Application of Genetically Engineered Finite-State-Machine Sequences to Sequential Circuit ATPG

Michael S. Hsiao, *Member, IEEE*, Elizabeth M. Rudnick, *Member, IEEE,* and Janak H. Patel, *Fellow, IEEE*

***Abstract*—New methods for fault-effect propagation and state justification that use finite-state-machine sequences are proposed for sequential circuit test generation. Distinguishing sequences are used to propagate the fault effects from the flip-flops to the primary outputs by distinguishing the faulty machine state from the fault-free machine state. Set, clear, and pseudoregister justification sequences are used for state justification via a combination of partial state justification solutions. Reengineering of existing finite-state machine sequences may be needed for specific target faults. Moreover, conflicts imposed by the use of multiple sequences may need to be resolved. Genetic-algorithm-based techniques are used to perform these tasks. Very high fault coverages have been obtained as a result of this technique.**

***Index Terms*—Automatic test generation, genetic engineering, pseudoregisters, state justification.**

I. Introduction

THE MAJORITY of the time spent by automatic test generators for sequential circuits is used to find test sequences for hard-to-detect faults. These hard faults are either hard to excite, hard to propagate, or both. Deterministic test generators have been proposed in the past [1]–[12], but they require backtracing through complex gates and flip-flops, and remodeling of such primitives is often required. Simulation-based test generators, on the other hand, avoid the complexity of backtracing by processing in the forward direction only. However, simulation-based approaches often fall short when targeting the hard faults.

Previously, homing, synchronizing, and distinguishing sequences have been used to aid the test generator in improving the fault coverage [6], [10]–[12], [26]. In [6], [10], and [12], symbolic and state-table-based techniques were used to derive these sequences in the fault-free machine. In [6], cube intersections of ON/OFF-set representations were used to derive distinguishing sequences. Binary decision diagrams (BDD's) and implicit state enumeration were used in [10] to derive synchronizing sequences. In the work by Park *et al.* [12], functional information was used to pregenerate sequences which simplified the propagation of fault effects from the flip-flops to the primary outputs, and state justification was done by using BDD's. Since these sequences are generated using the fault-free machine only, they may become invalid in a faulty machine. Homing sequences composed of specifying and distinguishing portions were used to aid ATPG in [11], but they had to be recomputed for each target fault.

The presence of a fault creates a faulty machine (circuit structure) which differs from the fault-free machine. The goal is to distinguish the faulty machine from the fault-free machine by activating the target fault and propagating its effects to the primary outputs. With the test generation process divided into fault activation and fault propagation phases, the principal approach taken in our work is to use finite-state-machine sequences in as many places as possible to reduce the work of rediscovering such sequences. The finite-state-machine sequences used in this work encompass distinguishing sequences, set/clear sequences, and justification sequences, all of which will be explained in the subsequent sections. No state diagrams are needed in this work.

Several questions remain. Since there are many finite-state-machine sequences for any large machine, what finite-state-machine sequences should be generated and stored? Sequences derived for a fault-free machine may not be valid for a faulty machine, or they may be valid for some faulty machines, but not for other faulty machines; how can invalid sequences be used to fit the specific needs of the target fault? Moreover, A finite-state-machine sequence may not always exist; can *partial* sequences be used? Finally, we cannot indiscriminately generate large numbers of sequences because potential problems of excessive storage and execution may result.

In this work, several classes of finite-state-machine sequences are generated statically for the fault-free machine, and also captured dynamically for the fault-free and faulty machines during the test generation process. The difficulty of deriving a sequence is taken into account at run time in the computation of flip-flop controllability and observability. These measures are much more accurate than the conventional controllability and observability metrics. They help to guide the test generator much more effectively, e.g., in propagating fault effects to flip-flops that are easy to observe. Modifications to the finite-state-machine sequences may be needed before they can be applied to fault propagation or state justification, and several useful sequences may exist for a particular problem. Genetic algorithms (GA's) [14] have been demonstrated to be effective in combining useful portions of several candidate solutions to a given problem. Therefore, we have chosen to use genetic algorithms in this work, both to

Manuscript received August 30, 1996. This work was supported in part by the Semiconductor Research Corporation under Contract SRC 95-DP-109, in part by DARPA under Contract DABT63-95-C-0069, and by Hewlett-Packard under an equipment grant. This paper was recommended by Associate Editor S. Reddy.

M. S. Hsiao is with the Department of Electrical and Computer Engineering, Rutgers, the State University of New Jersey, Piscataway, NJ 08854-8058 USA.

E. M. Rudnick and J. H. Patel are with the Center for Reliable and High-Performance Computing and Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801 USA.

Publisher Item Identifier S 0278-0070(98)03084-X.

0278–0070/98$10.00 © 1998 IEEE

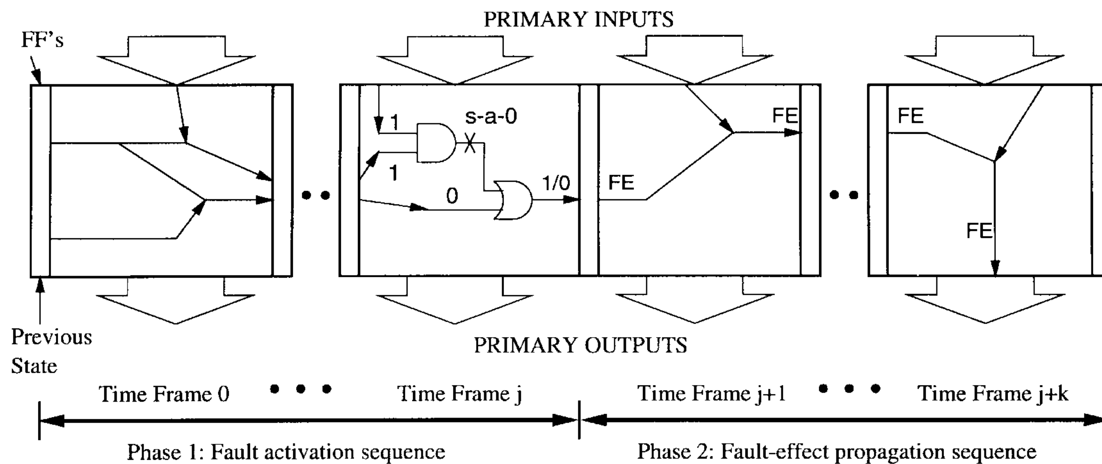

Fig. 1. Two-phase test generation strategy.

derive and manipulate finite-state-machine sequences and in the overall test generation process.

Several approaches to test generation using genetic algorithms have been proposed in the past [15]–[27]. Fitness functions were used to guide the GA in finding a test vector or sequence that maximizes given objectives for a single fault or group of faults. In GATEST [27] and ALT-TEST [25], the fitness functions were biased toward maximizing the number of faults detected and the number of fault effects propagated to flip-flops; increasing the circuit activity was a major objective in CRIS [15] and GATTO [20]. Maximizing the propagation of fault effects to flip-flops and increasing circuit activity have been shown to increase the probability of detecting faults at the primary outputs. Although the fault detection probability improves, activating a hard fault and propagating fault effects from flip-flops to a primary output remain difficult problems. Furthermore, propagation of fault effects was done indiscriminately (i.e., the GA-based test generator does not attempt to drive the fault effects to more observable flip-flops), resulting in much wasted effort. Increasing circuit activity may be ineffective in activating a given fault, and propagation of fault effects from certain flip-flops may not be possible. The hard-to-activate faults in some circuits may require specific states and justification sequences in order for them to be activated, and the previous GA-based test generators have failed to drive the circuit to these specific states for fault excitation, resulting in low fault coverages. For instance, GA-based test generators have obtained low fault coverages for ISCAS89 circuits [13] $s820$, $s832$, $s1488$, and $s1494$ due to frequently deep and specific sequences necessary to excite the faults, but deterministic test generators have been quite successful in generating tests for them. The differences in fault coverages were as high as 30% for such circuits. Even when a GA was specifically targeted at state justification, the simple fitness function used was inadequate for these circuits [22], [27]. Utilizing finite-state-machine information allows us to overcome the limitations of the previous genetic approaches, closing the 30% gap in fault coverage for these circuits; for other circuits, higher fault coverages than ever before have been obtained. The main difference between our present work and previously proposed GA-based techniques is that we utilize problem-specific knowledge during test generation and explore genetic engineering of sequences that exploit such knowledge to significantly improve the quality of test generation, both in terms of fault coverage and execution time.

The remainder of the paper is organized as follows. Section II gives an overview of this work; Section III briefly describes the genetic algorithm framework used in the test generator; Section IV gives details about the derivation and application of finite-state-machine sequences; Section V discusses the test generation algorithm, including fault activation in the single-time-frame mode, selection of target faults, and fitness evaluation; experimental results are given in Section VI; and Section VII concludes the paper.

II. Overview

Our test generation strategy uses several passes through the fault list, with faults targeted individually in two phases. The two-phase strategy is illustrated in Fig. 1. The first phase focuses on activating the target fault, while the second phase tries to propagate the fault effects (FE's) from the flip-flops (FF's) to the primary outputs. A target fault is selected from the fault list at the beginning of the fault activation phase, and an attempt is made to derive a sequence that excites the fault and propagates the fault effects to a primary output or to the flip-flops. Once the fault is activated, the faults effects are propagated from the flip-flops to the primary outputs in the second phase with the assistance of distinguishing sequences. The target fault is detected at the primary outputs when the faulty machine state is distinguished from the fault-free machine state. The distinguishing sequences corresponding to the flip-flops reached by the fault effects are used to *engineer* a valid sequence which successfully distinguishes the faulty state from the fault-free state.

During the fault activation phase, single-time-frame activation is entered if no single activation sequence can be found directly from the state in which the previous sequence left off. The state derived by single-time-frame activation is relaxed and the relaxed state is then justified to complete the fault activation, as illustrated in Fig. 2.

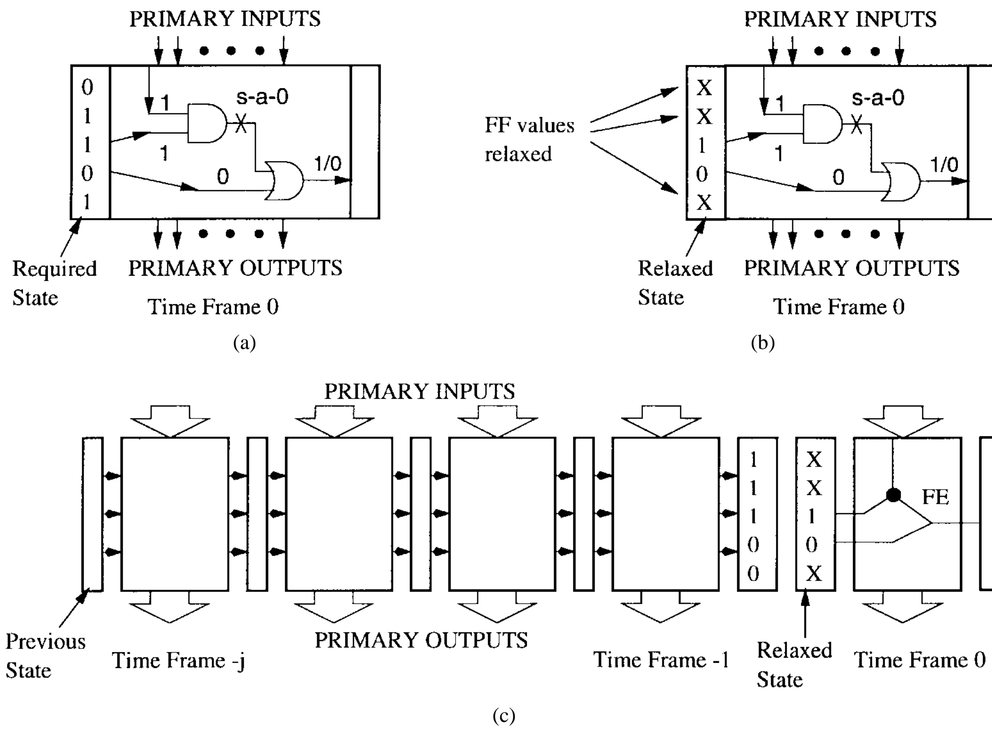

Fig. 2. State justification process.

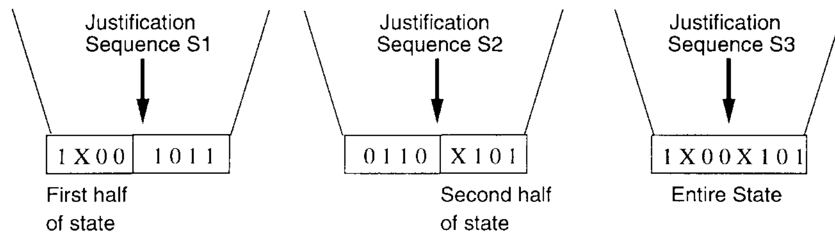

Fig. 3. Justification of partial states.

Flip-flops that are assigned the don't care value of $\boldsymbol{X}$ are considered to be unspecified. If state S_i is partially specified, then an exhaustive set of states can be obtained by enumerating don't care (unassigned) values of S_i. For example, state $\boldsymbol{X01}$ is partially specified, and it represents two states $\mathbf{001}$ and $\mathbf{101}$.

Definition 1: State S_j *covers* state S_i if the group of states represented by S_i is a subset of states represented by S_j.

Definition 2: A flip-flop is *relaxed* if its value is changed from 0 or 1 to a don't care value X.

In the single-time-frame fault activation, the aim is to find a test vector, composed of primary input and flip-flop values, that can activate the target fault in a single time frame. Once a vector (primary input and flip-flop values) is successfully derived, the state (flip-flop values) is first relaxed to one that has as many don't-care values (X) as possible, but is still capable of activating the target fault. State relaxation, which was first proposed by Niermann and Patel [7], [8], improves the success rate of the state-justification process which immediately follows. Next, finite-state-machine sequences for setting and clearing individual flip-flops and for justifying the values of groups of flip-flops (*pseudoregisters*) are used as seeds in the GA to aid state justification, and an attempt is made to justify the required state by evolving the GA population over several generations. In the process, the candidate sequences contained in the GA population are simulated, starting from the last state reached after the previous test sequences have been applied. The objective is to engineer a sequence that justifies the required state. Note that the state justified only needs to be covered by the relaxed state. Consider the situation shown in Fig. 3 in which an attempt is being made to justify state $\mathbf{1}\boldsymbol{X}\mathbf{00}\boldsymbol{X}\mathbf{101}$. Sequence S_1 successfully justifies the first half of the state, but fails to justify the second half; on the other hand, sequence S_2 justifies only the second half of the state. These two sequences S_1 and S_2 may provide important information in evolving the complete solution, S_3, which justifies the complete state. The use of the information provided by S_1 and S_2 in deriving sequence S_3 is explained later in this paper.

If a sequence is found that justifies the required state, the sequence is added to the test set, a fault simulator is invoked to remove any faults detected by the sequence, and the test generator proceeds to the fault propagation phase. Otherwise, test generation for the current target fault is aborted, and processing continues for the next fault in the fault list. In the

fault propagation phase, the GA is seeded with distinguishing sequences for the flip-flops to which fault effects have propagated. The distinguishing sequences used as seeds may have been derived for the fault-free machine or for a different faulty machine starting from a different state, and thus they may not be directly applicable to the current situation. Therefore, the GA population may have to evolve over several generations before an accurate distinguishing sequence is derived. Flip-flops that do not have distinguishing sequences are identified during the test generation process, and propagating fault effects to these hard-to-observe flip-flops is avoided. If a sequence that drives the fault effects to the primary outputs is successfully obtained, the sequence is added to the test set, and a fault simulator is invoked to remove any additional faults detected by the sequence. Test generation then continues with the next fault in the fault list.

A. Why Genetic Engineering?

Finite-state-machine sequences derived for the fault-free machine may not be valid for the faulty machines. Also, a sequence that is valid for one faulty machine may be invalid for a different faulty machine. A sequence S_p derived previously may be similar to the one needed for the current target fault; thus, reengineering S_p may provide a sequence that fits the specific needs of the current situation. Genetic algorithms are able to genetically reengineer valid sequences to fit those needs. Furthermore, conflicts encountered during state justification have to be resolved. During state justification, a sequence that correctly justifies one portion of the required state may simultaneously set an incorrect value on other portion(s), resulting in conflicts. Nevertheless, the justification sequences for each partial state may be viewed as partial solutions in finding the justification sequence for the complete state. Because important information about the assignment of primary inputs for justifying a specific part of a state is intrinsically implied by each sequence, this information may be useful in searching for the complete justification sequence. Stated differently, each partial solution is a chromosome in the evolutionary process; the desired solution may be evolved from the population of chromosomes with appropriate fitness functions. The GA is capable of combining several partial solutions, under arbitrary constraints, to form a complete solution to a problem via the evolutionary processes.

III. Genetic Algorithms

The GA framework used in our work is similar to the simple GA described by Goldberg [14]. The GA contains a population of *strings*, also called *chromosomes* or *individuals*, in which each individual represents a sequence of test vectors. A binary coding is used, and therefore, each character in a string represents the logic value to be applied to a primary input in a particular time frame. The population size used is a function of the string length, which depends on both the number of primary inputs and the test sequence length. Larger populations are needed to accommodate longer individual test sequences in order to maintain diversity. The test sequence length is a function of the structural sequential depth, where sequential depth is defined as the minimum number of flip-flops in a path between the primary inputs and the furthest gate. The population size is set equal to $4 \times \sqrt{\text{sequence length}}$ when the number of primary inputs is fewer than 16 and $16 \times \sqrt{\text{sequence length}}$ when the number of primary inputs is greater than or equal to 16. During the first stage of test generation in the first pass through the fault list, the sequence length is set equal to the structural sequential depth. The sequence length is doubled in the second stage of test generation and doubled again in the third stage since harder faults may require longer sequences for activation and/or propagation.

Each individual has an associated *fitness*, which measures the test sequence quality in terms of fault detection, dynamic controllability and observability measures, and other factors. The fitness function used in this work depends on the phase of test generation, and will be explained in a later section. The population is initialized with random strings, and if any appropriate finite-state-machine sequences exist for the current target fault, they are used as seeds as well. A fault simulator is used to compute the fitness of each individual. Then the evolutionary processes of *selection*, *crossover*, and *mutation* are used to generate an entirely new population from the existing population. Evolution from one generation to the next is continued until a sequence is found to activate the target fault or propagate its effects to the primary outputs or until a maximum number of generations is reached. To generate a new population from the existing one, two individuals are selected, with selection biased toward more highly fit individuals. The two individuals are crossed to create two entirely new individuals, and each character in a new string is mutated with some small mutation probability. A mutation probability of 0.01 is used in this work, and since a binary coding is used, mutation is done by simply flipping the bit. The two new individuals are then placed in the new population, and this process continues until the new generation is entirely filled. At this point, the previous generation can be discarded. In our work, we use tournament selection without replacement and uniform crossover. In *tournament selection without replacement*, two individuals are randomly chosen and removed from the population, and the best is selected; the two individuals are not replaced into the original (parent) population until all other individuals have also been removed. Thus, it takes two passes through the parent population to completely fill the new population. In *uniform crossover*, bits from the two parents are swapped with probability 1/2 at each string position in generating the two offspring. A crossover probability of 1 is used, i.e., the two parents are always crossed in generating the two offspring. Because selection is biased toward more highly fit individuals, the average fitness is expected to increase from one generation to the next. However, the best individual may appear in any generation.

IV. Finite-State-Machine Sequences

Distinguishing, set, clear, and pseudoregister justification sequences are the finite-state-machine sequences involved in this work. They are used as seeds for the GA during the cor-

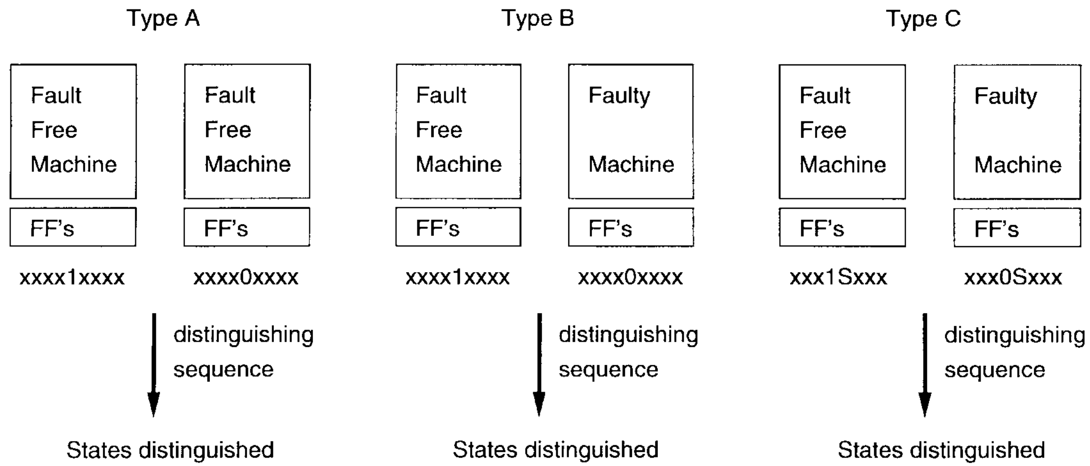

Fig. 4. Types of distinguishing sequences.

responding fault-propagation and state-justification phases. If the seeds are valid for a given situation, no further processing is required; otherwise, we attempt to genetically engineer valid sequences from the seeds. Each class of finite-state-machine sequences is described in this section.

A. Distinguishing Sequences

Distinguishing sequences are used to propagate fault effects from the flip-flops to the primary outputs. We define three different types of distinguishing sequences, as illustrated in Fig. 4.

Definition 3: A *distinguishing sequence of type A* for flip-flop i is a sequence that produces two distinct output responses when applied to the fault-free machine for two initial states that differ in the ith position and are independent of all other flip-flop values.

Definition 4: A *distinguishing sequence of type B* for flip-flop i is a sequence that produces two distinct output responses when applied to the fault-free machine with flip-flop $i = 0$ (or 1) and applied to the faulty machine with flip-flop $i = 1$ (or 0), independent of the values of all other flip-flops.

Definition 5: A *distinguishing sequence of type C* for flip-flop i is a sequence that produces two distinct output responses when applied to the fault-free machine with flip-flop $i = 0$ (or 1) and applied to the faulty machine with flip-flop $i = 1$ (or 0) while one or more of the other flip-flops have specific logic values.

Type A distinguishing sequences distinguish two states of the *same* machine, while types B and C distinguishing sequences distinguish states on two *different* machines. A distinguishing sequence of type C is similar to a type B sequence, except that a partial state (i.e., a subset of flip-flops) is assigned to a specific value. The value $\boldsymbol{x}$ in the state in Fig. 4 denotes an unknown, or more precisely, a *don't care* value, and $\boldsymbol{S}$ in a state represents a string of specific values (e.g., $\mathbf{1}$ or $\mathbf{0}$). Note that, because the distinguishing sequences of type C depend on a partial state S of the machine, they may not necessarily be applicable directly from any starting state.

A distinguishing sequence associated with a flip-flop can propagate a fault effect from the given flip-flop to the primary outputs. The most general case of generating a type A distinguishing sequence is as follows. A fault effect $\mathbf{D} = (\mathbf{1/0})$ is placed at the output of a flip-flop, while all other flip-flops in the circuit are set to unknown values. Any sequence that makes the $\boldsymbol{D}$ observable at the primary outputs is a distinguishing sequence of type A for the given flip-flop. This type of sequence is able to distinguish $2^{2(N-1)}$ pairs of states in the fault-free machine, where N is the total number of flip-flops in the circuit. In most circuits, however, the number of type A distinguishing sequences is small. In addition, this type of sequence may not successfully distinguish the states in the fault-free machine from those in the faulty machine, which is required when generating a test sequence for a target fault. Fortunately, fault effects are often propagated to many flip-flops, and flip-flop values do not typically remain unknown during the course of test generation. This gives rise to distinguishing sequences of types B and C.

A distinguishing sequence of type B or C for a flip-flop is specific to a target faulty machine (machine resulting from the presence of a target fault). Given a fault-free machine $\boldsymbol{G}$ and a corresponding faulty machine $\boldsymbol{F}$, a type B sequence would be able to distinguish $2^{2(N-1)}$ pairs of states between machines $\boldsymbol{G}$ and $\boldsymbol{F}$. On the other hand, a type C sequence would distinguish only $2^{2(N-M-1)}$ pairs of states, where M denotes the number of flip-flops with specified values. It should be noted that distinguishing sequences of types A and B are more powerful than those of type C since more pairs of states differing in a given flip-flop i can be distinguished by the sequences of types A and B for flip-flop i.

While distinguishing sequences of type A are capable of distinguishing two different fault-free states ST_1 and ST_2, they may not necessarily be able to distinguish the state ST_1 in fault-free machine $\boldsymbol{G}$ from the state ST_2 in faulty machine $\boldsymbol{F}$. Nevertheless, a type A sequence may be very similar to a sequence that is able to distinguish the two states in machines $\boldsymbol{G}$ and $\boldsymbol{F}$. Therefore, it is helpful to seed the GA with distinguishing sequences of type A in searching for a successful distinguishing sequence. Carrying this idea further, when a distinguishing sequence of type B or C is found for a certain fault $\boldsymbol{f_1}$, that sequence may not be directly

applicable under a different fault f_2. A similar argument applies in this case: the previously derived distinguishing sequence may be used as a seed for the GA to help find a valid sequence. The sequences generated in [12] are similar to the type A distinguishing sequences, except that they were generated using BDD's; no pruning of sequences was done, and dynamically generated sequences targeting specific faults were not used. When the sequences fail to distinguish the states for specific faulty machines, no procedure was given to modify the sequences. In contrast, we use a variety of distinguishing sequences and modify them to get valid sequences for each fault.

A distinguishing sequence of type C requires a specific partial state in order to successfully propagate a $\boldsymbol{D}$ from a given flip-flop to the primary outputs. Under this restriction, many flip-flops often have distinguishing sequences of type C when the more powerful sequences of types A and B do not exist. In many cases, a type C distinguishing sequence works as well as a type A or B sequence because the state reached at the end of fault activation is often contained within the required set of states for the distinguishing sequence.

Storing the type C distinguishing sequences may pose a problem, however. Including the specific values of the required partial state S for the sequences may adversely affect both the execution time and memory storage. Furthermore, when a distinguishing sequence of type C is derived dynamically during test generation, it is difficult to identify the flip-flops which require specific values. Thus, instead of storing the values for various subsets of flip-flops, a distinguishing power is associated with each distinguishing sequence to indicate how well the sequence distinguishes between states. As a consequence, the distinguishing power also indirectly indicates how well a fault effect will propagate from the corresponding flip-flop. Although the state-containment information is missing for these distinguishing sequences, they are still useful as seeds for the GA to evolve an effective distinguishing sequence. The distinguishing power of every corresponding sequence is updated for each successful and unsuccessful GA application.

All three types of distinguishing sequence can be generated by the GA. Before test generation begins, the GA is set in a preprocessing stage to compute any distinguishing sequences of type A. The sequence length of each individual in the GA population is set equal to the length used in the third stage of test generation, i.e., four times the sequential depth of the circuit. The actual length of a type A sequence depends on the time frame in which the states are distinguished. Type B sequences could be generated in a similar manner, given information about a particular faulty circuit, but it is likely to add to execution time without adding appreciable value. During the test generation process, derivation and pruning of distinguishing sequences of type C are done concurrently and adaptively. The GA is initialized with random sequences, and any distinguishing sequences of the flip-flops to which fault-effects have propagated are used as seeds in place of some of the random sequences in the fault propagation phase. If the seeds are longer than the length of a GA individual, the extra vectors at the end of the seed are truncated. If the seeds are shorter, they are padded at the end with extra random vectors. Sequences of type C may themselves be shorter than the length of a GA individual, depending on the time frame in which the states are distinguished.

For flip-flops that do not have an associated distinguishing sequence of any type, an observability value is used to indicate how observable the flip-flop is in the GA framework. Initially, all flip-flops in the circuit are set to a certain observability value. As time progresses, these observabilities for the flip-flops will decrease if no distinguishing sequence can be obtained for them. A low observability value indicates that it is difficult to generate a distinguishing sequence for that flip-flop. This measure of observability is much more accurate than conventional observability values, and it enables the test generator to avoid the propagation of fault effects to hard-to-observe flip-flops.

B. Set and Clear Sequences

Flip-flop set and clear sequences are used in engineering a sequence to justify a desired state. We define flip-flop set and clear sequences as follows:

Definition 6: A *flip-flop set sequence* is a sequence that sets the associated flip-flop to a logic value of one.

Definition 7: A *flip-flop clear sequence* is a sequence that clears the associated flip-flop to a logic value of zero.

For each flip-flop that requires a specific value, the corresponding set or clear sequence is used as a seed in the GA. A set of flip-flop set and clear sequences is pregenerated prior to test generation. Similar to distinguishing power, an associated setting (clearing) *power* is maintained which indicates the sequence's ability to set (clear) the given flip-flop starting from an unknown starting state. When a flip-flop set (clear) sequence is obtained, it is given a minimal power. As test generation progresses, the associated powers of these sequences are dynamically updated, and sequences having low powers are pruned. New flip-flop set and clear sequences may be dynamically generated and added to the set during the course of test generation.

A flip-flop set (clear) sequence associated with a flip-flop is intended to set (clear) the flip-flop starting from any given state. Any sequence that can set (clear) a given flip-flop i starting from an unknown state is a candidate set (clear) sequence for the ith flip-flop. Such sequences are called *type A* sequences. Type A sequences are generated in a preprocessing step prior to test generation with the sequence length set to four times the sequential depth of the circuit, but the actual length of a type A sequence depends on the time frame in which the flip-flop values are set or cleared. Although a large percentage of flip-flops have type A set/clear sequences, a small number of flip-flops remain hard to control. These flip-flops without type A set/clear sequences may have type B sequences, which are sequences dynamically generated for the fault-free machine during test generation. Type B sequences may require specific starting partial states. As the state becomes defined during the course of test generation, it may be easier to generate type B set/clear sequences. The length of a type B sequence depends on the time frame in which the flip-flop values are set or cleared. It should be noted that type B set/clear sequences are

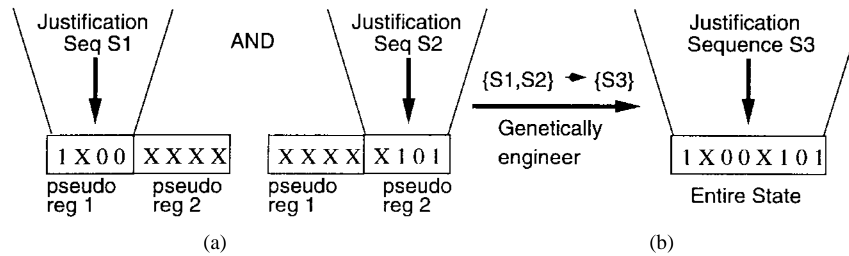

Fig. 5. Genetic engineering of state justification sequence.

not guaranteed to be successful in setting/clearing a given flip-flop from any starting state. Thus, type A set/clear sequences are more powerful than type B sequences. Both sequences of types A and B may become invalid in the presence of a fault. Furthermore, conflicting values may be encountered when applying the set and clear sequences for different flip-flops simultaneously. Conflicts can sometimes be resolved by the GA. However, more powerful sequences may be needed to reduce the frequency of conflicts.

C. Pseudoregister Justification Sequences

Sequences that are able to justify multiple flip-flop values simultaneously are more powerful than single flip-flop set and clear sequences. We partition the flip-flops into several groups, called *pseudoregisters*, and the justification sequences for pseudo-register states are defined as follows.

Definition 8: A *pseudoregister justification sequence* is a sequence that is able to justify the required flip-flop values for a particular pseudoregister.

Thus, set and clear sequences are a special case of pseudoregister justification sequences having a size of 1. For each pseudoregister that requires a specific value, any corresponding pseudoregister justification sequences are used as seeds in the GA. The goal is to genetically combine the pseudoregister justification sequences to engineer the desired solution for the complete state, as shown in Fig. 5.

A pseudoregister justification sequence S_i^t is a sequence capable of setting pseudoregister i to a specific state t. Justification sequences of types A and B are generated as for the set and clear sequences. A justification sequence of type A is a sequence that is able to justify a desired pseudoregister state starting from an unknown state. Type A sequences are generated statically in a preprocessing step; the GA sequence length used is four times the sequential depth of the circuit, but the actual length of a type A sequence depends on the time frame in which the pseudoregister state is justified. Type B sequences are generated dynamically for the fault-free machine during test generation, and may require specific partial starting states. The length of a type B sequence depends on the time frame in which the pseudoregister state is justified. Justification powers are maintained for each sequence and dynamically updated during the course of test generation. Depending on the sizes of the pseudoregisters, the number of type A pseudoregister justification sequences varies. Larger pseudoregisters are able to hold more states, but fewer of these states are likely to have type A sequences.

In terms of storage requirements, symbolic techniques for state justification introduced in the past are often memory inefficient for large circuits; moreover, storing justification information for the entire state space in the circuit would require a huge storage space. For example, in a circuit with N flip-flops, as many as 2^N states may have to be stored, along with justification sequences for each state. On the other hand, if the N flip-flops are partitioned into pseudoregisters, which are groups of k flip-flops representing portions of the entire state, only $(N/k)2^k$ pseudoregister states and sequences have to be stored. This is of linear order $(O(N))$ when k is small. In this work, typical values of k are less than six.

V. Test Generation Algorithm

The test generator is comprised of three stages; each stage involves several passes through the fault list, and a stage is finished when little or no improvement in fault coverage is achieved. Faults are targeted individually within each stage, and GA's are used to activate a fault and propagate the fault effects to the primary outputs. Different test sequence lengths for individuals are used in the GA population for the different stages. Since the time required for the fitness evaluation is directly proportional to the test sequence length, the shorter sequences are tried first, and faults are removed from the fault list once they are detected.

Test generation for a target fault is divided into *fault activation* and *fault propagation* phases. Fault activation excites the fault and propagates its effects to a primary output or at least one flip-flop. Fault propagation propagates the fault effects from one or more flip-flops to a primary output, possibly through several time frames, with the assistance of distinguishing sequences. Single-time-frame mode and state justification are used for fault activation when no sequence capable of exciting the target fault and propagating its effects to one or more flip-flops is generated. Fig. 6 displays the pseudocode for the two-phase test generation algorithm within each stage of the test generator.

Before test generation begins, the GA is used in a preprocessing stage to compute any type A finite-state-machine sequences. During the test generation process, derivation and pruning of sequences are done concurrently and adaptively. The GA is initialized with random sequences, and any relevant finite-state-machine sequences are used as seeds in place of some of the random sequences in the fault activation and fault propagation phases. The sequences and their corresponding power indexes are pruned after each successful and unsuccessful application.

```
Generate sets of type-A finite-state-machine sequences
For all undetected faults do
    Pick a target fault
    /* fault activation phase */
    Call GA to generate a sequence that activates the target fault
    If no sequence generated then
        /* single-time-frame activation */
        Generate a vector in single-time-frame mode
        If a vector found
            Relax the state obtained
            /* state justification */
            Seed GA with appropriate set, clear, or pseudo-register justification sequences
            Call GA to generate a justification sequence
            Update the justification powers of the required set, clear, and pseudo-register justification
                sequences and the controllabilities of the FF's
    If an activation sequence is found then
        Drop all faults detected by the activation sequence
        /* fault propagation phase */
        If target fault not detected then
            Seed GA with distinguishing sequences corresponding to FF's with fault effects
            Call GA to propagate fault effects to a PO
            If sequence derived by GA is successful then
                Drop all faults detected by the sequence
            Update distinguishing powers of the sequences and observabilities of the FF's
```

Fig. 6. Test generation algorithm.

Because the number of finite-state-machine sequences grows over time, the lists of sequences are pruned adaptively to increase the power and accuracy of these sequences in justifying and distinguishing the states. To improve state justification and fault detection, flip-flops which are hard to control or hard to observe are identified dynamically during the process; as a result, justification of difficult states in the first phase and propagation of fault effects to the hard-to-observe flip-flops in the first and second phases may be avoided. All faults are targeted until little or no more improvement is made.

A. The Single-Time-Frame Mode

When activation of the target fault is difficult and the GA fails to generate an activation sequence in the first phase, a second attempt is made to activate the fault in a single time frame. The aim here is to engineer a vector, composed of primary input and flip-flop values, capable of exciting the target fault and propagating its effects to at least one flip-flop in a single time frame. The target fault is activated when the fault is excited, and its effects propagate to one or more flip-flops or primary outputs in the single time frame. Initially, the GA is seeded with random vectors, and the evolution process is continued until a vector is found or a maximum of eight generations is reached. The fitness function guides the search by favoring individuals that activate the target fault. Because an unjustifiable state is undesirable, the fitness function uses the dynamic controllability values of the flip-flops to guide the search toward more easily justifiable states.

With these measures as guides to search for a solution, there is still no guarantee that the resulting state is indeed justifiable. Therefore, a further relaxation step is performed. Let S denote a state to be justified and s_i the ith flip-flop in state S. The state S' is obtained by inverting the value of s_i. If the target fault is still activated by S', then the ith flip-flop can be relaxed to the unknown value X. This implies that activation of the target fault does not depend on the assignment of s_i. The order in which the flip-flops are relaxed is determined in a two-level greedy fashion: first, the order of the pseudoregisters is determined from the least controllable to the most controllable pseudoregister state; then, the order of the flip-flops within the pseudoregister is determined from the least controllable to the most controllable flip-flop.

When state relaxation is finished, the relaxed state has to be justified. The GA is seeded with the set, clear, and pseudoregister justification sequences corresponding to the state to be justified. Pseudoregister justification sequences are selected such that the corresponding pseudoregister states they justify are *covered* by the desired relaxed state. The remaining individuals, if any, are seeded with random sequences. The GA begins its search by genetically reengineering and combining the partial solutions to form the complete solution that justifies the state with fitness functions that guide the GA to a sequence that successfully justifies the relaxed state.

B. State Justification Phase

The seeded sequences aim to justify the relaxed state. Up to five sequences are seeded for each pseudoregister state. The set, clear, and pseudoregister justification sequences may vary in length. Type B sequences may be shorter than the length of an individual in the GA population, and type A sequences may be shorter or longer. If a sequence is too long, the extra vectors from the beginning of the sequence are removed. The vectors at the end of the sequence may still be effective in achieving the desired state. The situation is different for sequences that are too short. Unlike distinguishing sequences, where the time frame in which the fault effects propagate to the primary

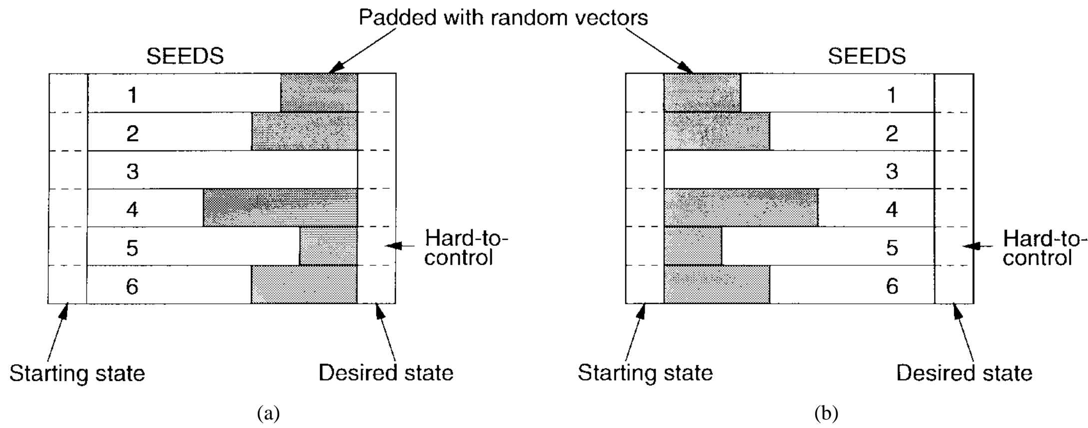

Fig. 7. Two methods of seeding the justification sequences.

outputs is not an important issue, the partial justification sequences have to arrive at their target states simultaneously. This imposes several constraints on the problem. Two possible ways of aligning the sequences corresponding to the target pseudoregister states are illustrated in Fig. 7. The first method left aligns all state justification seeds beginning with the state in which the previous sequence left off, as shown in Fig. 7(a). The second method right aligns the seeds ending at the time frame corresponding to the desired state, as shown in Fig. 7(b). Any missing vectors are padded with random vectors. The length of the state justification sequence derived may be shorter than the length of an individual in the GA population. Simulation is performed for each candidate sequence, and the state reached after each vector is compared against the desired state. Once the desired state is reached, any extra vectors are truncated. While the second method may appear to have a higher chance of success in justifying the desired state because all seeds end at the same time frame, the dominant factor in deriving the desired justification sequence is the ability to justify the hard-to-control portions of the state. The right-alignment scheme does not leave room for correction if the desired state is not justified at the end of the sequence length. On the other hand, left aligning the vectors allows the GA to truncate the sequence to the length of the seed that corresponds to the hardest to control pseudoregister. Combining partial solutions of different lengths to form a complete solution is a very difficult problem; neither a left nor a right alignment scheme guarantees convergence to a sequence that justifies the desired state. However, experiments conducted to compare the two alignment approaches confirmed that left alignment is more likely to derive the justifying sequence in most cases. Therefore, the left-alignment method is used.

The GA evolves the seeded sequences over several generations to find a sequence that justifies the desired relaxed state. The fitness function is biased toward favoring justification of harder to control pseudoregister states. As a result, the hard-to-control portions of the state will be justified first.

C. Secondary Effect of the Power Indicators

The different power indicators associated with the finite-state-machine sequences are not only able to assist in justifying or distinguishing states, but are also useful as controllability and observability measures to guide the test generation process.

Let C_i^t denote the controllability of pseudoregister i for state t. A higher C_i^t value indicates that pseudoregister i is more easily controlled to state t. Initially, C_i^t is set to 15 for all pseudoregister states t of each pseudoregister. A pseudoregister with a justification sequence for state t is given a controllability value $c_{\min}$, where $c_{\min} > 15$; $c_{\min}$ varies with different pseudoregister sizes, with higher $c_{\min}$ values given to larger pseudoregisters. During test generation, C_i^t is decremented by 1 every time justification of state t for pseudoregister i fails and is incremented by 1 otherwise. The power indicators of all corresponding sequences are updated for each successful and unsuccessful GA application.

This measure of controllability helps in relaxation of the state during the single-time-frame mode by avoiding hard-to-justify pseudoregister states. For example, a state that is easily justifiable is favored over a state that is hard to justify. The dynamic controllability also helps during state justification by guiding the GA toward discovering and/or traversing through hard-to-justify states.

Analogous reasoning goes for the observability of the flip-flops. Let O_i be the observability value for flip-flop i. O_i is initially set to $o_{\min}$ ($o_{\min} > 15$) for flip-flops having distinguishing sequences and 15 for the remaining flip-flops. During test generation, O_i is decremented by 1 if no distinguishing sequence can be obtained for flip-flop i and incremented by 1 if the application of its distinguishing sequence is successful. A low value indicates that it is difficult to propagate a fault effect to a primary output from that flip-flop. This measure enables propagation of fault effects to hard-to-observe flip-flops to be avoided.

D. Fitness Functions

Since the fault activation and fault propagation phases target different goals, their corresponding fitness functions differ. The parameters that affect the fitness of an individual in the GA are as follows.

P_1 Fault detection by the individual.

P_2 Sum of dynamic controllabilities for pseudoregisters.

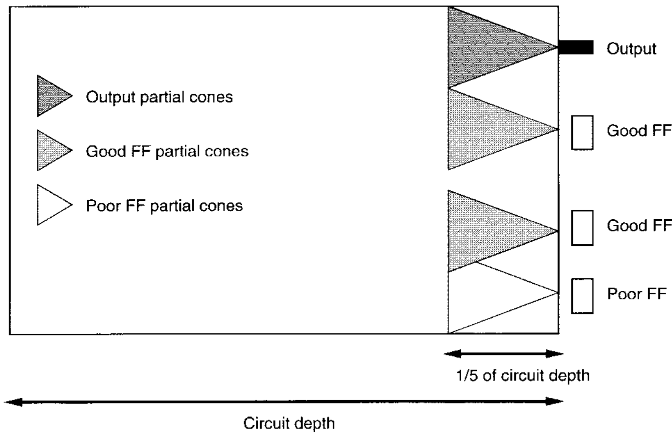

Fig. 8. Output and flip-flop partial cones.

P_3 Matches of pseudoregister values during state justification.

P_4 Sum of distinguishing powers for the distinguishing sequences of flip-flops with fault effects.

P_5 Weighted faulty circuit activity induced.

P_6 Number of new states visited by the individual.

Parameter P_1 is self-explanatory, in particular, during the fault propagation phase. It is included in the fault activation phase to cover faults that propagate directly to the primary outputs in the time frame in which they are excited. P_2 indicates the quality of the state to be justified. Maximizing P_2 makes the state more easily justifiable in the single-time-frame mode, and also avoids unjustifiable states. On the other hand, minimizing P_2 expands the search space by visiting hard-to-justify states. A sequence that justifies a hard-to-justify pseudoregister state is favored during test generation since the GA is more likely to bring the circuit to previously unexplored state spaces as a consequence. P_3 guides the GA to match the required pseudoregister values in the state to be justified, from the least controllable to the most controllable pseudoregister state. If only set and clear sequences are used, the pseudoregister states are replaced by flip-flop values in P_3. P_4 measures the quality of the set of flip-flops reached by the fault effects. Maximizing P_4 increases the probability that the fault effects reach flip-flops having more powerful distinguishing sequences, and thus indirectly improves the chances for detection. P_5 measures the number of events generated in the faulty circuit, with events on more observable gates weighted more heavily. Partial cones are computed and set up for the primary outputs and flip-flops. Fig. 8 illustrates the setup of partial cones; each partial cone has a depth of one-fifth the circuit's depth. Events are weighted more heavily if they are inside partial cones of the PO's or flip-flops with more powerful distinguishing sequences; events inside the partial cones of the hard-to-observe flip-flops are weighted more lightly. Events inside the partial cones of flip-flops having moderate observability values and events outside the partial cones are given a weight of 1. The partial cones are recomputed at the beginning of each of the three GA stages in order to include cones of flip-flops for which new distinguishing sequences have been obtained or observability values have decreased. P_6 is used to expand the search space. It was suggested in [29] and [30] that visiting as many different states as possible helps to detect more faults. The fitness functions thus favor visiting more states when the fault detection count drops very low. Hence, P_6 is considered in the final stage only. Different weights are given to each parameter in the fitness computation during the two phases (see the bottom of the page).

TABLE I
CHARACTERISTICS OF SYNTHESIZED CIRCUITS

Circuit	Seq. Depth	FF's	PI's	PO's	Faults
am2910	4	87	20	16	2391
mult16	9	55	18	33	1708
div16	19	50	33	34	2147
pcont2	3	24	9	8	11,300
piir8o	5	56	9	8	19,920
piir8	5	56	9	8	29,689

In the fault activation phase, the aim is to excite the fault and propagate the fault effects to as many good flip-flops as possible with short sequences and minimal time, where good flip-flops are those with more powerful distinguishing sequences; thus, the fitness function places a heavier weight on the quality of flip-flops reached by the fault effects. Any positive value on parameter P_4 implies that the target fault is excited and the fault effects propagated to at least one flip-flop. If no sequence is obtained to activate the current target fault, single-time-frame fault activation and state justification are used in a second attempt to activate the fault. In this case, the fitness function favors states that can be more easily justified in the single-time-frame activation, and the fitness function favors hard-to-reach states during state justification because hard-to-

Fault activation phase:

Multiple time frame:

$$\textit{fitness} = 0.2P_1 + 0.7P_4 + 0.1(P_5 + P_6^\dagger)$$

Single time frame:

$$\textit{fitness} = 0.1P_1 + 0.5P_2 + 0.3P_4 + 0.2(P_5 + P_6^\dagger)$$

State justification:

$$\textit{fitness} = 0.1P_1 + 0.7P_3 + 0.2(\min(P_2)^* + P_5 + P_6^\dagger)$$

$*: \min(x) = \text{Constant} - x$

Fault propagation phase:

$$\textit{fitness} = 0.8P_1 + 0.2(P_4 + P_5 + P_6^\dagger)$$

† evaluated only in the final GA stage.

TABLE II
GATE(0) Test Generation Results

Circuit	Total	HITEC[7]		GATEST[27]		CRIS[15]		GATTO[20]		IGATE(0)		
	Faults	Det	Vec	Det	Vec	Det	Vec	Det	Vec	Det	Vec	Dist
s298	308	**265**	306	**265**	161	253	476	-	-	264	239	7
s344	342	328	142	**329**	95	328	115	-	-	**329**	109	8
s382	399	**363**	4931	347	281	273	246	-	-	**363**	581	6
s400	426	**383**	4309	365	280	357	758	-	-	382	3369	11
s444	474	414	2240	406	275	397	519	-	-	**420**	1393	9
s526	555	365	2232	417	281	428	692	-	-	**446**	2867	9
s641	467	**404**	216	**404**	139	398	628	-	-	**404**	180	17
s713	581	**476**	194	**476**	128	475	1124	-	-	**476**	147	17
s820	850	**813**	984	517	146	451	1381	-	-	621	465	9
s832	870	**817**	981	539	150	370	1328	-	-	606	703	9
s1196	1242	**1239**	453	1232	347	1180	2744	1226	5202	1236	549	13
s1238	1355	**1283**	478	1274	383	1229	4313	1274	4672	1281	504	12
s1423	1515	776	177	1222	663	1167	2696	1265	3394	**1393**	4044	30
s1488	1486	**1444**	1294	1392	243	1355	1960	1344	631	1378	542	5
s1494	1506	**1453**	1407	1416	245	1357	1928	1277	912	1354	581	6
s5378	4603	3238	941	3175	511	3029	1255	3277	1132	**3447**	10,500	94
s35932	39,094	34,902	240	35,003	200	34,481	1525	32,943	563	**35,100**	386	301
am2910	2391	2164	874	2163	745	-	-	-	-	**2195**	2206	71
mult16	1708	1640	273	1653	204	-	-	-	-	**1664**	915	32
div16	2147	1665	189	1739	634	-	-	-	-	**1802**	4481	3
pcont2	11,300	3354	7	6826	272	-	-	-	-	**6837**	3452	0
piir8o	19,920	14,221	347	15,013	531	-	-	-	-	**15,072**	506	0
piir8	29,689	11,131	31	-	-	-	-	-	-	**18,140**	603	0

Det: number of faults detected Vec: test set length Dist: number of distinguishing sequences generated
Highest numbers of detections are highlighted

reach states may be necessary in order to reach a desired state that was previously unvisited. In the fault propagation phase, the goal is to find a sequence that will propagate the fault effects to a primary output, so the emphasis is placed on fault detection.

E. Adaptive Pruning of Sequences

The distinguishing or justification power associated with a sequence should indicate how well the sequence can propagate a fault effect or justify a state under various requirements for different faults. When a sequence S_0 is obtained, a minimal power is given to the sequence. S_0 is applied again when it is needed. For instance, when the same partial state needs to be justified at a later time, the corresponding pseudoregister justification sequence will be seeded. In the case of distinguishing sequences, when another fault effect reaches the same flip-flop, the associated distinguishing sequence is used again to guide the GA. If a sequence is found and is the same as S_0, the power for S_0 is incremented. If the sequence found differs from S_0, an additional sequence S_1 is added to the corresponding set of sequences, and the powers of both S_0 and S_1 are incremented. On the other hand, if no sequence is found, the power of S_0 is decremented. If the power drops below the minimal value, S_0 is removed from future consideration.

When the GA cannot be seeded with any useful sequences from a set of flip-flops, it is initialized with random sequences. If a sequence is derived, it becomes a candidate sequence for the respective pseudoregisters or flip-flops in the set. It should be noted that this sequence may be more powerful for some partial states, while less powerful for others in the set, but as the sequences are further pruned, the unfit ones are eventually weeded out.

F. Selection of the Target Fault

Since only forward propagation is involved, the target fault is selected intelligently. A fault is selected when its fault effects have propagated to a flip-flop having a distinguishing sequence of maximal distinguishing power. By selecting this fault, the activation phase can be omitted because the effects of the targeted fault have already reached at least one flip-flop. Thus, the fault propagation phase can be entered immediately. The target fault selected in this manner is likely to have a higher probability of detection.

If no fault has reached any flip-flop having a distinguishing sequence, selection of the target fault is biased toward the fault that has reached the greatest number of flip-flops. However, the activation phase is not omitted in this case.

G. Other Implementation Details

Because one fault is targeted at a time and the majority of time spent by the GA is in the fitness evaluation, parallelism among the individuals can be exploited. Therefore, parallel-pattern simulation [28] is used to speed up the process. During test generation, 32 sequences are simulated simultaneously, with values bit packed into 32-bit words during simulation. Fault-free simulation is first performed, followed by insertion of the fault and faulty circuit evaluation, in which events start exclusively from the faulty gate.

Targeting untestable faults is a waste of time because untestable faults cannot be identified using our approach. Thus, the HITEC deterministic test generator [7] is used after the first GA stage to identify and remove many of the untestable faults. A small time limit of 0.4 s/fault is used in an initial HITEC pass through the fault list to minimize the execution

TABLE III
COMPARISON OF THREE EXPERIMENTS

Circuit	Total Faults	IGATE(0)			IGATE(1)			IGATE(6)		
		Det	Vec	DI	Det	Vec	SC	Det	Vec	PRS
s298	308	264	239	7	**265**	204	23	**265**	232	60
s344	342	**329**	109	8	**329**	164	19	**329**	120	48
s382	399	**363**	581	6	362	1847	33	362	2047	104
s400	426	382	3369	11	382	2900	29	382	2162	112
s444	474	420	1393	9	423	1269	31	**424**	1970	104
s526	555	446	2867	9	**449**	3820	22	448	1840	100
s641	467	**404**	180	17	**404**	203	34	**404**	265	64
s713	581	**476**	147	17	**476**	172	30	**476**	160	80
s820	850	621	465	9	702	197	14	**814**	924	26
s832	870	606	703	9	718	370	10	**816**	965	26
s1196	1242	1236	549	13	1235	555	37	**1239**	633	142
s1238	1355	1281	504	12	*1282*	541	38	**1283**	589	143
s1423	1515	1393	4044	30	1394	4316	118	**1407**	3680	596
s1488	1486	1378	542	5	1442	392	23	**1443**	566	48
s1494	1506	1354	581	6	1450	465	29	**1452**	639	14
s5378	4603	3447	10,500	94	3487	11,500	142	**3556**	7664	1449
s35932	39,094	**35,100**	386	301	**35,100**	714	3289	**35,100**	223	13,365
am2910	2391	2195	2206	71	2195	2315	101	**2198**	3083	480
mult16	1708	**1664**	915	32	**1664**	985	87	**1664**	2121	280
div16	2147	1802	4481	3	1802	3705	100	**1815**	5589	446
pcont2	11,300	**6837**	3452	0	**6837**	3500	82	**6837**	5165	256
piir8o	19,920	**15,072**	506	0	**15,072**	604	75	**15,072**	600	300
piir8	29,689	18,140	603	0	18,140	700	75	**18,206**	493	580

Det: number of faults detected Vec: test set length DI: number of distinguishing sequences generated
SC: number of set and clear sequences generated
PRS: number of pseudo-register states with justification sequences
Highest numbers of detections are highlighted

time. If a large number of untestable faults is identified or if only a small number of faults remains in the fault list, a second HITEC pass with a time limit of 2 s/fault is used. Any test sequences generated by HITEC are discarded.

VI. EXPERIMENTAL RESULTS

The test generator was implemented in C++; both ISCAS89 sequential benchmark circuits [13] and several synthesized circuits were used to evaluate its performance. All circuits were evaluated on an HP 9000 J200 with 256-Mbyte RAM. The characteristics of the synthesized circuits are as follows: am2910 [31] is a 12-bit microprogram sequencer, mult16 is a 16-bit 2's-complement shift-and-add multiplier, div16 is a 16 bit divider using repeated subtractions, pcont2 is an 8-bit controller for DSP applications, and piir8o and piir8 are both 8-bit digital DSP filters. Table I displays the characteristics of the synthesized circuits. Structural sequential depth, number of flip-flops, number of PI's, number of PO's, and the total number of collapsed faults for each circuit are given in the table.

Three experiments were conducted. The first experiment involves the use of only distinguishing sequences during test generation. Neither single-time-frame fault activation nor state justification is applied in this case. Essentially, the distinguishing sequences are applied whenever the target fault could be activated directly from the state reached after the previous test sequences are applied. In the second experiment, pseudoregister justification sequences of size 1 (i.e., set and clear sequences for individual flip-flops) are used as seeds during the state justification phase, and distinguishing sequences are used during fault propagation. Finally, the results obtained by using justification sequences of larger pseudoregister sizes are reported in the third experiment. The test generators associated with these three schemes are labeled IGATE(n), where n is the pseudoregister size used in the test generator. IGATE(0) is the test generator that uses only distinguishing sequences (no pseudoregister sequences for state justification), IGATE(1) utilizes justification sequences of size-1 pseudoregisters, etc. Results for each test generator are given in this section. In addition, the effect of pseudoregister sizes on state justification is discussed as well.

A. Results of Using Distinguishing Sequences Only

The first experiment involves the use of distinguishing sequences only in the test generator IGATE(0). The fault coverages of IGATE(0) are compared to fault coverages for various other test generators in Table II. For each circuit, the total number of collapsed faults is given, followed by the number of faults detected and the test set length for each test generator. The number of distinguishing sequences generated by our approach is also reported for each circuit. The first test generator is HITEC [7], a deterministic test generator, followed by the GA-based test generators GATEST [27], CRIS [15], GATTO [20], and finally our test generator, IGATE(0). The GATEST results are an average of ten runs, each beginning with a different random seed.

Fault coverage is defined as the percentage of faults detected. The best fault coverages are highlighted in bold. From the table, the fault coverages achieved by IGATE(0) are significantly higher than those obtained by the other GA-based test generators for most of the circuits. For the hard-

TABLE IV
RESULTS AT VARIOUS CHECKPOINTS FOR IGATE(0)

Circuit	FF's	Best of previous GA-based Test Gen's			IGATE(0)				
		Det	Vec	TestGen	Chkpt	Det	Δ Det	Vec	Time
s526	21				1	431	+3	556	3.28 min
					2	439	+11	907	10.9 min
		428	692	CRIS[15]	3	446	+18	2867	35.0 min
s820	5				1	471	-46	117	32.0 sec
					2	547	+30	373	16.1 min
		517	146	GATEST[27]	3	621	+104	465	56.0 min
s832	5				1	476	-63	122	37.5 sec
					2	581	+42	303	15.3 min
		539	150	GATEST[27]	3	606	+67	703	1.08 hr
s1423	74				1	1381	+116	3760	23.5 min
					2	1393	+128	4044	1.42 hr
		1265	3394	GATTO[20]	3	1393	+128	4044	3.66 hr
s1488	6				1	1237	-155	194	4.07 min
					2	1271	-121	329	15.6 min
		1392	243	GATEST[27]	3	1378	-14	542	18.5 min
s1494	6				1	1203	-213	109	3.12 min
					2	1344	-72	295	18.1 min
		1416	245	GATEST[27]	3	1354	-62	581	30.1 min
s5378	179				1	3425	+148	2010	1.33 hr
					2	3436	+159	3170	4.00 hr
		3277	1132	GATTO[20]	3	3447	+170	10,500	13.7 hr
s35932	1728				1	35,099	+90	225	17.4 hr
					2	35,100	+91	386	22.2 hr
		35,003	200	GATEST[27]	3	35,100	+91	386	58.1 hr
am2910	87				1	2118	-48	512	8.80 min
					2	2167	+3	1190	39.9 min
		2163	745	GATEST[27]	3	2194	+30	2206	1.67 hr
div16	50				1	1713	-26	725	1.84 hr
					2	1770	+31	1530	8.89 hr
		1739	634	GATEST[27]	3	1802	+63	4481	15.0 hr

Check Point k: end of GA stage k
Δ Det: improvement over the best coverage among previous GA-based test generators

to-test ISCAS89 circuits, such as $s400$, $s444$, $s526$, $s1423$, $s5378$, and $s35932$, where long execution times are required by HITEC, the fault coverages achieved by IGATE(0) are significantly higher than those for the other GA-based test generators, and often higher than the HITEC fault coverages as well. IGATE(0) outperforms both HITEC and GATEST for all the synthesized circuits. However, IGATE(0) does not perform as well as HITEC for $s820$, $s832$, $s1488$, and $s1494$. These circuits contain faults that require specific and often long sequences for fault activation. None of the previous GA-based test generators could match the results of HITEC for these circuits since only HITEC was able to generate the exact sequences required to excite the faults. The test sets obtained by IGATE(0) are shorter than those obtained by HITEC, even when higher fault coverages are achieved by IGATE(0). The test sets are shorter than those obtained by CRIS and GATTO for most of the circuits. The test set lengths are comparable to those for GATEST, although sometimes longer due to the two-phase strategy of activating and propagating faults. Very high fault coverages are obtained very quickly for most of the circuits, as will be shown in the later part of this discussion.

B. Adding Single-Time-Frame Activation and State Justification

In the second and third experiments, single-time-frame mode with state justification is used to target hard-to-activate faults in the circuits. Pseudoregister justification sequences of various sizes are used to aid state justification in IGATE(n), where $n > 0$. A comparison of fault coverages for IGATE(0), IGATE(1), and IGATE(6) is given in Table III. The number of faults detected and the test set lengths are given for each test generator. The number of distinguishing sequences (DI), set/clear sequences (SC), and the number of pseudoregister states for which justification sequences were generated (PRS) are given for IGATE(0), IGATE(1), and IGATE(6), respectively. Results for IGATE(6) are obtained using pseudoregisters of size 6. The best fault coverage numbers are shown in bold. The numbers of distinguishing sequences generated by IGATE(1) and IGATE(6) are similar to the number obtained by IGATE(0).

For circuits that IGATE(0) and the previous GA-based test generators did poorly on, namely, $s820$, $s832$, $s1488$, and $s1494$, which require specific justification sequences, IGATE(1) and IGATE(6) were able to detect many more faults due to single-time-frame fault activation and state justification. The fault coverages are as high as those obtained by HITEC for IGATE(6). For $s820$, IGATE(1) detected 702 faults and IGATE(6) detected 814 faults, while only 517, 451, and 621 faults were detected by GATEST[27], CRIS[15], and IGATE(0), respectively; more than 30% improvement in fault coverage was obtained. For the larger and more complex circuits, $s1423$, $s5378$, and $s35932$, outstanding fault coverages

TABLE V
RESULTS AT VARIOUS CHECKPOINTS FOR IGATE(1) AND IGATE(6)

Circuit	FF's		IGATE(1)				IGATE(6)			
		Chkpt	Det	Δ Det	Vec	Time	Det	Δ Det	Vec	Time
s526	21	1	430	+2	525	1.10 min	438	+10	602	2.70 min
		2	440	+12	1305	10.0 min	448	+20	1840	15.2 min
		3	449	+21	3820	20.0 min	448	+20	1840	40.1 min
s820	5	1	471	-46	127	43.0 sec	649	+132	359	4.00 min
		2	702	+185	197	16.3 min	808	+291	858	15.0 min
		3	702	+185	197	36.0 min	814	+297	956	34.1 min
s832	5	1	422	-117	112	47.5 sec	781	+242	534	4.30 min
		2	582	+43	303	15.3 min	798	+259	687	18.2 min
		3	718	+179	370	38.1 min	817	+278	926	33.0 min
s1423	74	1	1381	+116	2651	13.5 min	1378	+113	991	25.0 min
		2	1393	+128	3611	20.1 min	1394	+129	2341	45.3 min
		3	1394	+129	4316	1.10 hr	1407	+142	3680	1.10 hr
s1488	6	1	1392	+0	235	4.12 min	1380	-12	188	3.00 min
		2	1435	+43	318	18.6 min	1442	+50	272	9.90 min
		3	1442	+50	392	23.5 min	1443	+51	566	10.1 min
s1494	6	1	1363	-53	239	4.12 min	1422	+6	309	4.50 min
		2	1448	+32	415	8.1 min	1443	+27	521	8.2 min
		3	1450	+34	436	12.1 min	1452	+36	639	11.0 min
s5378	179	1	3421	+144	2155	1.24 hr	3480	+203	2335	3.50 hr
		2	3471	+194	6742	9.90 hr	3515	+238	5035	9.1 hr
		3	3487	+210	11,500	13.7 hr	3556	+279	7664	21.5 hr
s35932	1728	1	35,096	+87	269	4.1 hr	35,100	+91	223	6.70 hr
		2	35,099	+90	482	7.0 hr	35,100	+91	223	7.1 hr
		3	35,100	+91	714	10.1 hr	35,100	+91	223	8.5 hr
am2910	87	1	2160	-3	654	6.90 min	2160	-3	758	7.00 min
		2	2189	+26	1707	23.8 min	2184	+21	2103	22.1 min
		3	2198	+35	2990	1.37 hr	2195	+32	2984	45.4 min
div16	50	1	1707	-32	623	20.5 min	1691	-48	669	18.9 min
		2	1768	+29	1456	1.19 hr	1773	+34	1455	1.10 hr
		3	1781	+42	3507	6.5 hr	1815	+76	5589	13.5 hr

Check Point k: end of GA stage k
Δ Det: improvement over the best coverage among previous GA-based test generators

were obtained by IGATE(6) when compared to IGATE(0), which already achieved very high fault coverages. Similar trends are seen in the results for the synthesized circuits. In terms of test set sizes, the test sets obtained by IGATE(1) and IGATE(6) are nearly the same size as those obtained by IGATE(0), which are generally shorter than those obtained by HITEC when comparable fault coverages are obtained. The test sets are also shorter than those obtained by CRIS for most of the circuits. IGATE(0), IGATE(1), and IGATE(6) also achieve very high fault coverages very quickly, as will be discussed next.

C. Execution Times

Very high fault coverages were obtained using a small number of vectors in a short time by all three test generators. This phenomenon is illustrated in Tables IV and V.

The best results of the previous GA-based test generators shown in Table II are listed in Table IV. The test generator, number of faults detected, and test set size are shown in Table IV, followed by the results of IGATE(0). Table V shows the results for IGATE(1) and IGATE(6). The run times at three checkpoints are displayed for IGATE(0), IGATE(1), and IGATE(6) in these two tables. These checkpoints were placed at the end of each GA stage. Recall that sequence lengths for the individuals in the population are doubled from one stage to the next, with longer sequences used to target the harder faults. The **ΔDet** column shows the difference in fault coverages between our test generators and the best achieved by any of the previous GA-based test generators. The fault coverages at the end of the first GA stage are already higher than the final fault coverages of the other test generators for many circuits. Many of these execution times are much shorter than those required by HITEC [7]. The user may wish to stop the test generation process if the fault coverage has reached a satisfactory level at the end of the first or second stage.

D. Effect of Pseudoregister Sizes

Table VI compares the effects of different pseudoregister sizes. The numbers of distinct pseudoregister states (PRS) for which justification sequences could be generated by IGATE(n) are reported, and results for three pseudoregister sizes are shown: 2, 4, and 6. The highest numbers of faults detected are shown in bold, and the italicized numbers indicate fault detections which are higher than any results previously obtained by other GA-based test generators.

One interesting statistic is the number of distinct pseudo-register states for which justification sequences could be derived. This number is indicated in the "PRS" column of the table. Take, for instance, circuit $s820$ which has only five flip-flops. When the pseudoregister size is 2, the flip-flops are

TABLE VI
EFFECTS OF PSEUDOREGISTER SIZES

Circuit	Total Faults	Total FF's	Various Pseudo-Register Sizes								
			IGATE(2)			IGATE(4)			IGATE(6)		
			Det	Vec	PRS	Det	Vec	PRS	Det	Vec	PRS
s298	308	14	**265**	236	27	**265**	243	52	**265**	232	60
s344	342	15	**329**	104	30	**329**	77	54	**329**	120	48
s382	399	21	360	1782	39	362	3420	76	362	2047	104
s400	426	21	**382**	2714	40	**382**	1763	72	**382**	2162	112
s444	474	21	*421*	950	39	**424**	2082	66	**424**	1970	104
s526	555	21	446	2113	41	**448**	3990	72	**448**	1840	100
s641	467	19	**404**	167	35	**404**	294	60	**404**	265	64
s713	581	19	**476**	147	32	**476**	155	48	**476**	160	80
s820	850	5	*812*	694	10	*812*	987	17	**814**	956	26
s832	870	5	**817**	926	10	*794*	959	14	*816*	965	26
s1196	1242	18	**1239**	505	36	**1239**	598	64	**1239**	633	142
s1238	1355	18	**1283**	578	35	**1283**	643	64	**1283**	589	143
s1423	1515	74	*1395*	4152	147	*1402*	5636	267	**1407**	3680	596
s1488	1486	6	**1444**	585	12	*1443*	540	17	*1443*	577	48
s1494	1506	6	**1452**	611	12	*1450*	635	17	**1452**	639	14
s5378	4603	179	**3568**	12,501	332	*3559*	10,391	676	*3556*	7664	1449
s35932	39,094	1728	**35,100**	250	3344	**35,100**	241	6070	**35,100**	223	13,365
am2910	2391	87	*2197*	2369	174	*2196*	3645	352	**2198**	3083	480
mult16	1708	55	**1665**	1807	110	1664	679	218	1664	2421	280
div16	2138	50	*1812*	4084	89	*1813*	4993	178	**1815**	5589	446
pcont2	11,300	24	**6837**	2563	48	**6837**	313	96	**6837**	5165	256
piir8o	19,920	56	15,071	340	112	**15,072**	600	224	**15,072**	600	300
piir8	29,689	56	**18,206**	604	112	**18,206**	491	224	**18,206**	493	580

Det: # of faults detected Vec: test set length
PRS: # of distinct pseudo-register states having justification sequences
Highest numbers of detections are shown in **bold**
Fault coverages higher than any previous GA-based ATG's are *italicized*

partitioned into three pseudoregisters of sizes 2, 2, and 1. The total number of states represented by the pseudoregisters is $2^2 + 2^2 + 2^1 = 10$ states. All ten pseudoregister states had justification sequences generated as shown in the table. When the pseudoregister size is increased to 4, the total number of states that can be represented now becomes $2^4 + 2^1 = 18$. Only a portion of the 18 pseudoregister states had justification sequences generated. As the pseudoregister sizes get larger, a smaller portion of the possible pseudoregister state space is obtainable. For $s1494$, only 14 out of 64 pseudoregister states had justification sequences because most of the faults were quickly detected without using the state justification phase, and the 14 states were the hard-to-reach states required by the remaining faults. In most circuits, larger pseudoregister sizes tend to give better fault coverage. Larger pseudoregisters, however, may adversely affect the fault coverage since justification sequences for some difficult but critical pseudoregister states may fail to be generated. For $s5378$, the number of size-2 pseudoregister states is $89 \times 2^2 + 1 = 357$; 332 out of 357 states had justification sequences derived for them. For a pseudoregister size of 6, only 1449 of 1888 (i.e., $29 \times 2^6 + 2^5$) states have justification sequences; fewer than 80% of the pseudoregister states have justification sequences in this case. The lack of information about justification sequences for the missing 20% of pseudoregister states makes state justification less successful for some crucial states. Therefore, it may be better to have justification sequences for some of these critical states with small pseudoregister sizes than having none at all when larger size pseudoregisters are used. This result confirms the ability of the GA to geneticallyengineer a solution when given useful starting seeds.

VII. CONCLUSION

A test generation framework which utilizes genetically-engineered finite-state-machine sequences for targeting hard-to-test faults was presented. The test generator is composed of three stages, and several passes through the fault list are made in each stage. Test generation for a targeted fault is carried out in two phases. The first phase excites a fault, and propagates its effects to the flip-flops with the assistance of pseudoregister justification sequences if necessary. The second phase drives the fault effects from the flip-flops to the primary outputs with the aid of distinguishing sequences. Various types of distinguishing, set, clear, and pseudoregister justification sequences are computed, both in a preprocessing step and during the test generation process. These sequences are seeded in the GA to evolve valid state justification and fault propagation sequences for the target fault. The GA is able to combine the finite-state-machine sequences, which are partial solutions to the problem, to engineer a complete solution. Pruning of these sequences is done dynamically during test generation to improve their effectiveness, quantified by the power indexes. Very high fault coverages obtained in short execution times result from the use of this approach. Significant improvements have been observed over previous GA-based and deterministic approaches.

REFERENCES

[1] S. Seshu and D. N. Freeman, "The diagnosis of asynchronous sequential switching systems," *IRE Trans. Electron. Comput.*, vol. EC-11, pp. 459–465, Aug. 1962.

[2] R. Marlett, "An effective test generation system for sequential circuits," in *Proc. Design Automation Conf.*, 1986, pp. 250–256.

[3] W.-T. Cheng, "The BACK algorithm for sequential test generation," in *Proc. Int. Conf. Comput. Design*, 1988, pp. 66–69.

[4] H.-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, "Test generation for sequential circuits," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 1081–1093, Oct. 1988.

[5] M. H. Schulz and E. Auth, "Essential: An efficient self-learning test pattern generation algorithm for sequential circuits," in *Proc. Int. Test Conf.*, 1989, pp. 28–37.

[6] A. Ghosh, S. Devadas, and A. R. Newton, "Test generation for highly sequential circuits," in *Proc. Int. Conf. Computer-Aided Design*, 1989, pp. 362–365.

[7] T. M. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *Proc. European Conf. Design Automation*, 1991, pp. 214–218.

[8] ——, "Method for automatically generating test vectors for digital integrated circuits," U.S. Patent 5 377 197, Dec. 1994.

[9] D. H. Lee and S. M. Reddy, "A new test generation method for sequential circuits," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 446–449.

[10] H. Cho, S.-W. Jeong, and F. Somenzi, "Synchronizing sequences and symbolic traversal techniques in test generation," *J. Electron. Testing: Theory Appl.*, vol. 4, no. 1, pp. 19–31, 1993.

[11] I. Pomeranz and S. M. Reddy, "Application of homing sequences to synchronous sequential circuit testing," *IEEE Trans. Comput.*, vol. 43, pp. 569–580, May 1994.

[12] J. Park, C. Oh, and M. R. Mercer, "Improved sequential ATPG based on functional observation information and new justification methods," in *Proc. European Design Test Conf.*, 1995, pp. 262–266.

[13] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Int. Symp. Circuits Syst.*, 1989, pp. 1929–1934.

[14] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[15] D. G. Saab, Y. G. Saab, and J. A. Abraham, "CRIS: A test cultivation program for sequential VLSI circuits," in *Proc. Int. Conf. Computer-Aided Design*, 1992, pp. 216–219.

[16] M. Srinivas and L. M. Patnaik, "A simulation-based test generation scheme using genetic algorithms," in *Proc. Int. Conf. VLSI Design*, 1993, pp. 132–135.

[17] E. M. Rudnick, J. G. Holm, D. G. Saab, and J. H. Patel, "Application of simple genetic algorithms to sequential circuit test generation," in *Proc. European Design Test Conf.*, 1994, pp. 40–45.

[18] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "Sequential circuit test generation in a genetic algorithm framework," in *Proc. Design Automation Conf.*, 1994, pp. 698–704.

[19] E. M. Rudnick, "Simulation-based techniques for sequential circuit testing," Ph.D. dissertation, Dept. Elect. Comput. Eng., Tech. Rep. CRHC-94-14/UILU-ENG-94-2229, Univ. Illinois, Aug. 1994.

[20] P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "An automatic test pattern generator for large sequential circuits based on genetic algorithms," in *Proc. Int. Test Conf.*, 1994, pp. 240–249.

[21] D. G. Saab, Y. G. Saab, and J. A. Abraham, "Iterative [simulation-based genetics + deterministic techniques] = complete ATPG," in *Proc. Int. Conf. Computer-Aided Design*, 1994, pp. 40–43.

[22] E. M. Rudnick and J. H. Patel, "Combining deterministic and genetic approaches for sequential circuit test generation," in *Proc. Design Automation Conf.*, 1995, pp. 183–188.

[23] F. Corno, P. Prinetto, M. Rebaudengo, and M. S. Reorda, "GATTO: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuits," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 991–1000, Aug. 1996.

[24] D. G. Saab, Y. G. Saab, and J. A. Abraham, "Automatic test vector cultivation for sequential VLSI circuits using genetic algorithms," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1278–1285, Oct. 1996.

[25] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Alternating strategies for sequential circuit ATPG," in *Proc. European Design Test Conf.*, 1996, pp. 368–374.

[26] ——, "Automatic test generation using genetically-engineered distinguishing sequences," in *Proc. VLSI Test Symp.*, 1996, pp. 216–223.

[27] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "A genetic algorithm framework for test generation," *IEEE Trans. Computer-Aided Design*, vol. 16, Sept. 1997.

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

[29] I. Pomeranz and S. M. Reddy, "LOCSTEP: A logic simulation based test generation procedure," in *Proc. Fault Tolerant Computing Symp.*, 1995, pp. 110–119.

[30] T. E. Marchok, A. El-Maleh, W. Maly and J. Rajski, "Complexity of sequential ATPG," in *Proc. European Design and Test Conf.*, Mar. 1995, pp. 252–261.

[31] Advanced Micro Devices, "The AM2910, a complete 12-bit microprogram sequence controller," in *AMD Data Book*, AMD Inc., Sunnyvale, CA, 1978.

Michael S. Hsiao (S'95–M'97) received the B.S. degree in computer engineering from the University of Illinois at Urbana–Champaign in 1992, and the M.S. and Ph.D. degrees in electrical engineering in 1993 and 1997, respectively, from the same university.

During academic year 1992, he was a Teaching Assistant in the Department of Electrical and Computer Engineering, University of Illinois. Between 1993 and 1997, he was a Research Assistant at the Center for Reliable and High-Performance Computing, University of Illinois. He was a Visiting Scientist at NEC USA, Princeton, NJ, during the summer of 1997. He is currently with Rutgers, the State University of New Jersey, Piscataway, where he is an Assistant Professor in the Department of Electrical and Computer Engineering. His current research focuses on VLSI testing, design for testability, power estimation, low-power design, and computer architecture.

Elizabeth M. Rudnick (S'90–M'95) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Illinois at Urbana–Champaign in 1983, 1990, and 1994, respectively.

She has worked at Motorola, Sunrise Test Systems, and Advanced Micro Devices in the areas of design verification, test generation, electronic design automation, and yield enhancement. She is currently an Assistant Professor in the Department of Electrical and Computer Engineering and a Research Assistant Professor in the Coordinated Science Laboratory, University of Illinois. Her research interests include test generation, defect diagnosis, design verification, and design for testability.

Janak H. Patel (S'73–M'76–SM'87–F'89) was born in Bhavnagar, India. He received the B.Sc. degree in physics from Gujarat University, India, the B.Tech. degree from the Indian Institute of Technology, Madras, and the M.S. and Ph.D. degree from Stanford University, Stanford, CA, all in electrical engineering.

He is with the University of Illinois at Urbana–Champaign, where he is currently a Codirector of the Center for Reliable and High Performance Computing and a Professor of Electrical and Computer Engineering and Computer Science, and a Research Professor with the Coordinated Science Laboratory. He is a cofounder of Sunrise Test Systems, an ATG and testability software company. He has also provided consulting and tutorial services to the industry on VLSI design and test. He is currently engaged in teaching, research, and consulting in the areas of automatic test generation, design for testability, fault simulation, and diagnosis.