European Design and Test Conferenece, pp. 22-28, Mar. 1997

Sequential Circuit Test Generation Using Dynamic State Traversal

Michael S. Hsiao, Elizabeth M. Rudnick, and Janak H. Patel
Center for Reliable and High-Performance Computing
University of Illinois, Urbana, IL 61801

Abstract

A new method for state justification is proposed for sequential circuit test generation. The *linear* list of states dynamically obtained during the derivation of test vectors is used to guide the search during state justification. State-transfer sequences may already be known that drive the circuit from the current state to the target state. Otherwise, genetic engineering of existing state-transfer sequences is required. In both cases, genetic-algorithm-based techniques are used to generate valid state justification sequences for the circuit in the presence of the target fault. This approach achieves extremely high fault coverages and thus outperforms previous deterministic and simulation-based techniques.

I Introduction

The majority of the time spent by automatic test generators for sequential circuits is used to find test sequences for hard-to-test faults. Deterministic test generators have been proposed in the past, but they often require backtracing through complex gates and flip-flops, and remodeling of such primitives is often required. In addition, large numbers of backtracks are often needed for the hard faults. Simulation-based test generators, on the other hand, avoid the complexity of backtracing by processing in the forward direction only. However, previous simulation-based approaches often fell short when targeting the hard faults because they lacked information about state justification.

Previously, homing, synchronizing, and distinguishing sequences have been used to aid the test generator in improving the fault coverage [1, 2, 3, 4, 5]. In [1, 2, 4], symbolic and state-table-based techniques were used to derive these sequences in the fault-free machine. Specifically, in [1], 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 [2] to derive synchronizing sequences. In the work by Park et al., [4], functional information was used to pre-generate sequences which simplified propagation of fault effects from the flip-flops to the primary outputs (PO's), 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 [3], but they had to be recomputed for each target fault.

Several approaches to test generation using genetic algorithms (GA's) have been proposed in the past [5-14]. 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. However, hard-to-test faults often could not be detected. In GATEST [9] and ALT-TEST [14], 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 [6] and GATTO [10]. Maximizing propagation of fault effects to flip-flops and increasing circuit activity have been shown to increase the probability of detecting faults at the PO's. Although the fault detection probability improves, activating a hard fault and propagating fault-effects from flip-flops to a PO remain difficult problems. Increasing circuit activity may be ineffective in activating a hard fault or propagating fault effects to PO's. DIGATE [5] tackled the problem of fault-effect propagation by intelligent use of distinguishing sequences. However, DIGATE requires that faults be activated in order for it to be effectively applied. 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 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 [12, 13].

Storing the state information for large circuits is impractical; similarly, keeping a list of sequences capable of reaching each reachable state is infeasible. The scheme proposed in our approach uses the *linear* list of states dynamically obtained during the derivation of test vectors to guide state justification. The storage necessary is on the order of the number of vectors generated in this case. When justifying states that have not been visited, several candidate sequences that lead to previously visited states are used to help find the target unvisited state. The candidate states are chosen such that they are similar to the target state. The sequences that reach the candidate states may be viewed as partial solutions to finding

*This research was supported in part by the Semiconductor Research Corporation under contract SRC 96-DP-109, in part by DARPA under contract DABT63-95-C-0069, and by Hewlett-Packard under an equipment grant.

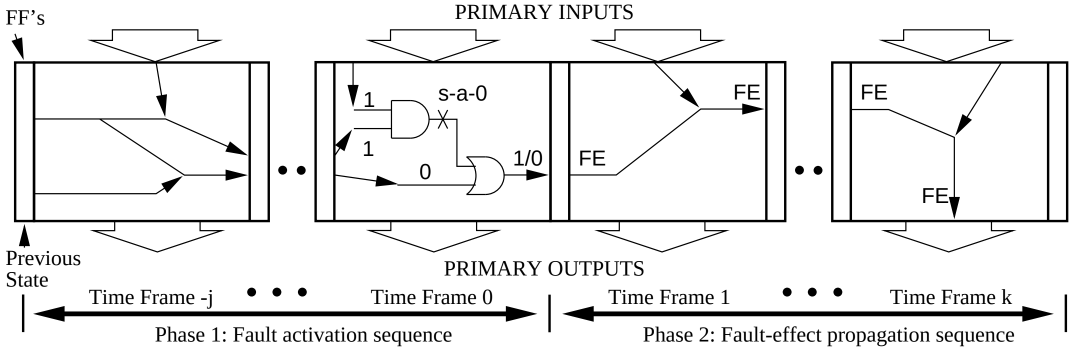

Figure 1: Two-phase strategy.

a sequence that justifies the target state. Genetic algorithms [15] 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 derive and manipulate dynamic state-transfer sequences and in the overall test generation process. Utilizing the dynamic state traversal information allows us to overcome the limitations of the previous genetic approaches and close the gap of 30% fault coverage difference in some of the circuits; for the other circuits, extremely high fault coverages have been obtained compared to other test generators.

II Algorithm 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 Figure 1. The first phase focuses on activating the target fault, while the second phase tries to propagate the fault effects from the flip-flops to the PO's. 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 (FE's) to a PO or to the flip-flops. Once the fault is activated, the fault effects are propagated from the flip-flops to the PO's in the second phase with the assistance of distinguishing sequences. The target fault is detected at the PO's when the faulty machine state is distinguished from the fault-free machine state. The second phase of test generation is similar to the algorithm in DIGATE [5] in which distinguishing sequences are used to propagate the fault-effects. The main contribution of this work is in the first phase, which consists of **single-time-frame fault activation** and **state justification using dynamic state-transfer sequences**.

During the fault activation phase, single-time-frame mode is entered if no activation sequence can be found directly from the state in which the previous sequence left off. The aim of single-time-frame fault activation is to find a test vector, composed of primary input (PI) and flip-flop values, that can activate the target fault in a single time frame. Single-time-frame fault activation is illustrated in Figure 2 and will be described in detail in a later section. Once a vector (PI and flip-flop values) is successfully derived, the state (FF values) is first relaxed to one that has as many don't-care values (X) as possible and still can activate the target fault; this improves the success rate of state-justification which immediately follows [16, 17].

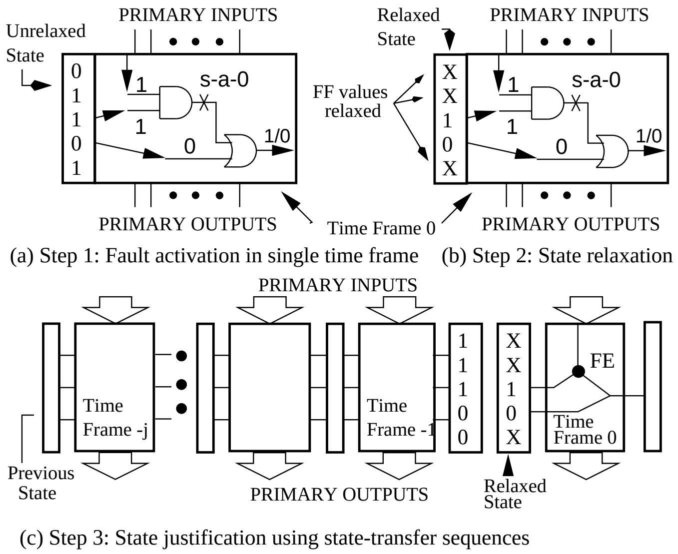

Figure 2: Multistage fault activation.

State justification is performed by using a GA with an initial population composed of random sequences and any useful state-transfer sequences. If the relaxed state S_r matches a previously visited state S_p, and a state-transfer sequence $T_{S_p}^{S_c}$ exists that drives the circuit from the current state S_c to state S_p, the state justification sequence $T_{S_p}^{S_c}$ is simply seeded into the GA. However, if the relaxed state S_r does not match any of the previously visited states, genetic-engineering of several sequences is performed to try to justify the target state. Several candidate states are selected from the set of previously visited states that most closely match the relaxed state S_r. The selection is based on the number of matching flip-flop values in the states. Let the set of selected candidate states be $\{S_i\}$; the set of sequences that justify these states from current state S_c is $\{T_{S_i}^{S_c}\}$. These sequences are used as seeds in the GA to aid in evolving an effective state justification sequence. Candidate sequences in the GA population are simulated, starting from the current state. The objective is to engineer a sequence that justifies the required state by genetically combining the candidate justification sequences. Consider the situation shown in Figure 3 in which an attempt is being made to justify state **1X0X10**. Sequence T_1 successfully justifies all but the third flip-flop value; on the other hand, sequence T_2 justifies all but

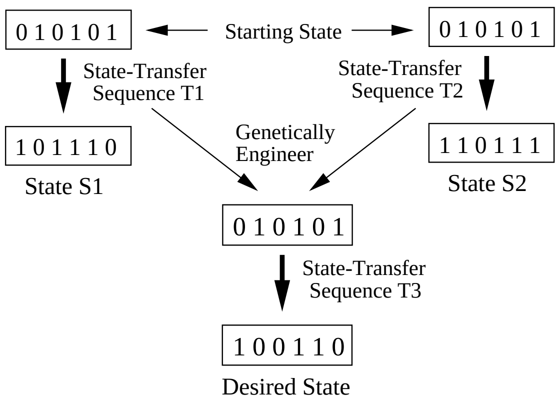

Figure 3: Genetic justification of desired state.

the final flip-flop value. These two sequences T_1 and T_2 may provide important information in evolving the complete solution, T_3, which justifies the complete state. They are used as seeds for the GA in an attempt to genetically engineer the sequence T_3. Presence of the fault may invalidate a sequence that was previously used to traverse a set of states. However, since the GA performs simulation in the presence of the fault to derive a sequence for the current situation, any sequence derived will be valid.

To facilitate the dynamic state traversal algorithm, a table of visited states is mapped to the list of vectors in the test set, as shown in Figure 4. During state justification, the goal is to generate a sequence that will justify the desired state from the current state. In Figure 4, the starting state (also the current state) is reached at the end of vectors i, k, and m, and the desired state (labeled *End State*) is reached at the end of vectors j and l. Therefore, either sequence T_1 (vectors $i+1$ to j) or T_2 (vectors $k+1$ to l) is sufficient to drive the circuit to the ending state. However, if the desired state

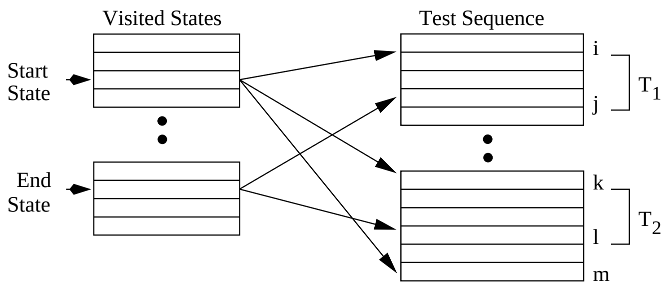

Figure 4: Data structure for dynamic state traversal.

has not been visited, a set of *ending states* that closely match the desired state is formed from the visited state table; the sequences corresponding to these states are seeded in the GA in an attempt to engineer a valid justification sequence for the desired state. If the current state was not visited before the ending states were reached, the n vectors that lead to the ending states are seeded into the GA, where n is the current GA sequence length.

During state justification, a sequence that correctly justifies one portion of the required state may simultaneously set an incorrect value on the 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 PI's for justifying a specific part of a state is intrinsically implied by each sequence, this information may be important and 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.

If a sequence is found that justifies the required state, the sequence is appended to the test set, a fault simulator is invoked to remove any additional faults detected by the sequence, and the test generator proceeds to the fault propagation phase. Otherwise, the current target fault is aborted, and test generation 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, and propagation of fault-effects is done in a manner similar to the algorithm introduced in DIGATE [5]. If a sequence that drives the fault-effects to the PO's is successfully obtained, the sequence is added to the test set, and the fault simulator is used to identify other detected faults that may be dropped. Test generation continues with the next target fault.

III Genetic Algorithms

The GA framework used in our work is similar to the simple GA described by Goldberg [15]. 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 PI in a particular time frame. 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 PI's and the furthest gate. In the first stage of test generation, the sequence length is set equal to the structural sequential depth. The sequence length is doubled in the second stage and doubled again in the third stage, since harder faults may require longer sequences for activation and/or propagation. The population size used is a function of the string length, which depends on both the number of PI's and the test sequence length. Larger populations are needed to accommodate longer individual test sequences in order to maintain diversity. The population size is set equal to $4 \times \textit{square root}(\textit{sequence length})$ when the number of PI's is less than 16 and $16 \times \textit{square root}(\textit{sequence length})$ when the number of PI's is greater than 15.

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 state-transfer or distinguishing sequences exist which are appropriate under the current situation, 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. Two individuals are selected from the existing population, with selection biased toward more highly fit individuals. The two individuals are crossed by randomly swapping bits between them to create two entirely new individuals, and each character in a new string is mutated with some small mutation probability. The two new individuals are then placed in the new population, and this process continues until the new generation is entirely filled. 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 PO's or until a maximum number of generations is reached. 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 Single Time Frame Mode

When a hard-to-activate fault is targeted and the GA fails to generate an activation sequence, a second attempt is made to activate the fault in a single time frame. The aim here is to engineer a vector, composed of PI and flip-flop values, capable of activating the target fault (i.e., exciting the target fault and propagating its effects to at least one flip-flop) in a single time frame. Initially the GA is seeded with random vectors, and the evolution process continues until a vector is found or a maximum of eight generations is reached.

To improve the chances of activation for hard faults, dynamic fitness objectives are set up for each target fault. Figure 5(a) illustrates the justification frontier for a stuck-at-0 fault at the output of gate k. The justification frontier consists of values necessary for justifying a desired value. During the single-time-frame fault activation, the fitness function for fault excitation tries to maximize the number of justification frontier values justified. Once a target fault is excited, its fault-effects need to be propagated to at least one PO or flip-flop. Figure 5(b) shows the propagation frontier for this case. The fitness function aims to dynamically advance the propagation of fault-effects beyond the current propagation frontier. In the example shown in the figure, the fault-effects are not yet propagated beyond gates b and d. Therefore, the dynamic fitness objectives will place emphasis on setting a **1** on line A and a **0** on line C to advance the fault-effects beyond gates b and d. The dynamic objectives are updated after every generation of the GA is evolved.

Because an unjustifiable state is undesirable, the fitness function also uses the dynamic controllability values of the flip-flops to guide the search toward more easily justifiable states. The dynamic controllability measures correspond to the frequency of setting/clearing a particular flip-flop. With these measures, 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 i^{th} 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 i^{th} flip-flop can be relaxed to the unknown value X.

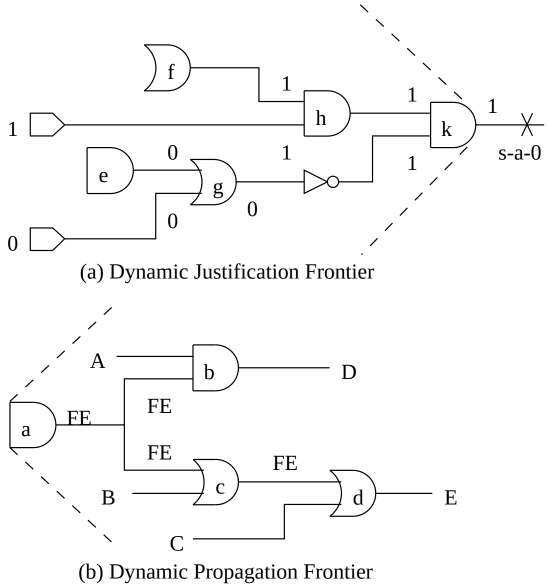

Figure 5: Dynamic objectives for the target fault.

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 greedy fashion: 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 state-transfer sequences as described in Section II. The remaining individuals, if any, are seeded with random sequences. The GA then attempts to combine the partial solutions to form the complete solution that justifies the relaxed state.

V Test Generation Procedure

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 PO's. 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 as described earlier. The GA is initialized with random sequences, and any useful state-transfer or distinguishing sequences are used as seeds in place of some of the random sequences in the fault activation and fault propagation phases.

A 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:

P1: Fault detection
P2: Sum of dynamic controllabilities
P3: Matches of flip-flop values

P4: Sum of distinguishing powers
P5: Induced faulty circuit activity
P6: Number of new states visited

Parameter *P1* 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 PO's in the time frame in which they are excited. To improve state justification and fault detection, flip-flops which are hard to control or hard to observe are identified dynamically during the process and are given lower controllability values; 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. *P2* indicates the quality of the state to be justified. Maximizing *P2* during single-time-frame fault activation makes the state more easily justifiable and also avoids unjustifiable states. On the other hand, minimizing *P2* during state justification expands the search space by forcing hard-to-justify states to be visited. A sequence that justifies a hard-to-justify state is favored during test generation, since the GA is more likely to bring the circuit to previously unexplored state spaces as a consequence. *P3* guides the GA to match the required flip-flop values in the state to be justified during state justification, from the least controllable to the most controllable flip-flop value. *P4* measures the quality of the set of flip-flops reached by the fault effects. Maximizing *P4* increases the probability that the fault-effects reach flip-flops having more powerful distinguishing sequences and thus indirectly improve the chances for detection. *P5* 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 PO's and flip-flops as in DIGATE [5]. Events are weighted more heavily inside the 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. *P6* is used to expand the search space. It was suggested in [18] and [19] 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, *P6* is considered in the final stage only. Different weights are given to each parameter in the fitness computation during the two phases:

Fault activation phase:

Multiple-time-frame:

1. $fitness = 0.2P1 + 0.8P4$

Single-time-frame:

2. $fitness = 0.1P1 + 0.5P2 + 0.2P4 + 0.2(P5 + P6^{\dagger})$

State justification:

3. $fitness = 0.1P1 + 0.2((k - P2) + 0.5P3 + 0.2(P5 + P6^{\dagger})$, where k is a constant

Fault propagation phase:

4. $fitness = 0.8P1 + 0.2(P4 + P5 + P6^{\dagger})$

†: included in the final GA stage only

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, 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 $P4$ implies that the target fault is excited and the fault-effects are 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 easily justified during single-time-frame fault activation, while hard-to-reach states are favored during state justification, because hard-to-reach states may be necessary in order to reach the desired unvisited target state. In the fault propagation phase, the goal is to find a sequence that will propagate the fault effects to a PO, so the emphasis is placed on fault detection.

B Selection of the target fault

A fault is selected when its fault effects have propagated to a flip-flop having a distinguishing sequence of maximal distinguishing power. Thus, the fault activation phase can be omitted, because the effects of the target fault have already reached at least one flip-flop, and the fault propagation phase can be entered immediately. The target fault selected in this manner has a higher probability of detection. If no fault-effect has reached any flip-flop having a distinguishing sequence, selection of the target fault is biased toward the fault whose effects have reached the greatest number of flip-flops. However, the fault activation phase is not omitted in this case.

C 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 [20] is used to speed up the process. During fitness evaluation, 32 candidate sequences from the population are simulated simultaneously, with values bit-packed into 32-bit words during simulation. Fault-free simulation is first performed, followed by 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 [16] is used after the first GA stage to identify and remove many of the untestable faults. A small time limit of 0.4 seconds per fault is used in an initial HITEC pass through the fault list to minimize the execution time. If a large number of untestable faults are identified or if only a small number of faults remain in the fault list, a second HITEC pass with a time limit of 2 seconds per fault is used. Any test sequences generated by HITEC are discarded.

VI Experimental Results

The new test generator, STRATEGATE, was implemented in C++; both ISCAS89 sequential benchmark circuits and several synthesized circuits were used to evaluate its performance on an HP 9000 J200 with 256 MB RAM. The characteristics of the synthesized circuits are discussed in [5, 14]. STRATEGATE is compared to various other test generators in Table 1. 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 states visited

Table 1: STRATEGATE Results

Circuit	Total Faults	HITEC[16]		GATEST[9]		CRIS[6]		DIGATE[5]		STRATEGATE		
		Det	Vec	Det	Vec	Det	Vec	Det	Vec	Det	Vec	States
s298	308	**265**	306	264	161	253	476	264	239	**265**	306	154
s344	342	328	142	**329**	95	328	115	**329**	109	**329**	86	272
s382	399	363	4931	347	281	273	246	363	581	**364**	1486	1159
s400	426	383	4309	365	280	357	758	382	3369	**384**	2424	1954
s444	474	414	2240	405	275	397	519	420	1393	**424**	1945	1085
s526	555	365	2232	417	281	428	692	446	2867	**454**	2642	1764
s641	467	**404**	216	**404**	139	398	628	**404**	180	**404**	166	115
s713	581	**476**	194	**476**	128	475	1124	**476**	147	**476**	176	109
s820	850	813	984	517	146	451	1381	621	465	**814**	590	25
s832	870	817	981	539	150	370	1328	606	703	**818**	701	25
s1196	1242	**1239**	453	1232	347	1180	2744	1236	549	**1239**	574	386
s1238	1355	**1283**	478	1274	383	1229	4313	1281	504	1282	624	406
s1423	1515	776	177	1222	663	1167	2696	1393	4044	**1414**	3943	3605
s1488	1486	**1444**	1294	1392	243	1355	1960	1378	542	**1444**	593	48
s1494	1506	**1453**	1407	1416	245	1357	1928	1354	581	**1453**	540	48
s5378	4603	3238	941	3175	511	3029	1255	3447	10500	**3639**	11571	9550
s35932	39094	34902	240	35009	197	34481	1525	**35100**	386	**35100**	257	195
am2910	2391	2164	874	2163	745	-	-	2195	2206	**2198**	2509	2233
mult16	1708	1640	273	1653	204	-	-	1664	915	**1665**	1530	943
div16	2147	1665	189	1739	634	-	-	1802	4481	**1814**	3476	2425
pcont2	11300	3354	7	6826	272	-	-	**6837**	3452	**6837**	194	152
piir8o	19920	14221	347	15013	531	-	-	**15072**	506	15071	354	305
piir8	29689	11131	31	-	-	-	-	18140	603	**18206**	443	326

Det: number of faults detected Vec: test set length States: number of states visited

by STRATEGATE is also reported for each circuit. The first test generator is HITEC [16], a deterministic test generator, followed by the GA-based test generators GATEST [9], CRIS [6], DIGATE [5], and finally STRATEGATE.

Fault coverage is defined as the percentage of total faults detected. The best fault coverages are highlighted in bold. As shown in the table, the fault coverages achieved by STRATEGATE match or surpass those obtained by all other test generators for all circuits except two, s1238 and piir8o, where only one less fault was detected. In many cases the fault coverages obtained by STRATEGATE are significantly higher. For the hard-to-test circuits, such as s444, s526, s1423, s5378, s35932, and the synthesized circuits, where long execution times are required by HITEC, the fault coverages achieved by STRATEGATE are much higher, and execution times are much shorter. Even in the circuits where previous GA-based test generators did not perform well, such as s820, s832, s1488, and s1494, STRATEGATE is able to detect all of the detectable faults. These four 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; STRATEGATE, however, is able to reach all the required states via dynamic state traversal. STRATEGATE is able to visit more states than HITEC in the larger circuits where higher fault coverages are obtained. The test sets obtained by STRATEGATE are more compact than those obtained by HITEC, even when higher fault coverages are achieved by STRATEGATE. The test sets are more compact than those obtained by CRIS or DIGATE for most of the circuits.

STRATEGATE achieves very high fault coverages very quickly using a small number of vectors. This phenomenon is illustrated in Table 2 for twelve circuits at different checkpoints 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 checkpoint, number of faults detected, test set size, and execution time are displayed in Table 2 for each circuit. The fault coverages at the end of the first or second GA stages are already higher than the final fault coverages of the other test generators for many circuits. For example, STRATEGATE detects 1410 faults in 13.2 minutes for circuit s1423; all other test generators spend hours of execution time and still do not reach this fault coverage. This phenomenon is consistent for many circuits shown in the table. 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.

VII Conclusions

A test generation framework which utilizes dynamic state traversal for targeting hard-to-test faults was presented. 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; single-time-frame fault activation and state justification using dynamic state traversal are performed for hard-to-test faults. The second phase drives the fault effects from the flip-flops to the PO's with the aid of distinguishing sequences. The dynamic state-transfer and distinguishing sequences are seeded in a GA to evolve valid state justification and fault propagation sequences, respectively, for the target fault. The

Table 2: Results at Various Checkpoints for STRATEGATE

Circuit	Ckpt	Det	Vec	Time	Circuit	Ckpt	Det	Vec	Time
s382	1	361	601	1.07 min	s1423	1	**1410**	2065	13.2 min
	2	362	1285	5.9 min		2	**1410**	2965	40.1 min
	3	**364**	1486	8.1 min		3	**1414**	3943	1.27 hr
s444	1	408	354	38.5 sec	s1494	1	1393	295	5.34 min
	2	**420**	753	2.3 min		2	**1453**	540	7.50 min
	3	**424**	1945	20.1 min		3	**1453**	540	7.60 min
s526	1	431	486	1.37 sec	s5378	1	**3562**	2175	4.60 hr
	2	442	1098	8.3 min		2	**3607**	4461	25.1 hr
	3	**454**	2642	54.5 min		3	**3639**	11571	37.8 hr
s713	1	475	157	1.1 min	s35932	1	**35100**	257	10.1 hr
	2	**476**	176	1.30 min		2	**35100**	257	10.2 hr
	3	**476**	176	1.31 min		3	**35100**	257	10.9 hr
s820	1	812	572	3.07 min	am2910	1	2190	953	6.25 min
	2	**814**	590	3.60 min		2	**2197**	1761	13.5 min
	3	**814**	590	3.63 min		3	**2198**	2509	29.4 min
s1196	1	1235	521	1.12 min	div16	1	1727	352	32.0 min
	2	1237	536	1.21 min		2	**1810**	1168	2.62 hr
	3	**1239**	574	1.49 min		3	**1814**	3476	8.1 hr

Check Point k: end of GA stage k

GA is able to combine the various sequences, which are partial solutions to the problem, to engineer a complete solution. 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 approaches, especially for the larger circuits and for circuits having hard-to-activate faults. More than 30% improvement in fault coverages were obtained for some circuits.

References

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

[2] H. Cho, S.-W. Jeong, and F. Somenzi, "Synchronizing sequences and symbolic traversal techniques in test generation," *Journal of Electronic Testing: Theory and Application*, vol. 4, no. 1, pp. 19-31, 1993.

[3] I. Pomeranz and S. M. Reddy, "Application of homing sequences to synchronous sequential circuit testing," *IEEE Trans. Computers*, vol. 43, no. 5, pp. 569-580, 1994.

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

[5] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Automatic test generation using genetically-engineered distinguishing sequences," *Proc. VLSI Test Symp.*, pp. 216-223, 1996.

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

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

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

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

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

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

[12] E. M. Rudnick and J. H. Patel, "Combining deterministic and genetic approaches for sequential circuit test generation," *Proc. Design Automation Conf.*, 1995.

[13] E. M. Rudnick and J. H. Patel, "State justification using genetic algorithms in sequential circuit test generation," Coordinated Science Laboratory, University of Illinois, Urbana, IL, Tech. Report CRHC-96-01/UILU-ENG-96-2201, Jan. 1996.

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

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

[16] T. M. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," *Proc. European Conf. Design Automation (EDAC)*, pp. 214-218, 1991.

[17] T. M. Niermann and J. H. Patel, "Method for automatically generating test vectors for digital integrated circuits," *U.S. Patent No.* 5,377,197, December, 1994.

[18] I. Pomeranz and S. M. Reddy, "LOCSTEP: A logic simulation based test generation procedure," *Proc. Fault Tolerant Computing Symp.*, June, 1995.

[19] T. E. Marchok, Aiman El-Maleh, W. Maly and J. Rajski, "Complexity of sequential ATPG," *Proc. European Design and Test Conf.*, March 1995.

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