Computer Science
Technical Report

Colorado State University

Antirandom Test Patterns Generation Tool

Huifang Yin
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
Fall, 1996

Technical Report CS-98-101

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Antirandom Test Patterns Generation Tool

Huifang Yin
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
Fall, 1996

ABSTRACT

Random testing is a well known concept that each test is selected randomly regardless of the test previously applied. This paper introduces the antirandom testing which each test applied is chosen such that its total distance from all previous tests is maximum. And one automatic test patterns generation tool using this concept is implemented. At the same time, one popular test generation and simulation tool Nemesis is used to simulate the test patterns generated from Random testing, Antirandom testing and Nemesis which used the Boolean satisfiability method, by comparing their fault coverage for stuck-at and bridgeIDDQ in some Benchmark circuits, the advantages and disadvantages in these three methods are obvious, the suitable situation for each method can be easily found.

1 Introduction

To produce reliable computer systems, defect-free components must be available. In the overall hardware development time, the testing time is a significant fraction. How to make the testing as efficient as possible is a big challenging task for testers. For minimize the testing time, finding an efficient algorithm for the test generation is very important [4].

Random testing and its variations have been extensively used and studied for hardware systems. However, *random testing* does not exploit some information like the previous tests applied. It is very difficult to ascertain the fault coverage that is provided by the test procedure, and for large circuits, it is really not efficient.

If an experienced tester is generating tests by hand, he would select each new test such that it covers some part of the functionality not yet covered by tests already generated. So Dr. Malaiya gave this algorithm one formal definition **Antirandom** test generation algorithm. In this project, I implement this algorithm in one automatic test patterns generation tool **ARTG** (see appendix) and compare it with the *Random testing* and the Boolean satisfiability method used in *Nemesis*.

Section 2, Some definitions in Antirandom Test Patterns Generation Tool.

2.1. Some terminologies in Antirandom tool.

2.2. The antirandom algorithm definition.

Section 3, The algorithm description.

3.1. Construction of MHDATS and MCDATS.

3.2. The theorems used in tool for reducing the computational complexity.

Section 4, Brief introduction to Nemesis.
4.1. The Boolean satisfiability method.
4.2. Terminologies in Nemesis.
4.3. Benchmark Circuits used in this project
4.4. Some important files in Nemesis

Section 5, The comparison of Antirandom, Random and Nemesis.

Section 6, The analysis for the fault coverage diagrams.

Section 7, Conclusion.

2 Some definitions in Antirandom Test patterns Generation Tool

Dr. Malaiya formally define the *antirandom* testing algorithm using two kinds of distances- *Hamming distance* and *Cartesian distance* as the measures of difference [1]. Then he proposed the *antirandom* algorithm to construct the test sequences.

2.1 Some terminology in Antirandom Tool

Antirandom test sequence (ATS): A test sequence such that a test t_i is chosen such that it satisfies some criterion with respect to all tests $t_0, t_1, ..., t_{i-1}$ applied before.

Distance: A measure of how different two vectors t_i and t_j are. Here we use two measures of distance defined below.

Hamming Distance (HD): The number of bits in which two binary vectors differ. It is not defined for vectors containing continuous values.

Cartesian Distance (CD): Between the two vectors, $A = a_N, a_{N-1}, ..., a_1, a_0$ and $B = b_N, b_{N-1}, ..., b_1, b_0$, if all variables in the two vectors are binary, it is given by:

$$CD(A, B) = \sqrt{(HD(A, B)}$$

Total Hamming Distance (THD): For any vector, the sum of its Hamming distances with respect to all previous vectors.

Total Cartesian Distance (TCD): For any vector, the sum of its Cartesian distances with respect to all previous vectors.

2.2 The antirandom algorithm definition

Maximal Distance Antirandom Test Sequence (MDATS): A test sequence such that each test t_i is chosen to make the total distance between t_i and each of $t_0, t_1, ...t_{i-1}$ maximum, i.e:

$$TD(t_i) = D(t_i, t_0) + D(t_i, t_1) + \cdots + D(t_i, t_{i-1})$$

is maximum for all possible choices of t_i. We will use Hamming distance and Cartesian distance to construct MHDATs and MCDATs.

3 The algorithm description

In antirandom algorithm, we use maximal distance criterion, every time we attempt to find a test vector as different as possible from all previously applied vectors. Suppose the distance between two vectors is large, then the set of faults detected by one is likely to contain only a few of the faults detected by the other. I will have some experiment data to verify this hypothesis.

3.1 Construction of MHDATs and MCDATs

- For each of N input variables, assign an arbitrarily chosen value to obtain the first test vector. We will know this does not result in any loss of generality.
- To obtain each new vector, evaluate the THD (TCD) for each of the remaining combinations with respect to the combinations already chosen and choose one that gives maximal distance, let the THD and TCD are MHDAT and MCDAT. Add it to the set of selected vectors.
- Repeat step 2 until all 2^N combinations have been used.

3.2 The theorems used in tool for reducing the computational complexity

For N test vectors, we need 2^N times computation. Dr. Malaiya proposed some theorems to be used to reduce the computational complexity.

Theorem 1: A MHDATs (MCDATs) will always contain complementary pair of vectors, i.e. t_{2k} will always be followed by t_{2k+1} which is complementary for all bits in t_{2k} where $k = 1, 2, ...$ (Proof omitted here)

According to this theorem, for N test vectors, we only need 2^{N-1} times computation.

Theorem 2: Expansion of MHDATs (MCDATs).

- Start with a complete MHDATs of N variables, $X_{N-1}, X_{N-2}, ..., X_1, X_0$.
- For each vector $t_i, i = 0, 1, ...(2^{N-1})$, add an additional bit corresponding to an added variable X_N, such that t_i has the maximum total HD (CD) with respect to all previous vectors. (Formal proof is to be sought)

Theorem 3: Expansion and Unfolding of a MHDATs (MCDATs).

- Expand by adding a variable using Theorem 2. We now have the first $(2^N/2)$ vectors needed.
- Complement one of the columns and append the resulting vectors to first set of vectors obtained in Step 1.

In the antirandom test patterns generation tool, I use these three theorems to implement the antirandom algorithm, the speed is greatly increased.

4 Brief introduction to Nemesis

Nemesis is a diverse program that simulates and generates test patterns for circuits with a variety of different types of faults [2]. Nemesis generates and simulates tests for stuck-at and bridgeIDDQ faults in combinational circuits. The test patterns which Nemesis generates are done using the Boolean satisfiability method [5].

4.1 The Boolean satisfiability method

Nemesis generates test patterns in two steps:

- It constructs a formula expressing the Boolean difference between the unfaulted and faulted circuits.
- It applied a Boolean satisfiability algorithm to the resulting formula.

4.2 Terminologies in Nemesis

Test Generation: Test generation is the process of finding a test for a fault or showing that no test for that fault exists.

Test Simulation: Test simulation is the process of simulating a potential test against a list of faults.

Stuck-At Fault: A stuck-at fault is an abstraction wherein the faulty circuit behaves as if one line were permanently tied to a logic 1 or logic 0 instead of varying as a function of the circuit inputs.

Bridge Fault: A bridge fault is a short between two signal wires in the circuit which are interconnected with a logical behavior specified by a Ptbridge tta file.

BridgeIDDQ Test: For a bridge fault, any test that causes the two bridged wires to take on opposite logic values in the fault free circuit is an IDDQ test for that fault.

Configuration File: To specify the information necessary for Nemesis to run, one configuration file includes the following information is needed: Simulate or generate tests, Fault type, Root directory, Circuit directory, Faultlist directory, Test directory, Cell library and for bridge faults, a truth table directory. The circuit name can be given in command line or in the configuration file.

subsectionBenchmark Circuits used in this project

I choose two kinds of combination circuits C432 and C1908 for the experiment. There is 36 bit input in C432 circuit, there is 33 bit input in C1908 circuit. Because in antirandom test patterns generation tool, we only consider the input bits, so we ignore output here.

4.3 Some important files in Nemesis

Nemesis.log: It is an output file created by the initial run of Nemesis and written to the directory you are running Nemesis from. Every successive time Nemesis is run, the results are appended to the Nemesis.log file.

Circuit name.deteted: It contains a list of detected faults.

Circuit name.undetected: It contains a list of undetected faults.

Circuit name.tdl: It's the gate level netlist description of the fault free circuit.

Circuit name.faultlist: Nemesis uses faultlist files to determine what faults to look for when simulating or generating tests. The two formats used for faultlists are bridge and stuck-at.

Circuit name.tta: Truth table files are necessary to give a more accurate representation of what occurs at a bridge fault site. There are tta files for all MCNC cell bridge faults. These files were created by Ptbridge.

Circuit name.tests: Nemesis communicates test information in the test format. This format is used for output test files when Nemesis is generating tests, and for input test files when Nemesis is simulating tests. As a result, Nemesis may simulate tests that it has generated.

5 The comparison of Antirandom, Random and Nemesis test generation algorithms

As I said before, Nemesis is a test generation and test simulation tool. So in this project, I used this useful tool to do the following things:

- Using generate.tests configuration in Nemesis to generate test patterns for stuck-at faults and bridgeIDDQ faults for C432 and C1908.
- Using Antirandom test generation tool to generate test patterns for C432 and C1908. (Same algorithms for Stuck-At faults and bridgeIDDQ faults).
- Randomly choosing the same test set No. to construct each corresponding random test patterns.
- Using simulate.tests configuration in Nemesis to simulate these three groups of test patterns.

The experiment results are represented in the following tables and diagrams. Here are two examples which are *stuck-at* faults tested for C432 and C1908. We tested *antirandom testing*, and *random testing* with seed 0 - Random1, seed 9830160 - Random2, seed 2147483647 - Random3.

Table 1: Stuck-At faults tested in C432 (Total: 524)

Test No.	Antirandom	Random1	Random2	Random3
1	122	19	58	62
2	193	52	88	92
6	248	188	185	171
10	289	263	236	234
15	365	338	274	347
25	440	409	325	396
35	462	438	364	420
45	470	458	390	438
60	481	477	441	449

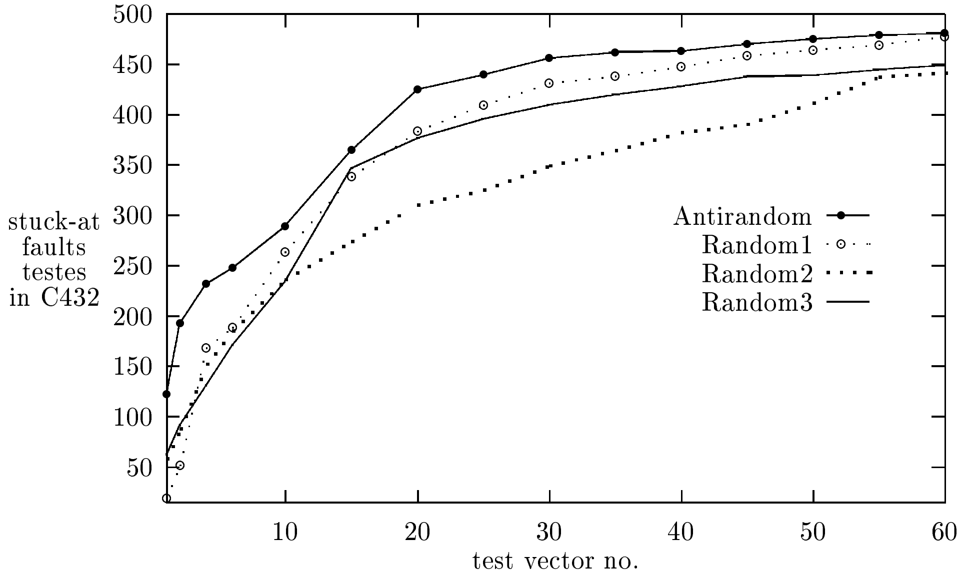

Figure 1: Stuck-At faults tested in C432

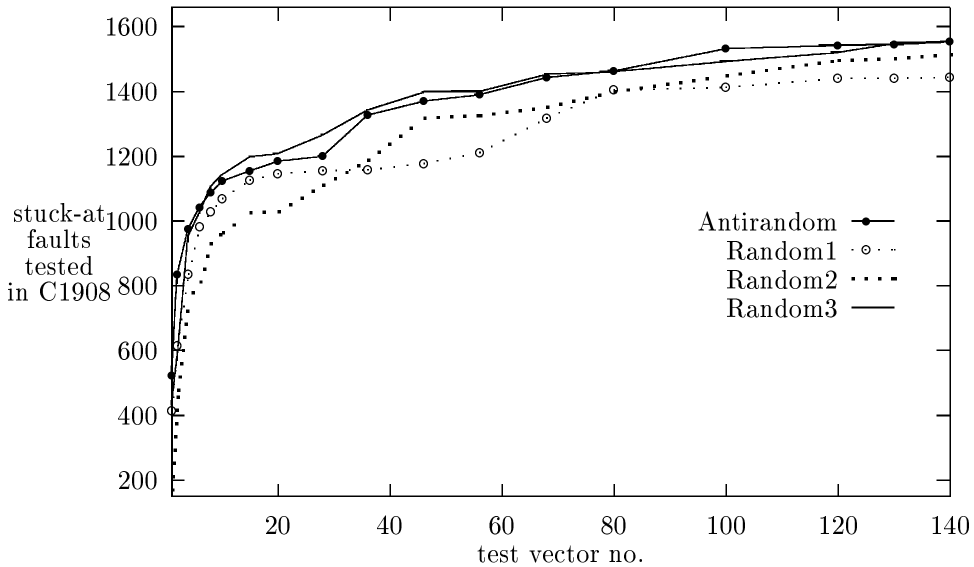

Figure 2: Stuck-At faults tested in C1908

Table 2: Stuck-At faults tested in C1908 (Total: 1879)

Test No.	Antirandom	Random1	Random2	Random3
1	523	414	169	432
2	835	615	478	583
6	1044	983	813	1028
10	1125	1070	965	1144
20	1186	1147	1030	1210
36	1328	1158	1187	1344
56	1392	1212	1326	1402
80	1464	1406	1399	1462
100	1534	1412	1448	1493
120	1543	1441	1495	1521
140	1556	1445	1513	1553

6 The analysis for the fault coverage diagrams

From the two diagrams for C432 and C1908, we can see:

The Boolean satisfiability method in Nemesis is almost always better than Random and Antirandom, However, when the test patterns No. is less than 60, the Antirandom testing method is better than Random, after 60, the Random and Antirandom becomes same also.

7 Conclusions

From the results in experiment, we can find the *Antirandom testing* is more efficient than *Random testing* for finding the Stuck-At faults.

So how to combine the antirandom algorithm with some other methods to be a very efficient and powerful automatic test generation tool will be our future work.

References

[1] Y. K. Malaiya, "Antirandom Testing: Getting the most out of black-box testing," *Proc. International Symposium On Software Reliability Engineering*, Oct. 1995, pp. 86-95.

[2] Craig Hall and Brian Chess, "The Nemesis User Manual", Computer Engineering University of California, Santa Cruz 95064.

[3] Barry W. Johnson, "Design and Analysis of Fault-Tolerant Digital Systems", University of Virginia, Charlttesville.

[4] Naixin Li, "Measurement and Enhancement of Software Reliability through Testing", Colorado State University.

[5] Tracy Larrabee, "Test pattern Generation Using Boolean Satisfiability", IEEE Transactions On Computer-Aided Design, Vol. 11, No. 1, Jan. 1992. *IEEE Software*, Sept. 1995, pp. 6-17.

8 APPENDIX

8.1 Appendix A - ARTG program performance

In measuring the performance for ARTG (Antirandom Testing Generation program), I used the *times* function to record the CPU time, and the *purify* testing tool to record the memory usage.

The experiment is done for getting 50 binary testing vectors, and the binary bit no. is from 2 to 100. It runs in Sun-OS machine. The CPU time is recorded by Clock Tick, the memory usage is based on the basic memory usage (including Purify overhead). The code memory is about 250K, the data memory is about 50K, the stack memory is 1.6K, only the heap memory is expanding when the binary bit no. is expanding. So I represented the heap memory no. and CPU time with the binary bit using the following table 3 and graphs 3, 4.

In this experiment, I found the CPU time varied with the load in machine, and the results from Purify-instrmented program demonstrated UMR (uninitialized memory usage). So for these results, I think they are not accurate, they only represent the varying trends for CPU time and Heap memory. More research and investigation for these are needed.

Table 3: Heap memory and CPU time for ARTG program

Bit No.	Heap(K)	CPU(Clock Tick)
2	24.6	1
4	41.0	3
8	73.7	30
12	106.5	73
16	139.3	130
20	172.0	193
25	213.0	272
30	254.0	347
35	294.9	465
40	335.9	566
60	499.7	1104
80	663.6	1778
100	835.6	3132

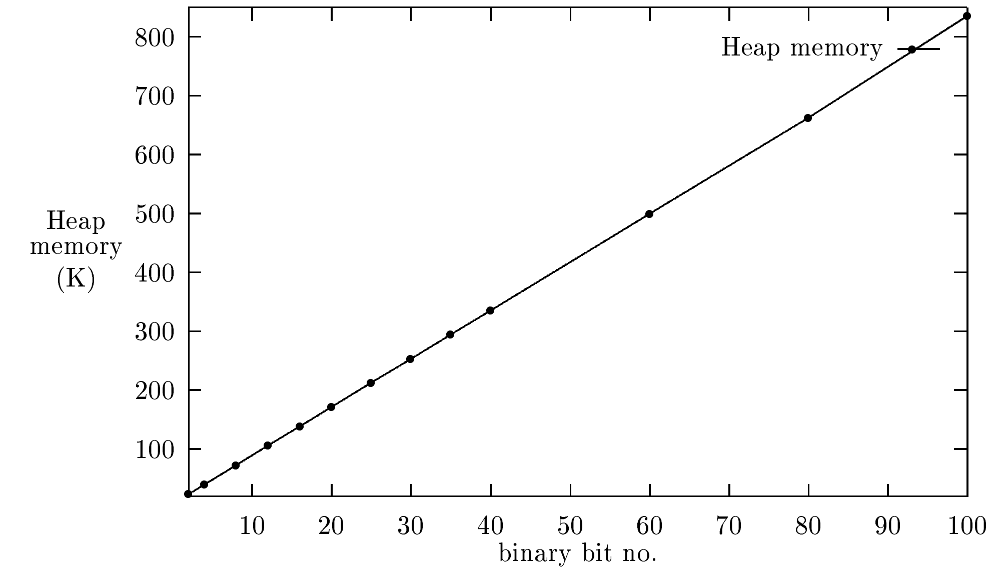

Figure 3: Heap memory usage for running ARTG

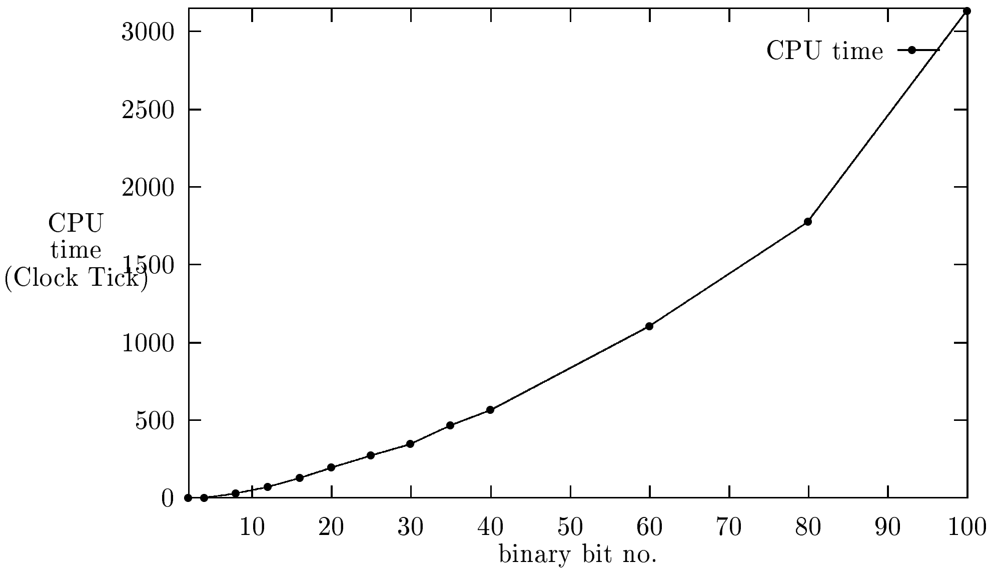

Figure 4: CPU time for running ARTG