_{c-k+1}, a_{c-k+2}, a_{c-k+3}, a_{c-k+3}, \ldots, a_c, b_{c+1}
\end{array}\right\} \text{new } k\text{-neighborhoods in } A'$$

$$\left.\begin{array}{c}
b_c, a_{c+1}, a_{c+2}, a_{c+3}, \ldots, a_{c+k-2}, a_{c+k-1} \\
b_{c-1}, b_c, a_{c+1}, a_{c+2}, \ldots, a_{c+k-3}, a_{c+k-2} \\
b_{c-2}, b_{c-1}, b_c, a_{c+1}, \ldots, a_{c+k-4}, a_{c+k-3} \\
\vdots \\
b_{c-k+1}, b_{c-k+2}, b_{c-k+3}, b_{c-k+3}, \ldots, b_c, a_{c+1}
\end{array}\right\} \text{new } k\text{-neighborhoods in } B'$$

To ensure that no k-"bad neighborhoods" arise from the proposed crossover operation, we must verify that each of these k-vectors is a good neighborhood. If this is not satisfied, the crossover point c is rejected and the sequence of crossover points $c-1, c+1, c-2, c+2, \ldots (mod(n-1))$ is followed until a point is found that does not induce a bad neighborhood. We call this mechanism *k-good neighborhood enforcement* (k-GNE). Note that k-GNE does not necessarily imply that no k'-bad neighborhoods will arise for $k' < k$. When GNE is used for all $k' \leq k$, we describe the mechanism as *k-maximal good neighborhood enforcement* (k-MGNE).

Recall that for $k = 1$, "sink" and "abandon" degenerate to equivalent definitions. In this case, the concepts of sinking and abandoning do not reflect interactions between neighboring rules, but are an intrinsic property of single rules. We call such rules *sinking rules*. It is easy to show that a rule $f(a, b, c)$ is sinking if and only if $f|_{b=\alpha} = \alpha$ for some $\alpha \in \{0, 1\}$, i.e. a cofacter of f when the center variable is fixed is precisely the constant function of the fixed value. There are 31 sinking rules for radius 1 CA; by eliminating them from the possible rules that may be introduced via full mutation, we are guaranteed to have no 1-sinks.

5.5.2 Linear Time Sink Detection

Let $r = r_j, \ldots, r_{j+k-1}$ be the respective rules of k adjacent CA cells $C = \{c_j, \ldots, c_{j+k-1}\}$. We may determine if C is a sink with time complexity $\mathcal{O}(k)$. To aid comprehension, we use as an example the rule vector 216, 245, 37, 228, 108, the truth tables of which are given in table 5.9. Recall from section 5.4.2.1 that *the symbol at cell i* is defined as the aggregate of cell states $c_{i-1}c_ic_{i+1}$. Also, we refer to the two cells c_{j-1} and c_{j+k} as the *bordering* cells, while the two cells c_j and c_{j+k-1} are the *endmost cells*. We wish to find a k-vector $u = u_0, \ldots, u_{k-1}$ such that the k equations 5.2 hold; r is a sink if and only if such a u exists.

$$\begin{aligned}
r_j(-, u_0, u_1) &= u_0 \\
r_{j+1}(u_0, u_1, u_2) &= u_1 \\
r_{j+2}(u_1, u_2, u_3) &= u_2 \\
&\vdots \\
r_{j+k-2}(u_{k-3}, u_{k-2}, u_{k-1}) &= u_{k-2} \\
r_{j+k-1}(u_{k-2}, u_{k-1}, -) &= u_{k-1}
\end{aligned} \tag{5.2}$$

The '$-$' characters in the first and last of equations 5.2 denote don't care, i.e. these two equalities must hold independent of the respective bordering cell. For each solution of each equation, the algorithm creates a node labeled with the corresponding symbol. The nodes of the two endmost cells are labeled using '-' to represent the bordering cell value (thus there are but four possible nodes for each endmost cell). Since there are exactly eight symbols per cell (with the previously noted exceptions), and determining if each is a solution involves a single rule evaluation, this phase requires a constant time per cell. In table 5.9 the solutions to equations 5.2 for the example rule vector are indicated by a box around the function value. Note that the solutions for the endmost cells adhere to the additional requirement of independence from the corresponding border cell. Figure 5.3 exemplifies this step. Each column of symbols corresponds to the set of possible nodes for the associated cell; those that are actually manifested (i.e. the solutions) are indicated with ovals.

The final phase of the algorithm is to create a *directed acyclic graph* (DAG) with the existing nodes. Beginning with the nodes of the leftmost cell c_j and proceeding rightwards, arcs are introduced as follows. An arc is created from a node labeled *abc* of c_i to a node labeled *bcd* of c_{i+1} when either of the following hold: 1) the indegree of *abc* is greater than 0, or 2) c_i is the left endmost cell. Creation of all arcs incident from all nodes of a cell takes constant time. This follows from the facts that there are at most 8 nodes per cell, and there are at most 2 arcs that may be made incident from each (given a node labeled *abc* we have but two choices for d, thus two choices for *bcd*). In Figure 5.3 we observe several nodes that are left unconsidered when adding arcs, since they have indegree of 0.

If during the algorithm it should occur that no arcs are added between nodes of cell c_i and cell c_{i+1}, then C is not a sink. However if we successfully create a directed path from a node of the left endmost cell c_j to one of the right endmost cell c_{j+k-1}, then C is a sink. Furthermore, each such path in the DAG represents a unique subvector that C sinks to. In our example there are exactly 4 paths from the left side to the right side of the DAG, corresponding to the 4 vectors $\{01000, 01001, 11000, 11001\}$.

In conclusion, the algorithm outlined in this section will detect a k-sink with time complexity $\mathcal{O}(k)$.

$c_{i-1}c_ic_{i+1}$	$r_j = f_{216}$	$r_{j+1} = f_{245}$	$r_{j+2} = f_{37}$	$r_{j+3} = f_{228}$	$r_{j+4} = f_{108}$
000	0	1	1	$\boxed{0}$	$\boxed{0}$
001	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$
010	0	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$
011	$\boxed{1}$	0	0	0	$\boxed{1}$
100	1	1	$\boxed{0}$	$\boxed{0}$	0
101	$\boxed{0}$	1	1	1	1
110	1	$\boxed{1}$	0	$\boxed{1}$	1
111	$\boxed{1}$	$\boxed{1}$	0	$\boxed{1}$	0

Table 5.9: Preprocessing phase of the sink detection algorithm

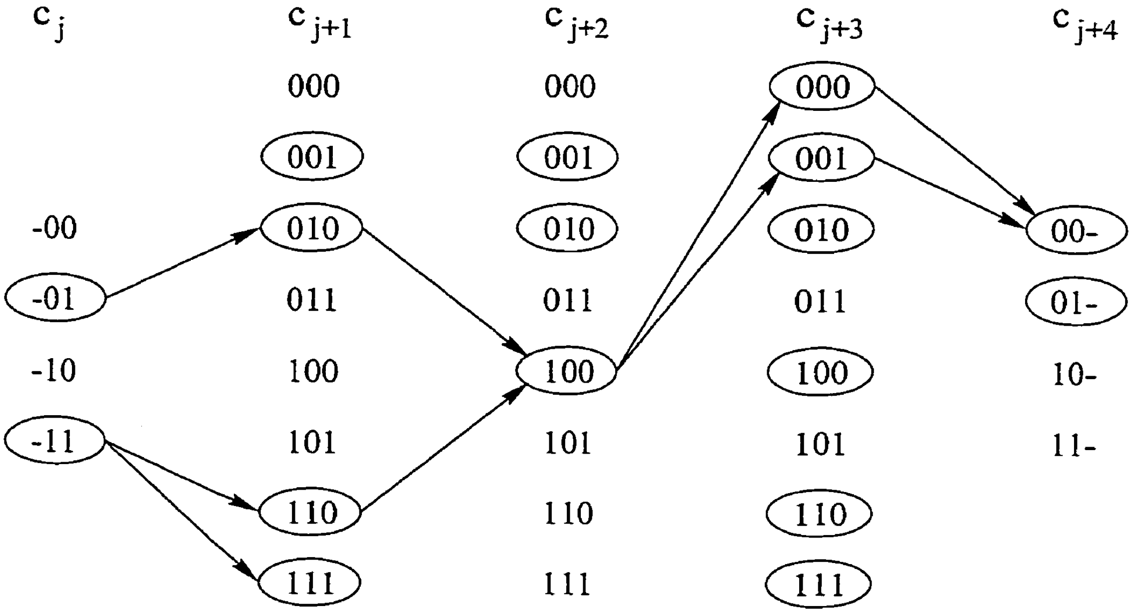


Figure 5.3: A DAG of the sink detection algorithm

5.5.3 Dynamic Max Time Reduction

An auxiliary mechanism not previously used was employed in the experiments of this section. Recall that CAGA requires the parameter T_{max}, which dictates the maximum number of vectors to produce from an individual before "giving up", i.e. if the sequence generated by the CA has not met the target fault coverage F_{cov} by time T_{max}, the individual is assigned a fitness of 0 and fault simulation terminates. The value of T_{max} was chosen such that most of the initial population $P(0)$ had positive fitness. However, in the advanced generations the population as a whole has an average raw fitness much greater than that of $P(0)$; i.e. the majority of the individuals require much less than T_{max} clock ticks to achieve F_{cov}. In these generations, the very few individuals that require close to T_{max} vectors or those that never detect F_{cov} of the faults become progressively less valuable to the genetic algorithm, yet they take the most computation time during fault simulation.

The natural remedy to this inefficiency is to reduce T_{max} as CAGA evolves. We call this *dynamic max time reduction* (DMTR). DMTR was implemented such that the value used for T_{max} in generation g, $T_{max}(g)$, is such that 90% of $P(g-1)$ require less than $T_{max}(g)$ vectors, as long as this value is at least 3 times that of the fittest individual in $P(g-1)$ (otherwise $T_{max}(g) = T_{max}(g-1)$). The reason for this second constraint is to prevent T_{max} from becoming too small, i.e. so that in future generations it won't be "hard" for an individual to receive nonzero fitness.

5.5.4 Null Boundary CA Results

Original experiments employing 3-GNE revealed less than satisfactory results that are not repeated here. These experiments used full mutation and FIV, with the usual parameter settings of population size $N = 300$ and GA halting after $\Delta g = 200$ generations without improvement. At first, this was thought to indicate that, contrary to intuition, GNE does not improve CAGA. However, after closer investigation, we hypothesize that this may be accounted for as follows. Consider the nondegenerate linear rules 90 and 150 (recall $f_{90}(x_{i-1}, x_i, x_{i+1}) = x_{i-1} \oplus x_{i+1}$ and $f_{150}(x_{i-1}, x_i, x_{i+1}) = x_{i-1} \oplus x_i \oplus x_{i+1}$); these are the only two rules present in primitive CA. With the exception of the leftmost and rightmost cell in the CA, neither of these rules can exist with the endmost cells of a sink. This is a direct implication of the fact that both of these functions have the property that negation of one of the dependent variables x_{i-1} or x_{i+1} always negates the function itself.

When CAGA works against an initial population $P(0)$ consisting entirely of primitive CA, these two rules are the dominant rules (count wise) for many generations, since only the secondary genetic operator of mutation can introduce different

		MGNE		SA	
circuit	CTRL	TL	%impr	TL	%impr
c1355	864	896	-3.7	848	1.9
c1908	1632	1592	2.5	1760	-7.8
c2670	800	608	24.0	608	24.0
c3540	3936	1536	61.0	2272	42.3
c432	136	156	-14.7	144	-5.9
c499	316	292	7.6	300	5.1
c5315	752	688	8.5	728	3.2
c6288	30	30	0.0	30	0.0
c7552	960	1120	-16.7	1120	-16.7
c880	448	384	8.3	336	25.0
average			8.3		7.1

Table 5.10: Good neighborhood enforcement for null boundary CA

rules. Inspection of the fittest individuals in the final generation of these experiments confirm that even after hundreds of generations, the rules 90 and 150 are still by far the most numerous rules. Thus, in these experiments, the event that GNE actually rejects a candidate crossover point is rare. In conclusion, GNE has little impact on CAGA when the majority of the rules in the population are 90 and 150.

Under this observation, we attempt to seed CAGA with parameters that will result in a greater proportion of nonlinear rules. Boosting both N and Δg intuitively has this consequence; if CAGA is allowed to evolve a larger population for a longer time, the effect of mutation should become more dominant. The values $N = 500$ and $\Delta g = 1000$ were used to verify this theory.

Results of these CAGA runs are given in table 5.10. The control experiment (CTRL) uses full mutation, FIV, and the complete rule space, i.e. $\{0, \ldots, 255\}$. Experiments of the columns labeled MGNE employed 3-MGNE and used the same parameter settings as CTRL. Under "TL" the minimum test lengths are listed, while the "%impr" column gives the percentage improvement over CTRL. The improvements range from -16.7% to 61.0% with a mean value of 8.3%, which is substantial.

Intuitively, sinks are bad for testing, however, abandons might not be as problematic. Suppose there is a k-abandon A and the abandoned subvector is w. Such a defect in randomness only negatively affects testing when there is a hard to test fault f with a significant number of vectors in the test set of f having $A = w$. Based on this intuition, we performed experiments using GNE with no abandon checking; we call this mechanism *sink avoidance* (SA).

The columns in table 5.10 labeled "SA" lists the results of experiments that differ from MGNE only in that no abandon checking is done, i.e. no k-sinks for $k \in \{1, 2, 3\}$ are created by any genetic operation. Here the range of improvements is -16.7% to 42.3%, with mean 7.1%. It is interesting to note that both MGNE and SA provide a positive augmentation to CAGA against benchmarks c2670, c3540, c499, c5315, and c880, while for c432 and c7552 it appears that MGNE and SA are detrimental[12] to CAGA performance. Clearly there is some consistency in GNE effectiveness against various CUTS. However it is unclear if there exist techniques to determine a priori if GNE should be used for a given circuit, and, if so, which "flavor" of GNE should be applied.

In the next section we discuss a degenerate CA behavior that is a generalization of sinking.

5.5.5 Boundary Subcycles

Again we turn to informal inspection of the worst CA in an attempt to assess what phenomenon causes birth of undesirable individuals in the experiments of section 5.5.4. There are various unwanted rule interactions that are not filtered out by GNE and would seem to be computationally hard to detect.

These problematic rule interactions cause a *subcycle* which is a generalization of a sink in which some neighborhood of cells $C = \{c_j, \ldots, c_{j+k-1}\}$ of size k enters a cycle of length at most 2^k that is never broken by the bit sequences seen at c_{j-1} and c_{j+k}. In other words, the next configuration of C is independent of both c_{j-1} and c_{j+k} for all times after the subcycle is enterred[13]. We found that when subcycles occur they usually reside on the left or right end of the CA. This is expected, since

---

[12] of course this consistency could be attributed to the CTRL runs being "lucky"

[13] a subcycle with cycle length of 1 is simply a sink

(under the null boundary condition) on the left end c_{j-1} is always 0 and on the right end c_{j+k} is always 0. Thus at the left (right) end, the next substate of C is trivially independent of c_{j-1} (c_{j+k}) for all configurations. It follows that subcycles are more likely to occur on the CA boundary; we call these ***boundary subcycles.***

A subcycle in a CA growth appears as a column with a vertically repeating pattern. In the right side growth of figure B.5 there is a very wide boundary subcycle. The nonlinear growth of figure B.7 has a 3-cell internal subcycle which appears to degrade randomness of nearby cells; it is possible that this subcycle is part of a wider and longer subcycle.

The fact that subcycles are more likely to occur on CA boundaries is the primary motivator for the work of the next section.

5.5.6 Periodic Boundary CA Results

In this section we employ CAGA to run against periodic boundary CA (PBCA). Recall that null boundary CA assume the constant value 0 at the nonexistent cells left of c_0 and right of c_{n-1}. In PBCA, c_0 and c_{n-1} are logically adjacent, i.e. the left neighbor of c_0 is c_{n-1}, while c_{n-1}'s right neighbor is c_0. A worthy analogy states that a null boundary CA is to a line segment as a PBCA is to a circle. The hope here is that eliminating the null boundary condition will result in fewer subcycles because many potential boundary subcycles will be eliminated.

An issue we face when using PBCA in CAGA pertains to the initial population. It has been shown that no PBCA has an irreducible characteristic polynomial [5]. Hence $P(0)$ must be populated with individuals that are inferior to primitive CA in terms of pseudorandomness. We are forced to evolve humans from mice rather than from monkeys; the PBCA of $P(0)$ are created by randomly assigning each cell to either rule 90 or rule 150.

Another consequence of the periodic boundary condition is that it becomes natural to think of the chromosomes as being circular, i.e. as having no "ends". As such, we use a variation of crossover called *circular* crossover. Circular crossover

circuit	CTRL	MGNE TL	MGNE %impr	SA TL	SA %impr
c1355	920	880	4.4	952	-3.5
c1908	1136	1272	-12.0	1368	-20.4
c2670	1440	512	64.4	416	71.1
c3540	2128	2112	0.8	1968	7.5
c432	144	132	8.3	144	0.0
c499	312	304	5.1	276	11.5
c5315	552	784	-42.0	656	-18.8
c6288	32	34	-6.3	32	0.0
c7552	1120	928	17.1	960	14.3
c880	304	304	0.0	272	10.5
average			4.0		7.2

Table 5.11: Good neighborhood enforcement for periodic boundary CA

selects two crossover points and swaps the segments of the parent's chromosomes between these points to obtain the two offspring. We add the additional constraint that the distance between the two crossover points is at least 5 positions in both directions around the circular chromosome.

Referring to table 5.11, we again find that MGNE and SA both give positive average improvement. All experiments ran against the same parameter set as those of table 5.10, except PBCA were used and $P(0)$ was created randomly. For MGNE, the improvements were as low as -42.0% and as high as 64.4%, and averaged 4.0%. Under SA, the range shifted up to -20.4% to 71.1% and the mean improvement over CTRL was 7.2%.

Table 5.12 provides a comparison of the data of table 5.10 and 5.11, i.e. a comparison of the results of the null boundary CAGA runs of section 5.5.4 and the PBCA CAGA runs of this section. The numeric columns of table 5.12 give the percent improvement of PBCA experiments over null boundary experiments for the corresponding CAGA settings. Evolution of PBCA with sink avoidance seems to be the most powerful combination; an average improvement of 10.0% is achieved when SA is used against PBCA rather than null boundary CA.

We hypothesize that sink avoidance attains greater results for PBCA than for

circuit	CTRL	MGNE	SA
c1355	-6.5	1.8	-12.3
c1908	30.4	20.1	22.3
c2670	-80.0	15.8	31.6
c3540	45.9	-37.5	13.4
c432	-5.9	15.4	0.0
c499	1.3	-1.4	8.0
c5315	26.6	-14.0	9.9
c6288	-6.7	-13.3	-6.7
c7552	-16.7	17.1	14.3
c880	32.1	20.8	19.1
average	2.1	2.5	10.0

Table 5.12: Improvement of PBCA over null boundary

null boundary CA as follows. Boundary subcycles are a hindrance in evolution of null boundary CA, possibly more so than sinks. Since boundary subcycles are less frequent in PBCA, sinks may be a more common reason for low fitness. By preventing sinks in PBCA, CAGA avoids production of many degenerate CA, and in the end usually finds a better solution to the CA testing problem.

5.6 Test Sequence Truncation

5.6.1 Description

In this section we present a simple technique that can potentially further shorten the required test length for a test generator. The technique, call *test sequence truncation* (TST), is shown to be effective against many of the "good" CA produced by genetic means.

Recall that the fault simulator takes a parameter b that controls the number of vectors that are fault simulated in parallel. Let T be the true test length and T' the test length returned by the fault simulator. Then T' is actually the smallest multiple of b greater than or equal to T. If $b = 1$ or we are lucky, $T' = T$. Otherwise it follows that there are up to $b - 1$ extra test vectors at the end of the sequence that are unnecessary for the target coverage. The true value of T can easily be computed following completion of the GA by setting $b = 1$ and performing a final fault simulation. Therefore several vectors may be shaved off of the end of the test sequence; we call this tail truncation.

However, we may also remove vectors from the beginning of the sequence. Let $S = v^{(0)}, \ldots, v^{(T-1)}$ be a tail truncated test sequence. It follows that removal of $v^{(T-1)}$ from S will not provide the desired fault coverage F_{cov}. This is because $v^{(T-1)}$ is the vector that causes the fault coverage of S to go from being less than F_{cov} to being greater or equal to F_{cov}. But the same cannot necessarily be said about the initial vector $v^{(0)}$. As long as each fault detected by $v^{(0)}$ is also covered by some vector in $v^{(1)}, \ldots, v^{(T-1)}$, we may clearly remove $v^{(0)}$ from the test sequence. This reasoning can be extended to a maximal number of vectors as follows: Let i be the minimum integer such that $v^{(i)}$ detects at least one fault f such that f is not detected by any vector in $v^{(i+1)}, \ldots, v^{(T-1)}$. Then the sequence $S' = v^{(i)}, \ldots, v^{(T-1)}$ has the same fault coverage as S, but is shorter than S when $i > 0$. By using the same machine that generates S, but seeding with initial state $v^{(i)}$ rather than $v^{(0)}$, we have found a generator/initial state combination that produces S'. This can be done algorithmically quite quickly by using a binary search and fault simulations to determine i. We call this form of truncation *head truncation.*

5.6.2 Results

To demonstrate the effectiveness of TST, the CA resulting from a previous experiment were subjected to TST. The set of experiments used were those using full mutation and FIV that were summarized in table 5.1. Each CA underwent first tail and then head truncation.

The results are shown in table 5.13. We see that TST is an effective device; on average the required test length was reduced by %6.2, which is certainly worth the effort, which is minor.

circuit	original	after TST	% impr
c1355	1000	966	3.4
c1908	2056	1799	12.5
c2670	1248	1246	0.0
c3540	3072	3013	1.9
c432	180	164	8.9
c499	280	252	10.0
c5315	992	895	9.8
c6288	34	33	2.9
c7552	2144	1932	9.9
c880	352	341	3.1
average			6.2

Table 5.13: Test sequence truncation results

5.7 Conclusion

In this chapter we have not only successfully applied the simple genetic algorithm to evolve CA that are good at testing a target circuit, we have also explored several mechanisms that improve the GA.

Empirical evidence was presented that indicates that none of the four combinations of choices regarding FIV/VIV and slight or full mutation is especially superior or inferior. The standard GA setting of $p_c = 0.6$ was verified to be appropriate.

Two new mutation operators were proposed, BRF and LPM. BRF proved to work well against some CUTs, while poorly against others. However, when other forms of mutation are omitted, BRF will only yield CA with rules 90, 105, 150, and 165 (when the initial population is entirely primitive CA), which may be desirable if CA regularity is an asset. LPM did not seem to out perform slight mutation when radius 1 neighborhoods were used, however the extension LPM2 proved to be more beneficial than radius 2 slight mutation. The mathematics of Markov chains was used to compute the approximate expected time of last possible mutation, and experimentation showed that the resulting equation approximates reality quite closely.

A mechanism that avoids selection of bad crossover points or mutation choices was proposed, which avoids creation of sinks and abandons. We found that this mechanism improves CAGA, as does a simplified version that only prevents sinks.

These improvements held for both null boundary and periodic boundary CA; the most powerful nonlinear CA were found when CAGA evolved PBCA using sink avoidance. A linear time algorithm for sink detection was defined.

Finally we give a simple technique to further reduce test length called test sequence truncation. Test sequence truncation provided a mean test length reduction of %6.9 for a set of CA evolved by CAGA.

The next chapter describes auxiliary work that gives a new method to obtain a stopping condition for BIST testing.

6. Minimizing a Test Stopping Condition

6.1 Problem Definition and Background

In this chapter we apply a genetic algorithm to another problem in BIST, namely that of stopping the test pattern generation. A genetic algorithm is used to evolve a small subset of a test generator's state variables that can be used to uniquely distinguish the final vector from all other patterns.

Suppose one has chosen a suitable pseudorandom test pattern generator G and initial state $v^{(0)}$ for a target circuit. Furthermore, fault simulation has determined that $T+1$ test vectors are necessary to achieve the target coverage when G is seeded with $v^{(0)}$. The designer proceeds by obtaining the fault free output sequence of the CUT for the $T+1$ input vectors, and computing the signature left by the sequence in the output compactor. For the BIST to work, the output compactor device needs to stop accepting input exactly $T+1$ clock ticks following instigation of testing. Clearly an additional input line to the output compactor must be implemented such that when asserted, the compactor halts and the signature comparison is performed. The question remains: what logic drives this signal?

Though details pertaining to BIST controller architecture are rare in the literature, the typical solution involves the use of a dedicated counter to keep track of the number of tests. More specifically, if the test sequence is of length T, an accumulator of $\lceil log_2 T \rceil$ bits is loaded with the binary representation of $T+1$ upon test initialization. The accumulator is decremented after each test is applied, i.e. with the system clock. The contents of each memory element in the accumulator are fed into a NOR gate, the output of which will rise only when all cells are zero indicating that all tests have been applied [65].

In this chapter a new technique is proposed, which reduces BIST circuitry area by eliminating the need for the accumulator.

A definition is needed to describe the approach to the test stopping problem taken herein. Let S be a k-subset of Z_n, say $S = \{i_1, i_2, \ldots, i_k\}$ where $i_1 < i_2 < \cdots < i_k$. Let v be a binary n-vector. Then define $substate(S, v)$ as the vector $(v_{i_1}, v_{i_2}, \ldots, v_{i_k})$. We say vectors v and u are *equal in substate* S if $substate(S, v) = substate(S, u)$.

Now consider a subset $S = \{i_1, i_2, \ldots, i_k\}$ of Z_n with the following property. For all t in $[0, T-1]$, $substate(S, v^{(t)}) \neq substate(S, v^{(T)})$. Then the subset of generator cells $\{c_{i_1}, c_{i_2}, \ldots, c_{i_k}\}$ uniquely distinguishes $v^{(T)}$ from all other test vectors in the test sequence. If k is small enough, then it becomes feasible to create a k-input AND gate (or, equivalently, a cascade of lesser input AND gates) with input j hooked into the state of c_{i_j} if $v_{i_j}^{(T)} = 1$, or the state negated $\overline{c_{i_j}}$ otherwise. We call S a *stopping condition* (SC), more specifically, if $k = |S|$, we call S a k-SC. If there does not exist a k'-SC for any $k' < k$, we call S a *minimal stopping condition* (MSC). Clearly the size of the chosen SC directly affects the hardware requirements of the logic driving the stop-test signal, thus finding a minimal or near-minimal SC is beneficial.

Let us exemplify this approach. Suppose some 10-input CUT is to be tested by the primitive CA with rule vector (150, 90, 90, 90, 90, 90, 90, 90, 150, 150). When seeded with the initial state vector $v^{(0)} = 0100110001$, 26 vectors are required to achieved the desired fault coverage, i.e. we wish to stop the test procedure immediately following CA time $t = 25$. Table 6.1 gives the content of the 10 CA cells $c_0, \ldots, c_9$ for $0 \leq t \leq 25$. The rightmost column in table 6.1 gives the substate vector $substate(\{4, 6, 9\}, v^{(t)}) = (v_4 v_6 v_9)^{(t)}$. Note that the first value of t such that $(v_4 v_6 v_9)^{(t)} = 100$ is $t = 25$. Thus this substate can be used to uniquely identify the 25th vector, i.e. 4, 6, 9 is a 3-SC. If one wires the inputs of a 3-input AND gate to the CA state variables c_4, $\overline{c_6}$, and $\overline{c_9}$, the output of this gate can be used as a stop-test signal for whichever devices require such an assertion. In the example of table 6.1, $\{4, 6, 9\}$ is the only 3-SC; furthermore it is a MSC since there are no 2-SCs or 1-SCs.

t	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	$v_4v_6v_9$
0	0	1	0	0	**1**	1	**0**	0	0	**1**	101
1	1	0	1	1	**1**	1	**1**	0	1	**1**	111
2	1	0	1	0	**0**	0	**1**	0	0	**0**	010
3	1	0	0	1	**0**	1	**0**	1	0	**0**	000
4	1	1	1	0	**0**	0	**0**	0	1	**0**	000
5	0	0	1	1	**0**	0	**0**	1	1	**1**	001
6	0	1	1	1	**1**	0	**1**	1	1	**0**	110
7	1	1	0	0	**1**	0	**1**	0	0	**1**	111
8	0	1	1	1	**0**	0	**0**	1	1	**1**	001
9	1	1	0	1	**1**	0	**1**	1	1	**0**	110
10	0	1	0	1	**1**	0	**1**	0	0	**1**	111
11	1	0	0	1	**1**	0	**0**	1	1	**1**	101
12	1	1	1	1	**1**	1	**1**	1	1	**0**	110
13	0	0	0	0	**0**	0	**0**	0	0	**1**	001
14	0	0	0	0	**0**	0	**0**	0	1	**1**	001
15	0	0	0	0	**0**	0	**0**	1	0	**0**	000
16	0	0	0	0	**0**	0	**1**	0	1	**0**	010
17	0	0	0	0	**0**	1	**0**	0	1	**1**	001
18	0	0	0	0	**1**	0	**1**	1	0	**0**	110
19	0	0	0	1	**0**	0	**1**	1	1	**0**	010
20	0	0	1	0	**1**	1	**1**	0	0	**1**	111
21	0	1	0	0	**1**	0	**1**	1	1	**1**	111
22	1	0	1	1	**0**	0	**1**	0	1	**0**	010
23	1	0	1	1	**1**	1	**0**	0	1	**1**	101
24	1	0	1	0	**0**	1	**1**	1	0	**0**	010
25	1	0	0	1	**1**	1	**0**	1	1	**0**	100

Table 6.1: Minimal test stopping condition example

6.2 Analysis

Here we wish to determine the expected cardinality of the MSC. Suppose we are given a sequence $v^{(0)}, \ldots, v^{(T)}$ of $T + 1$ n-vectors, assumed to be produced randomly. Let K be the random variable denoting the cardinality of a MSC; we seek the expectation $E(K)$ in terms of T and n. Deducing an exact equation is hard, here we simply derive a lower bound on $E(K)$. We will see that the cardinalities of the MSCs found via a GA are very close to this lower bound, allowing us to conclude that the GA performs well. We assume that the vectors are produced at random, with each vector being drawn from a uniform distribution. The fact that there are no repeat vectors in a sequence from a test generator is only taken into account when determining the probability that there exists an n-SC (this probability is 1 if there are no repeated vectors allowed).

We first determine $s(k, T)$, the probability that a fixed k element substate vector w of $v^{(T)}$ is a SC. Given a value of t, with $0 \leq t < T$, the probability that w is not equal to the same substate of $v^{(t)}$ is simply $1 - 2^{-k}$. The probability that w is not equal to the same substate for *any* t, $0 \leq t < T$, is therefore $(1-2^{-k})^T$, as the vectors are independently produced. Thus $s(k, T) = (1 - 2^{-k})^T$.

Letting $e(k)$ denote the probability of existence of a k-SC, we now consider the complementary probability $e^c(k) = 1 - e(k)$; this is the probability there is no k-SC. Choosing some fixed k-substate w, the probability that w is not a SC is $s^c(k, T) = 1 - s(k, T) = 1 - (1 - 2^{-k})^T$. Let w' be a k-substate different from w, and let $m = |w \cap w'|$. Let A denote the event that w is not a SC and B denote the event that w' is not a SC. If $m = 0$, i.e. w and w' do not share any bit positions, then the probability of B given A, $P(B|A) = s^c(k, T)$, as the two events are independent. However when $m > 0$ we find that $P(B|A) > s^c(k, T)$. This is explained as follows.

Suppose w is not a SC, then we know that there exists $\tau > 0$ vectors in $v^{(0)} \ldots v^{(T-1)}$ that equal $v^{(T)}$ in w; let u be one of these vectors. The probability that u matches w' is no longer 2^{-k}, it is increased to 2^{m-k}. (Since $m > 0$, we have $2^{m-k} > 2^{-k}$). Thus the probability that w is not a SC is:

$$\begin{aligned} P(B|A) &= 1 - (1 - 2^{m-k})^{\tau}(1 - 2^{-k})^{T-\tau} \\ &> 1 - (1 - 2^{-k})^{\tau}(1 - 2^{-k})^{T-\tau} \\ &= 1 - (1 - 2^{-k})^{T} \\ &= s^c(k, T) \end{aligned}$$

If the events that each of all $N = \binom{n}{k}$ possible k-substate are not SCs were independent, we could express $e^c(k) = s^c(k, T)^N$. However, as the above argument demonstrates, even assuming that one k-substate is not a SC implies that the probabilities of $\binom{n}{k} - \binom{n-k}{k} - 1$ other k-substates are each increased to some value greater than $s^c(k,T)$. Through similar reasoning it follows that this effect is compounded the more substates we mark as non-SCs. We must conclude $e^c(k) \geq s^c(k,T)^{\binom{n}{k}}$, which implies

$$e(k) \leq 1 - s^c(k,T)^{\binom{n}{k}} \tag{6.1}$$

We may now use equation 6.1 to obtain a lower bound on $E(K)$. Consider the cumulative distribution function $p(K \leq k)$, i.e. the probability that the cardinality of a MSC is *at most* k. It is trivially true that if a SC exists with cardinality k, then for all k' with $k \leq k' \leq n$ there exist SCs of size k'. Conversely, if there does not exist a SC of cardinality k, then it is implied that for all k' with $0 < k' \leq k$ there does not exist a k'-SC. It follows that $p(K \leq k) = e(k)$. We proceed:

$$\begin{aligned} E(K) &= \textstyle\sum_{k=1}^{n} kp(K = k) \\ &= \textstyle\sum_{k=1}^{n} k(p(K \leq k) - p(K \leq k-1)) \\ &= \textstyle\sum_{k=1}^{n} kp(K \leq k) - \sum_{k=0}^{n-1}(k+1)p(K \leq k) \end{aligned}$$

Since $p(K \leq n) = 1$ and $p(K \leq 0) = 0$:

$$\begin{aligned} E(K) &= n + \textstyle\sum_{k=1}^{n-1} kp(K \leq k) - \sum_{k=1}^{n-1}(k+1)p(K \leq k) \\ &= n + \textstyle\sum_{k=1}^{n-1}(k-(k+1))p(K \leq k) \\ &= n - \textstyle\sum_{k=1}^{n-1} p(K \leq k) \end{aligned}$$

Now using 6.1 we obtain:

$$\begin{aligned} E(K) &\geq n - \textstyle\sum_{k=1}^{n-1}(1 - s^c(k,T)^{\binom{n}{k}}) \\ &= n - \textstyle\sum_{k=1}^{n-1}(1 - (1-(1-2^{-k})^T)^{\binom{n}{k}}) \\ &= 1 + \textstyle\sum_{k=1}^{n-1}(1-(1-2^{-k})^T)^{\binom{n}{k}} \\ &= E'(K) \end{aligned} \tag{6.2}$$

Figure 6.1 plots the bound $E'(K)$ from equation 6.2 for $T \in \{10^3, 10^4, 10^5\}$ and $n \in [20, 200]$. It appears that once n is large enough, the lower bound becomes

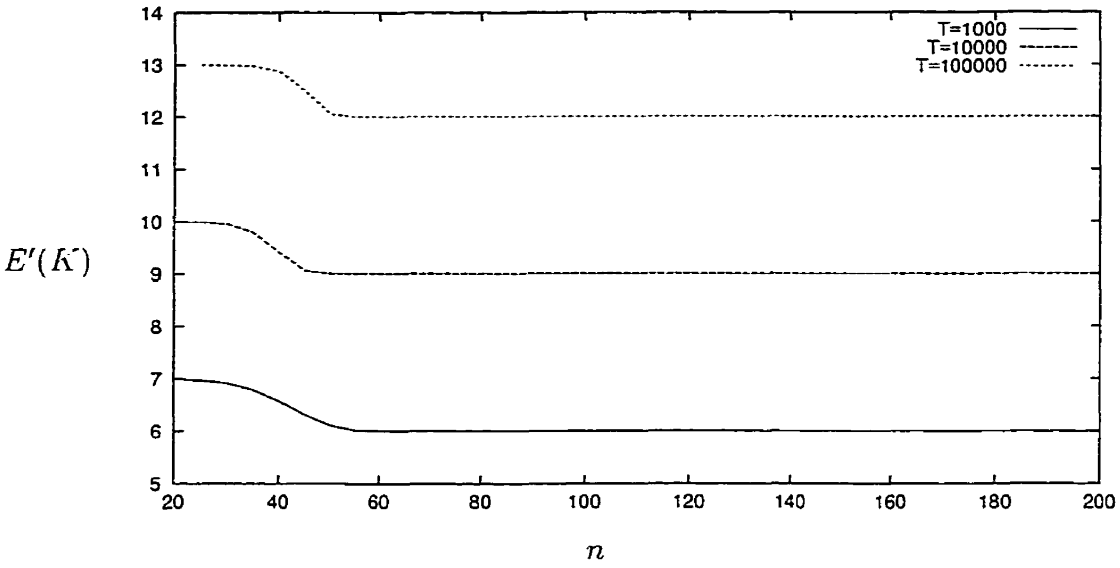


Figure 6.1: Lower bound on $E(K)$ for various n and T

only dependent on T, assuming the values of 6, 9, and 12 for the respective values of T. Therefore, somewhat counterintuitively, if T is fixed and n is allowed to grow indefinitely, the expected size of a MSC will never be less than some constant value.

6.3 Implementation of MSCGA

In this section we describe the *Minimal Stopping Condition Genetic Algorithm* (MSCGA), which attempts to find a small SC for a given sequence of vectors. The individuals in MSCGA are simply various substates. If an individual is a k-SC, it is assigned fitness proportional to $(n-k)^2$. On the other hand, if an individual is not a SC, it is assigned a fitness of 0.

We first discuss the encoding of substates, and then describe the crossover and mutation operators in MSCGA.

6.3.1 Encoding

In theory, an individual is represented by an n symbol string over the alphabet $\{X, -\}$; finding X ($-$) at position j in the string means that cell j of the generator is included (not included) in the substate. In practice, an individual I is stored as two bitstrings called $B_1(I)$ and $B_0(I)$. A 1 at position j in $B_1(I)$ means that j is included in the substate and $v_j^{(T)} = 1$. Conversely, a 1 at position j in $B_0(I)$ also means that j appears in the substate, but $v_j^{(T)} = 0$. When position j of neither strings is set, this indicates that cell j is not used in the substate. Of course, since the final vector is known to the GA a priori, only one of $B_1(I)$ or $B_0(I)$ can ever be set at each position over all possible individuals. This encoding lends itself to bit-level machine manipulation when performing vector comparisons, and thus aids in accelerating the algorithm. For example, the SC illustrated in table 6.1 would be encoded with

$$B_1(I) = 0000100000$$
$$B_0(I) = 0000001001$$

6.3.2 Genetic Operators

The genetic operator of crossover is carried out as expected. Suppose A and B are individuals that are to partake in crossover. The crossover point x is chosen uniformly from the set Z_{n+1}. The first offspring C is created such that $B_i(C)$ is the concatenation of the leftmost x bits of $B_i(A)$ with the rightmost $n-x$ bits of $B_i(B)$. The second offspring D is created such that $B_i(D)$ is the concatenation of the leftmost x bits of $B_i(B)$ with the rightmost $n-x$ bits of $B_i(A)$.

When an individual I is to be mutated, the straightforward approach would be to select a bit position x at random, and then either add x to the substate if it is not included, or remove x if it is already a member. However, if the GA evolves to the point at which the average substate cardinality is significantly less than $n/2$ (a common occurrence), then mutation would most likely cause elements to be added to substates. As we are seeking to minimize a SC, this is undesirable.

To avoid this, the employed mutation operator increases or decreases the size of the effected substate with equal probability. If the operator chooses to add an element, one of the missing elements is selected at random and the appropriate bit of either $B_0(I)$ or $B_1(I)$ is set. Otherwise a random member of the substate is removed by clearing a bit in either of the bitstrings.

6.4 Experiments and Results

Experiments were first performed by applying MSCGA to the test stopping problem. "Pseudo" test sequences were created using the standard C library function `drand48()`, each bit in each vector being set with probability 1/2. A mechanism was used to ensure that no repeat vectors would be present in the sequence. The sequence was then fed to the GA. Members of the initial population were also created randomly, with each position being included in the SC with probability 1/2. Other parameters are:

- crossover probability $p_c = 0.6$
- mutation probability $p_m = 0.13$
- population size $N = 300$
- the fitness scaling coefficient $C_{mult} = 1.5$
- the number of individuals reproduced unchanged in the next generation $k_{rep} = 10$
- the number of generations without improvement before MSCGA halting $\Delta g = 200$

Note that p_m is slightly smaller than the mutation probability used in CAGA (0.15). This parameter was lowered here because of the hunch that mutation plays a less important role in MSCGA than in CAGA. Table 6.2 summarizes the results. Each number in the table is the average minimal SC cardinality found by the GA over 5 runs, each run using a different randomly generated test sequence. The actual integral values averaged for each entry differed by at most 1, suggesting a low variance. The values in each row correspond to the same generator width n, while columns indicate test length T.

Given that our lower bounds on the expected cardinalities of MSCs for $T = 10^3$, $T = 10^4$, and $T = 10^5$, were respectively 6, 9, and 12, we conclude that MSCGA

n	$T = 10^3$	$T = 10^4$	$T = 10^5$
20	7.0	10.4	14.0
40	7.0	10.2	13.6
60	7.2	10.6	13.4
80	7.2	10.4	14.0
100	7.4	10.6	13.6
120	7.2	10.6	13.6
140	7.6	10.6	13.8
160	7.0	10.4	13.6
180	7.4	10.6	13.6
200	7.2	10.6	13.6

Table 6.2: Average stopping condition cardinalities over various test lengths and generator widths

performs quite well, as the average best individual found was always no more than 2 bits larger than these numbers. Of course, given that we have only computed a lower bound on the expected cardinality of a MSC, it is quite possible that the actual expected values are even closer to these results.

For further experiments, we modify MSCGA such that the fitness evaluation is not only based on the SC size, but also a second metric – the span of the SC. The span of the SC is the value of $b_R - b_L + 1$, where b_R and b_L are the rightmost bit and the leftmost bit in the SC, respectively. In a design environment in which the logical relative positions of test generator cells is correlated to the physical layout at fabrication, a smaller span is preferable. This is because the longer the span, the longer the wires that carry the SC cell values to the conjunctive network that detects the final vector.

As in the previous experiments, a substate (individual) that is not a SC receives a raw fitness of 0. Those that are SCs obtain a raw fitness proportional to the the square of the difference of the total number of cells and those involved in the SC. However, to promote low spans, an additional quantity based on the span is added to this value. If w and s are the weight and span of a SC, then the raw fitness is

n	$T = 10^3$		$T = 10^4$		$T = 10^5$	
	size	span	size	span	size	span
20	7.4	11.2	10.4	15.6	13.6	17.4
40	7.0	18.0	10.0	25.0	13.6	27.0
60	7.0	18.2	10.2	32.4	13.2	35.0
80	7.0	27.0	10.0	34.0	13.2	38.6
100	7.0	38.2	10.0	45.8	13.4	56.2
120	7.0	31.8	10.0	55.2	13.0	57.4
140	7.0	22.2	10.2	60.8	13.4	83.6
160	6.8	41.4	10.0	73.6	13.2	76.8
180	7.2	31.4	10.0	67.6	13.2	80.6
200	6.8	58.0	10.0	94.8	13.4	110.4

Table 6.3: MSCGA with span minimization results

computed as:

$$f_{raw} = (n-w)^2 + ((n-w+1)^2 - (n-w)^2)(n-s)/(n-w+1)$$

Essentially this defines a two tiered fitness measure in which a SC with weight w will always receive a higher raw fitness than one with weight $w+1$, while two SCs with equal weight will be assigned raw fitnesses such that the one with lower span will be fitter.

The results of these experiments are given in table 6.3. For each (n, T) pair, the numbers in the table are averages over 5 separate MSCGA runs. It seems that the span of the resulting best individual is usually not much more than 1/2 of the total width of the generator. We note that the size of the best SC found by MSCGA is not affected by this slight change in fitness measure, thus we gain a more "compact" SC without sacrificing SC cardinality.

6.5 Comparison Against Brute Force Search

In the previous section it was demonstrated that a SC with cardinality close to the expected minimum can be found via GA techniques. Utilizing bit level manipulation for vector comparison, these GAs run much faster than CAGA of the previous chapter, however for test lengths of 10^5, MSCGA still takes about an hour to complete. The question remains: is exhaustive search of all SCs of some cardinality k a superior approach?

Suppose we have determined that the expected MSC cardinality is k, then the previous question is very similar to the question: what is the ratio r of the expected number of k-SCs to $\binom{n}{k}$, the total number of k-substates? There exist algorithms to generate all $\binom{n}{k}$ k-subsets of an n element set that run in constant amortized time [66]. We would expect to have to examine only about $r/\binom{n}{k}$ substates before finding a k-SC. It is quite conceivable that this scheme would find a k-SC much faster than MSCGA.

To test this hypothesis, random test sequences of length $T = 1000$ were created as before, but rather than using MSCGA to find a SC, brute force search over all 7-substates was used in an attempt to identify a 7-SC. The results in the second column of table 6.2 suggest that 7-SCs usually exist for this test length. The results indicated that in this case r is sufficiently high so that a 7-SC is always found within seconds. This appears to render MSCGA an inefficient approach to the problem.

However, use of MSCGA is warranted by the following observation. MSCGA does not use the parameter k passed to the brute force search, and thus is capable of finding a SC with cardinality less than k, if one exists. If a MSC with cardinality $k' < E(K)$ exists, one would expect the number of such MSCs to be very small. For such a test sequence, brute force search would typically need to exhaust a large fraction of the whole search space. Noting that the size of the search space is $\binom{n}{k'}$, the time requirements of the brute-force algorithm could easily be prohibitive.

Experiments were performed to determine if MSCGA could locate a MSC with lower than expected cardinality. Similar to those of section 6.4, these experiments worked with randomly generated test sequences. However, an additional mechanism was added to the code such that given parameters k and m, a set S of m random k-substates is created. To create the actual test sequence, a final vector $v^{(T)}$ is randomly chosen, and then the remainder of the test sequence $v^{(0)}, \ldots, v^{(T-1)}$ are selected such

	$m = 100$		$m = 1000$	
n	average $\|SC\|$	$100/\binom{n}{6}$	average $\|SC\|$	$1000/\binom{n}{6}$
80	6.60	332.8×10^{-9}	5.07	332.8×10^{-8}
100	6.73	83.9×10^{-9}	5.27	83.9×10^{-8}
120	6.80	27.4×10^{-9}	5.33	27.4×10^{-8}
140	6.87	10.7×10^{-9}	5.53	10.7×10^{-8}
160	6.93	4.7×10^{-9}	5.73	4.7×10^{-8}
180	7.00	2.3×10^{-9}	5.67	2.3×10^{-8}
200	7.07	1.2×10^{-9}	5.93	1.2×10^{-8}

Table 6.4: Artificial SC injection with $k = 6$ and $T = 1000$

that no vector $v^{(i)}$, $0 \leq i < T$ matches $v^{(T)}$ in any substate $w \in S$. In this fashion, m k-SCs are artificially injected into the sequence.

Table 6.4 summarized the MSCGA runs that with artificial injection of 6-SCs for test sequences of length 1000 (recall that the experiments of table 6.2 suggested that 7 was the smallest SC normally found by MSCGA for $T = 1000$). Experiments were performed for both $m = 100$ and $m = 1000$. Each value in the "average $|SC|$" columns in table 6.4 is taken over 15 runs. One observes that even when only 100 of the $\binom{n}{6}$ possible 6-substates are forced to be SCs, MSCGA is still capable of locating one of them. When $m = 1000$ it appears that there are enough "forced" 6-SCs such that their intersections imply a high proability of the existence of 5-SCs; furthermore MSCGA seems to be able to locate these 5-SCs quite frequently. Also included in table 6.4 are the values of $m/\binom{n}{6}$ for the various n and m values. These give an indication that, for these test sequences, brute force search is analogous to looking for a needle in a haystack.

At this point one might suggest that real pseudorandom test generators are highly unlikely to have smaller than expected SCs in their sequences. However, suppose we were to obtain an exact expression for the expected minimal stopping condition, say $E(K) = f(n, T)$. We know that $f(n, T)$ increases continuously with T, thus there are many values of (n, T) such that $f(n, T)$ is close to $k + \frac{1}{2}$ for some integer k, i.e. $f(n, T)$ is near equidistant from k and $k + 1$. Now does one run the brute force search against all k-substates or all $(k + 1)$-substates? Of course finding a k-SC is preferable, yet we are not very confident that one exists. If no k-SC exists, running the brute force search for a k-SC is futile and costly. Running the brute force search for a $(k+1)$-SC will certainly be successful, but will result in a suboptimal solution if there indeed exists a k-SC. In such a situation, use of MSCGA is clearly the superior option, as the previous experiments demonstrate.

6.6 Conclusion

This chapter's contributions are threefold. A new technique for obtaining a test halt signal has been proposed. This technique uses only a subset of the test generator's cells and requires no additional memory elements. A lower bound on the number of generator cells needed to be tapped to create this signal is derived. Finally, the use of a genetic algorithm to locate such cells is explored and contrasted with using brute force search.

7. Conclusions and Future Work

It is unclear to what extent CA conform to the *building block hypothesis*, which is a means of determining a priori how effective GA techniques will be against a problem. However, results of this thesis indicate that CAGA can usually improve significantly over primitive CA for testing. Empirical evidence was presented that indicates that in the simple CAGA, slight mutation is slightly more effective than full mutation.

Two advanced mutation operators were introduced: Bit Role Flipping and Last Possible Mutation. BRF seemed to work well against some benchmarks but was detrimental for others. The mathematics of Markov chains was used to compute the expected time of LPM, and experimentation with primitive CA showed that the resulting equation is a good predictor. Though uneffective for the test lengths required by our target benchmarks, when extended to LPM2 the operator was shown to be a powerful augmentation to CAGA, outperforming radius 2 slight mutation. The price of using LPM2 is the emergence of radius 2 rules, which are more costly to implement in hardware.

Mechanisms that avoid degenerate CA behavior were explored. Preventing narrow abandons and sinks or simply sinks alone proved to be good practice when evolving both null boundary and periodic boundary CA. A fast algorithm to prevent sinks was given. On average the shortest test lengths were obtained by periodic boundary CA when evolved using sink avoidance.

Auxiliary work on obtaining a BIST stopping condition was presented. A lower bound on the expected minimum number of pseudorandom pattern generator cells needed to uniquely distinguish the final test vector was derived. Use of GA and brute force search techniques for stopping condition identification were compared. The results indicate that under typical test lengths and generator widths, only a fraction of the cells are necessary to form a stopping condition.

On a higher level, we have shown that genetic evolution of CA for test generation can be enhanced by the utilization of advanced genetic operators that are devised specifically for CA. Such a philosophy is surely applicable to other genetic algorithm applications.

The potential avenues of research that could emerge from the work of this thesis are abundant; the remainder proposes several.

7.1 GA fine tuning

Due to the large number of parameters controlling the genetic algorithms presented in this thesis, it is doubtful that the combinations of values chosen herein are the best. Experimentation with different values of any of the CAGA parameters listed in section 5.3.1 could reveal a more productive actual parameter set.

Using completely random nonlinear CA rather than primitive CA in the initial population will clearly result in a less fit initial population and would require a much larger value of T_{max}. However, it is quite plausible that over many generations such a starting point would result with better test generators.

7.2 Alteration of GA goal

At the onset of this research, successful experiments (not discussed in the thesis) were executed that took a different approach to the testing problem. Rather than fixing F_{cov} and attempting to minimize the number of vectors needed, the number of vectors generated was fixed and the genetic algorithm sought to maximize the fault coverage. Further research along these lines would certainly be interesting.

Other test generator attributes could potentially be optimized via GA techniques. For example, limiting the power consumption incurred during testing (as in [50]) is a worthwhile quest. Another option is to consider more complex fault models during fault simulation.

7.3 GA Distribution

GAs lend themselves to immediate parallelization in that the evaluations of the fitness function against each individual in a generation are totally independent. Thus distributing these evaluations across a network can vastly accelerate the algorithm, allowing for more generations and larger population sizes to be processed. Aside from this "obvious" means of GA distribution, there are more advanced paradigms such as the *island model.*

7.4 Evolution of Feedback Shift Registers

In practice, feedback shift registers are used as BIST generators more often than CA. This is likely due to the fact that in general a shift register cell takes up less area than a CA cell [1].

A nonlinear type II FSR is defined totally by one function of n variables, i.e. the logic driving the leftmost cell. These machines are discussed in [7]. Because of the successes of this thesis with the evolution of CA, it would seem reasonable to use a GA to evolve nonlinear FSR that minimize test length. Of course the encoding scheme used for the feedback function is an issue that would need intelligent resolution, since manipulation of entire truth tables is clearly infeasible when n is much greater than 10. A possible solution might employ a decision diagram representation.

7.5 Sequential Circuits

Based on the results of this thesis, extension of the CAGA to evolve CA with cells involved in a CUT's scan path should certainly be worth the effort. When testing a combinational circuit, the temporal cost of a single test is typically one clock tick, while under a scan-based architecture application of a single test takes multiple ticks because of the scan-in/scan-out operations. Thus reduction of test length is an important pursuit for scan-based designs.

7.6 Signature Analysis

Moving our attention to "the other shore" of BIST, synthesis of a signature analyzer that somehow reduces the probability of aliasing given a CUT and test sequence might be a job for a GA. However, the obvious way to evaluate an individual's fitness is to compute the signature left by each of the faulty output streams and count how many match the fault free signature. Given the typical number of faults and test vectors, the time requirements of such a program would be very high. Perhaps a more cunning approach exists.

7.7 MSCGA

Recall that the MSCGA of chapter 6 assigns a zero fitness to substates that are not stopping conditions. The algorithm may perform better when a more intelligent fitness function is used, i.e. one that incorporates the number of times a substate matches into its computation. This stems from the fact that a substate that matches only a small number of times is likely converted to a true stopping condition in a small edit distance. Of course non-genetic means of obtaining a MSC exist and warrant further research.

7.8 Detection of Abandons

Though checking for small abandons in section 5.5 suggested that abandons do not impede CAGA (avoiding sinks alone gave similar results), existence of a fast detection algorithm would facilitate experimentation with larger abandons. Finding such an algorithm or proving the problem NP-complete would be beneficial.

7.9 Further Sink Prevention Experimentation

The experiments of section 5.5 were performed prior to discover of the fast sink detection algorithm. Running experiments with k-sink prevention for larger k will certainly reveal more information with regards to the value of this mechanism.

Generalization to short subcycle detection and determining probabilities of sinks and short subcycles are other possible areas of valuable research.

Bibliography

[1] P. D. Hortensius, R. D. McLeod, W. Pries, D. M. Miller, and H. C. Card. Cellular Automata-Based Pseudorandom Number Generators for Built-In Self-Test. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 8(8):842–859, August 1989.

[2] M. Serra, T. Slater, J. C. Muzio, and D. M. Miller. The Analysis of One-Dimensional Linear Cellular Automata and Their Aliasing Properties. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 9(7):767–778, July 1990.

[3] P.P.Chaudhuri, D.R.Chowdhury, S. Nandi, and S. Chattopadhyay. *Additive Cellular Automata Theory and Applications Volume 1.* IEEE Computer Society Press, 1997.

[4] Paul H. Bardell, William H. McAnney, and Jacob Savir. *Built-In Test for VLSI: Pseudorandom Techniques.* John Wiley & Sons, 1987.

[5] Kevin Cattell. *Characteristic Polynomials of One-Dimensional Linear Hybrid Cellular Automata.* PhD thesis, University of Victoria, Victoria, Canada, 1995.

[6] Rene David. *Random Testing of Digital Circuits Theory and Applications.* Marcel Dekker, 1998.

[7] Solomon W. Golomb. *Shift Register Sequences.* Holden-Day, Inc., 1967.

[8] M. Serra, D. M. Miller, and J. C. Muzio. Linear Cellular Automata and LFSRs are Isomorphic. *Third Tech. Workshop, New Directions IC Testing*, pages 195–205, October 1988.

[9] Harold S. Stone. *Discrete Mathematical Structures and Their Applications.* Science Research Associates, Inc., 1973.

[10] S. Wolfram, O. Martin, and A.M. Odlyzko. Algebraic Properties of Cellular Automata. *Communications in Mathematical Physics*, 93:219–258, March 1984.

[11] S. Zhang, R. Byrne, J. C. Muzio, and D. M. Miller. Why Cellular Automata are better than LFSRs as Built-In Self-Test Generators for Sequential-type Faults. In *Proc. IEEE Int. Symp. on Circuits and Systems*, pages 69–72, May 1994.

[12] S. Zhang, R. Byrne, J. C. Muzio, and D. M. Miller. Quantitative Analysis for Linear Hybrid Cellular Automata and LFSR as Built-In Self-Test Generators for Sequential Faults. *J. of Electronic Testing: Theory and Applications*, 7:209–221, 1995.

[13] S. Wolfram. Cellular Automata as Simple Self-Organizing Systems. Caltech preprint CALT-68-938, 1982.

[14] S. Wolfram. Statistical Mechanics of Cellular Automata. *Reviews of Modern Physics*, 55:601–644, July 1983.

[15] S. Wolfram. Universality and Complexity in Cellular Automata. *Physica D*, 10:1–35, jan 1984.

[16] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[17] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. distributed on tape to participants of the Special Session on ATPG and Fault Simulation, Int. Symposium on Circuits and Systems, June 1985.

[18] Georges R. Harik and Fernando G. Lobo. A Parameter-less Genetic Algorithm. Illinois Genetic Algorithms Laboratory (ILLEGAL) Report No. 99009, 1999.

[19] Micaela Serra. *The Electrical Engineering Handbook*, chapter 85.1. CRC Press and IEEE Press, second edition, 1997.

[20] R. D. Eldred. Test Routines Based on Symbolic Logical Statements. *Journal of Association of Computer Machines*, 6(1):33–36, 1959.

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

[22] J. P. Roth. Diagnosis of Automata Failures: A Calculus and a Method. *IBM J. Res. Develop*, 10(4):278–291, 1966.

[23] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, C-30(3):215–222, 1981.

[24] P. Goel and B. C. Rosales. PODEM-X: An Automatic Test Generation System for VLSI Logic Structures. In *18th IEEE Design Automation Conference*, pages 260–268, June 1981.

[25] Hyung Ki Lee and Dong Sam Ha. An Efficient, Forward Fault Simulation Algorithm Based on Parallel Pattern Single Fault Propagation. In *IEEE International Test Conference*, pages 946–955, 1991.

[26] R. A. Frohwerk. Signature Analysis: A New Digital Field Service Method. *Hewlett-Packard J.*, 28(9):2–8, 1977.

[27] P. D. Hortensius. *Parallel Computation of Non-deterministic Algorithms in VLSI.* PhD thesis, University of Manitoba, Winnipeg, Canada, 1987.

[28] E. B. Eichelberger, E. Lindbloom, J. A. Waicukauski, and T. W. Williams. *Structured Logic Testing.* Prentice Hall, 1991.

[29] S. Funatsu, N. Wakatsuki, and T. Arima. Test Generation Systems in Japan. In *12th Design Automation Symposium*, pages 114–122, June 1975.

[30] H. Ando. Testing VLSI with Random Access Scan. *Digest of Papers, COMPCON 80*, pages 50–52, February 1980.

[31] J. H. Stewart. Future Testing of Large LSI Circuit Cards. In *Semiconductor Test Symposium*, pages 6–15, October 1977.

[32] J. H. Stewart. Application of Scan/Set for Error Detection and Diagnostics. In *Semiconductor Test Conference*, pages 152–158, October-November 1978.

[33] A. W. Burks, editor. *Essays on Cellular Automata.* University of Illinois Press, 1970.

[34] K. Preston Jr. and M. J. B. Duff. *Modern Cellular Automata Theory and Applications.* Plenum Press, 1984.

[35] E. F. Codd. *Propagation, Computation, and Construction in Two-Dimensional Cellular Spaces.* PhD thesis, University of Michigan, 1965. published as *Cellular Automata.* New York: Academic Press, 1968.

[36] J. H. Conway. *Winning Ways for your Mathematical Plays*, volume 2, chapter 25. Academic Press, 1982.

[37] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman and Company, 1982.

[38] Stephen Wolfram. Random Sequences Generated by Cellular Automata. *Advances in Applied Mathematics*, pages 123–169, 1986.

[39] S. Wolfram. Twenty Problems in the Theory of Cellular Automata. *Physica Scripta*, T9:170–183, 1985.

[40] K. Cattell, F. Ruskey, J. Sawada, M. Serra, and C.R. Miers. Fast Algorithms to Generate Necklaces, Unlabeled Necklaces and Irreducible Polynomials over GF(2). *Journal of Algorithms*, 37(2), Nov 2000.

[41] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley, 1966.

[42] J. H. Holland and J. S. Reitman. *Cognitive Systems Based on Adaptive Algorithms*, pages 313–329. Academic Press, 1978.

[43] D. E. Goldberg. *Computer-aided Gas Pipeline Operation using Genetic Algorithms and Rule Learning.* PhD thesis, University of Michigan, 1983.

[44] Marianne Haslev. A Classifier System for the Production by Computer of Past Tense Verb-Forms. Presented at a Genetic Algorithms Workshop at the Rowland Institute, Nov 1986.

[45] John R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection.* Massachusetts Institute of Technology, 1992.

[46] D. E. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms.* Addison-Wesley, Reading, MA, second edition, 1981.

[47] Moshe Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D*, pages 193–208, 1996.

[48] Moshe Sipper and Marco Tomassini. Generating Parallel Random Number Generators by Cellular Programming. *International Journal of Modern Physics*, 7(2):181–190, 1996.

[49] S. Chiusano, F. Corno, P. Prinetto, and M. Sonza Reorda. Cellular Automata for Deterministic Sequential Test Pattern Generation. pages 60–65, 1997.

[50] F. Corno, M. Rebaudengo, M. Sonza Reorda, G. Squillero, and M. Violante. Low Power BIST via Non-Linear Hybrid Cellular Automata. In *Proceedings of the IEEE VLSI Test Symposium*, pages 29–34, 2000.

[51] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, and R. Mosca. Advanced Techniques for GA-based sequential ATPGs. In *IEEE Design & Test Conference*, March 1996.

[52] F. Corno, M. Sonza Reorda, and G. Squillero. An Improved Cellular Automata-Based BIST Architecture for Sequential Circuits. In *ISCAS2000: IEEE International Symposium on Circuits and Systems*, Geneve (CH), May 2000.

[53] F. Corno, N. Gaudenzi, P. Prinetto, and M. Sonza Reorda. On the Identification of Optimal Cellular Automata for Built-In Self-Test of Sequential Circuits. In *VTS'98: 16th IEEE VLSI Test Symposium*, Monterey, CA, USA, April 1998.

[54] M. Rebaudengo and M. Sonza Reorda. Floorplan Area Optimization Using Genetic Algorithms. In *GLS'94: 4th IEEE Great Lakes Symposium on VLSI*, Notre Dame, IN, USA, March 1994.

[55] P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms.

In *ITC'94: IEEE International Test Conference*, Washington D. C., USA, October 1994.

[56] F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. Garda: a Diagnostic ATPG for Large Synchronous Sequential Circuits. In *ED&TC'95: IEEE European Design and Test Conference*, Paris, France, March 1995.

[57] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, and R. Mosca. Advanced Techniques for GA-based Sequential ATPGs. In *ED&TC'96: IEEE European Design and Test Conference*, Paris, France, March 1996.

[58] F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. Partial Scan Flip Flop Selection for Simulation-based Sequential ATPGs. In *IEEE International Test Conference*, Washington, USA, October 1996.

[59] F. Corno, M. Rebaudengo, P. Prinetto, and M. Sonza Reorda. New Static Compaction Techniques of Test Sequences for Sequential Circuits. In *ED&TC'97: IEEE European Design and Test Conference*, pages 37–43, Paris, France, March 1997.

[60] F. Corno, M. Sonza Reorda, and G. Squillero. Approximate Equivalence Verification of Sequential Circuits via Genetic Algorithms. poster at DATE'99: IEEE Design, Automation & Test in Europe, Munich (Germany), March 1999.

[61] Philippe Giguère and David E. Goldberg. Population Sizing for Optimum Sampling with Genetic Algorithms: A Case Study of the Onemax Problem. In *Third Annual Genetic Programming Conference*, Madison, WI, U.S.A., July 1998.

[62] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, Ann Arbor, 1975.

[63] R. Ramakumar. *Engineering Reliability Fundamentals and Applications.* Prentice Hall, New Jersey, 1993.

[64] Eric Agyekum and Michael Hermary, June 2000. personal correspondence.

[65] M. A. Breuer, R. Gupta, and J. C. Lien. Concurrent Control of Multiple BIT Structures. In *Proceedings of the International Test Conference*, pages 431–442, September 1988.

[66] C. Mifsud. Algorithm 154: Combination in lexicographic order. *Communications of the ACM 6*, page 103, 1963.

[67] David Bryan. The ISCAS '85 benchmark circuits and netlist format. MCNC (formerly Microelectronics Center of North Carolina) technical note, 1988.

APPENDIX A

Benchmark Circuits

Table A.1 lists various attributes of the ISCAS '85 Benchmark Circuits [17, 67]. Note that the numeric part of the circuit name is the number of internal wires, and the number of faults is counted *after* equivalence fault collapsing. Circuits c499 and c1355 are functionally equivalent; all EXOR gates in c499 have been expanded into the four NAND gate implementation in c1355.

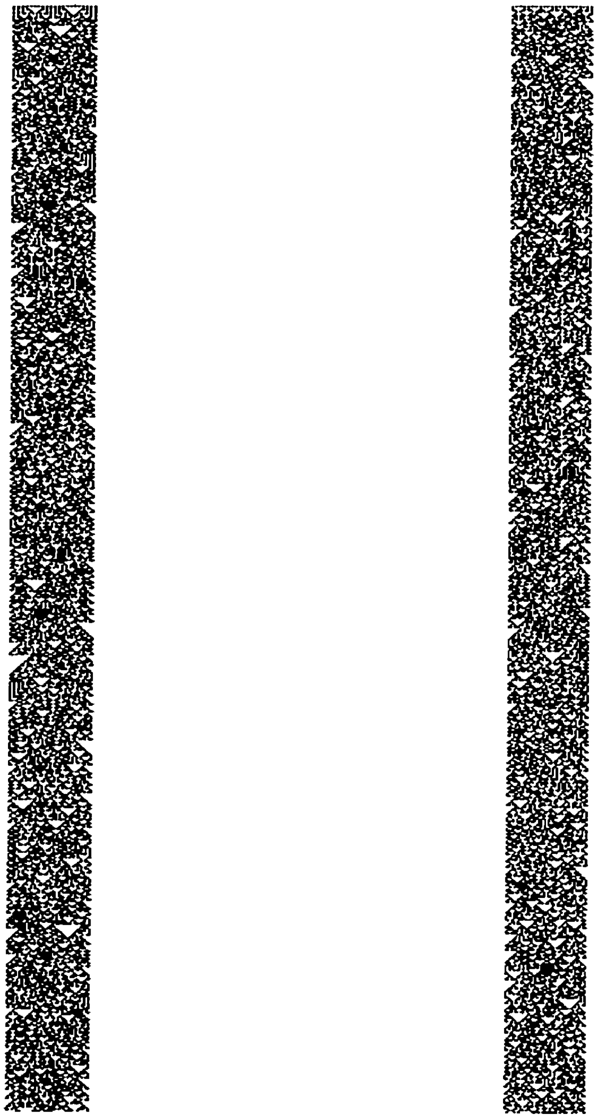
circuit	function	total gates	inputs	outputs	faults	T_{max}
c432	Priority Decoder	160 (18 EXOR)	36	7	524	1000
c499	ECAT	202 (18 EXOR)	41	32	758	5000
c880	ALU and Control	383	60	26	942	20000
c1355	ECAT	546	41	32	1574	5000
c1908	ECAT	880	33	25	1879	10000
c2670	ALU and Control	1193	233	140	2747	100000
c3540	ALU and Control	1669	50	22	3428	20000
c5315	ALU and Control	2307	178	123	5350	5000
c6288	16-bit Multiplier	2406	32	32	7744	500
c7552	ALU and Control	3512	207	108	7550	15000

Table A.1: The Benchmark Combinational Circuits

APPENDIX B

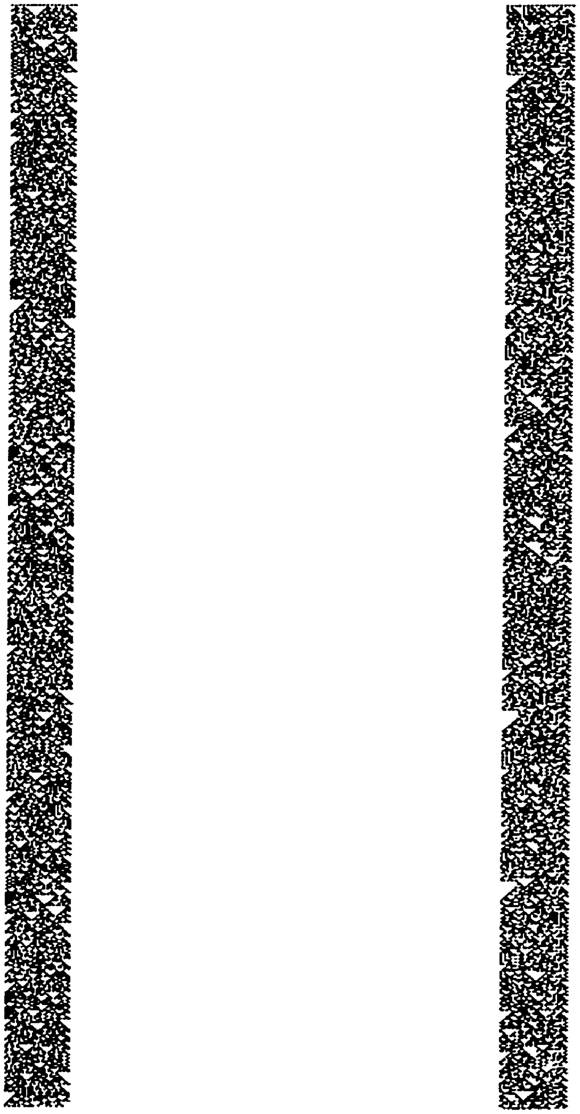
Example CA Output Plots

This appendix provides CA growths for the best of initial population and best evolved CA for the experiments using full mutation and fixed initial vector of section 5.3.2.3. The growths use the same initial vector as used in the CAGA runs, and span 600 time steps. Note that the benchmark c432 is omitted since CAGA provided no improvement over the initial population in this experimental run.



Best primitive CA Best evolved CA

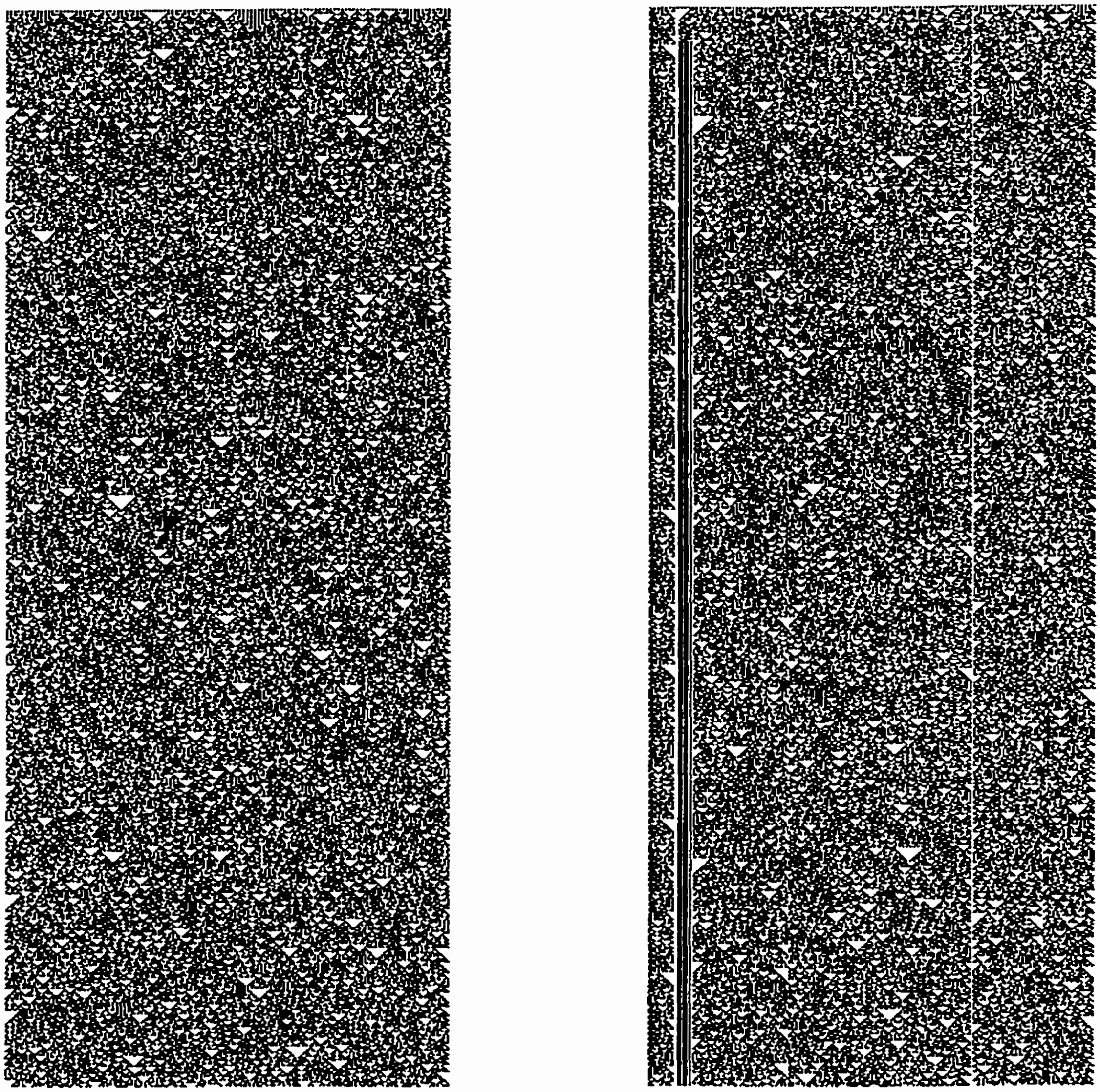
Figure B.1: CA growths for c1355



Best primitive CA

Best evolved CA

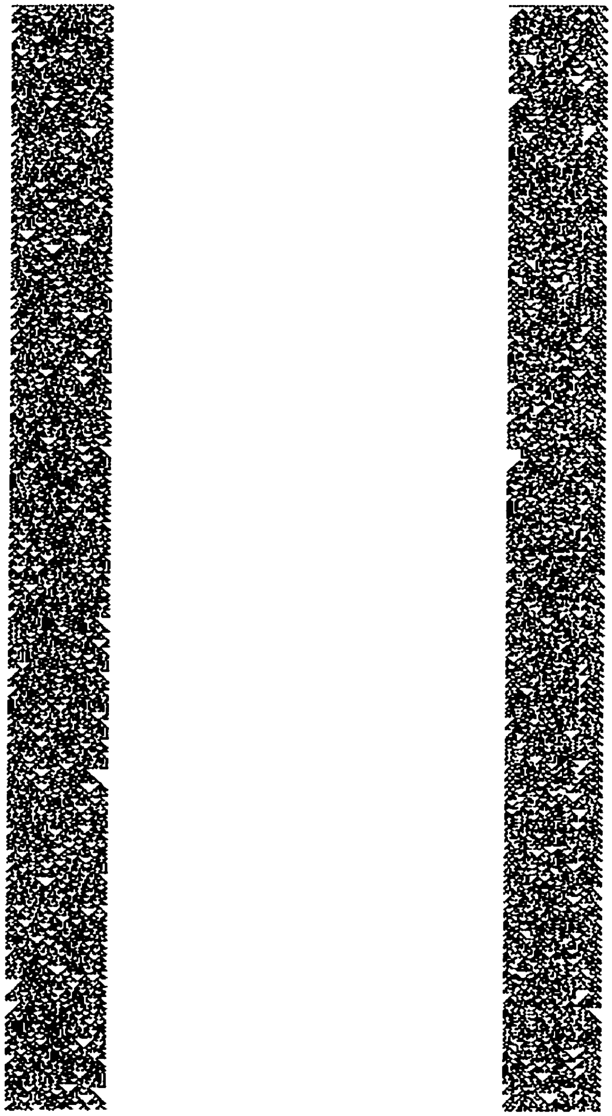
Figure B.2: CA growths for c1908



Best primitive CA

Best evolved CA

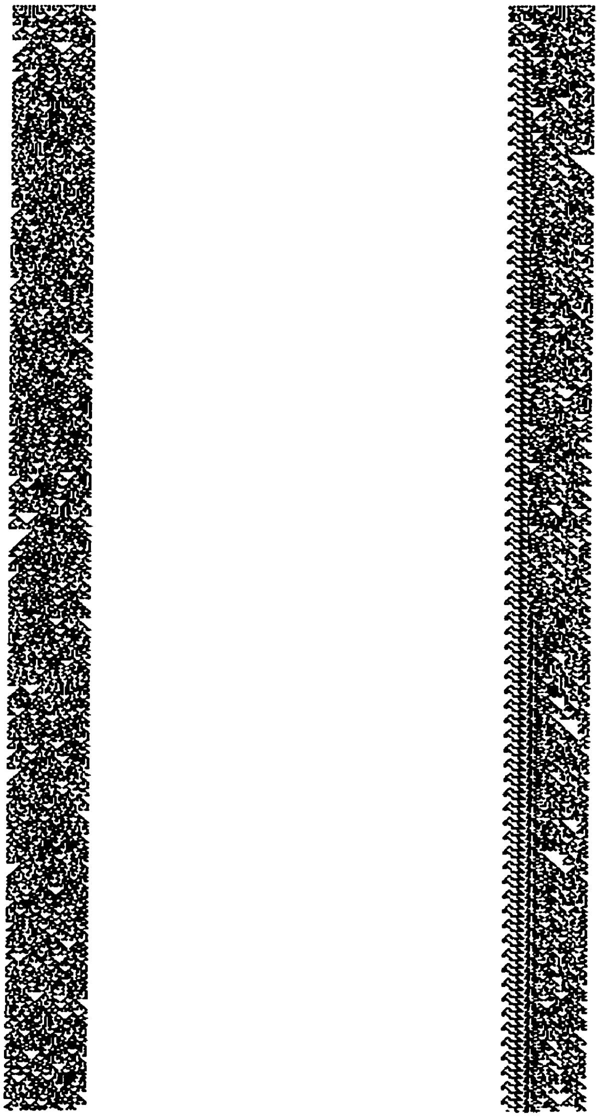
Figure B.3: CA growths for c2670



Best primitive CA

Best evolved CA

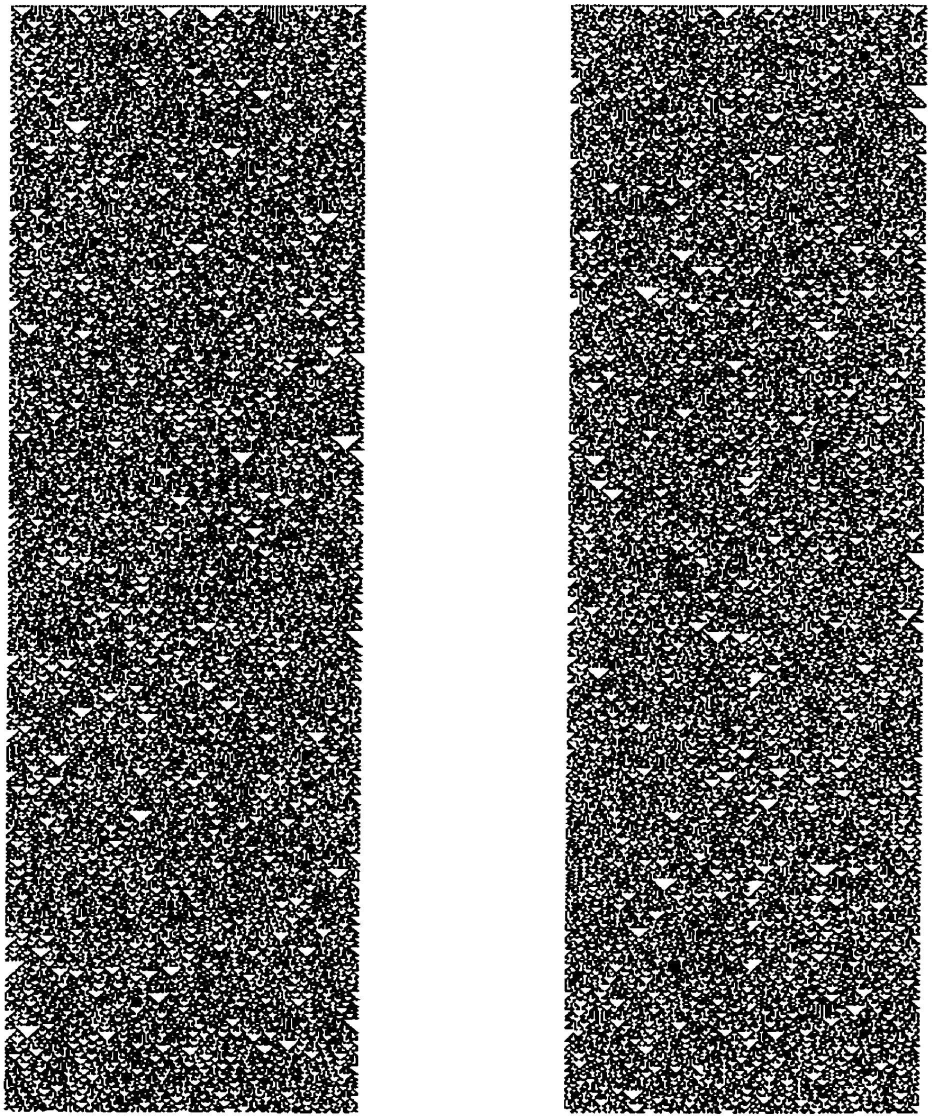
Figure B.4: CA growths for c3540



Best primitive CA

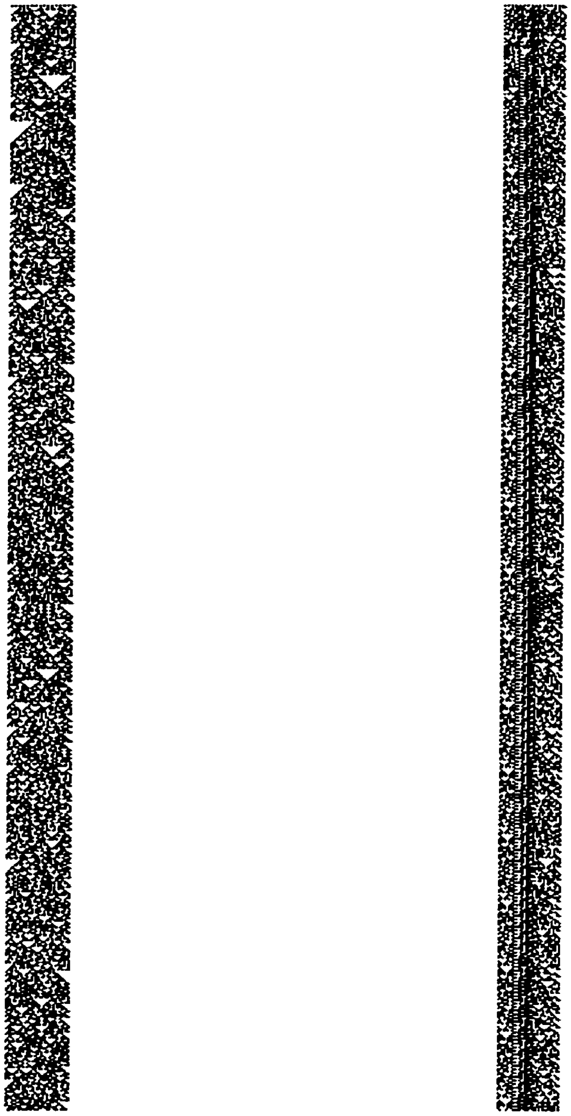
Best evolved CA

Figure B.5: CA growths for c499



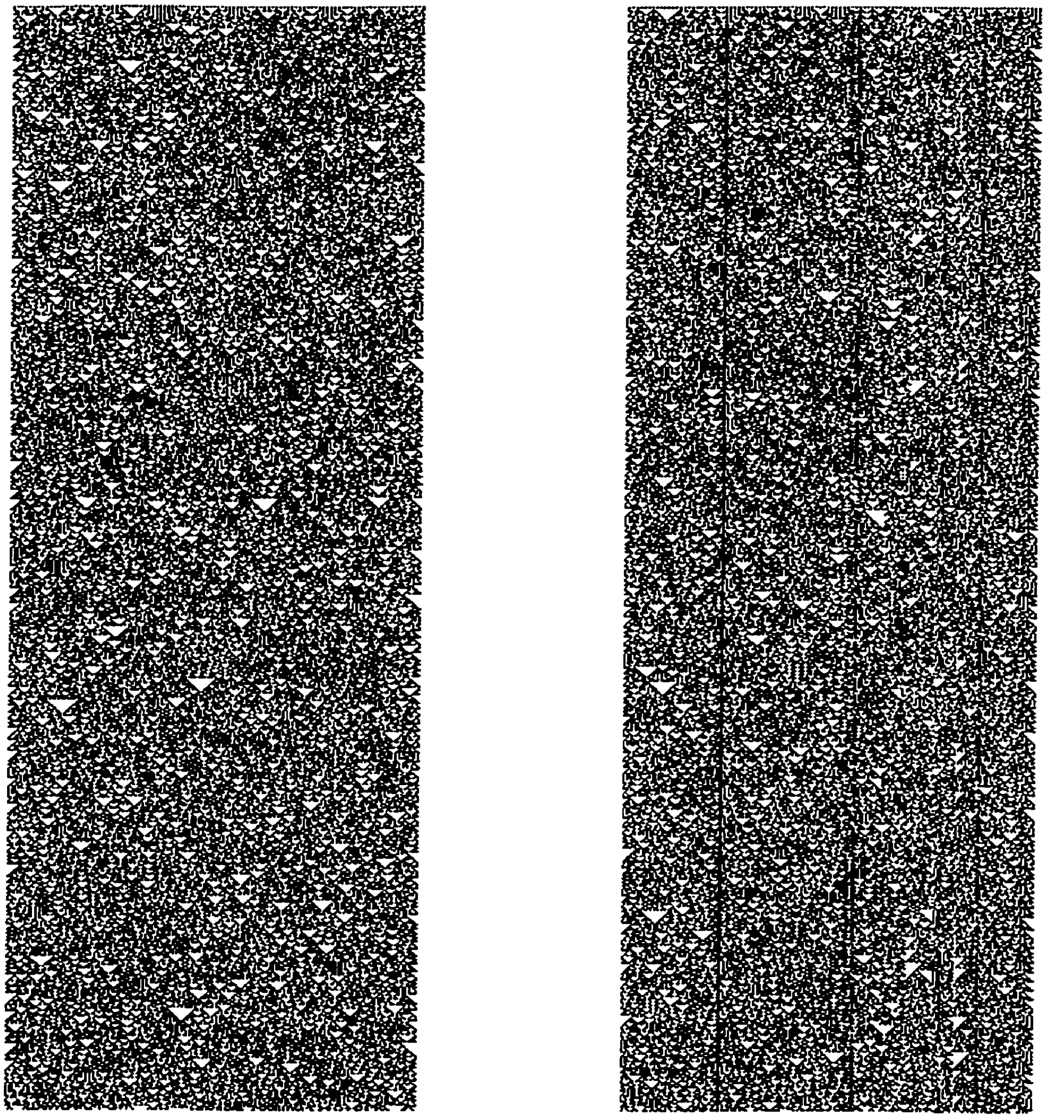
Best primitive CA **Best evolved CA**

Figure B.6: CA growths for c5315



Best primitive CA Best evolved CA

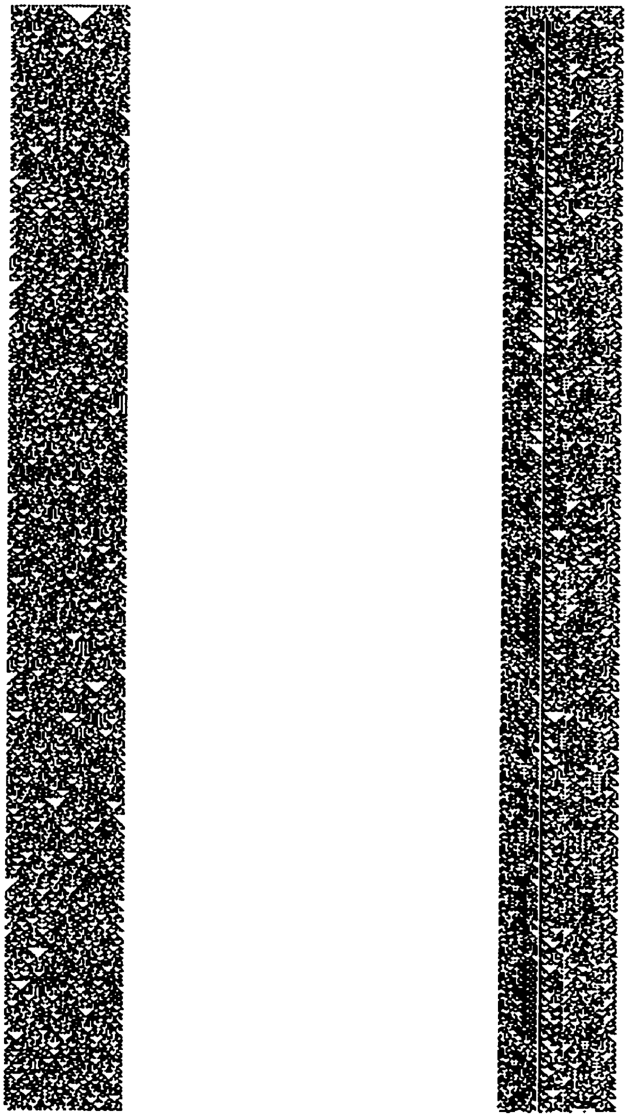
Figure B.7: CA growths for c6288



Best primitive CA

Best evolved CA

Figure B.8: CA growths for c7552



Best primitive CA

Best evolved CA

Figure B.9: CA growths for c880

APPENDIX C
libfsim User Manual

The fault simulator used in this thesis is based on the algorithm of [25]. This algorithm was implemented in a C program called `fsim` by the authors of [25]. To facilitate the massive number of fault simulations needed during the execution of CAGA, this code was modified so that the fault simulator could be accessed via function calls into a library called `libfsim`. Prior to these modifications the fault simulation was performed by invoking an external `fsim` process for each simulation and passing test vectors via temporary files. For information on using the `fsim` program (found at /home/csvlsi/sw/bin/fsim), see the document /home/csvlsi/sw/doc/fsim.doc (both on the UVic VLSI group research machine *shannon*). The remainder of this appendix explains how use `libfims` in a C or C++ program and describes its three access routines.

The libfsim code is available on *shannon*, under /home/jbingham/libfsim/. This directory contains the libfsim source, header, and object files, a makefile, and the source for two example main programs (`c_prog.c` and `cpp_prog.cc`) that run some fault simulations. To use `libfsim`, `fsim.h` must be #included in the source, and `libfsim.a` must be linked during compilation; the source may be compiled as either C or C++.

`libfsim` requires the user to have implemented a function adhering to the following prototype:

```
void getTestPatterns(int n,int a[],int batchSize)
```

`libfsim` calls `getTestPatterns()` to retrieve a batch of test vectors. These vectors are of width `n` and are returned in the array `a[]`; `batchSize` determines the number of vectors. If b is equal to `batchSize` then the b n-vectors $v^{(0)}, \ldots, v^{(b-1)}$ are stored in `a[]` as follows. The low order bit of `a[`i`]` stores the value $v_i^{(0)}$, the second lowest order bit of `a[i]` stores the value $v_i^{(1)}$, etc., for $0 \leq i < n$. Thus, when `getTestPatterns()` returns, the j^{th} lowest order bit of `a[i]` must store the bit $v_i^{(j-1)}$.

The `libfsim` initialization function is:

```
void fsim_init(char *name1, int batchSize)
```

The character pointer argument must point to a string containing the path to the circuit description file (in ISCAS85 format [67]). The second argument must be a power of 2 between 1 and 32 inclusive and tells fsim how many vectors to process in parallel; the higher this number the faster the fault simulation algorithms performs, but the resulting test lengths are rounded up to the nearest multiple of `batchSize`. This same value will always be passed to `getTestPatterns()` when called by the library.

Actual fault simulation is performed by:

```
int fsim_sim(float cov, int maxVectors)
```

`fsim_sim` takes a float in the unit interval that specifies the desired fault coverage; for 100% fault coverage this parameter should be 1.0. The second argument determines the maximum number of vectors to process before returning. The return value is the lowest multiple of `batchSize` (as passed to `fsim_init()`) greater or equal to the number of vectors needed to reach the desired fault coverage or `maxVectors`.

Finally, before subsequent calls to `fsim_sim` are made, this function must be called:

```
void fsim_reset()
```

It cleans up all internal data structures used by the fault simulator, and need not be called before the very first simulation during runtime.

Vita

Surname: Bingham Given Names: Jesse David

Place of Birth: Abbotsford, British Columbia, Canada

Educational Institutions Attended:

University of Victoria	1992-2001

Degrees Awarded:

B.Sc.	University of Victoria	1998

Honours and Awards:

NSERC PGS B	2001-
Canada Scholarship for Science and Engineering	1992-1993

Publications:

J. Bingham and M. Serra. Solving Hamiltonian Cycle on FPGA Technology via Instance to Circuit Mappings, In *Workshop on Engineering of Reconfigurable Hardware/Software Objects, PDPTA*, Las Vegas, U.S.A., June 2000.

Partial Copyright License

I hereby grand the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the library of any other university , or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Genetic Evolution of Nonlinear Cellular Automata for Built-In Self-Test of Combinational Circuits

Author: ____________________
Jesse David Bingham
April 30, 2001