Fast Sequential Circuit Test Generation Using High-Level and Gate-Level Techniques

Elizabeth M. Rudnick[†] Roberto Vietti[††] Akilah Ellis[†]
Fulvio Corno[††] Paolo Prinetto[††] Matteo Sonza Reorda[††]

[†]Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL USA
[††]Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy

Abstract

A new approach for sequential circuit test generation is proposed that combines software testing based techniques at the high level with test enhancement techniques at the gate level. Several sequences are derived to ensure 100% coverage of all statements in a high-level VHDL description, or to maximize coverage of paths. The sequences are then enhanced at the gate level to maximize coverage of single stuck-at faults. High fault coverages have been achieved very quickly on several benchmark circuits using this approach.

1 Introduction

Most recent work in the area of sequential circuit test generation has focused on the gate level and has been targeted at single stuck-at faults. Both deterministic fault-oriented and simulation-based approaches have been used effectively, although execution times are often long. The key factor limiting the efficiency of these approaches has been the lack of knowledge about circuit behavior. Architectural-level test generation has been proposed as a means of exploiting high-level information while maintaining the capability to handle stuck-at faults [1]. However, the high-level information must be derived from a structural description at the register transfer level (RTL), and sequences generated are targeted at detecting specific stuck-at faults in modules for which gate-level descriptions are available. Circuits with modules for which gate-level descriptions are not available can be handled, but better fault coverages are obtained by a gate-level test generator in less time [2].

Several approaches have been proposed for automatic generation of functional test vectors for circuits described at a high level, including [3][4][5]. The functional test vectors can be used for design verification and power estimation, in addition to screening for manufacturing defects. Vemuri and Kalyanaraman enumerate paths in an annotated VHDL description and translate them into a set of constraints [3]. A constraint solver is then used to obtain a test sequence to traverse the specified path. Fault coverages of test sets generated for statement coverage were low, but higher fault coverages were obtained by covering each statement multiple times. Cheng and Krishnakumar transform the high-level description in VHDL or C into an extended finite state machine (EFSM) model and then use the EFSM model to generate test sequences that exercise all specified functions [4]. Traversing all transitions in an EFSM model was shown to guarantee coverage of all functions. Execution time was very low for generation of test sequences, and good fault coverages were achieved for several circuits. The approach proposed by Corno et al., implemented in the test generator RAGE, aims to generate test sequences that cover each *read* or *write* operation on a variable in a high-level VHDL description [5]. The operations are each covered a specified number of times. Good fault coverages were achieved for several benchmark circuits by covering each operation at least three times, and fault coverages were sometimes higher than those obtained by a deterministic, gate-level test generator. Execution was fast for all but the larger circuits.

These functional test generation approaches are based upon a technique commonly used for software testing: generating tests that cover all statements in the system description. Another software testing technique, which has not been implemented in the previous work on functional test generation, is to generate tests that traverse all possible paths in the system description [6]. In this work, we address path coverage, as well as statement coverage. A VHDL circuit description may contain multiple processes that execute concurrently. Since a path is defined only *within* a single process, we apply path coverage to single-process designs or to the main process of multiple-process designs only. Here, limitations must be placed upon path length to bound the number of tests generated. Generation of tests for path coverage, in addition to statement coverage, may enable higher fault coverages to be achieved.

[*]This research was supported in part by DARPA under Contract DABT63-95-C-0069, in part by the European Union through the FOST Project, and by Hewlett-Packard under an equipment grant.

Whether statement coverage or path coverage is used as the coverage metric, generation of test sequences using software testing based techniques is limited by an inability to specify values of variables that will maximize detection of faults at the gate level. We propose to combine a software testing based approach at the high level with test sequence enhancement techniques at the gate level to achieve high fault coverages in sequential circuits very quickly. The gate-level test sequence enhancement techniques that we use borrow from techniques already developed for dynamic compaction of tests generated at the gate level [7][8]. The objective is to maximize the number of faults that can be detected by each test sequence generated at the high level.

We begin with an overview of the test generation process in Section 2. Generation of test sequences at the high level using software testing based techniques is then described in Section 3, followed by a discussion about test sequence enhancement at the gate level in Section 4. Results are presented in Section 5 for several benchmark circuits, and Section 6 concludes the paper.

2 Overview

We propose to combine software testing based techniques for test sequence generation at a high level with gate-level techniques for test sequence enhancement. The overall test generation process is illustrated in Figure 1. Several partially-specified test sequences are derived from the high-level circuit description using various coverage goals, e.g., coverage of all statements. An automatic synthesis tool is used to obtain a gate-level implementation of the circuit, and then the gate-level test sequence enhancement tool is executed to generate a complete test set targeted at high coverage of single stuck-at faults, using test sequences generated at the high level as input. The high-level sequences generated are aimed at traversing through a number of control states in the system, and values of variables are left unspecified as much as possible. The gate-level tool then has more freedom to select values that will maximize fault coverage. The same sequence may be reused a number of times, but modifications made at the gate level, which essentially specify the values of variables in the datapath, are likely to result in different fully-specified test sequences. Furthermore, any sequences or subsequences that do not contribute to improving the fault coverage are not added to the test set.

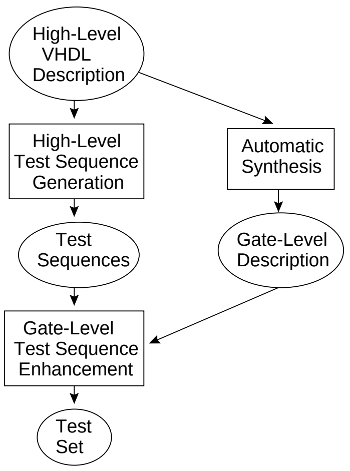
High-Level VHDL Description
High-Level Test Sequence Generation
Automatic Synthesis
Test Sequences
Gate-Level Description
Gate-Level Test Sequence Enhancement
Test Set

Figure 1: Overview of test generation.

3 High-Level Test Generation

The first step in our test generation procedure is to obtain a set of partially-specified test sequences using the high-level circuit description and various coverage goals. Ideally, we would like to automate this process, but automatic generation of tests for both statement and path coverage is itself a very difficult problem, and no implementation is currently available. Therefore, in the current work, the sequences are derived manually. One of our goals in this work is to provide guidelines on the types of high-level sequences that are most useful for stuck-at fault testing. It may be possible to avoid using sequences that are difficult to derive automatically and still achieve high fault coverages. In particular, our experiments indicate that statement coverage usually suffices and is easier to achieve than path coverage.

Various high-level benchmark circuits are used in our work, and most of these have been derived from VHDL descriptions found at various ftp sites. Circuits *b01–b08* range from simple filters to more complex microprocessor fetch and execution units and are available from the authors.

The simplest coverage metric is statement coverage. A test set with 100% statement coverage exercises all statements in the VHDL description. Every branch must be exercised at least once in the set of sequences derived, but all paths are not necessarily taken. Path coverage is a more comprehensive metric that does aim to ensure that all paths are taken. To obtain a set of sequences with 100% statement coverage, the datapath and control portions of the description are identified, and the state transition graph (STG) for the control machine is derived. Then test sequences are assembled to traverse all control states and all blocks of code for each state. Each sequence begins by resetting the circuit. In the benchmark circuits that we are using, a reset signal is available. However, the only necessary assumption is that the circuit is initializable. This assumption is satisfied at the gate level by either using a reset signal or an initialization sequence. Several vectors are then added to traverse between states and exercise various statements. Finally, vectors are added to the end of the sequence to ensure that the circuit ends in a state in which the output is observed. For many circuits, the outputs are observable in any state, so these vectors are unnecessary. Portions of the test sequences that determine the values of variables are left unspecified as much as possible so that the gate-level tool has more freedom in choosing values to maximize stuck-at fault coverage.

Derivation of test sequences for 100% statement coverage is best illustrated by an example. The STG for the control machine of benchmark circuit *b03* is shown in Figure 2. When the reset signal is asserted, the circuit is placed in state *INIT*. In state *INIT*, the (bit) variables *request1* through *request4* are read from the primary inputs, and the next state is set to *ANALISI_REQ*. In state *ANALISI_REQ*, the 4-bit *grant* variable is written to the primary outputs, one of four blocks of code is executed, depending on the *request* variables read in the previous state, and the next state is set to *ASSIGN*. In state *ASSIGN*, the *grant* variable is updated, the variables *request1* through *request4* are read from the primary inputs, and the next state is set to *ANALISI_REQ*. The set of test sequences derived for 100% statement coverage therefore contains four partially-specified sequences, each having five vectors. The first vector resets the circuit. The second vector sets the *request* variables to exercise one of the four code blocks in the following state. The last three vectors are used to traverse from the *ANALISI_REQ* state to the *ASSIGN* state, where the *grant* variable is set, and back to the *ANALISI_REQ* state, where the *grant* variable is written to the primary outputs.

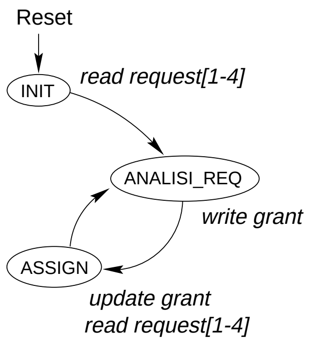

Figure 2: STG for benchmark circuit *b03*.

In obtaining a set of sequences for path coverage, we start with a sequence with 100% statement coverage. Several sequences are then added to maximize coverage of paths. Paths are considered within each state of the STG and also across several states. The procedure for deriving sequences to cover paths within a state is first explained for benchmark circuit *b04*. The STG for *b04* is shown in Figure 3 along with a flow chart for state *sC*. All assignments in the flow chart are carried out in the same clock cycle. When the circuit is reset, state *sA* is entered. States *sB* and *sC* are reached in the next two clock cycles, regardless of the inputs, as long as the reset line is not asserted. No particular patterns are needed to reach all statements and to cover all paths in states *sA* and *sB*. However, many paths are possible in state *sC*. The circuit must be in state *sC* for a minimum of four clock cycles to exercise all statements at least once. Either four separate sequences or one long sequence can be used. We have opted to use a larger number of shorter sequences in order to provide more sequences for optimization by the gate-level tool. Fifteen sequences are needed to cover all paths. (Note that the *ENA* variable is used at two separate decision points.) Only five sequences are needed if the last two decision points are not considered. We consider the 8-bit variable *D_IN* used in the last two decision points to be part of the datapath, and therefore, in our experiments, we have left specification of values for this variable to the gate-level tool.

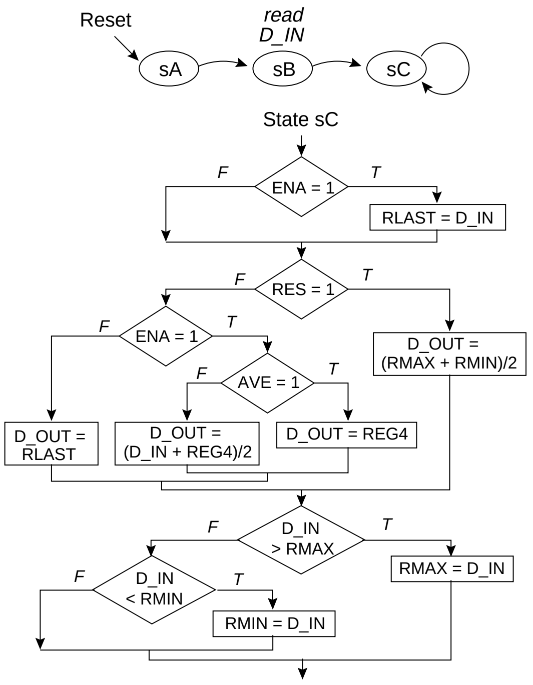

Figure 3: STG for benchmark circuit *b04* and flow chart for state *sC*.

Paths that occur across multiple states must also be considered. Consider the STG for the control unit of benchmark circuit *b06* shown in Figure 4. This STG contains several cycles. In order to limit the number of sequences derived, we place restrictions on sequences that traverse a cycle. Self-loops are traversed at most once in any sequence, and for other cycles, the

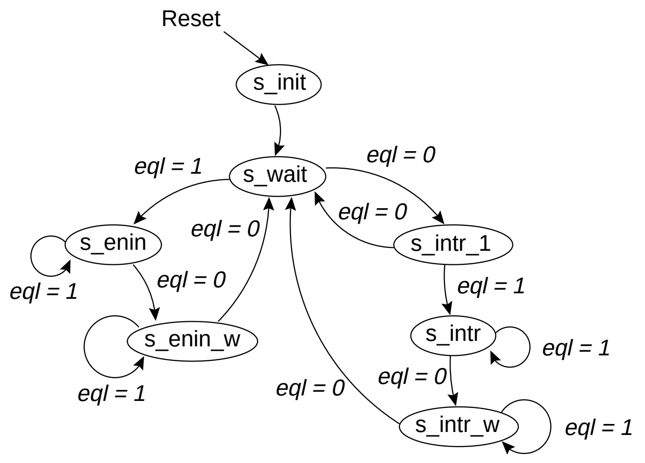

Figure 4: STG for benchmark circuit *b06*.

sequences are terminated when a state is repeated. Four sequences are required to fully cover paths involving states *s_wait*, *s_enin*, and *s_enin_w*. Five sequences are needed to cover paths involving states *s_wait*, *s_intr_1*, *s_intr*, and *s_intr_w*. Nine sequences are thus required for path coverage.

In general, both the STG and the statement flow for each reached state must be considered when deriving sequences for path coverage.

4 Gate-Level Test Enhancement

Functional tests generated at a high level are effective in traversing through much of the control space of a machine. However, they cannot exercise all values of variables, except in very small circuits, due to the large number of possible values. Selecting good values to use at the high level is an unsolved problem, and a gate-level approach may be more effective in finding values that exercise potential faults.

4.1 Architecture of Gate-Level Tool

Our gate-level test enhancement tool repeatedly selects a partially-specified sequence provided by the high-level test generator and attempts to evolve a fully-specified sequence that maximizes fault coverage. The number of times that test sequence evolution is attempted is a parameter specified by the user. Sequences may be selected randomly or sequentially from the list of sequences provided by the high-level test generator. If sequences are selected randomly, a random number generator is used to decide which sequence to select. Random selection does not guarantee that every sequence will be used, but it does not restrict the order in which the sequences are selected. If sequences are selected sequentially, the first sequence is selected first, the second sequence is selected second, and so on. Every sequence will be selected at least once if the number of attempts at test sequence evolution is greater than or equal to the number of sequences.

The main function of the gate-level test enhancement tool is to repeatedly solve an optimization problem: maximizing the number of faults detected by each sequence. Genetic algorithms (GAs) have been used effectively for many different optimization problems, including sequential circuit test generation [9]–[11],[2]. Thus, we use a GA for test sequence enhancement. We simply seed the GA with a sequence obtained at the high-level and then set the GA fitness function to maximize fault detection. The GA will explore several alternative sequences through a number of generations, and the best one is added to the test set if it improves the fault coverage. Any vectors at the end of the sequence that do not contribute to the fault coverage are removed. Then the next high-level sequence is selected, and the genetic enhancement procedure is repeated. This process continues until the number of attempts at test sequence enhancement reaches the user-specified limit.

4.2 A GA for Test Sequence Enhancement

In this work, we use a simple GA, rather than a steady-state GA [12], since exploration of the search space is paramount. The simple GA contains a population of *strings*, or individuals [13]. In our application, each individual represents a test sequence, with successive vectors in the sequence placed in adjacent positions along the string. Each individual has an associated *fitness*, and in our application, the fitness measure indicates the number of faults detected by each sequence. The population is initialized with a set of sequences derived from a single sequence generated at the high level, and the evolutionary processes of *selection*, *crossover*, and *mutation* are used to generate an entirely new population from the existing population. This process is repeated for several generations. 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. The two new individuals are then placed in the new population, and this process continues until the new generation is entirely filled. Binary tournament selection without replacement and uniform crossover are used, as was done previously for gate-level test generation [10]. The goal of the evolutionary process is to improve the fitness of the best individual in each successive generation by combining the good portions of fit individuals from the preceding generation. However the best individual may appear in any generation, so we save the best individual found.

The GA is seeded with copies of the partially-specified test sequence provided by the high-level test generator. The specified bits are the same for every individual. Bits that are not specified are filled randomly. Each fully-specified test sequence is then fault simulated to obtain its fitness value; the fitness value measures the quality of the corresponding solution, primarily in terms of fault coverage. The GA is evolved over several generations, and by the time the last generation is reached, several of the values specified by the high-level test generator may have changed in many of the individuals due to the mutation operator; i.e., the sequences may no longer be covered by the original partially-specified sequence. However, such a sequence will only be added to the test set if it covers some additional faults not already covered by previous vectors in the test set and if it has the highest fault coverage.

4.3 Fitness Function

The PROOFS sequential circuit fault simulator [14] is used to evaluate the fitness of each candidate test sequence and again to update the state of the circuit after the best test sequence is selected. The number of faults detected is the primary metric in the fitness function, since the objective of the GA is to maximize the number of faults detected by a given test sequence. To differentiate test sequences that detect the same number of faults, we include the number of fault effects propagated to flip-flops in the fitness function, since fault effects at the flip-flops may be propagated to the primary outputs in subsequent time frames. However, the number of fault effects propagated is offset by the number of faults simulated and the number of flip-flops to ensure that the number of faults detected is the dominant factor in the fitness function:

$$\begin{aligned} fitness &= \#\ faults\ detected \\ &+ \frac{\#\ fault\ effects\ propagated\ to\ flip\ flops}{(\#\ faults\ simulated)(\#\ flip\ flops)} \end{aligned}$$

While an accurate fitness function is essential in achieving a good solution, the high computational cost of fault simulation may be prohibitive, especially for large circuits. To avoid excessive computations, we can approximate the fitness of a candidate test by using a small random sample of faults. In this work, we use a sample size of about 100 faults if the number of faults remaining in the fault list is greater than 100.

5 Results

Experiments were carried out to evaluate the proposed approach for combining high-level and gate-level techniques for sequential circuit test generation. Test sequences were derived manually at the high level by extracting the STG of the control machine and then ensuring that all VHDL statements or paths were covered within each control state. For *diffeq*, a short C program was written to assist in obtaining high-level sequences. This circuit contains a single loop, and the loop must be exited to observe the output. The C program was used to determine the number of loop iterations executed for a given input. Gate-level implementations of the circuits were synthesized using a commercial synthesis tool. Test sequence enhancement was then performed at the gate level using a new GA-based tool implemented using the existing PROOFS [14] source code and 2100 additional lines of C++ code. A small GA population size of 32 was used, and the number of generations was limited to 8 to minimize execution time. Nonoverlapping generations and crossover and mutation probabilities of 1 and 1/64 were used.

Tests were generated for several high-level benchmark circuits on an HP 9000 J200 with 256 MB memory. Characteristics of the benchmark circuits are summarized in Table 1, including number of VHDL lines in the high-level description, number of control states, number of logic gates in the gate-level circuit, number of flip-flops (**FFs**), number of primary inputs (**PIs**), number of primary outputs (**POs**), and number of collapsed faults. Circuits *b01*–*b08* have been used previously for research on functional test generation [5]. Circuits *barcode*, *gcd*, *dhrc*, and *diffeq* were taken from the HLSynth92 and HLSynth95 high-level synthesis benchmarks. All circuits were translated into a synthesizable subset of VHDL before they were used.

Test generation results are shown in Table 2 for sequences derived at the high level to maximize path coverage. Results for HITEC [15], a gate-level deterministic test generator, and GATEST [10], a gate-level GA-based test generator, are also shown for comparison. Three passes through the fault list were made by HITEC for all circuits unless all faults were identified as detected or untestable earlier. Time limits for the three passes were 0.5, 5, and 50 seconds per fault. For each circuit, the number of faults detected (**Det**), the number of test vectors generated (**Vec**), and the execution time are shown for each test generator. The execution time for the proposed approach includes the time for gate-level test enhancement only, but the time for generating sequences from high-level circuit descriptions is expected to be of the same order of magnitude, based on previous work [5]. The number of attempts at generating a useful test sequence (**Seq**) and the sequence selection strategy (**Strat**), whether sequential or random, are also shown in the table, as well as the number of faults identified as untestable by HITEC. Results are shown for the sequence selection strategy and number of attempts that gave the highest fault coverage, while using a minimal number of test vectors. If more

Table 1: High-Level Benchmark Circuits

Circuit	High Level		Gate Level				
	VHDL Lines	Control States	Gates	Flip-Flops	PIs	POs	Faults
b01	102	8	60	5	3	2	135
b02	70	7	37	4	2	1	72
b03	134	3	210	30	5	4	452
b04	79	3	676	66	12	8	1396
b05	297	5	892	34	2	36	1884
b06	127	7	92	9	3	6	206
b07	92	7	600	51	2	8	1271
b08	88	4	210	21	10	4	489
barcode	97	4	617	46	12	18	1091
gcd	44	1	1191	49	33	16	2199
dhrc	135	2	4420	202	65	8	9468
diffeq	66	1	9340	129	81	48	18,216

Table 2: Combining High-Level Test Generation with Gate-Level Test Enhancement

Circuit	High-Level + Gate-Level					HITEC				GATEST		
	Det	Vec	Time	Seq	Strat	Det	Vec	Time	Unt	Det	Vec	Time
b01	133	44	6.02s	10	seq	133	110	0.50s	2	133	80	12.0s
b02	69	27	5.80s	20	seq	69	54	0.29s	3	70	69	10.1s
b03	334	105	46.3s	20	rand	333	214	1.25h	41	334	169	1.12m
b04	1204	113	1.17m	20	rand	1177	303	1.42h	136	1217	220	4.60m
b05	905	85	2.36m	20	rand	913	396	11.9h	236	902	129	1.41m
b06	190	31	17.7s	20	seq	190	89	0.89s	16	190	62	19.6s
b07	888	100	6.80m	20	seq	878	206	4.87h	140	871	88	1.39m
b08	311	54	1.37m	40	seq	461	563	1.16m	28	261	84	46.3s
barcode	580	77	1.68m	20	rand	689	1816	28.7h	12	552	161	4.52m
gcd	1988	356	17.8m	90	rand	1638	206	13.7h	3	1377	227	10.6m
dhrc	8861	317	48.9m	60	seq	8864	1094	15.2h	150	8860	820	1.55h
diffeq	17,881	335	1.80h	100	rand	17,730	803	23.6h	46	18,009	662	7.71h

attempts are made at test sequence enhancement, the execution time will increase, but higher fault coverages were not achieved in our experiments.

For most circuits, the fault coverages for the proposed approach are competitive with the fault coverages achieved by HITEC. For *barcode*, the fault coverage is about the same as that achieved by HITEC after two passes through the fault list and 51.1 minutes of execution, although more faults are detected by HITEC in the third pass. For *b08*, HITEC achieves higher fault coverage in the first pass. In some cases, such as *b04*, *b07*, and *gcd*, higher fault coverages are obtained by combining the high-level and gate-level techniques. Furthermore, for a given level of fault coverage, the test sets generated using the proposed approach are much more compact. Execution times for gate-level test enhancement are often orders of magnitude smaller than those for HITEC. Nevertheless, untestable faults cannot be identified using the proposed approach. Thus, the designer may choose to run a gate-level test generator such as HITEC in a postprocessing step. Fault coverages for the proposed approach are significantly higher than those for GATEST for several circuits. For some circuits, GATEST achieves the same fault coverage as the proposed approach, but test set lengths and execution times are significantly higher. For *diffeq*, the GATEST fault coverage was higher, but execution time was also significantly higher. The gate-level test enhancement is very similar to the procedure used in GATEST, except that GATEST uses random sequences in the initial GA population. The seeds used by the gate-level test enhancement tool are critical in providing information to the GA about sequences that can activate faults and propagate fault effects.

The two sequence selection strategies are compared in Table 3 for sequences derived at the high level to maximize path coverage or for 100% statement coverage. Statement and path coverage are the same for *diffeq*, since this circuit contains only a single path. For path coverage, the sequential selection strategy gives better results in terms of fault coverage and test set size for some circuits, but in a few cases, the fault coverages are significantly higher for random selection. Random selection is therefore preferred in general. For statement

Table 3: Sequential vs. Random Selection of Sequences Derived for Path Coverage or Statement Coverage

		Path Coverage						Statement Coverage					
		Sequential			Random			Sequential			Random		
Circuit	Seq	Det	Vec	Time	Det	Vec	Time	Det	Vec	Time	Det	Vec	Time
b01	10	133	44	6.02s	133	55	6.42s	128	38	4.76s	132	39	5.08s
b02	20	69	27	5.80s	69	30	6.37s	67	20	5.37s	69	29	5.44s
b03	20	334	108	36.4s	334	105	46.3s	334	108	36.3s	334	105	46.2s
b04	20	1189	95	1.16m	1204	113	1.17m	1199	91	1.26m	1200	104	1.28m
b05	20	153	31	2.55m	905	85	2.36m	232	71	2.48m	232	71	2.48m
b06	20	190	31	17.7s	190	41	17.8s	190	37	33.1s	190	30	33.1s
b07	20	888	100	6.80m	887	97	6.81m	877	67	6.45m	877	67	6.46m
b08	40	311	54	1.37m	301	43	1.50m	311	54	1.34m	302	40	1.47m
barcode	20	573	91	1.93m	580	77	1.68m	575	110	3.36m	575	110	3.35m
gcd	90	1914	302	18.6m	1988	356	17.8m	1662	304	16.8m	1769	283	13.6m
dhrc	60	8861	317	48.9m	8843	312	51.6m	8860	404	1.29h	8860	404	1.24h
diffeq	100	17,881	335	1.79h	17,881	335	1.80h	17,881	335	1.79h	17,881	335	1.80h

coverage, the random selection strategy tends to give fault coverages that are as good as or better than those for sequential selection. Fault coverages are sometimes higher than those for sequences derived for path coverage. However, fault coverages may be significantly lower, as is the case for circuit *b05.* These results are not unexpected, since certain paths may need to be traversed in order to excite some faults and propagate their effects to the primary outputs. Nevertheless, since good results are often obtained for sequences derived for 100% statement coverage alone, and these sequences are easier to derive, this approach may be preferred.

6 Conclusions

High fault coverages have been obtained very quickly by combining high-level and gate-level techniques for test generation. Sequences derived to maximize coverage of statements or paths in the high-level VHDL description are enhanced at the gate level to maximize coverage of single stuck-at faults. This approach may be used as a preprocessing step to gate-level test generation to speed up the process, and it sometimes results in improved fault coverages as well. Higher fault coverages were obtained for sequences derived for path coverage, but good results were also obtained for 100% statement coverage. A random selection of sequences for gate-level enhancement was shown to provide consistently good results.

References

[1] J. Lee and J. H. Patel "Architectural level test generation for microprocessors," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 10, pp. 1288–1300, Oct. 1994.

[2] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Sequential circuit test generation using dynamic state traversal," *Proc. European Design and Test Conf.*, pp. 22–28, 1997.

[3] R. Vemuri and R. Kalyanaraman, "Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming," *IEEE Trans. VLSI Systems*, vol. 3, no. 2, pp. 201–214, June 1995.

[4] K.- T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Trans. Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 57–79, Jan. 1996.

[5] F. Corno, P. Prinetto, and M. Sonza Reorda, "Testability analysis and ATPG on behavioral RT-level VHDL," *Proc. Int. Test Conf.*, Nov. 1997.

[6] M. W. Johnson, "High level test generation using software metrics," M.S. thesis, Department of Electrical and Computer Engineering, Tech. Report CRHC-95-06/UILU-ENG-95-2204, University of Illinois, Feb. 1995.

[7] E. M. Rudnick and J. H. Patel, "Simulation-based techniques for dynamic test sequence compaction," *Proc. Int. Conf. Computer-Aided Design*, pp. 67–73, 1996.

[8] E. M. Rudnick and J. H. Patel, "Putting the squeeze on test sequences," *Proc. Int. Test Conf.*, pp. 723–732, Nov. 1997.

[9] 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, Nov. 1992.

[10] 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.

[11] F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "GATTO: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuits," *IEEE Trans. Computer-Aided Design*, vol. 15, no. 8, pp. 991–1000, Aug. 1996.

[12] J. H. Holland, *Adaptation in Natural and Artificial Systems,* Ann Arbor, MI: University of Michigan Press, 1975.

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

[14] T. M. Niermann, W. -T. Cheng, and J. H. Patel, "PROOFS: A fast, memory-efficient sequential circuit fault simulator," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 2, pp. 198–207, Feb. 1992.

[15] 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.