Simulation Based Sequential Circuit Automated Test Pattern Generation

Jing Yuan

28 July 2004

Preface

This thesis constitutes my Masters thesis project for a degree in Master of Science in Engineering (M.Sc.Eng.). It was written during the period of 1st of February to 31st of July 2004 at Informatics and Mathematical Modelling (IMM), Technical University of Denmark (DTU). Associate Professor Flemming Stassen, IMM, was supervisor of my M.Sc. Thesis project.

I would like to thank the entire staff and my fellow students at the institute for creating a friendly and inspiring environment. Specially, I would like to take this opportunity to thank Flemming Stassen for his counseling, support and not least his interest in my M.Sc. Thesis project. Furthermore, I would like to thank my girlfriend Yingdi Lu for her patience and understanding.

Abstract

The aim with this paper is to design a high efficient sequential ATPG on single stack-at fault model. A new approach for sequential circuit test generation is proposed in this paper. With combining the advantage of logic simulation based ATPG and fault simulation based ATPG, Higher fault coverage and shorter test sequential length are achieved for benchmark circuit instead of pure logic or fault simulation based ATPG.

A new high efficient fault simulation algorithm which is based on PROOFs [39] is presented. Here two new techniques are used to accelerate parallel fault simulation: 1) X algorithm preprocessing, 2) Dynamic fault ordering method. Based on experiment result, these two heuristic accelerate fault simulation by 1.2 time in fault simulation.

Two metaheuristic algorithms, genetic algorithm and Tabu search, are investigated in test generation process. These algorithms are used to generate population of candidate test vectors and optimize vectors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and motivation

With the persistent increase in integration density and new technologies, test generation becomes more and more complex and expensive. Compare with combination ATPG, sequential ATPG is much more complicated. Because sequential ATPG not only depend on primary inputs, but also depends on the flip-flop's states. State and time expansion have caused a virtual explosion of the circuit. So there exists a crucial need to develop sequential ATPG that can handle VLSI chips at a reasonable computing cost and provide high fault coverage.

There are three types of test generator algorithms for sequential circuit [4], structure-based algorithms, fault simulation based algorithms and logic simulation based algorithms.

The principle to all structure-based algorithms is to construct of a combinational model of the circuit. Deterministic approach like D-algorithm, PODEM is used to generate test pattern. FTP (forward time processing) is used to propagate the effect of the fault and RTP (reverse time processing) is used for getting the required state. Because of ever-increasing complexity of digital circuit, the numbers of backtracks in deterministic approach is exponentially growing (NP problem), the structure-based algorithms algorithms are not feasible for large circuit.

In stead, in the past decades, the focus has been set on simulation-based techniques and metaheuristics. Simulation-based techniques are used to evaluate each candidate solution by fitness function. Metaheuristic algorithm is used to find out the best solution, which has highest fitness value, quickly.

In general, this type of ATPG is called simulation based ATPG. Because it needn't branch and bound, the computational complex is only square or linear to the size of circuit.

The class of simulation-based ATPG is divided into logic simulation based test generators (LSG) and fault simulation based test generators (FSG). LSG targets a property of the fault-free circuit such as the activity or the number of states (flip-flops' values) visited, and, based on fault-free simulation, generate a test sequence maximizing this property. FSG, on the other hand, fault simulate is used to evaluate the effectiveness of each candidate test sequence, thus providing a fitness function while collecting information on targeted faults, i.e. guiding the search process.

To LSG, only fault-free simulation (logic simulation) is used to guide the search process, whose computational complex is N, whereas to FSG, fault simulation is used to guide the search process, whose computational complex is N^2. LSG executes relatively quickly compared to FSG, especially for large circuits. But every coin has two sides. The deficient of LSG is that less information (only fault-free circuit is simulated) used in fitness function than FSG, so fitness function has less guidance and generates longer test sequence than FSG. But static compaction technology can remove unwanted vector of LSG and guarantee to retain fault coverage. These let LSG a more promising technology because of scalability to large circuits.

1.2 Overview

The focus of this document is simulation based sequential ATPG for single stack-at fault. The overview of ATPG is showed in Figure 1.1. From Figure 1.1, we can see the major part of this project is metaheuristic algorithm and fault simulation. The function of metaheuristic algorithm is to find the best test sequence according to fitness value. Fault simulation is used to find new faults detected by new inserted vector and drop the detected fault from fault list (fault dropping).

The Pros and Cons of LSG and FSG are complementary. LSG is fast but has less guidance information and FSG is slow and has relative more guidance information. In this project, both technologies are investigated and a new simulation based generator which combine both advantage of two ATPGs are proposed.

There are many excellent metaheuristic algorithms for large-scale optimization problem. But there is no such thing as a best metaheuristic, which has

```
Main
{
        do
        {
                Use metaheuristic algorithm, such as gentic algorithm, to find a new test vector

                Insert test vector to test set and checked new faults detected by the new test
                Vector (fault simulation)
        }
        While(finished_test_pattern_generation())
}

bool finished_test_pattern_generation()
{      // fault coverage: total fault detect / the number of fault in fault List
        If(fault coverage >= N) //N's default value is 1
                return false;

        If(test sequence's length >= M) //M's default value is 20,000
                return false;

        If(The last O number of test vectors can't detect any fault)
                //O's default value is 4,000
                return false;

        return true;
}
```

Figure 1.1: ATPG Overview

actually been proven mathematically. Here two classical algorithms, genetic algorithm and Tabu search are investigated to find suitable metaheuristic for ATPG.

Of course, a high efficient fault simulator can speed up ATPG. Here the proposed fault simulation is based PROOF[39]. Two new heuristics are used to improve the speed of fault simulation furthermore.

1.3 Document organization

Chapter 1 **Introduction**. This chapter gives a short overview of the project. It briefly describes the history of sequential ATPG and gives the overview of the project.

Chapter 2 **System architecture**. This chapter describes the framework of the design.

Chapter 3 **Fault simulator**. This chapter describes the related work and current design of the fault simulator.

Chapter 4 **Logic simulator**. This chapter describes the current design of the logic simulator.

Chapter 5 **Cost Function**. This chapter describes the fitness function, which used in simulation based ATPG.

Chapter 6 **Metaheuristic algorithm**. This chapter presents the implementation of GA algorithm and Tabu search algorithm, which used in ATPG.

Chapter 7 **Test**. This chapter describes the the project's test work.

Chapter 8 **Benchmark test run on C++ and C#**. Because many of published ATPG is implemented with C++ language, whereas the proposed ATPG is implemented with C# language. In this chapter, benchmark of C++ and C# is proposed to compare the performance of other published ATPG.

Chapter 9 **Experiment Result**. This chapter presents the experiment result of proposed ATPG.

Chapter 10 **Conclusion**. This chapter concludes this project.

Chapter 2

System architecture

In this work, my mandatory task is to design sequential ATPG, which can generate single stuck-at fault test pattern for sequential circuit. The framework is showed in Figure 2.1.

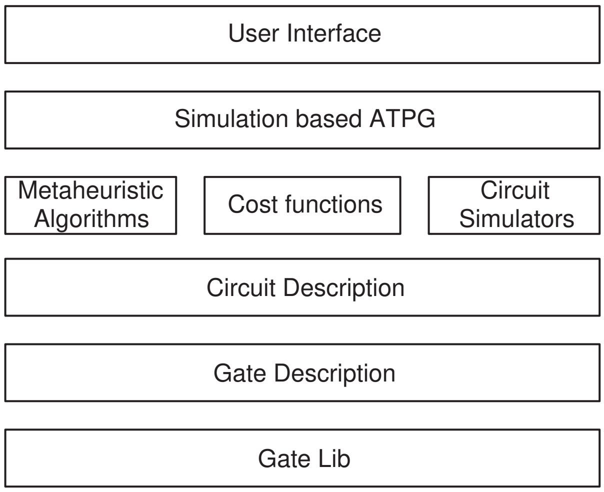


Figure 2.1: System Framework

The detail of each function block showed below

- **Gate lib** describes the functional of different gates. In this project, it support 9 types of gate, i.e. "AND", "NAND", "OR", "NOR", "XOR", "XNOR", "INVERSION", "BUFFER" and "D type flip flop".
- **Node description** describes attributes and functions in the node level. The attributes include information about node type, node value, input, output, fan-out and fault Information. The functions include node update, reset, etc.

- **Circuit description** describes attributes and functions in the circuit level. The attributes include information about D flip-flop node, internal node, primary input nodes, primary output nodes, fan-out nodes and fault info. For minimizing lookup time, these information are saved in hash table. The function includes input circuit from circuit file, circuit update, etc. Now, the circuit file supported is ISCAS89 benchmark file.
- **Metaheuristic Algorithms** includes two large-scale optimization algorithms, genetic algorithm and Tabu search.
- **Simulation Algorithms** implements fault-free circuit simulation and several fault simulation algorithms, such as X algorithm, parallel differential fault simulation algorithm.
- **Fitness functions** contain several fitness functions for LSG and FSG.
- **Simulation based ATPG engine**. It includes several simulation based ATPG engine, eg. GA based LSG and Tabu search based FSG.
- **User interface**. User can change parameters and operate the system through it.

In this project, many different technologies and heuristics are investigated, so in system design period, abstract, refinement and modularity are specially considered. From Figure 2.1, the system is comprised of 5 layers and 7 modules. Each module can only allow to invoke function of other modules which is in the same layer or one layer below. This design has advantages showed below.

1. Easy for sharing code. For instance, in this project, I implement six types of ATPG, these ATPG uses 2 different cost functions and 3 metaheuristic algorithms. Each function in "cost functions" block and "metaheristic algorithms" block are shared in these ATPG engines. So it needn't write independent cost function and metaheuristic algorithm for each ATPG.
2. Decoupling and easy refinement. After determining the interface of each block, any refinement in each block doesn't influence other block. For instance, modifying fault simulation algorithm will not influence cost function block, which invoke functions of simulation algorithms block.

Chapter 3

Fault Simulation

3.1 Overview

Fault simulation consists of simulating a circuit in the presence of faults. Comparing the fault simulation results with those of the fault-free simulation, the faults detected by that test can be determined. An example is showed in Figure 3.1.

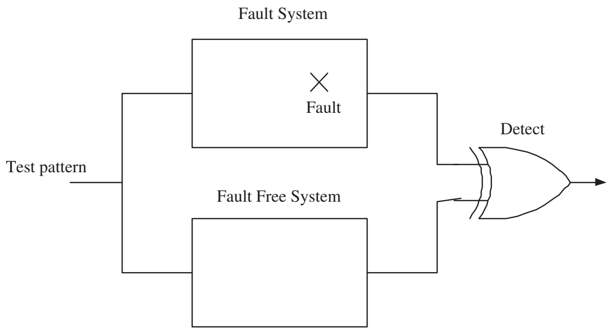


Figure 3.1: Fault simulation

There are two purposes of fault simulation during design cycle. The first is to evaluate the candidate test vector and guide the test pattern generation process (FSG). The second is to measure the fault coverage of inserted test sequence.

3.2 Terms

Synchronous sequential circuit is comprised of combinational logic block (CLB) and flip-flop. The inputs of the flip-flop are called PPO and outputs of the flip-flop are called PPI. Compare with combination circuit, sequential circuit has a new parameter, time frame. Under the zero gate delay model, Sequential circuit can be simulated by method of time-frame expansion. Each time frame just represents circuit in each clock cycle. CLB is copied in each time frame. Flip-flops are removed and their output are directly linked to input of next time frame's flip-flop. Test vector is inserted in each time frame one by one. For instance, the circuit in Figure 3.2 is expanded three time frames (clock cycle) in Figure 3.3. After time-frame expansion and insert test vector to each time frame, we can simulate the sequential circuit and get the primary output result in the third clock cycle.

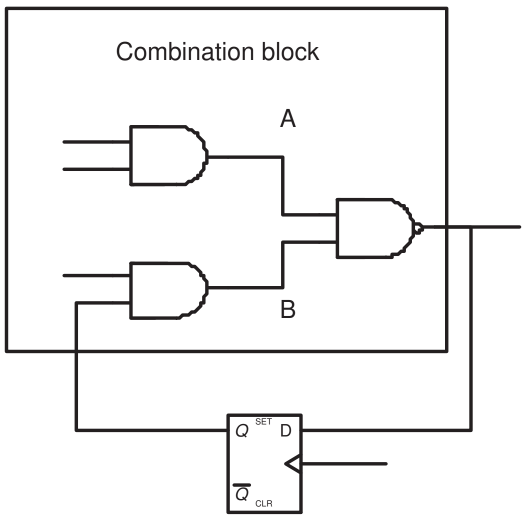


Figure 3.2: Sequential circuit

For sequential circuit fault simulation, if an undetected single stack-at fault propagated faults to PPO in the time frame, then in next time frame, it is called as **multiple event fault**. Otherwise, if it didn't propagate faults to PPI, it is called as **single event fault** in next time frame. A multiple event fault is the same as multiple stack-at faults. Fault simulation must consider effect of the faulty value of all the PPIs whose fault-free value and fault value are different.

As large CPU time is used in ATPG's fault simulation process, an efficient

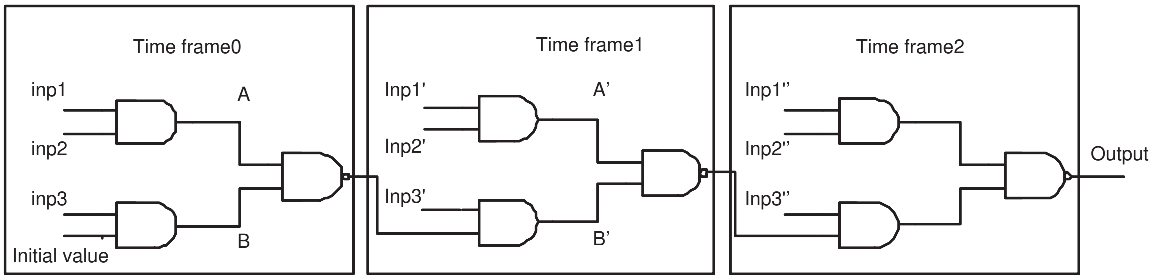


Figure 3.3: Time Expansion.

fault simulation algorithm is very important for ATPG. Five fault simulation algorithms are popularly used [37]. They are serial algorithm, parallel algorithm, differential algorithm, deductive algorithm, concurrent algorithm.

Serial algorithm is single pattern single fault propagation algorithm. Parallel algorithm is a single pattern parallel fault propagation algorithm. It directly improves on serial algorithm by taking advantage of existing computer instructions doing bit-parallel logical operations. In 32-bit INTEL CPU, 32 faults can be simulated in one pass. Pure serial algorithm and parallel algorithm is infeasible for VLSI circuit, because their computational complex is N^3, N is the number of the gates in the circuit.

Deductive algorithm explicitly simulating the fault-free circuit, and simultaneously deducing all detectable faults from the current good state of the circuit. Concurrent algorithm simulates the fault-free circuit and concurrently simulats the faulty circuit only if the faulty circuit's activity actually differs from the fault-free circuit. Deductive algorithm, and concurrent algorithm's computational complex is N^2, so they are fast algorithm, but they need large storage.

The difference algorithm simulates the fault-free circuit first and only simulate the part of fault circuit which is difference with adjacent fault circuit (or the fault free circuit). Its computational complex is N^2 and it need very little memory because it only stores one copy of adjacent fault cricuit (or fault free circuit) and the difference between adjacent fault cricuit (or fault free circuit). Among deductive concurrent and differential algorithms, differential algorithm [37] is not only the fastest one, but also the one requiring least memory.

PROOFS [39] [5] is a super fast parallel differential fault simulator for sequential circuit. Through exploiting parallel simulation of faults and utilizing several efficient heuristic, it accelerates fault simulation and reduces memory requirements by about five times.

Unlike the deterministic method showed above, X algorithm [30] and Critical path tracing [22] is an approximate method. X algorithm can identify most of undetectable faults and Critical path tracing can identify most of detectable faults. Their computational complexes are all N. The detail of X algorithm are showed in APPENDIX 12.1.

In this paper, I proposed two new techniques that substantially reduce the parallel fault simulation time of PROOFS. 1) Reducing fault simulation time by dynamic ordering. 2) Removing most of undetectable fault by X algorithm. We incorporated the proposed technique into my version of Proofs to develop a new fault simulator. According experiment results, my fault simulator is on average about 2.5 times faster than the Proofs implemented by memy version of Proofs for 10 ISCAS89 benchmark circuits.

In the rest of this chapter,

- Chapter 3.3 introduce technique of Proofs.
- Chapter 3.4 describes new heuristics used in my ATPG.
- Chapter 3.5 is experiment result.
- Chapter 3.6 is ongoing work.

3.3 Proofs

Proofs is a parallel differential fault simulator. In PROOFs [39], fault free circuit is simulated first and then fault circuits are simulated. Because one word in PC is 32 bits and a gate's value only occupy one bit, in Proofs, 32 faults are injected each time and simulated in parallel with the same test vector. If a fault is detected, it is dropped from fault list.

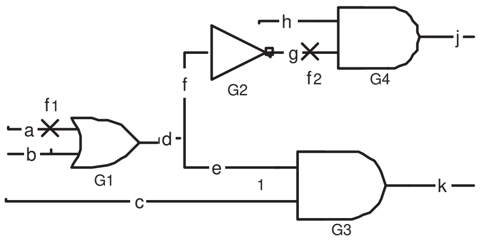
In [39], several heuristic have been used into Proofs to accelerate fault simulation. There are:

1. Event ordering
2. Inactive fault removing
3. Fault ordering

Next, I describe these heuristic by detail.

1. *Event ordering* To take advantage of parallel, same events should be deal with at the same time. But if using FIFO(First In and First Out) queue to store events, same events are hardly to be deal with in the same time. For instance, two faults, f1 and f2, showed in Figure 3.4, are simulated in parallel. F1 is assumed to propagate to the primary

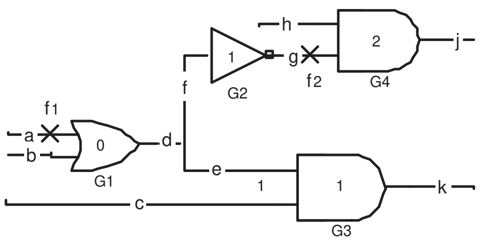
output j and k and f2 is assumed to propagate to the primary output j. At first, f1 and f2 are injected in the circuits and events G1 and G4, which is in the gate G1 and G4, are inserted in the event queue. After dealing with the events G1 and G4, f1 and f2 is propagated to G2 and G3, two new events G2 and G3 are inserted. In the end, f1 is propaged into G4 and event G4 is inserted in the queue again. There are totally 5 events in simulation and the number of events doesn't reduce after applying parallel simulation.



Turn	Event queue
0	G1, G4
1	G2, G3
2	G4

Figure 3.4: FIFO event queue.

In Proofs, event is ordered according to the level of the circuit. Every time, the event at the lowest level is retrieved from the event queue. For example, assuming the faults in Figure 3.4 are simulated by event ordering method, which is showed in Figure 3.5. In Figure 3.5, the number in the center of each gate represents the gate's level no. Each time, when new events are inserted into the event queue, they are ordered by the level no. Initially, event G1 and G4 is inserted in the queue. In the first turn, event G1 is lowest level event in the queue, so it is simulated and event G2 is inserted into the queue. In the second run, event G2 is lowest level event in queue, so it is simulated and event G4 is inserted in the queue. Because G4 is already exist in the queue, so it only need to simulate one time and the number of simulated events is reduced.



Turn	Event queue
0	G1, G4
1	G2, G3, G4
2	G3, G4
3	G4

Figure 3.5: Event ordering.

2. *Inactive fault removing:* If the value of a single event fault f is the same as the fault-free value, fault f is called **inactive fault**. Otherwise, it is called active fault. Inactive faults don't need to be simulated in parallel, so it is removed from fault list before fault simulation.
3. Fault ordering: In parallel fault simulation, in order to take advantage of parallelism, faults that cause the same events should be grouped together. PROOFS shows that depth first ordering often reduce the number of events more than breadth first ordering, so depth first ordering is used as event ordering algorithm in Proofs.

3.4 Proposed method

In the proposed fault simulator, in addition to the three heuristics showed above, two new heuristics have been incorporated into Proofs to accelerate fault simulation. The two heuristics are

1. Reduction of faults to be simulated in parallel by X algorithm
2. Dynamic Fault ordering

Next, I describe these heuristics in detail.

1. Using X algorithm to reduce the number of faults simulated by differential simulator.
   X algorithm [30] is a one-pass, linear-time algorithm that determines most of undetectable faults for a given test vector. Because differential fault simulator is a square-time algorithm, X algorithm can be used as a preprocessing step to reduce the number of faults simulated by differential simulator and significantly reduce fault simulation time.
   Critical path tracing [22] is another one-pass, linear-time algorithm that determine most of detectable faults for a given test vector. Use critical path tracing can reduce fault simulation furthermore. Here it wont't be considered because of two reasons. Firstly, normally, most of detectable faults are easily detected and they can be detected in the early stage of test and after that stage, the number of faults detected by each test vector generally is very limited and using critical path tracing can't get much improvement. In experiment, I found the improvement by critical path tracing in general less than the overhead of the algorithm. Secondly, this algorithm needs lots of memory to store preprocessing information, such as FFR information and parity information. In my implementation, it consumes more than 400M-byte-memory for some large circuits such as s35932.

The deficient of X algorithm is that it can't get any benefit from fault dropping. In other words, the simulation time is constant no matter how many faults are simulated. Considering in test generation process, the more faults are dropped, the less simulation time is improved by X algorithm. Especially in the end of test pattern generation, most of the faults are detected, the overhead of X algorithm may be greater than the improvement. In proposed method, X algorithm is stoped before the time when the overhead of X algorithm is greater than the improvement. . My idea is using "threshhold". If the number of faults, which need to be simulated, is over than threshold, X algorithm will be used. Otherwise, it will not be used (just use parallel differential simulator).
The value of threshold is calculated after simulating the first test vector by the Formula 3.1, 3.2 and 3.3.

$$\text{XEff} = \frac{An - Ax}{An} \tag{3.1}$$

$$\text{FaultNo} = \frac{XTime \times Ax}{DiffTime} \tag{3.2}$$

$$\text{Threshold} = \frac{FaultNo}{XEff} \tag{3.3}$$

An: the number of simulated fault per test vector on average. An is got from experiment result, showed in Table 3.1.
Ax: the number of fault simulated using X algorithm per test vector on average. Ax is got from experiment result, showed in Table 3.1.
XEff: the percentage of improvement (decrease) of the number of simulated faults after using X algorithm. Eff is calculated from Formula 3.1, which showed in Table 3.1. The value of Eff is conservatively set to 0.5 to calculate threshold.
XTime: the time that is spent in X algorithm simulation. Because in experiment, X time's Standard deviation is low, for each circuit, it is estimated according to the simulation result of the first test vector.
DiffTime: the time used in diffTime algorithm simulation. It is also estimated according to the first test vector.
SimFaultNo: the number of simulated faults. It is also estimated according to the first test vector.
FaultNo: the number of faults simulated by differential simulator after applying X algorithm.
Threshold: It is represented the overhead of X algorithm. If the number of rest faults is less than threshold, X algorithm is stopped and only differential fault simulation is used in fault simulation.

Circuit	An	Ax	Xeff
s349	80.8	32.6	0.60
s510	92.7	15.3	0.83
s641	212	43	0.80
s713	262	47.6	0.82
s838	739.9	134.2	0.82
s1196	323.1	50.3	0.54
s1238	417.5	52.8	0.65
s1423	946.7	235.2	0.76
s1488	583.7	12.5	0.97
s1494	603.2	12.5	0.98
s5378	2017.6	472.8	0.76

Table 3.1: Experiment result of Eff

I implement extended X algorithm [22] with the star detection heuristic and the fault propagation heuristic. The benchmark circuits used in experiment are ISCAS89 benchmark circuits. For each of benchmark circuits, the experiment is performed 3 times to study the consistence of the results. For each run, a sequence of random vectors is generated. In all experiments, each of the standard deviation value is lower than 10 percent of the average value. This value shows the consistency of the algorithm. For small circuits, s349, s510, s641, s713 and s838.1, the length of test vector is set as 200. For large circuits, i.e. s1196, s1238, s1423, s1488, s1494 and s5378, the length of test vector is set to 2000 to achieve high fault coverage. X algorithm is written with C# dotnet language and runs on HP-Compaq workstation. The operation system is windows XP and CPU is INTEL Pentium 4 2GHz. In the rest of experiments in this chapter, all experiment environment is same.
In Figure 3.6, the griding part shows the simulated faults reduced by X algorithm.

2. Dynamic Fault ordering In Proofs, faults is ordered by depth first ordering and grouping together in parallel simulation. According to my methods, because multiple event faults have faults effect in PPI and PPI is the lowest level in the circuit, Grouping multiple event faults with the same PPI can help to decrease the events. For instance, there are three faults in Figure 3.7. Faults f1 and f2 are neighbors by depth first ordering. They are grouped together by Proofs. Faults f1 and f3 have fault effect in the same PPI. In my method, they are grouped together. It can be seen that grouping f1 and f3 leads to more large intersection area.

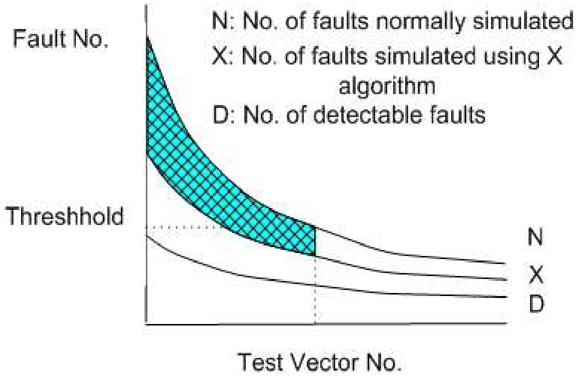


Figure 3.6: Number of fault simulated

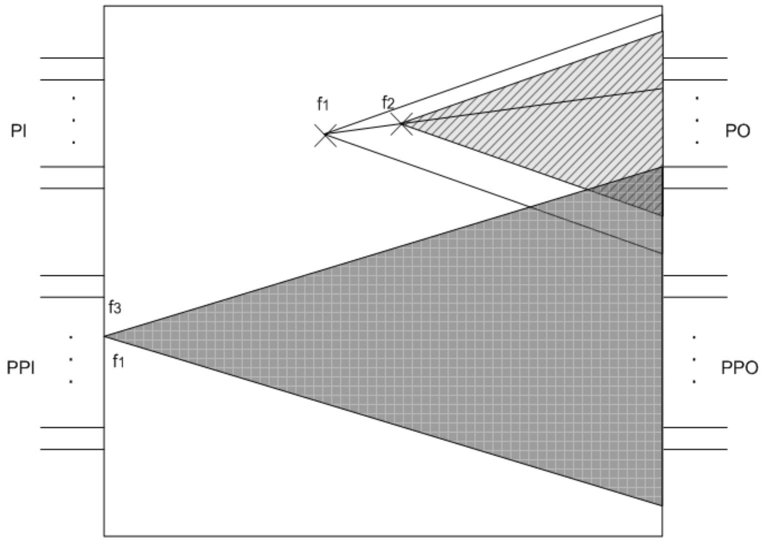


Figure 3.7: First example of fault grouping.

Like Proofs, I propose that ordering these stems by depth first traversing. Before each simulation, the single event faults are mapped to stems. Then, multiple event faults which has event in the same PPI are grouped. Last stem and multiple event faults are simulated in parallel. The details are showed in Figure 3.8.

There are two deficiencies in this method. Firstly, it needs to order faults and has overhead in each simulation. Secondly, for some multiple event faults (fault positions are in internal node and PPIs), which need longer distance to be propogated from internal node to output than from PPIs, dynamic Fault ordering can't help decrease the event number. For instance, in Figure 3.9, grouping f1 and f3 doesn't help to

```
Main
{
    Order_Prepocess();

    For each test vector
   {
       Dynamic_Order();

       FaultSimulation();
    }
}

void Order_Prepocess()
{
    ordering faults in fault list by depth first traversing
}

void Dynamic_Order()
{
    Map single event faults in FFR to the FFR  output stem;

    For each PPI
   {
       for each multiple event fault which has faults effect in the PPI
       {
           If (multiple event fault hasn  been grouped by other PPI)
               Grouped the fault with PPI
       }
   }
}
```

Figure 3.8: Dynamic fault grouping process

reduce the number of events.

3.5 Experiment results

In last section, two new heuristics are used to improve the speed of parallel fault simulation. To measure the effectiveness of the proposed heuristic, I implemented the Proofs, which is introduced in the second section of this chapter and a new parallel heuristic XProofs which incorporate the two proposed new heuristics. The performance of these individual techniques are reported in the next two subsections. All of fault simulators are written in the C# dotnet language and run on HP-Compaq workstation. The operation system is windows XP and CPU is INTEL Pentium 2G Hz. The other experiment environment is same as last section.

- Dynamic ordering.

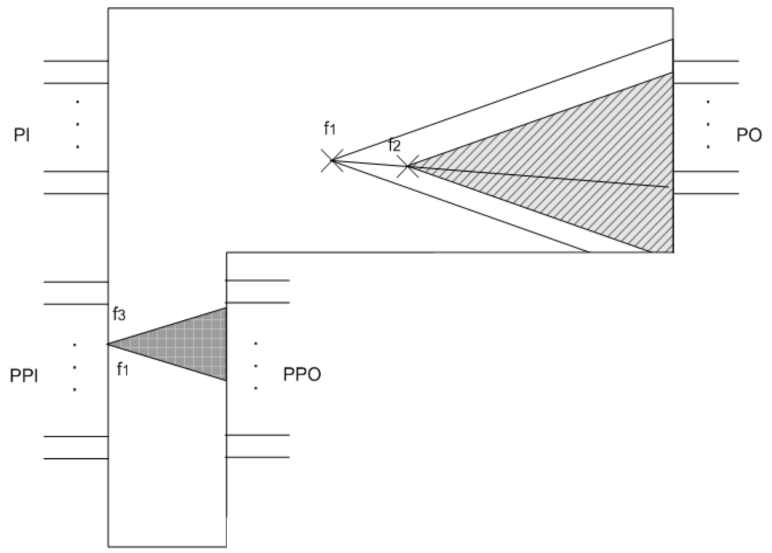


Figure 3.9: The second example of fault grouping.

The first experiment measures the performance of the heuristic "dynamic ordering". I implement a fault simulator called XProofs1 which incorporates the proposed heuristic and compare it with Proofs which incorporates the static fault ordering.
The aim of dynamic order is to reduce simulation time via reducing the number of events. The number of event per test vector and CPU times in each simulation are showed in Table 3.2.
From the table, we can see that the number of events in 8 of the benchmark circuits is reduced and on average, the number of events is decreased by 7 percent. In the rest of 3 circuits, the number of events increased because the distance between some FFRs' output and PO or PPO is shorter than PPI. If group these faults together, the performance will become poor.
Though the number of events is decreased 7 percent on average, simulation time is only decreased 1 percent. This is because the overhead of dynamic ordering is larger than static ordering. But code optimization can help to decrease dynamic ordering's overhead and improve the performance. Because time limitation, I haven't time to optimize code. It will be done in ongoing work.
Because the simulation time has only slightly reduced and in order to decrease the amount of test work, this heursitic is not included in proposed ATPG.

- Performance of incorporation of X algorithm

Circuit	Average Event Number per Vector			Time(Sec)		
	Proofs	XProofs1	Speed up ratio	Proofs	XProofs1	Speed up ratio
s349	205,5	203,1	1,01	1,13	1,17	0,96
s510	312,7	306,8	1,01	1,4	1,54	0,9
s641	712	738,7	0,96	3,54	3,72	0,95
s713	1055,6	927,9	1,13	4,58	4,13	1,1
s838	3276,7	2885,8	1,13	17,1	16,4	1,04
s1196	1812,8	1802,6	1,01	74,5	76,1	0,97
s1238	2636,9	2644,2	0,99	105,1	111	0,94
s1423	6549,6	5044	1,29	334,2	294,3	1,13
s1488	3988,9	3989,9	0,99	152,2	163,8	0,92
s1494	4089,2	4049,8	1,01	156,8	158,5	0,98
s5378	14974,1	11478,4	1,3	1644	1311,1	1,25
Average			1,07			1,01

Table 3.2: Experiment results of Dynamic ordering and static ordering

In this section, I report experimental results for incorporation of X algorithm. Fault simulators implemented here are comprised of X algorithm and differential simulation. Extended X algorithm [22] with the star detection heuristic and the fault propagation heuristic is used to reduce the number of faults simulated by differential simulator. Two fault simulators, XProofs2 and XProofs3 are implemented. threshold isnot used in XProofs2, whereas threshold is used in XProofs3. The algorithm of XProofs3 is showed in Figure 3.10.

```
Fault ordering by depth first algorithm (preprocessing step);

XProofs3()
{
      Inactive fault moving;

      if (the number of fault in fault list > threshold)
      {
            Simulate the circuit by X algorithm to determine most of
            undetected fault;
      }

      Fault Simulate the rest of faults by parallel differential simulator

      fault dropping;
};
```

Figure 3.10: Algorithm of XProofs3

From the Table 3.3, it can be seen that the fault reduced by the X

Circuit	Fault Coverage	Number of faults simulated by differential simulator per vector on average				
		Proofs	XProofs2	Speedup	XProofs3	Speedup
s349	93.4%	80,8	32,6	2,47	79,4	1,01
s510	97.8%	92,7	4.89	18.95	15,3	6,05
s641	70.6%	212	43	4,93	43	4.93
s713	69.3%	262	47,6	5,5	55,4	4,72
s838	25.1%	739,9	134,2	5,51	134,2	5,51
s1196	86.1%	323,1	50,3	6,42	150,5	2,14
s1238	81.3%	417,5	52,8	7,9	52,8	7,9
s1423	48.9%	946,7	235,2	4,02	235,2	4,02
s1488	68.4%	583,7	12,5	46,69	12,5	46,69
s1494	67.7%	603,2	12,5	48,25	12,5	48,25
s5378	66.8%	2017,6	472,8	4,26	472,8	4,26
Average				14.08		12,31

Table 3.3: Experiment result 1 of fault simulator with X algorithm

algorithm is great. Because of threshold, the fault reduced by XProofs3 is less than XProofs2 in four of the benchmark circuits. In the rest of the circuits, because the number of undetected faults is always greater than the threshold in simulation, the fault reduced by XProofs3 is same as XProofs2.

Table 3.3 shows the improvement of simulation time. It can be seen that XProofs2 and XProofs3 can reduce the simulation time about 60 percent. Notablely, for circuit s349, simulation time of XProofs2 is larger than Proofs. This is because in this case, the overhead of X algorithm is greater than improvement. Whereas, because of threshold which can stop X algorithm when overhead is greater than improvement, XProofs3 is faster than Proofs. This case shows the efficiency of threshold.

In conclusion, incorporation X algorithm can effectively reduce the simulation time and using threshold can effectively prevent overhead great than the improvement of X algorithm when fault coverage is high and reduce the simulation time furthermore.

In proposed ATPG, X algorithm with threshold is incorporated with parallel differential simulator.

Circuit	Fault Coverage	Time(Sec) Proofs	Time(Sec) XProofs2	Speedup	Time(Sec) XProofs3	Speedup
s349	93,40%	1,13	1,22	0,92	1,02	1,1
s510	97,80%	1,4	1,14	1,22	1,05	1,33
s641	70,60%	3,54	2,67	1,32	2,67	1,32
s713	69,30%	4,58	3,16	1,44	3,16	1,44
s838	25,10%	17,1	6,01	2,84	6,01	2,84
s1196	86,10%	74,5	30,1	2,47	30,1	2,47
s1238	81,30%	105,1	32,2	3,26	32,2	3,26
s1423	48,90%	334,2	151,4	2,2	149,3	2,23
s1488	68,40%	152,2	36,4	4,18	36,3	4,19
s1494	67,70%	156,8	36,1	4,34	36,1	4,34
s5378	66,80%	1644	672	2,44	672	2,44
Average				2,42		2,45

Table 3.4: Experiment result 2 of fault simulator with X algorithm

3.6 Ongoing work

Hyung Ki Lee et al. proposed HOPE [20] which further speeds up PROOFS. It used three new heuristics to improve the speed of Proofs.

- fault mapping
- fault ordering
- efficient fault injection

These heuristics can also improve proposed fault simulator in experiment of [20]. But in my implementation, the first two heuristic decrease the fault simulation's performance, this is because these heuristics are sophistic and complicated. If don't take care of code optimization, the improvement will be offset by high overhead. Unfortunately, because of time limitation, I can't do much code optimization. I plan to optimize these code and implement the third heuristic "efficient fault indection" in the future work.

Next, I show these techniques in detail.

- *fault mapping:*
  After removing stem, CLB can be partitioned into fan out free region (FFR). Any single event fault in FFR must propagate into output of FFR before propagating to any PPO or PO. In other words, any single event fault in FFR can be mapped to stem faults, which is output of FFR. After mapping, only stem fault can be simulated. For example, Figure 3.11 is a FFR in circuit. Faults f1, f2, f3, f4 and f5 can be

mapped to stack-at 1 and stack-at 0 fault in the stem s. This heuristic can reduce faults to be simulated.

- *fault ordering*
  HOPE improves the ordering method furthermore. Because multiple event faults can't be mapped and they should be propagated through stem, grouping faults in the same FFR together is a good idea to reduce events. For instance, faults f1, f2, f3, f4 and f5 in Figure 3.11 should be grouped together. HOPE proposes that nonstem faults inside each FFR be grouped first and then ordering these fault groups by depth first traversing FFR.

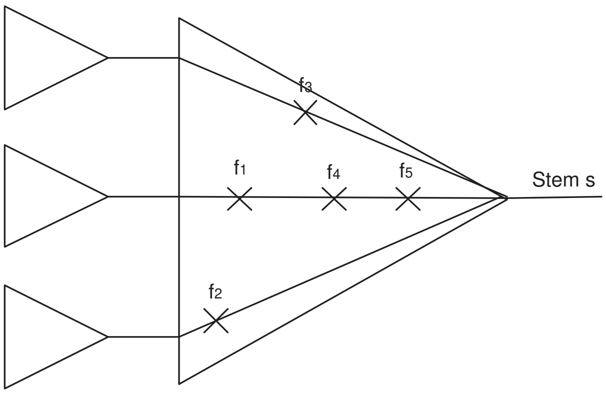


Figure 3.11: Fault ordering.

- *efficient fault injection*
  In proposed fault simulation, traditional fault simulation [11] method is used. Faults are injected by masking the appropriate bits of words and it is slow because it needs to check each gate during the simulation to determine whether bit-masking has to be performed or not.
  Hope improves the amelioration in the fault injection method. When a fault is introduced at an input or output of the gate, the gate function is changed. In other words, we can use a new Boolean expression to reflect the presence of the fault. For instance, suppose a stack-at 0 fault is injected into a gate's output, then the function of the fault gate is $f = 0$;.
  In HOPE, the types of fault free gates and fault gates are coded together, which is showed in Table 3.5.
  A new fault type of gate is created, when a fault is injected into circuit. The new fault type of gate is assigned a function index (20 + the bit position of the fault in the word). Because one word contain 32 bits, here 20 to 51 is reserved for faulty gates.

function index	gate type
1	AND
2	NAND
3	OR
4	NOR
5	XOR
6	XNOR
7	INVERT
8	BUFFER
9	D flip-flop
20 and above	reserved for faulty gates

Table 3.5: Gate Type

Assuming three stack-at faults f1, f2 and f3 are injected into circuits, as shown in Figure 3.12. Suppose these three faults are simulated in parallel and saved in the bit position 0 to 2. Gate A has two faults f1 and f2. The two faults are added into link list. Because f1's bit position is lower than f2, the type of Gate A is changed to 20, represented the first fault in bit position 0. Gate B has no fault. Gate B's type is 1 represented AND gate. Gate C has one fault f3, which is in the position 2, so gate C's type changed to 22.
After all faults are inserted, the gates are evaluated according to function index and fault information.
The advantage of the method is it needn't circuit modification and has no any overhead for fault-free gates.

In experiment results [20], on average, HOPE is faster than Proof by 1.6 times.

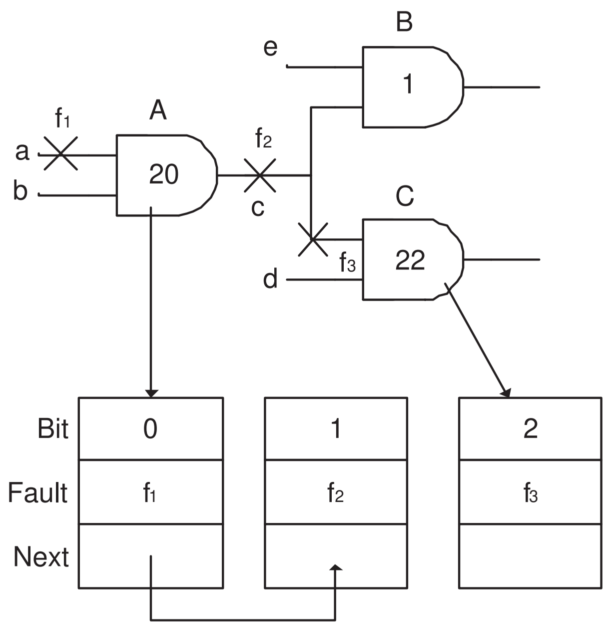


Figure 3.12: Example of Fault Injection.

Chapter 4

Logic Simulation

Logic simulator is used to simulated the fault free circuit. In the beginning of each time frame, not only new test vector is inserted in the circuit, but also flip-flops's states are updated. These changes influence the internal node of the circuits. Logic simulator is used to simulation these influence to fault free circuit. In LSG, Logic simulator is also used to evaluate the effectiveness of each candidate test sequence.

Logic simulator implemented in this project has following characters.

- event-driven simulator.
- Influence of flip-flops' new state is simulated first. (The test vector is same as test vector in last time frame) This is because flip-flop's state in current time frame is not depending on new test vector, but only depending on PPO of last time frame. In other words, in the same time frame, flip-flops' new states are always the same whatever test vector changes. If simulating influence of flip-flops' new state first, the circuit can be used as initial circuit for new test vector simulation.
- Differential logic simulator. Like differential fault simulator, only different part of circuits is simulated. The initial circuit is the circuit after simulating the new flip-flop's state.
- parallel logic simulator. 32 different test vectors are simulated in one pass.
- Event ordering. Like Proofs, event is ordered according to the level of the circuit.

Chapter 5

Fitness function

5.1 Overview

The two types of simulation-based ATPG, FSG and LSG, all use fitness function to evaluate the fitness of test sequence and guide the test generation, but they use different method to calculate fitness value. For FSG, fault simulation makes up the bulk of computation. LSG repeats logic simulation to gather information for fitness function. Both FSG and LSG is complementary. LSG is less complex than FSG, so the execution time can be far shorter. But at the same time, LSG gathers less information for fitness function, generally, the fault coverage of LSG is inferior than FSG.

In this project, at first, a LSG is designed. Because of complementary of LSG and FSG, ATPG is improved by combination technique of LSG and FSG. The organization of the rest chapter are showed below.

- Chapter 5.2 Fitness function of LSG
- Chapter 5.3 Fitness function of FSG
- Chapter 5.4 Combinational fitness function

5.2 Fitness function of LSG

The proposed fitness function of LSG targets on property of the fault free circuit and depends on the observations showed below.

1. Observation 1: The more new states are accessed, the more faults may be detected.

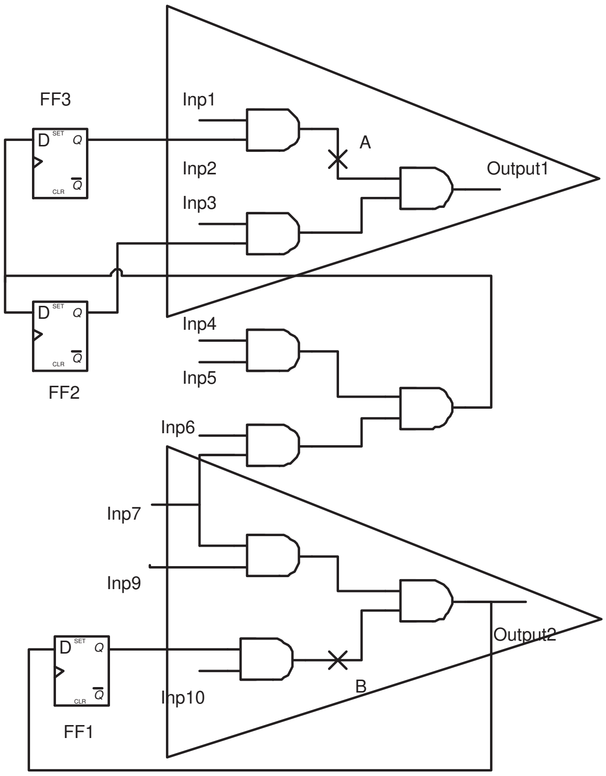


Figure 5.1: Useful states and useless states.

Fault detection in sequential circuit is depended on circuit states (flip-flops' value). More new states are accessed, more faults may be detected. Therefore, Pomeranz [28] developed a test generator that aims to drive the fault-free circuit to as many new states as possible.

2. Observation 2: New states which have more difference with old states, more useful.

   Large sequential circuit has numerous states, assuming the number of circuit's states is equal 2^N, N is the number of flip-flop, most of these states are noise and useless. Indiscriminate addition of new states is inefficient and fault coverage can level off prematurely.

   Which test pattern is important, which is not? We can see an example first. A fault circuit showed in Figure 5.1. New states in flip-flop FF2 and FF3 is useful to test the fault A in line. New state in FF1 is useless. But it is always useful when more than one flip-flop's state changes. In other words, more different with old states, more useful.

   Hamming distance can be used to guide the selection and remove noise. Longer distance means there is more difference with old states. The choice, which has long distance with old states, is a good choice. For example, in Figure 5.2, a circuit has 8 flip-flops. In history table, there are 2 old states and in candidate table, there are 2 candidate states. Hamming distance between state A and old states is 9 and that of state B is 5. A is better than B.

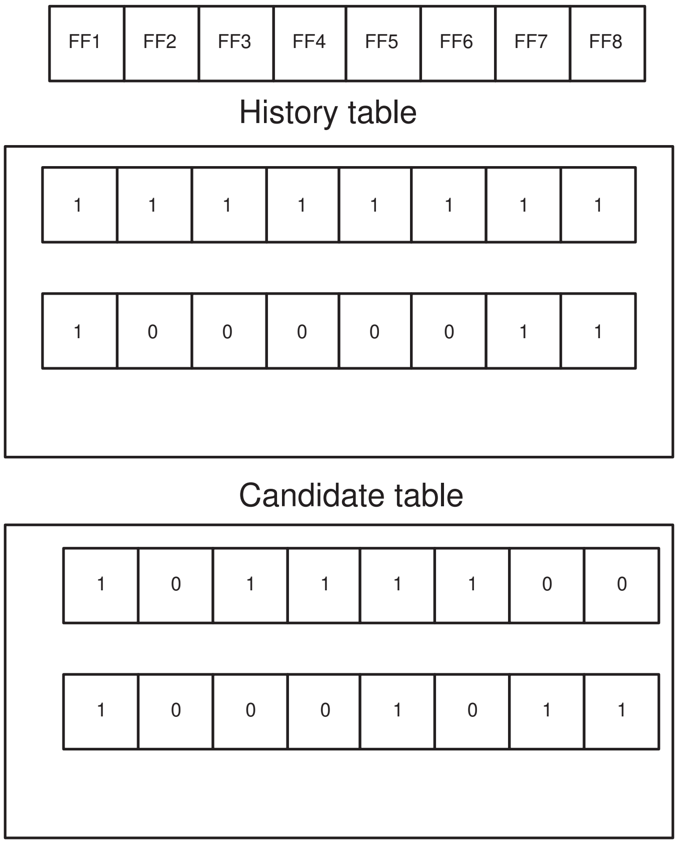


Figure 5.2: Example of hamming distance

3. Observation 3: State partitioning provides better guidance in the search space.
   As shown above, for each fault, some state variables are useful, while others are useless and unimportant to detected faults, so maximizing the new states on these useless states doesn't play any work. Obviously, these unimportant states are noise and could misguide the test generator. For instance FF1 is useless states for fault A. Unfortunately, Hamming distance can't weed out these noises.
   Another choice is partitioning global state into partial states [31]. Fitness value is calculated according to how many new partial states ac-

cess.
For example, Figure 5.3 showed circuit with eight flip-flops. Global state is partitioned into two partial states S1 and S2. There are two old states in history table. Here it can be found that candidate A has 2 different partial states and B has 1 different partial state, so A is winner.

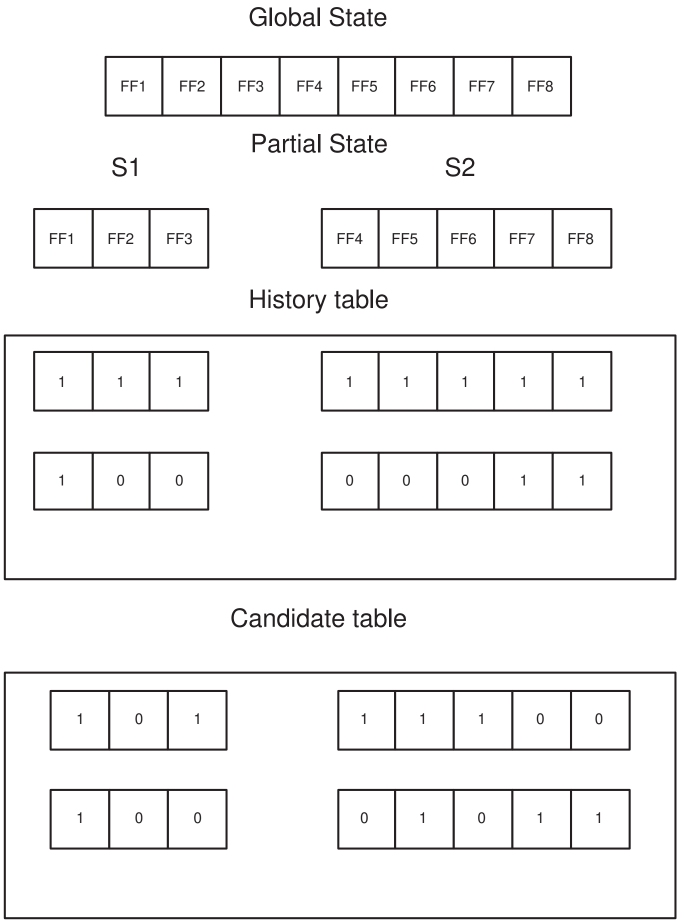


Figure 5.3: Example of state partitioning

4. Observation 4: Hard-to-detect fault generally has to access hard to control states during testing [31].
   Shuo Sheng and Michael S. Hsiao [31] proposed that partitioning states according to their controllability and assigning hard to control states higher weight is useful to weed out the noise. Here I use the same partition method as [31]. The details is showed below.
   Partition algorithm showed below
   (a) Calculate each flip-flop's controllability value.

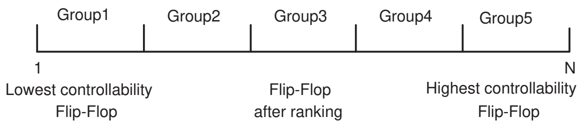


Figure 5.4: State Partitioning

The circuit is simulated by logic simulation with 1000 random vectors and the number of times in each FFi is set to 0 is recorded , denoted by Ni^0. The 0 controllability of FF_i is calculated as $CC0_i = Ni^0 \div 1000$, and the 1 controllability is $CC1_i = 1 - CC0_i$. The bias of the controllability values is called as FFbias and defined as $FFbias(i) = |CC0_i - CC1_i|$, If FFi is easily controlled as both 0 and 1, then FFbias(i) is close to 0. A large FFbias(i) indicates that a strong bias exists for this $CC0_i$, $CC0_i$ and It is hard to control 0 or 1 to FF_i. Because reducing this bias is most interesting in state traversal, bias-to-0 or 1 isn't differentiated.

(b) Sort the entire flip-flops by FFbias.

(c) Partition total flip-flop into 5 sub groups, each group has 1/5 flip-flop, which is shown Figure 5.4.

5. Observation 5: Best former test patterns can not guarantee later test patterns are also best. If only optimizing single test vector for each time frame, then only local optimization result can be achieved.
As mentioned above, for sequential circuit, primary output values not only depend on primary inputs, but also depend on old circuit states. If only optimizing test pattern for each time frame, only local optimization result is achieved, because best former test patterns can't guarantee later test patterns also best. For example, in Figure 5.5, a circuit has 8 flip-flops. It is partitioned into 3 partial states, S1, S2 and S3. In history state table, it has three old states. There are two test sequences A, B in candidate table. Each sequence has three test patterns, which are winners after optimization. For instance, A1 is the best test pattern after applying test pattern A0. I also denote partial states a decimal number instead of binary number. For example, '3' appearing on S1 represents partial states "011". It can be found that after applying A0, three new partial states are accessed. For B0, only 2 new states are accessed. But when applying all three test vectors in test sequence A, 5 new partial states are accessed. For B, 8 new partial states are accessed. Obviously, B is better than A globally. But if only optimizing for each time frame, A0 is better than B0, so A is selected. Then a local optimization result is achieved. To improve it, I optimize test

sequence instead of single vector. Figure 5.6 shows the structure of an optimization unit (chromosome in GA) . In theory, longer sequence can get more global optimal results. But because Longer sequence also exponentially extends search space, which is showed in Equation 5.1. This increases generation time and decreases the probability of finding optimal result. I tried 5 different lengths of sequence, 1, 10, 20, 40 and 100, to find the optimal value.

$$SearchSpace = 2^{N \times M} \tag{5.1}$$

- N is the number of primary inputs.
- M is sequence's length.

According to the observation showed above, Fitness value is computed by the Formula 5.2 :

$$\text{Fitness} = \sum_{j=1}^{M} \sum_{i=1}^{N} W_{ij} \times U_{ij} \div 10^{K} \tag{5.2}$$

- N is the total number of partitions. In my implementation, N is set to 5.
- M is the length of test vectors in a chromosome,
- U and K's meaning is showed in below Table 5.1, 5.2 and 5.3.

In Formula 5.2, "$\sum_{j=1}^{M}$" is used for global optimization (Observation 5). "$\sum_{i=1}^{N} W_{ij} \times U_{ij}$" is used to maximum new partial states(Observation 2, 3, 4).
"$\div 10^{K}$" is used to achieve maximize new global states (Observation 1),

	W: weight assigned to each sub group.
Group 1	1
Group 2	2
Group 3	3
Group 4	4
Group 5	5

Table 5.1: Parameter W

Because the goal is to get as many useful states as possible, the search must not frequently return the circuit with an old or reset state. In sequential circuit, there is reset signal to reset circuit's state. These signals must be masked. S. Sheng [27] uses reset masking technology to prevent it, i.e. use parallel simulation to identify the circuit's reset

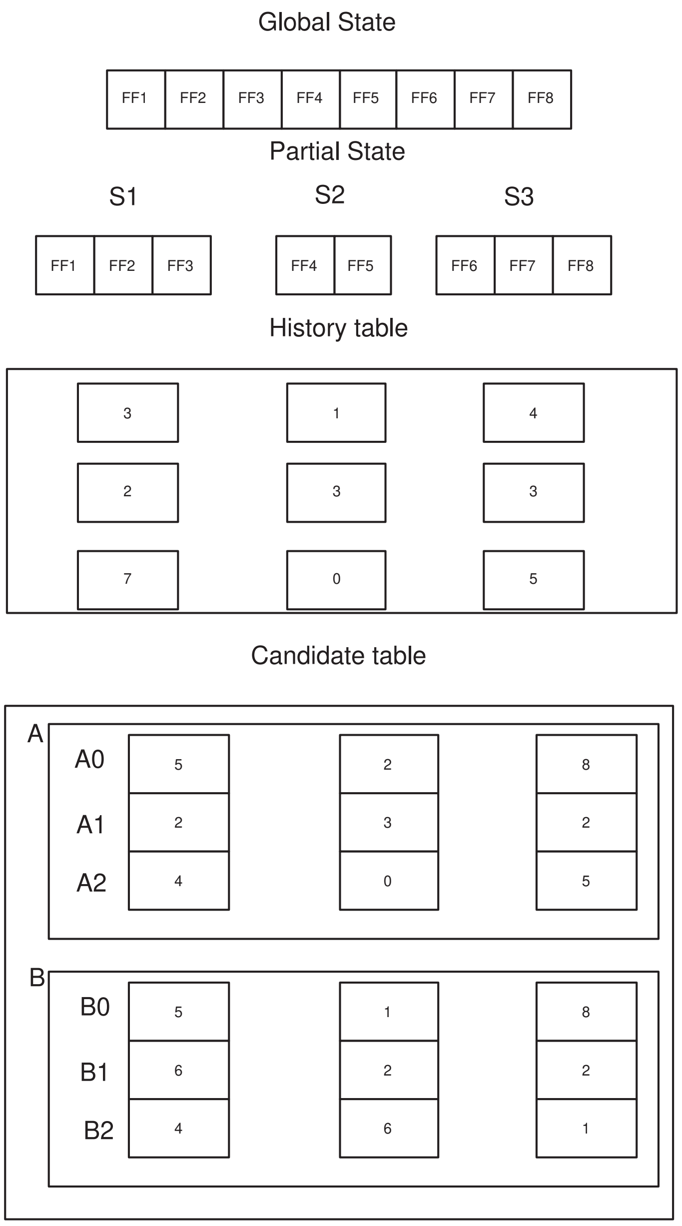


Figure 5.5: Global Optimization and Local Optimization

Test Vector 1	Test Vector 2	.....	Test Vector M - 1	Test Vector M

Figure 5.6: structure of an optimization unit

PIs and perform the masking as part of the ATPG process. Here I use a new simple method. In any times, the circuit returns an old or

	U
New Partial State	10
Old Partial State	1/n, n is the number of times this partial state has been visited during logic simulation

Table 5.2: Parameter U

	K
New Global State	0
Old Global State	n, n is the number of times this global state has been visited during logic simulation

Table 5.3: Parameter K

reset global state, fitness value is divided by 10^k, k is the number of times this global state has been visited during logic simulation. This way is better than reset masking technology, because it prevents circuit from returning not only reset state, but also old states, whereas reset masking technology only prevent circuit from returning reset state.

5.3 FSG fitness function

FSG [12] [32] [9] [13] gathers information for fitness function from PROOFS fault simulation. The fitness function of [13] is showed in Formula 5.3.

$$fitness = \sum fautls\ detected + \frac{\sum fault\ effects\ propagated\ to\ flip\ flops}{\sum faults\ simulated \sum flip\ flops} \tag{5.3}$$

Because the aim of ATPG is to achieve high fault coverage, the number of detected faults ($\sum fautls\ detected$) is the primary metric in the fitness function. The number of fault effects propagated to flip-flops are induced to differentiate test sequences that detect the same number of faults. To ensure that the number of faults detected is dominant fraction in the fitness function, the number of fault propagated is offset by the number of fault simulated and the number of flip flops.

Though an accurate fitness function is essential for achieving a good solution, the high computational cost of fault simulation may be prohibitive. To avoid excessive computations, [12] [32] [9] [13] approximate the fitness of a candidate test by using a small random sample of faults. In these work, they use a sample size of about 100 faults if the number of faults remaining in the fault list is greater than 100.

In this project, Formula 5.3 is chosen as fitness function and X algorithm is used instead of fault sampling if the remaining faults are greater than threshold, as X algorithm can determine most of undetected faults. The detected faults and fault effects propagating to each PPO are computed by Formula 5.4 and 5.5.

$$FXALG = FFL - XUDF \tag{5.4}$$

$$XFPPPO = XUDF - FXPALG \tag{5.5}$$

- FXALG: detectable faults, which is determined by X algorithm
- FFL: faults in fault list.
- XUDF: undetectable faults, which is determined by X algorithm
- XFPPPO: undetectable faults but whose effects can propagated to each PPO, determined by X algorithm
- FXPALG: undetectable faults and whose effects also can not propagated to each PPO, determined by X algorithm

X algorithm is chosen because of two reasons.

The first, X algorithm is more accurate than fault sampling. Table 5.4 shows the precision of X algorithm and Precision of Fault Sampling. From Table 5.4, it can be seen that X algorithm is much preciser than fault sampling.

Formula 5.6 is used to calculate the precision of X algorithm in Table 5.4. Formula 5.7 is used to compute the precision of fault sampling in Table 5.4. Like [12] [32] [9] [13], the sample size is 100 in the experiment. The experiment environment is the same as last chapter.

$$Precision = \frac{\sum_{i=1}^{LTS} \frac{NODF}{NODFX}}{LTS} \tag{5.6}$$

$$Precision = \frac{\sum_{i=1}^{LTS} (NODF < NODFS)?(\frac{NODF}{NODFS} : \frac{NODFS}{NODF})}{LTS} \tag{5.7}$$

```
Sampling faults from fault list

Estimate the number of detected faults with formula showed below

NOFFS = the number of detected sample faults / sample size * the length of fault list

For each test vector
{
        if(NOFFS  greater than NOFXHOPE)
        {
                TotalPrecision += NOFXHOPE/ NOFFS;
        }
        else
        {
                TotalPrecision += NOFFS / NOFXHOPE;
        }
}

Precision of fault sampling  = TotalPrecision / the length of test sequence;
```

Figure 5.7: Precision of fault sampling

- LTS: the length of test sequence.
- NODF: the number of detectable faults, which are determined by parallel differential fault simulator.
- NODFX, the number of detectable faults, which are determined by X algorithm.
- NODFS, the number of detectable faults determined by fault sampling.

The second, any faults detected by differential fault simulator also can be determined by X algorithm. This is a very useful character which fault sampling hasn't, because most of faults are easily detected faults and they are dropped in the early stage of test generation. In addition, the number of faults detected by single vector are very little, generally not greater than 2. Because there is large probability that these detectable faults are not sampled, the fitness value may be no difference with useless vectors which can't detect any faults.

5.4 Combinational fitness function

As showed above, The Pros and Cons of LSG and FSG are complementary. LSG is fast but containing less guidance information and FSG is slow but with relative more guidance information. In this section combination-simulated based generator (CSG) is proposed to combine the advantage of LSG and

Circuit	Precision		
	the length of Test vector	X algorithm	Fault Sampling
s349	200	0.99	0,87
s510	200	0.97	0,92
s641	200	0.98	0.75
s713	200	0.85	0,71
s838	200	0.79	0,83
s1196	2000	0.52	0,37
s1238	2000	0,59	0,39
s1423	2000	0.92	0,48
s1488	2000	0,98	0,42
s1494	2000	0.96	0,38
s5378	2000	0.93	0.70

Table 5.4: Precision of X algorithm

FSG, which is showed in Figure 5.8.

```
Main
{
    do
    {
        Logic simulation and metaheuristic algorithm are used to find best test vector

        X algorithm are used to evaluate the fitness value of the best 32 vector, which
        has highest LSG fitness value, again and selected the best vector (FSG)

        Insert the best test vector to test set and checked new faults detected by the new
        test  vector (fault simulation)
    }
    While(finished_test_pattern_generation())
}
```

Figure 5.8: Combination ATPG Processing

Comparing with pure LSG, CSG needs 32 more times of X algorithm for each new test vector. But this overhead is not large, because of two reasons.

1. For each new inserted test vector, ATPG also needs fault simulation to determine the number of detectable fault. Table 9.3 in Chapter 9 shows that in most of test circuits, only about 10%'s CPU time is used for fault simulation, whereas the computation cost of X algorithm is much less than fault simulation. The cost of 32 times X algorithm haven't much influence in running time .

2. Table 9.7 shows that CSG is faster than LSG, because CSG decreases the

length of test sequence (without decreasing fault coverage). In other words, the number of test generation is decreased by CSG. This improvement is greater than overhead.

In conclusion, CSG not only improves the fault coverage, but also decreases the test generation time.

Chapter 6

Metaheuristic algorithm

In art of state simulation-based [10] test generation ATPG, metaheuristic algorithms, such as genetic algorithm, are used to search for best test sequence and fitness function is used to guide the test generation's search.

Six metaheuristic algorithms are commonly used. They are Tabu Search(TS), genetic Aglorithm(GA), Simulated Annealing(SA), Deterministic Annealing(DA), Ant Systems(AS) and Neural Networks(NN) [3]. GA is a iterative procedure that maintains a population of candidate solutions encoded in form of chromosome string. SA, DA and TS move from one solution to another in the neighborhood to find good solution [3]. AS discovers good solutions by using positive feedback, constructive greedy heuristic and distributed computation. NN is a learning method, which gradually adjusts weights until a satisfactory solution is reached.

A suitable metaheuristic algorithm for ATPG can speedup test generation, reduce test sequence's length and improve fault coverage. Because of time limitation, it is impossible to implement and compare all these metaheuristics in this project. In order to choose the best one, I consult metaheursitic solution of another large-scale problem, Vehicle Routing Problem (VRP), which is to find the efficient use of a fleet of vehicles that must make a number of stops to pick up and deliver passengers or produces. I think efficient metaheursitic for VRP also can be used in ATPG, because of two reasons. Firstly, like ATPG, VRP is a NP problem and an algorithm for solving NP problem can be translated into the one for solving another NP problem. Secondly, like ATPG, VRP also has large neighborhood and the size of neighborhood is an important metric to select metaheuristic algorithm to solute problem.

Comparing the best-known methods, NN, SA, DA and AS have not shown out competitive results [15]. TS [14] and GA [1] have been got a lot of attention and are the most effective approachs for VRP. In this project, I implemented GA and TS. They will be described in the rest of the Chapter,

The deficiency of metaheuristic algorithms [1] is that their running time is unknown and they involve many parameters that need to be tuned for each problem before they are applied. In this project, I limited running time via limiting the number of repeating and tried to find best parameters for ATPG. The details are shown later.

6.1 Genetic algorithm

1. **Background**
   In 1859, Charles Darwin proposed the theory of Natural Selection [18]. It states the procession that results in adaptation of an organism to its environment by means of selectively reproducing changes in its genotype. The individual with higher adaptation characteristics is more likely to survive and mate. In nature, the characteristics of every organism are stored in the gene. When the organisms mate, parents' genes are reproduced, crossover, mutated and pass on to the offspring. The population will gradually improved by the natural selection.
   Genetic algorithm is modeled on Darwin's theory [18]. The basic terms and ideas are widely accepted. Here I just explain it in brief.
   Normally, species-evolution acts on a population of individuals to complete natural selection of best ones, which has high fitness to environment. To be the same as species-evolution, GA acts on a population of individuals and seeks patterns (genetic material) of the individuals, which upgrade fitness. For optimization problem, individuals are represented as solutions of problem. Generally, Patterns (genetic material) of the individuals are encoded to bit string like gene on a DNA chromosome, which determines the feature of species and patterns of the individuals. Bit string is often called as chromosome. These chromosomes are each evaluated by the fitness function. Population can be overlapped (there are individuals that contain same chromosome in the population) or no overlapping (each chromosome of individuals are different).
   Like natural selection, GA also uses fitness value to select individuals in the current population to mate and reproduce new offspring. Reproduction is the recombination of the individuals, which has two parent individuals per offspring typically. Each new set of offspring and any surviving parents is called a generation.

The outline of standard GA [1] is showed in Figure 6.1.

```
GA()
{
    do{
        Select promising individuals from current population

        Select individuals and the previous population left off

        Cross and Mutate to reproduce selected individuals

        Evaluate created individuals
    }
    while(finished())

    return best from population
}
```

Figure 6.1: Standard GA Procession

Evolutionary operators, crossover and mutation, are mimicking the basic process of reproduction.
Crossover places the role of combination the parents's gene to create a new offset. According fitness value, the best chromosomes' data are selected and mixed, hopefully producing a better next generation. Generally, each offset has two parents.
During mutation, each part of chromosome has a certain probability, generally very low, to change. New individual in a population is introduced by this operation and we can search for a new solution (not only congregate the individuals that already exist in the population).
The Pros of Genetic algorithms is that they are adaptive learning heuristic. GAs are also robust and effective algorithms, whose computational complex is simple [3]. GA is significantly different from heuristics such as SA and TS and thus complements these algorithms. The difference are showed below [16]:

- GAs search from a set of solutions and do a multi directional search in search space. This is similar to parallelly running several Tabu search. There is higher probability to get global optimal result.
- GAs needn't continuous searching space. It is more suitable for real life example, which generally has discontinuous search space.
- GAs are nondeterministic algorithms which are stochastic in decision. This makes them more robust.
- GAs have many parameters and these parameters are hidden and

depend non-linearly on each other. It is hard to find optimal combination of these parameters.

GA is suitable for many different problems. Until now, many different versions of GAs are proposed for different problems. But in order to work as effective as possible, the following basic items need to be carefully considered for all genetic algorithms[24].

- A GA framework
- A good genetic representation of a solution in a form of a chromosome.
- Fitness function to evaluate fitness value of each solution.
- A initial population constructor, generally it is a random constructor.
- Genetic operators, such as selection.
- Values for parameters, population size, mutation probabiltiy, etc.

The argument against GA is GA has so many parameters and it is very time-consuming to find out the best combination of all these praramters. For sparing time, in this project, I tried to find the best value by experiment for critical GA parameters, such as mutation rate. Other parameters is selected according to the optimal value found in the related works. The detail showed in the next section.
Please note, this chapter doesn't describe fitness function because it is introduced in previous chapter.

2. GA framework for ATPG
   Besides standard GA, there is another best-known GA [1], steady state GA (SS), showed in Figure 6.2. In many optimization problems, it works better than standard GA. But here it isn't considered, because it can't exploit parallel logic simulate. In order to utilize parallel logic simulation, a new generation should be evolved first, then parallel logic simulation can be used to calculate these individual's fitness value. Whereas, for steady state GA, two new individuals are generated at first, then their fitness values should be calculated immediately. Here I focus on standard GA, which is showed in Figure 6.1.
3. **Selection**
   Various selection schemes will generate different results. There are many selection schemes have been used. Here I will focus on proportional selection (PS) and stochastic universal selection (SUS).
   Proportional selection uses a roulette wheel, which is sized according to the fitness of each individual in the population. An individual is selected by spinning the roulette wheel. An example is showed in Figure 6.3. Stochastic universal selection uses a roulette wheel, which has

```
Stead-State_GA()
{
    do{
        p1; p2; p3; p4 ← Select 2 individual from population

        o1; o2 ← Crossover(Best(p1; p2); Best(p3; p4))

        o01; o02 ← Mutate(o1; o2)

        r1; r2; r3; r4 ← Random Select 4 individual from population

        Replace the worst individual between r1and r2 with o01 in population

        Replace the worst individual between r3 and r4 with o02 in population
    }
    while(finished())

    return best from population
}
```

Figure 6.2: Stead state GA Procession

N equidistant marks and N is the number of individuals in the population. The number of copies of each individual selection is equaling to the number of marks inside the corresponding slot. An example is showed in Figure 6.4.

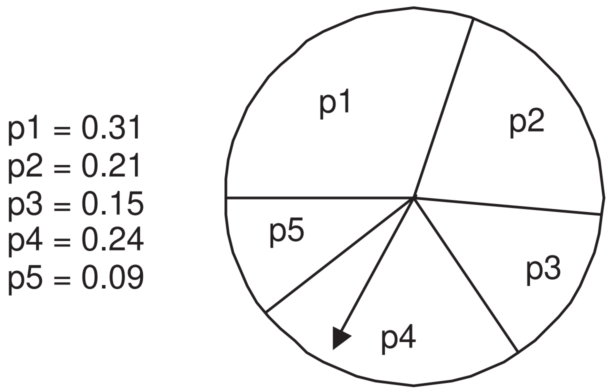


Figure 6.3: Proportional selection

[9] shows stochastic universal selection is a more fair method and with less noise in the sampling of new individuals than Proportional selection. In this implementation, I compare these two selection methods by experiment results.

p1 = 0.31
p2 = 0.21
p3 = 0.15
p4 = 0.24
p5 = 0.09

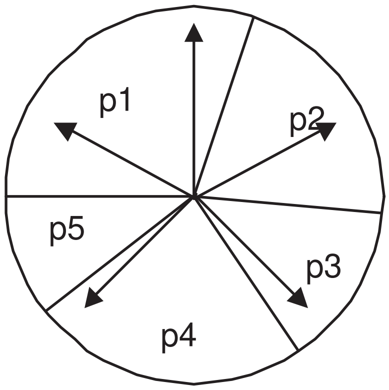


Figure 6.4: Stochastic universal selection

The population size is small in this implementation, which will be showed in next section and controlling of the number of copies is very important in small population genetic algorithms. In general, at the start of GA's running, there are some extraordinary individuals in a population of ordinary colleagues. If using proportional selection or stochastic universal selection, the extraordinary individuals would take over a significant proportion of the population in a single generation. This causes the premature convergence and it is undesirable. In the beginning, the difference of fitness should be shrunk. Later when the run becomes matures, there may still be a significant diversity within the population, but the average fitness is close to the population's best fitness. Average members and best members get almost same numbers of copies in future generation. It doesn't encourage a healthy competition among near equal fitness values. At that time, the difference of fitness should be amplified.

Fitness scaling [18] can help in these situations and linear scaling is a useful and simple scaling method. Linear scaling requires a Linear relationship between the raw fitness f and the scaled fitness f'.

$$f^{'} = af + b \tag{6.1}$$

Because that the best individual is copied one time more than average individual in generation is desirable in generation for small population [18], so a and b is selected to scale f like Figure 6.5.

Toward the end of the run, the average fitness may be near Max fitness. If fitness value is scaled by above method at that time, minimal fitness value will be negative, which is showed in Figure 6.6. This is also undesirable.

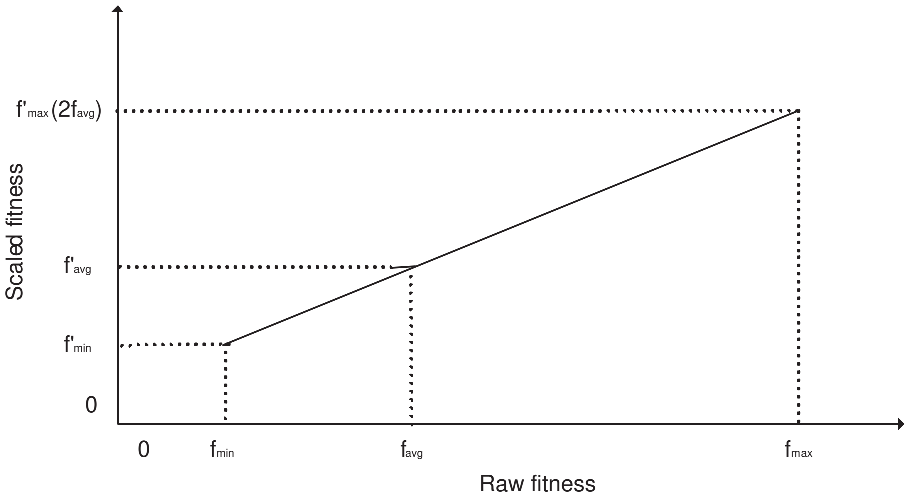


Figure 6.5: Linear scale

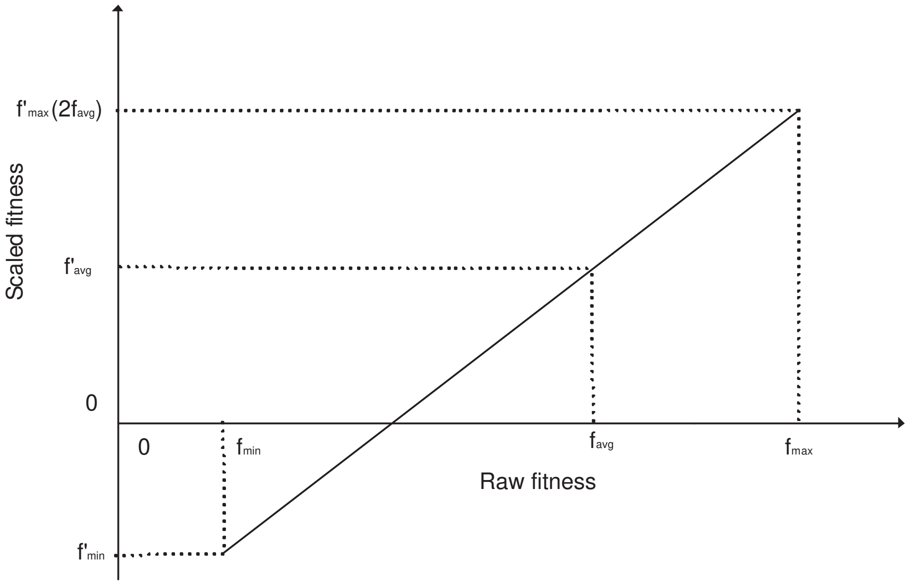


Figure 6.6: Undesirable Linear Scale

At that time, I can't scale

$$f^{'}_{m}ax = 2f^{'}_{a}vg \tag{6.2}$$

I just map the minimal raw fitness 'fmin' to a scaled fitness 'f'min' = 0 to scale it as much as possible, which is shown in Figure 6.7.
In this way, linear scaling helps prevent premature in early stage and speed up improvement in the end run.

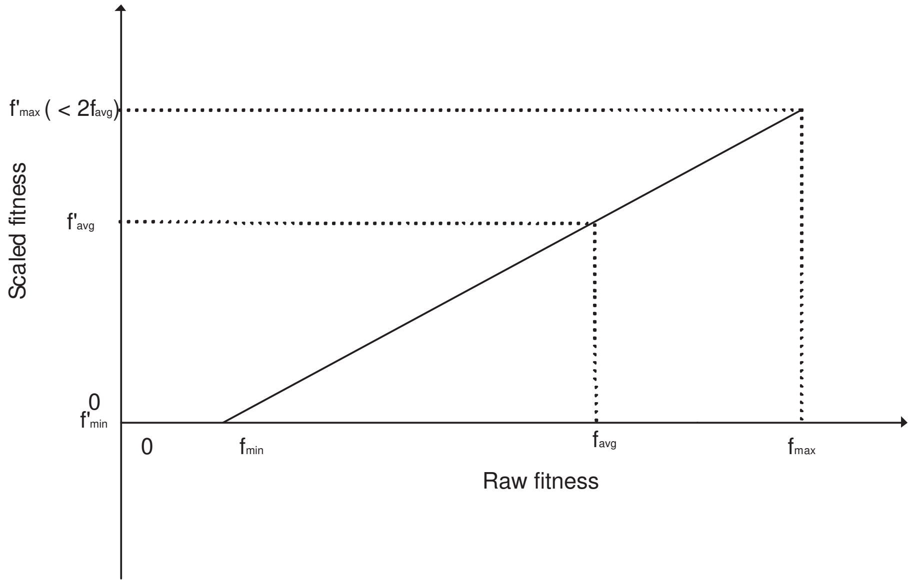


Figure 6.7: Modified Linear Scale

4. GA parameter
   Various GA parameters are important in achieving good results.
   Firstly, **chromosome's organization** is considered. In my case, binary-coded string is used to represent chromosome (population). Each bit in string represents a primary input and each string represents one test vector. **The initial population** is generated with random seed.
   Next, I consider **the size of population**. A sufficient population is needed to provide good combinations of characters. However, a reasonable limit on the population size is needed to reduce computation. As [9] presented, the best size of population in ATPG GA is 16 or 32. Increasing size can't get much improvement but increase run time. [9] also shows that for ATPG, overlapping population is faster than no overlapping population, almost without any negative influence with fault coverage. Considering ability of parallel simulation, I select size of population 32 with overlapping. Termination condition is another critical condition, which can limit the number of repeatation. There are three conditions that will terminate generation.

   (a) Convergence condition. The longest hamming distance between highest fit individual and other individual is smaller than N / 10, N is the number of flip-flops.
   (b) The best fitness value has no improvement in the latest ten iteration.

(c) The number of iteration is greater than 600.

Crossover is a very important operation in GAs, which provides a method for creating new solution from two fit parents, which are likely to be helpful in improving the fitness. There are several techniques can be used in crossover.

(a) **One point crossover**
One index (swap point) in the chromosome string is selected. All data beyond that point in the chromosome string is swapped between the two parent organisms. The getting chromosomes are the children. An example is shown in Figure 6.8. There are two methods to implement one point crossover, fixed-index and random-index. Each time, when crossover, for fixed-indes, the swap point is fixed and for random -index, the swap point is randomly.

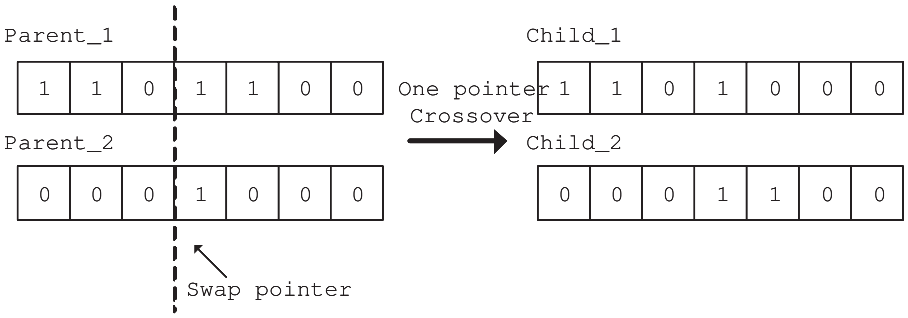


Figure 6.8: One point crossover

(b) **Two point crossover**
Two indices (swap points) are selected in the chromosome. Everything between the two points is swapped between the parent chromosomes, rendering two child chromosomes. An example is showed in Figure 6.9. There are also two methods to implement two point crossover, i.e. fixed-index and random-index.

(c) **Uniform crossover**
Uniform crossover is the extreme case of multipoint crossover, for each bit: it takes the value from one of the parents at random. An example is showed in Figure 6.10.
Studies [5] have shown that uniform crossover is superior to one and two-point crossover. The advantage of uniform crossover is that it performs many different combinations of schemata much more quickly than one or two point crossover. When chromosome length is long the emphasis of a GA is on performing more combinations of chromosome to obtain new schemata. Therefore,

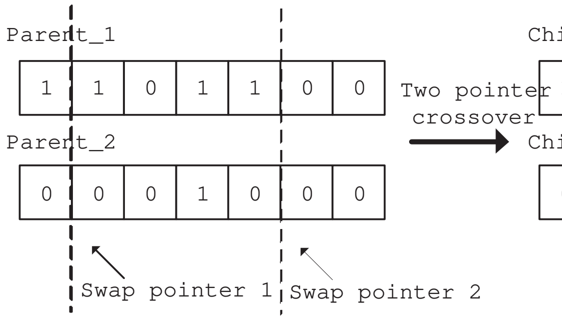


Figure 6.9: Two point crossover

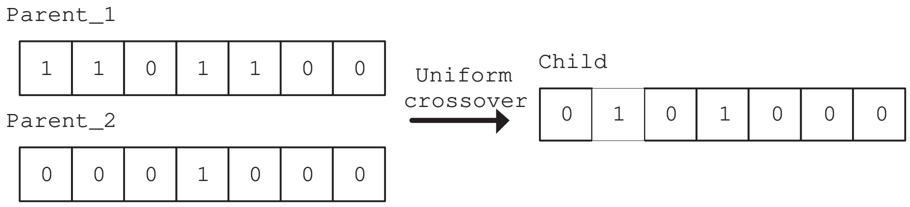


Figure 6.10: Uniform crossover

uniform crossover surges forward in this area. Because in the sequential ATPG, the chromosome's length generally is quite longer, which is shown in Equation 6.3, so uniform crossover is selected for this implementation.

$$The\ length\ of\ Chromsome = N \times M \tag{6.3}$$

- N is the number of primary input.
- M is the number of test vectors in one chromosome.

Mutation rate is a very critical and problem depend parameter, which is used to prevent the loss of key characters at the various string positions. But, mutation also destroys good combinations of characters, so a balance must be found. I tried mutation probabilities from 0.01 to 0.5 respectively to find the best mutation rate of the circuits.
In summary, the GA parameters, selected in this implementation are shown in Table 6.1.

6.2 Tabu Search algorithm

I wish one could have more chance to gain an efficient way for solving this problem. Therefore here I choose another method-Tabu search algorithm.

Parameter Name	Value or technology selected	Comment
GA algorithm	Standard GA	
Chromosome	Bit string	
Population size	32	
Population type	Overlapping population	
Selection Method	PS or SUS	Determined according to experiment results
Fitness scaling	Linear scaling	
Crossover	Uniform crossover	
Mutation	Not decided	Determined according to experiment results

Table 6.1: GA parameter

Tabu search (TS)[17] is proposed in its current form by Fred Glover in 1986 and An enormous amount of applications and variations of TS has been proposed since then to solve a wide range of hard optimization problems such as job shop scheduling, the traveling salesman problem, resource planning, telecommunications, VLSI design, financial analysis, scheduling, space planning and energy distribution.

TS is an iterative procedure designed for the solution of optimization problems. It is "a meta-heuristic superimposed on another heuristic". The basic idea of Tabu search is to explore the search space of all feasible solutions by a sequence of moves and some moves in Tabu list are forbidden to avoid entrainment in cycles (Hence "Tabu") and escape from locally optimal but not globally optimal solutions. Tabu-list contains moves which have been made in the recent past but are forbidden for a certain number of iterations. Sometimes, a Tabu move can be overridden, when it is deemed favorable. Forgetting that a move is Tabu could lead to a solution which is the best obtained so far.

An iteration in a Tabu search showed in Figure Figure 6.11.

From Figure 6.11, it can be seen there are three most important elements in the Tabu search. The first is neighbor definition (move definition). The second is the Tabu list, which help identify cycling. The use of Tabu list is showed in Figure 6.12. The last is neighbor evaluation, which help to find best neighbor.

Processing in Figure 6.13 is the basic steps I design and several important variables is emphasized here.

First, the neighborhood, search space in one iteration, is created by inverting

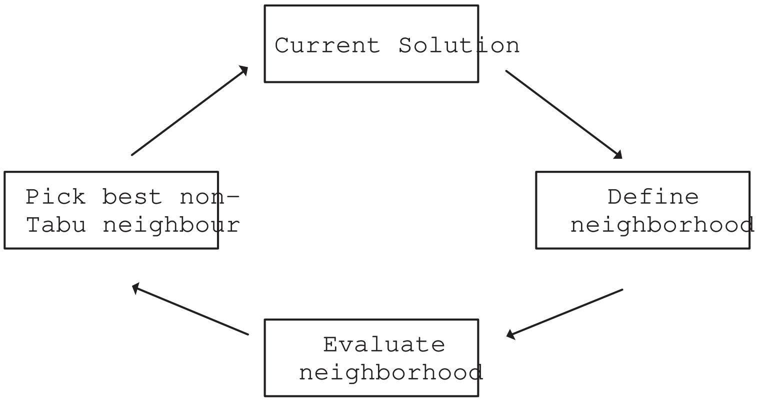


Figure 6.11: Tabu iteration

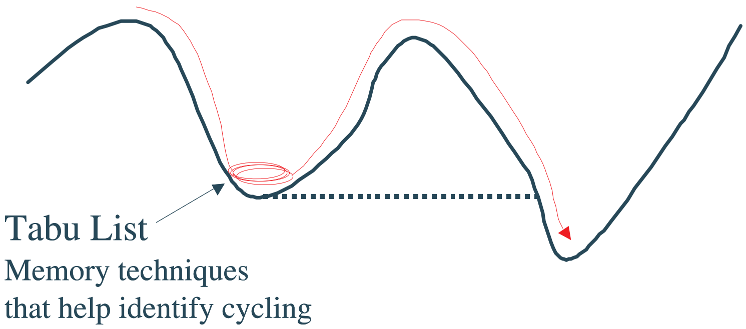


Figure 6.12: The use of Tabu list

calculation for each binary number of the original binary string or the best one from last calculation and hamming distance is only one between original one. Selecting hamming distance one is because the length of binary string (like chromosome in GA) is general long, the number of neighborhood is exponentially growing with the length of difference and Tabu search need calculate all neighborhood's fitness value. This let longer hamming distance impossible. Compare GA, Tabu search's search space is too narrow. Second, to speed up searching in Tabu list. A hash table is created to be Tabu List that contains calculated and discarded binary string. To evaluate neighborhood, fitness function which showed in chapter 5.2 is used. Finally, for limiting the calculation and comparing with GA, stopping criterion for this problem is the same as GA that, if cannot find an objective value which is bigger than that of 10 times before and most iteration for 600 times, stop the calculation.

[40] shows the advantage of Tabu search is not bounded by linearity and yields relatively better solutions than previously intractable problems, the

```
TABU()
{
    Choose initial solution s          //randomly input a sequence of  a binary code string;

    s*=s; k=1                          //s*=current best solution which is derived from a binary
    repeat                             //code string I just input; k: the number of circulation
    {                                  //that responds to the stopping condition

        Generate VN(s, k) N(s)         //here I create the neighborhood of this binary number by
                                       //Boolean inversion calculation for every digit, and a
                                       //Tabu list memory that contains history moves

        Choose the best s^ in V        //choose a best binary number in the neighborhood I just
                                       //created that can bring the maximum value for the
                                       //polynomial

        s=s^                           //this binary number that denoted by s^ is the best
                                       //solution in the neighborhood in the whole
                                       //neighborhood

        if f(s)<f(s*)then s*=s         //compare the best binary number in the neighborhood to
                                       //the original number, choose a better one that can bring
                                       //the maximum value for the polynomial.

        k=k+1                          //repeat above steps for the best binary number I just
    }                                  //chose

    until stopping condition is met    //here I decide that the number of iteration great than
                                       //600 and after 10 times circular calculation for the
                                       //above steps, if no improvement (here means cannot
                                       //find an objective value which is bigger than that of 10
                                       //times before) happens for the objective value, stop the
                                       //calculation.
    return s*
}                                      //the best solution in search processing
```

Figure 6.13: Tabu search processing

deficient of Tabu Search is it assumes fast performance evaluation. In other words, if neighborhood evaluation needs long time, the performance of Tabu search maybe poor. Unfortunately, evaluating a test vector's fitness generally takes relative long time, this foreshow Tabu search yields poor performance than GA. Experiment results also proof it, which is showed in Table 9.4.

Chapter 7

Test

In this project, test is a really hard work and this is because of two reasons. The first, it is a very complicated system and there are about 20,000 lines C# code in the system. The second, the algorithm used in this work is undertermined algorithm, it needs pseudorandom generator and each run of test generation is different.

For the second problem, I just fix the seed of pseudorandom generator to generate same sequence input steam. Then test generator can generate same test sequence.

For the first problem, divided and conquer design principle is used to solve problem. Total test work is divided into four stages and I try to debug bugs as early stages as possible, because the cost of tracing a bug is much cheaper in early stages.

1. Test for each procedure.
2. Test for each class.
3. Test for each function block in system framework, which is showed in Figure 2.1.
4. Test for whole system.

In the first stage and second stage, statement coverage and marginal test is used. Simple test case is designed for test. In these two stages, because test block is not complicated and I try to find as many bugs as possible, I trace each sentence to find potential faults. Most of bugs are found in these two stages in this project.

In the third and last stage, because the test block is too complicated for simple test case to cover each statement, ISCAS89 benchmark circuits are used in test. Because benchmark circuits are general very large, it is hard to judge the test result whether it is right or not. In this case, test program is designed to analyze result. For instance, I test circuit simulator block, which includes X algorithm, critical path tracing and differential fault simulator, by benchmark circuit 's5378' which has 3000 gates and the result showed that X algorithm detected 200 faults, differential fault simulator detected 190 faults and critical path tracing detected 170 faults. Of course, I don't know whether these faults are all really detected by the test vector or not. So I use two test programs to analyze the result. Because the first test program is designed according to Formula 7.1. If violating the rule, there are bugs in system. Another test program is compare the results from my simulator with a parallel fault simulator download from CAD web site to determine whether there are bugs or not.

$$CPTDF \subseteq DFDF \subseteq XDF \tag{7.1}$$

- CPTDF: detectable faults determined by critical path tracing
- DFDF: detectable faults determined by differential fault simulation
- XDF: detectable faults determined by X algorithm

In some cases, I can determine the result by funtional specification. For instance, I test a logic simulated based ATPG (LSG). The aim of LSG is to generates test sequence that contain new partial states as much as possible, if some old partial states are frequently appears,then there are bugs in program.

In the last stage, bug is hard to trace and fix. It often takes one or two days to fix a bug. In this project, all detected bugs are fixed.

Chapter 8

Benchmark test run on C++ and C#

This project is implemented by C# on Visual Studio .NET 2002 development environment. The .NET languages are ”semi-compiled” like Java. Source code in the .NET world is compiled into the Microsoft Intermediate Language (MSIL) and is run on the .NET Common Language Runtime (CLR) engine. Other ATPG already published is mostly written by C++. Fully compiled language like C++ is faster than semi-compiled language. Benchmark test will compare the algorithm's efficiency between these ATPGs.

In proposed ATPG, two data structures, hash table and array list (a data structure combination array and list), and two operations, float point comparison and addition, are widely used. So I wrote three benchmark test cases.

- Hashtable: The hashtable test primarily measures the efficiency of the default hash implementation and function, i.e. create a new hash table, add item and access item. For c#, It uses class hashtable, and for C++, map list is used:
- The List test primarily measures the efficiency of the default list implementation. For C#, class Arraylist is used, and for C++, vector is used.
- Bubble sort: bubble sort measure the comparison and addition of float point number. In each item, there are two float point numbers. Item's value equals the addition of the two numbers in the item. Bubble sort test is sorting a number of items according to item's value.

C# is compiled by Microsoft's Visual Studio .NET 2002. C++ is compiled by Microsoft's Visual C++ version 6.0. I wanted to let the compilers optimize as much as possible. The optimization settings I settled on were as follows:

	Bubble (s)	Hash (s)	List (s)
C#	10.56	6.24	0.41
C++	0.581	0.51	0.11
Times of speed up	18.2	12.2	3.7

Table 8.1: Comparison C# and C++

- Visual C#: used ”release” configuration, turned on ”optimize code” within Visual Studio.
- Visual C++: used ”release” configuration, turned on ”whole program optimization”, set ”optimization” to ”maximize speed.”

Before running each set of benchmarks I defragged the hard disk, rebooted, and shut down unnecessary background services. I ran each benchmark at least three times and selected the best score from each component, assuming that slower scores were the result of unrelated background processes getting in the way of the CPU and/or hard disk. Start-up time for each benchmark was not included in the performance results. The benchmarks ran on the following hardware:

Type: Compaq HP D530 CMP

Desktop CPU: Pentium 4- 2GHz

RAM: 512MB

OS: Windows XP Pro SP 1

File System: NTFS

From Table 8.1, I can observe C++ is much faster than C#. Because hash table is much more dominated in my ATPG (gate node and fault list is saved in hash table) than array list, I estimate that if my ATPG is written by C++, the speed in performance will be increased more than 15 times.

Chapter 9

Experiment Result

The GA-based and Tabu search based test generator includes sequential circuit parallel differential fault simulation and X algorithm fault simulation. The circuit analysis and simulation part is about 6000 lines C# code. I implemented two GA algorithms, standard GA with linear scale proportional selection and standard GA with linear scale stochastic universal selection. Each GA algorithm and Tabu search algorithm are about 1200 lines including 200 lines code for fitness function calculation. It is compiled in Microsoft's Visual Studio .NET 2002 Version 7.0.9466 and running on Microsoft .NET Framework 1.0 Version 1.0.3705. Tests were generated for the ISCAS89 sequential benchmark circuits on a Pentium 4 2 GHz processor with 512M memory.

As showed in section, I fixed the number of population to 32, max number of generation to 600 to limit the execution time, max length of test sequence to 20000. In finish condition, I fixed max length of last generation, which is not improve fitness value, to 10 and max length of last test sequence undetected fault to 4000 to limit the execution time. I also mark the last detected vector to the sequence length and run 3 times for each circuit. (I can't run more times for each circuit because it is too time consuming. Just for running three times, it take me 4 weeks to simulated all the test cases in four workstations). Each result is the average from three run and round to integer. A new random seed was used for each run.

There are three parameters, selection method, length of test sequence in a chromosome and mutation rate, need to be determined by experiment results. To find the best combination parameters for proposed LSG. The best selection method and length of test sequence in a chromosome are found in the first experiment, which fixed mutation rate. Then best mutation rate are

Sequence length in a chromosome	Best numbers		Total detected faults		Avg time of generation a vector (s)	
	SUS_GA	PS_GA	SUS_GA	PS_GA	SUS_GA	PS_GA
1	2	**9**	8102	**8160**	0.12	0.15
10	5	**6**	8938	**8971**	0.28	0.3
20	6	**8**	8919	**8990**	0.37	0.38
40	**9**	5	**8907**	8868	0.6	0.61
100	6	**8**	8727	**8743**	0.69	0.96

Table 9.1: Experiment results for various sequence length in a chromosome

found in the second experiment, which use best selection method and length of test sequence in a chromosome found in the first experiment.

In the first experiment, I want to compare proportional selection and stochastic universal selection and find the optimal length of test sequence in a chromosome. I fixed mutation rate to 0.01. I tried 5 different lengths of sequence, 1, 10, 20, 40 and 100. Standard deviations are given in parentheses.

The result is showed in appendix's Table A.1 to A.10. In Table A.1, A.2, A.8, I also calculate the standard deviation for detected faults. The results show all deviation is less than 2% of the number of detected faults. It is a very low value. The critical data is counted in Table 9.1.

In Table 9.1,

- "SUS_GA" means genetic algorithm with linear scale stochastic universal selection.
- "PS_GA" means genetic algorithm with linear scale proportional selection.
- "Best numbers" means how many best results (detect the most faults) each selection method achieves. For instance, compare PS_GA, SUS_GA find more fault in circuit S838 and s1494 in sequence length 1, so Best numbers is 2.
- "Total detected faults" is addition of detected fault in all test circuits.
- "Avg time of generation a vector (s)" is average time GA finds a vector. The default time unit is second.

Though stochastic universal selection is less noise than proportional selection, from Column "Best numbers" and "Total detected faults", proportional selection looks better than stochastic universal selection. For natural selection, individual is randomly selected one by one according their fitness value, which is like proportional selection and GA is an algorithm that mimics natural selection. I think this is one reason why proportional selection's performance is better than stochastic universal selection here. Another reason is, according to the column "Avg time of generation a vector (s)", "PS_GA" use longer time than "SUS_GA" in generating a vector. This means there is more iteration in generation of "PS_GA" than "SUS_GA". More iteration leads to find better result.

The effectiveness of global optimization in the end of section can be proofed here. From "Avg time of generation a vector (s)", I also can observe the time increases when sequence length increases. This is because search space is exponentially extended by sequence's length in a chromosome. To find optimal result, it needs more iteration. Column "Best numbers" and "Total detected faults" shows with sequence length increasing, the results become better at first, but results become worse when length is larger than 10 (SUS_GA) and 20 (PS_GA). This is because increase sequence's length can help to find global optimal value, but this will extend search space and finding optimal value will be harder and harder. Table 9.1 also shows the balance sequence length for "SUS_GA" is 10 and for "PS_GA" is 20.

Next, the effects of mutation rate on fault coverage were investigated. Because Table 9.1 shows that PS_GA with sequence length 20 can detect the most faults, I just use its GA parameters and various mutation probabilities from 0.01 to 0.5 respectively to find the best mutation rate. The result shows in appendix's Table A.8 and Table A.11 to Table A.16.

Table 9.2 shows the critical value.

mutation rate	Best numbers	Total detected faults	Total sequences' length
0.01	6	**8990**	109320
0.025	3	8955	86140
0.05	**7**	8981	103080
0.075	4	8933	98820
0.1	3	8876	115780
0.25	2	8726	99660
0.5	3	8692	120100

Table 9.2: Experiment results 1 for various mutation rate

Meaning of "Best numbers" and "Total detected faults" is the same as Table 9.2. "Total sequences' length" means total test sequences' length generated for every test circuit. Mutation rate 0.05 gets the best result in column "Best numbers" and mutation rate 0.01 gets the best result in the results of column "Total detected faults" and when mutation rate greater than 0.05, total detected faults just decreased with mutation rate increasing. Especially for mutation rate 0.5, total sequences' length is the largest but faults detected is least. From these data, it can be concluded that mutation rate around between 0.01 and 0.05 is best for proposed ATPG.

According Figure 1.1, ATPG's run time is divided into meta-heuristic algorithm and fault simulation run time. Which one is dominated? Table 9.3 shows GA run time is dominated. In most of test circuit, about 90%'s CPU time is used for GA algorithm to generate test sequence and only 10% CPU time used in fault simulation. This tells us that metaheuristic algorithm is very important to ATPG, improving performance of metaheuristic algorithm will lead great improvement of ATPG.

As shown in chapter 5.4, in the end of test generation, the combination GA need 32 times X algorithm to find best test vectors. X algorithm needs much shorter time than fault simulation, so the overhead of 32 times of X algorithm is not high.

Circuit	GA Run Time(s)	Total Run Time(s)	GA Run time / Total Run Time	Fault coverage
s349	678	725	0.935	0,922
s510	614	654	0.939	0,957
s641	1390	1544	0.900	0,865
s713	1366	1529	0.893	0,826
s838	5516	7741	0.712	0,431
s1196	6071	6547	0.923	0,987
s1238	5937	6450	0.920	0,938
s1423	9087	11950	0.760	0,793
s1488	7121	7711	0.923	0,952
s1494	7618	8259	0.922	0,948
Total	45398	53110	-	-
Average	-	-	-	0.8619

Table 9.3: Experiment results 2 for various mutation rate

Then, I simulated Tabu search algorithm based ATPG. Like GA, I fixed max number of generation to 600 to limit the execution time, max length of test sequence to 20000. In finish condition, I fixed max length of last generation, which is not improve fitness value, to 10 and max length of last test sequence undetected fault to 4000 to limit the execution time. I also optimize for test sequence length of 10 instead of only one test vector. Table A.17 in appendix shows the simulation result. Table 9.1 shows the comparison of PS_GA and Tabu search.

PS_GA's Data is from appendix's Table A.7. One chromosome also contains 10 test vectors like Tabu search. Table 9.4 shows in each test circuits, PS_GA can find more faults than Tabu Search, whereas total run time and sequences' length generated by Tabu search is much longer than PS_GA. This is because of two reasons.

- In ATPG, neighborhood evaluation generally takes long time, this lead Tabu search to yield poor performance.
- GA has more probability to get global optimal result than Tabu search. GA search from a set of solutions and do a multi directional search in search space. This like parallel running several Tabu search.

Comparing with GA, Tabu search is not a good choice in our case.

	Best numbers	Total detected faults	Total sequences length	Total run time
Tabu Search	0	7180	97550	95446
PS_GA	6	7658	81350	32087

Table 9.4: Comparison of PS_GA and Tabu

Then I tried to simulate and find best parameter for combination method based ATPG (CSG), which combining LSG and FSG. In last experiment, it can be observed that proportional selection is better than stochastic universal selection, best mutation probability is in the range from 0.01 to 0.05 and the best length of test sequence are in the range from 10 to 20. This observation is used in this Experiment to select the parameters. I fix the selection method in proportional selection and change mutation rate from 0.01 to 0.05 and step is 0.01 and select the length of test sequence in chromosome 10 and 20. According to the mutation rate and length of test sequence in a chromosome, simulation is divided into 10 groups. Which is showed in Table 9.5.

The benchmark circuits used in simulation are s349, s510, s641, s713, s838, s1196, s1238, s1423, s1488, s1494. The result is showed in appendix Table A.18 to Table A.27. The critical data is counted in Table 9.6.

From Table 9.6, it can be seen that selecting 10 test vector in a chromosome is better than 20. Group 1 and 3 can detect the most faults. Group 2 has shortest test sequence and run time. Trade-off test sequence's length and fault coverage, I compare Group 2 with PS_GA which Data is from appendix's Table A.7. The result showed in Table 9.7.

From the Table 9.7, it can be observed that in all benchmark circuits, CSG's performance is better than PS_GA. On average, CSG improves the fault

Group	Mutation rate	length of test sequence in a chromosome
Group 1	0.01	10
Group 2	0.02	10
Group 3	0.03	10
Group 4	0.04	10
Group 5	0.05	10
Group 1	0.01	20
Group 2	0.02	20
Group 3	0.03	20
Group 4	0.04	20
Group 5	0.05	20

Table 9.5: Groups in experiment

Group No.	Total detected faults	Total sequences length	Total run time
Group 1	**9127**	66340	37089
Group 2	9112	**32170**	**20203**
Group 3	**9127**	53560	31490
Group 4	9096	52540	34118
Group 5	9117	71170	48193
Group 6	9063	85820	46430
Group 7	9063	65860	36886
Group 8	9078	70410	47667
Group 9	9090	87600	61101
Group 10	9081	83600	53746
total (Sequence length in a chromosome is 10)	45579	275780	169693
total (Sequence length in a chromosome is 20)	45375	393290	245830

Table 9.6: Experiment of CSG

coverage about 1.4%, decrases sequence's lengh about 68.8% and accelerate the run time about 52.5%. Because CSG need 32 times more X algorithm in each test generation, the improvement of run time is less than improvement of sequence's lengh. From the data shows in Table 9.7, compare LSG, CSG improves the performance much.

Last, I compare CSG with the following art of state ATPG.

Circuit	Flt.det			Sequence Length			Run Time		
	CSG	PS_GA	Improve	CSG	PS_GA	Improve	CSG	PS_GA	Improve
s349	342	332	3%	70	300	76.7%	16	50	68%
s510	564	558	1.1%	530	4200	87.4%	212	334	36.5%
s641	406	404	0.5%	470	1620	71%	227	445	49%
s713	480	480	0%	530	1940	72.7%	169	499	66.1%
s838	478	462	3.5%	510	17500	97.1%	488	6773	92.8%
s1196	1236	1229	0.6%	6300	10980	42.6%	3321	4798	30.8%
s1238	1277	1279	0.2%	3080	15080	79.6%	1569	5097	69.2%
s1423	1428	1347	6%	7230	19160	62.3%	7244	11448	36.7%
s1488	1446	1446	0%	9250	16820	45%	4123	6484	36.4%
s1494	1455	1453	0.1%	4200	15480	72.9%	2834	6563	56.8%
Total	9112	8990	1.4%	32170	103080	68.8%	20203	42491	52.5%

Table 9.7: Compare PSGAand CombinationGA

- Hitec [25], a deterministic ATPG.
- Strategate [23], a GA-fault simulation based test generator.
- Proptest [28], a compaction fault simulation based test generator.
- Locstep [26], a logic-simulation-based test generator that targets maximizing global states visited (not state partitioning).

The result in Table 9.8 show that in all test cases, our approach which use state partitioning detects more fault than Locstep which is also a logic-simulation-based ATPG, but doesn't use state partitioning. This shows state partitioning is useful in removing the noise and guide the search direction. Compare other fault-simulation-based ATPG and deterministic ATPG, Our approach is good. It detects the most faults in six of nine test circuits. In the rest of the test circuits, our approach performed worse than two fault-simulation-based test generators. This is because our partition algorithm is a heuristic and might not always provide best searching direction than others. Our approach's run time is much longer than other approach. There are two reasons. The first, C# compiler is a semi-compiler. The performance of program written by C# is much slower than C and C++. Considering this, our implementation is not much slower than others. The second, I just finish these code in three months, because of time limitation, there are sveral optimization techniques not implemented and time limitation also doesn't allow me do any code optimization, whereas, other ATPG used their university's high performance library of fault simulation and GA.

circuit	Proposed		Hitec		STRATEGATE		Protest		LOCSTEP	
	flt.det	Time	flt.det	Time	flt.det	Time	flt.det	Time	flt.det	Time
s349	**342**	16	332	462	NA	NA	NA	NA	NA	NA
s641	**406**	227	404	4.8	404	NA	404	30	404	NA
s713	**480**	169	475	6.7	476	79	476	37	475	NA
s1196	1236	3321	**1239**	5.5	**1239**	89	**1239**	28	NA	NA
s1238	1277	1569	**1283**	8.2	1282	NA	**1283**	43	1268	NA
s1423	**1428**	7244	723	50040	1414	4572	1416	277	1274	NA
s1488	**1446**	4123	1444	990	1444	NA	1444	119	1425	NA
s1494	**1455**	2834	1453	576	1453	456	1453	126	NA	NA
s5378	3635	62454	3238	941	3639	136080	**3643**	676	3059	7545

Table 9.8: Comparison with other ATPG

Chapter 10

Conclusion

An efficient sequential ATPG, which combines the advantage of LSG and FSG, is presented. A highly accurate fitness function based on state partition is used to evaluate candidate test vector in order to achieve good quality test sets. In metaheuristic algorithm, I optimize test sequence instead of single test vector to get global optimization. Because increasing length of test sequence in a chromosome will exponential increase search space and lead to search optimal result harder. I use experiment to find optimal result.

I use GA and Tabu search algorithm to find optimal test vector. Compare these two algorithms, Tabu search is not the ideal metaheuristic where neighborhood evaluation takes long time and it is not suitable in our case.

In GA, I investigate the effectiveness of stochastic universal selection and proportional selection. Based on experiment, I find proportional selection, which achieves high fault coverage than stochastic universal selection. Variations in mutation rate also had effect on fault coverage. Result from the ISCAS89 sequential benchmark circuits indicates the best mutation rate.

At the last, I compare CSG ATPG with other art of state ATPG. In most of test circuits, our ATPG detects the most faults. It approves our algorithm is efficient.

Chapter 11

Acronym

AS:	Ant systems
ATPG:	Automated test pattern generation
CLB:	Combinational logic block
CLR:	Common language runtime
CSG:	Combination-simulated based generator
DA:	Deterministic annealing
FF:	Flip-Flop
FFR:	Fan-out free region
FIFO:	First in first out
FSG:	Fault simulation based generators
FTP:	Forward time processing
GA:	Genetic algorithm
LSG:	Logic simulation based test generators
MSIL	Microsoft Intermediate Language
NN:	Neural Networks
PI:	Primary input
PO:	Primary output
PPO:	Output of the flip-flop
PPI:	Input of the flip-flop
PS:	proportional selection
PS_GA:	Genetic algorithm with proportional selection
RTP:	Reverse time processing
SA:	Simulated annealing
SUS:	Stochastic universal selection
SUS_GA:	Genetic algorithm with stochastic universal selection

Bibliography

[1] Evolutionary algorithms. *Lecture slides on Large-Scale Optimization 02715, DTU. URL http://www.imm.dtu.dk/courses/02715/*, 2004.

[2] T. M. Barnes. Using genetic algorithms to find the best generators for half-rate convolutional coding. *North Carolina State University*.

[3] A. S. Bjarnadoettir. Solving the vehicle routing problem with genetic algorithms. Master's thesis, DTU, 2004.

[4] K. K. Chang Kim. Yong, Saluja. Sequential test generators: past, present and future. *Integration, the VLSI Journal*, 26(1-2):41–54, December 1998.

[5] W.-T. Cheng and J. H. Patel. Proofs: a super fast fault simulator for sequential circuits. *European Design Automation Conference*, pages 475–479, March 1990.

[6] Y. S. D. G. Saab and J. A. Abraham. Cris: A test cultivation program for sequential vlsi circuits. *Proc. Int'l Conf. Computer-aided Design (ICCAD 92), IEEE CS Press, Los Alamitos, Calif.*, pages 216–219, 1992.

[7] C. Darwin. Britannica concise encyclopedia from encyclopia britannica. *URL http://concise.britannica.com/ebc/article?eu=387589*, 2004.

[8] T. M. N. E. M. Rudnick and J. H. Patel. Methods of reducing events in sequential circuit fault simulation. *Proc. Int. Conf. on Computer Aided Design*, pages 546–549, Nov. 1991.

[9] G. S. G. Elizabeth M. Rudnick, Janak H. Patel and T. M. Niermann. Sequential circuit test generation in a genetic algorithm framework. *IEEE-CAS : Circuits and Systems, ACM Press, New York, NY*, pages 698–704, 1994.

[10] A. G. et al. Efficient spectral techniques for sequential atpg. *Proc. IEEE Design Automation and Test in Europe Conf. (DATE 01), IEEE CS Press, Los Alamitos, Calif.*, pages 204–208, 2001.

[11] E. E. et al. Digital logic simulation in a time based, table driven environment-part 2 parallel fault simulation. *Computer*, 8:38–79, May 1975.

[12] E. M. R. et al. Application of simple genetic algorithms to sequential circuit test generation. *Proc. European Design and Test Conf. (ED and TC 94), IEEE CS Press, Los Alamitos, Calif.*, pages 40–45, 1994.

[13] E. M. R. et al. Fast sequential circuit test generation using high-level and gate-level techniques. *Design Automation and Test in Europe (DATE '98)*, pages 570–576, February 1998.

[14] J. B. et al. A new hybrid genetic algorithm for the capacitated vehicle routing problem. *Journal of the Operational Research Society, 41(2)*, pages 179–194, 2003.

[15] J.-F. C. et al. a guide to vehicle routing heuristics. *Journal of the Operational Research Society, 53*, pages 512–522, 2002.

[16] S. S. et al. Iterative computer algorithms with application in engineering: Solving combinatorial optimization problems, chapter 3. *IEEE Computer Society*, 1999.

[17] G. F. Tabu search A tutorial. *Interfaces*, 20(4), 1990.

[18] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Leraning*. Addison-Wesley, Reading, Mass., 1989.

[19] F. J. F. J.P. Shen, W. Maly. Inductive fault analysis of mos ic's. *IEEE Design and Test of Computers Vol.*, 2/12:13–26, December 1985.

[20] H. K. Lee and D. S. Ha. Hope: an efficient parallel fault simulator for synchronous sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9), September 1999.

[21] M. B. M. Abramovici and A. Friedman. *Fault Simulation in Digital Systems Testing and Testable Design*. Computer Science Press, Reading, MA, 1992.

[22] P. M. M. Abramovici and D. Miller. Critical path tracing - an alternative to fault simulation. *IEEE Design and Test of Computers*.

[23] E. M. R. M. S. Hsiao and J. H. Patel. Sequential circuit test generation using dynamic state traversal. *Proc. European Design and Test Conf. (ED and TC 96), IEEE CS Press, Los Alamitos, Calif.*, pages 22–28, 1996.

[24] Z. Michalewicz. *Genetic Algorithm + Data Structures = Evolution Programs*, volume 3rd. Springer-Verlag, revised and extended edition edition, 1996.

[25] T. M. Niermann and J. H. Patel. Hitec: A test generation package for sequential circuits. *Proc. European Design Automation Conf. (EURODAC 91), IEEE CS Press, Los Alamitos, Calif.*, pages 214–218, 1991.

[26] I. Pomeranz and S. M. Reddy. Locstep: A logic-simulation-based test generation procedure. *IEEE Trans. CAD of Integrated Circuits and System*, 16(5):544–554, May 1997.

[27] I. Pomeranz and S.M.Reddy. Fault simulation based test generation for combinational circuits using dynamically selected subcircuits. *Proc. Int'l Conf. Computer Design (ICCD 99), IEEE CS Press, Los Alamitos, Calif.*, pages 412–417, 1999.

[28] I. P. R. Guo and S.M.Reddy. A fault simulation based test pattern generator for synchronous sequential circuits. *Proc. VLSI Test Symp. (VTS 99), IEEE CS Press, Los Alamitos, Calif.*, pages 260–267, 1999.

[29] K. T. S. Sheng and M. Hsiao. Effective safety property checking using simulation-based sequential atpg. *Proc. Design Automation Conf. (DAC 02), ACM Press, New York*, 2/12:813–818, 2002.

[30] B. K. S.B. Akers, S. Park and A. Swaminathan. Why is less information from logic simulation more useful in fault simulation? *1990 IEEE International Test Conference*, pages 786–800, 1990.

[31] S. Sheng and S. Hsiao. Efficient sequential test generation based on logic simulation. *IEEE Design and Test of Computers*, 19(5):56–64, May 2002.

[32] T. Siriwan and P. Nilagupta. Hpgast: High performance ga-based sequential circuits test generation on beowulf pc-cluster. *ANSCSE2000*, 2000.

[33] K. Son. Fault simulation with parallel value list algorithm. *VLSI Systems Design*, 6/12:36–43, December 1985.

[34] F. Stassen. Test pattern generation. *in Lecture Notes on Test of Digital Logic ID, DTU*, pages 3.9–3.19, January 1992.

[35] F. Stassen. Testability analysis. *Lecture Notes on Test of Digital Logic ID, DTU*, pages 5.1–5.9, January 1992.

[36] F. Stassen. Fault modeling. *Lecture slides on Test of Digital Logic ID, DTU*, 2003.

[37] F. Stassen. Fault simulation. *Lecture slides on Test of Digital Logic ID, DTU*, 2003.

[38] G. Syswerda. Uniform crossover in genetic algorithms. *International Conference on Genetic Algorithms. Los Altos, CA: Morgan Kaufman*, 3, 1989.

[39] J. H. P. Thomas M. Niermann, Wu-Tung Cheng. Proofs: a fast, memory efficient sequential circuit fault simulator. *Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 535–540, June 1990.

[40] E. R. TYanto Prasetio. *Tabu search: overview and example. IndE 510*, May 2002.

Appendix A

Appendix

A.1 X algorithm

X algorithm [30] is a one-pass, linear-time algorithm that determines a set of undetectable faults for a given test vector. It is a conservative algorithm which can't detected all undetectable faults.

The basic version of X algorithm is used in combination circuit. The idea is that the fault free circuit is simulated first, then circuit is scan from each PO to PI to see whether each fault can find a sensitization path to propagate fault to PO. X algorithm can easily extended to sequential circuit by considering PPO and PPI in scan.

Next, I will introduce some terms and lemma used in X algorithm first, then I will describe the procedure of X algorithm. The last, I will propose some heuristic used in improving efficiency of X algorithm. All text, showed later, is come from [30]. To keep the descriptions of the algorithms simple and easy-to-read, I will restrict myself to circuits made up of NAND gates and fanout nodes, the algorithms can easily generalized to accommodate other Boolean gate type.

1. **Term and Lemma**
    - **Sensitization Path:** Given a circuit and an input vector, a path from a lead L1 to a lead L2 is said to be a sensitization path if a fault on lead L1 will change the value at L2 and at every intermediate lead on that math.
    - **Control vector:** we define an input vector to an n-input NAND gateas a controlling vector if it puts 0's on two or more input lines

of that gate and no primary input has iso-parity paths to all of these 0-valued lines.
- Lemma 1: If the input vector of a NAND-gate is a controlling vector then the output of that gate will remain unchanged under the presence of a single stuck fault on any other lead in the circuit.

2. **procedure of Algorithm**
Input: A combinational circuit, a test vector and values on every lead obtained through logic simulation Output: A partial assignment of the circuit indicated by marking a subset of leads with a "*" and with an "X". The logic values on the leads marked "X" indicated undetected fault.
Procedure:
   (a) Mark all of the output leads .
   (b) processing the nodes of the circuit in reverse topological order(from the outputs to the inputs), for each node, based on the type of the node do:
   NAND gate:
      - If the output o fthe NAND gate is marked X then mark all of its inputs as X.
      - If the output of the NAND gate is marked and the output value is a 0 then mark all of the input leads
      - If the output of the NAND gate is marked the output value is a 1 and the input is a contolling vector then mark all input leads X.
      - If the output of the NAND gate is marked , the output value is a 1 and the input is not a controlling vector; then arbitrarily choose one of the input leads with the value 0 and mark it with a *(there will be at least one such input lead) and mark all other input leads X.

   Fanout Node: If all of the branches are marked X then mark the stem X; otherwise, mark the stem .

   End.
3. Other heuristics X algorithm offers a certain amount of choices when processing the NAND-gate. The choice consists of choosing an input lead with the value0, when more than one input to the gate has the value 0. THe chosen lead will be marked . In [30], Two heuristic, Star detected faults heuristics and Fan-out propagation heuristic are proposed.
   - **Star detected faults heuristics** when we faced with a choice of more than one 0 valued input to a NAND-gate, we ask if one of

the 0-valued input leads to the gate has already been detected. If so, we choose that input lead to mark with a .

- **Fan-out propagation heuristic** when we faced with a choice of more than one 0 valued input to a NAND-gate, we check to see if one of these inputs is a branch of a fanout, one of whose other branches has already been marked .

A.2 Experiment results

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	317(0.3)	4713	117
s510	564	377(0.7)	4045	69
s641	467	398(4.5)	2690	211
s713	581	472(0.2)	1542	131
s838	931	422(0)	19456	2744
s1196	1242	1108(0.9)	7392	742
s1238	1355	1140(4.8)	19836	1999
s1423	1515	1398(20.1)	14354	3231
s1488	1486	1218(22.3)	6728	835
s1494	1506	1251(15.9)	2878	350

Table A.1: SUS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 1

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323(0)	1500	108
s510	564	498(0.3)	2200	125
s641	467	404(0)	11900	1047
s713	581	480(0.3)	15900	1239
s838	931	445(3.2)	17080	4626
s1196	1242	1232(2.9)	14110	5060
s1238	1355	1276(6.0)	17160	6140
s1423	1515	1389(19.5)	12550	8012
s1488	1486	1442(11.7)	14100	4187
s1494	1506	1449(6.9)	17590	4800

Table A.2: SUS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 10

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	640	102
s510	564	560	7820	871
s641	467	404	1020	298
s713	581	480	760	243
s838	931	402	17860	5914
s1196	1242	1231	17600	6491
s1238	1355	1272	11380	4132
s1423	1515	1361	11580	8172
s1488	1486	1437	8840	3387
s1494	1506	1449	8180	2675

Table A.3: SUS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	720	220
s510	564	564	880	210
s641	467	404	3240	1540
s713	581	480	1760	880
s838	931	404	18680	12077
s1196	1242	1236	19680	11381
s1238	1355	1280	14560	9182
s1423	1515	1334	13200	12161
s1488	1486	1439	18320	8863
s1494	1506	1443	12680	6414

Table A.4: SUS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 40

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	900	587
s510	564	564	1500	782
s641	467	404	7600	1945
s713	581	480	6600	1871
s838	931	390	12500	16490
s1196	1242	1227	16500	13008
s1238	1355	1277	19400	11359
s1423	1515	1244	16500	13008
s1488	1486	1407	18800	11950
s1494	1506	1411	17000	10740

Table A.5: SUS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 100

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	318	2528	104
s510	564	382	37	1
s641	467	402	4750	528
s713	581	475	7005	829
s838	931	422	19456	3585
s1196	1242	1148	19542	2448
s1238	1355	1166	18395	2316
s1423	1515	1400	12971	3343
s1488	1486	1222	5244	582
s1494	1506	1225	3799	421

Table A.6: PS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 1

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	760	73
s510	564	542	370	29
s641	467	404	770	182
s713	581	480	1440	374
s838	931	448	17440	4484
s1196	1242	1236	14330	4020
s1238	1355	1279	16360	4659
s1423	1515	1371	19070	9970
s1488	1486	1440	12030	2774
s1494	1506	1448	17350	3948

Table A.7: PS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 10

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323(0)	160	27
s510	564	560(0.3)	7820	871
s641	467	404(0)	1660	519
s713	581	480(0)	2200	715
s838	931	464(0.3)	19480	6607
s1196	1242	1237(3.1)	17920	6244
s1238	1355	1274(11.7)	10260	3474
s1423	1515	1360(10.9)	18780	12126
s1488	1486	1441(5.7)	16860	5983
s1494	1506	1447(1.9)	14180	5250

Table A.8: PS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	920	244
s510	564	564	960	163
s641	467	404	1600	613
s713	581	480	1320	517
s838	931	407	14480	8469
s1196	1242	1224	9080	4392
s1238	1355	1278	14600	6913
s1423	1515	1319	17200	17278
s1488	1486	1428	18840	10361
s1494	1506	1441	17480	9272

Table A.9: PS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 40

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	800	514
s510	564	564	1200	493
s641	467	404	4800	3544
s713	581	480	2900	2325
s838	931	437	11000	12141
s1196	1242	1229	17000	12846
s1238	1355	1279	19800	16056
s1423	1515	1207	17500	29538
s1488	1486	1410	17200	14579
s1494	1506	1410	19000	14960

Table A.10: PS_GA: mutation rate is 0.01, length of test sequence in a chromosome is 100

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	140	25
s510	564	543	600	85
s641	467	404	1240	404
s713	581	480	900	305
s838	931	463	17100	7836
s1196	1242	1236	19480	7596
s1238	1355	1278	17580	7726
s1423	1515	1336	13080	9839
s1488	1486	1442	6960	3255
s1494	1506	1450	9060	4174

Table A.11: PS_GA: mutation rate is 0.025, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	300	50
s510	564	558	4200	334
s641	467	404	1620	445
s713	581	480	1940	499
s838	931	462	17500	6773
s1196	1242	1229	10980	4798
s1238	1355	1279	15080	5097
s1423	1515	1347	19160	11448
s1488	1486	1446	16820	6484
s1494	1506	1453	15480	6563

Table A.12: PS_GA: mutation rate is 0.05, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	520	93
s510	564	561	10480	1721
s641	467	404	1140	338
s713	581	480	820	257
s838	931	452	19200	9093
s1196	1242	1233	14900	5296
s1238	1355	1277	17160	5974
s1423	1515	1310	14440	10195
s1488	1486	1445	11980	5998
s1494	1506	1448	8180	4512

Table A.13: PS_GA: mutation rate is 0.075, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	440	76
s510	564	559	4060	668
s641	467	404	1480	420
s713	581	480	1760	544
s838	931	397	18740	8597
s1196	1242	1234	19820	6749
s1238	1355	1274	16700	6699
s1423	1515	1320	18520	13139
s1488	1486	1435	19660	8979
s1494	1506	1450	14600	6519

Table A.14: PS_GA: mutation rate is 0.1, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	680	108
s510	564	498	80	12
s641	467	404	1160	309
s713	581	479	1580	431
s838	931	394	14720	5983
s1196	1242	1232	19300	6256
s1238	1355	1272	11420	3743
s1423	1515	1274	19400	11569
s1488	1486	1421	18580	7350
s1494	1506	1429	12740	5233

Table A.15: PS_GA: mutation rate is 0.25, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	323	900	133
s510	564	540	140	22
s641	467	404	2180	544
s713	581	480	1720	459
s838	931	402	19060	7377
s1196	1242	1226	17920	5866
s1238	1355	1271	19940	6430
s1423	1515	1202	19240	11495
s1488	1486	1415	19380	7471
s1494	1506	1429	19620	8102

Table A.16: PS_GA: mutation rate is 0.5, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s641	467	403	1900	2090
s713	581	388	6990	10043
s1196	1242	1217	16900	15963
s1238	1355	1262	18720	15807
s1423	1515	813	19380	31721
s1488	1486	993	17290	11726
s1494	1506	994	16570	8043

Table A.17: Tabu search: length of test sequence in a chromosome is 10

Circuit	Total fault	Flt.det	Vector length	Time
s349	350	341	70	14
s510	564	564	530	306
s641	467	406	410	211
s713	581	480	480	194
s838	931	490	16890	6341
s1196	1242	1237	12010	6643
s1238	1355	1275	9110	4846
s1423	1515	1433	11200	11152
s1488	1486	1446	8640	3825
s1494	1506	1455	7000	3557

Table A.18: CSG: mutation rate is 0.01, length of test sequence in a chromosome is 10

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	341	100	14
s510	564	564	900	1225
s641	467	406	460	194
s713	581	476	520	260
s838	931	490	18220	7332
s1196	1242	1236	16460	8192
s1238	1355	1276	11820	5630
s1423	1515	1379	14240	13316
s1488	1486	1443	12600	5619
s1494	1506	1452	10500	4653

Table A.19: CSG: mutation rate is 0.01, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	342	70	16
s510	564	564	530	212
s641	467	406	470	227
s713	581	480	530	169
s838	931	478	510	288
s1196	1242	1236	6300	3321
s1238	1355	1277	3080	1569
s1423	1515	1428	7230	7244
s1488	1486	1446	9250	4123
s1494	1506	1455	4200	2834
s5378	4603	3635	62454	16900

Table A.20: CSG: mutation rate is 0.02, length of test sequence in a chromosome is 10

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	341	100	14
s510	564	564	900	1125
s641	467	406	460	194
s713	581	476	520	260
s838	931	490	1260	507
s1196	1242	1236	8460	4210
s1238	1355	1276	15820	7535
s1423	1515	1379	12240	11446
s1488	1486	1443	10600	4727
s1494	1506	1452	15500	6868

Table A.21: CSG: mutation rate is 0.02, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	340	50	12
s510	564	564	930	560
s641	467	406	320	172
s713	581	480	210	190
s838	931	496	16400	7172
s1196	1242	1236	8680	5089
s1238	1355	1276	14700	8285
s1423	1515	1428	5640	6449
s1488	1486	1446	3030	1623
s1494	1506	1455	3600	1907

Table A.22: CSG: mutation rate is 0.03, length of test sequence in a chromosome is 10

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	343	60	12
s510	564	564	600	859
s641	467	406	280	135
s713	581	480	430	247
s838	931	476	16840	9158
s1196	1242	1236	13800	8393
s1238	1355	1278	8720	5530
s1423	1515	1387	14800	14499
s1488	1486	1446	7380	4396
s1494	1506	1455	7500	4438

Table A.23: CSG: mutation rate is 0.03, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	341	1100	23
s510	564	564	1370	1971
s641	467	406	210	121
s713	581	480	1410	878
s838	931	461	19970	9289
s1196	1242	1237	5840	3541
s1238	1355	1276	9130	5242
s1423	1515	1430	8080	9444
s1488	1486	1446	2680	1988
s1494	1506	1455	2750	1621

Table A.24: CSG: mutation rate is 0.04, length of test sequence in a chromosome is 10

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	343	120	30
s510	564	564	400	539
s641	467	406	220	133
s713	581	480	460	311
s838	931	482	16680	11799
s1196	1242	1239	13200	7752
s1238	1355	1277	18460	10367
s1423	1515	1398	12220	13915
s1488	1486	1446	15500	9733
s1494	1506	1455	10340	6522

Table A.25: CSG: mutation rate is 0.04, length of test sequence in a chromosome is 20

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	341	70	16
s510	564	564	950	1056
s641	467	406	350	171
s713	581	480	260	148
s838	931	480	16400	6565
s1196	1242	1238	9720	5140
s1238	1355	1278	18750	10864
s1423	1515	1429	16450	17228
s1488	1486	1446	7080	3716
s1494	1506	1455	1140	3289

Table A.26: CSG: mutation rate is 0.05, length of test sequence in a chromosome is 10

Circuit	Total Fault	#flt.det	Vector length	Time
s349	350	343	60	14
s510	564	564	480	648
s641	467	406	300	147
s713	581	480	620	345
s838	931	491	16880	8738
s1196	1242	1236	13200	9631
s1238	1355	1275	14980	7462
s1423	1515	1385	13880	13982
s1488	1486	1446	10420	5747
s1494	1506	1455	12780	7032

Table A.27: CSG: mutation rate is 0.05, length of test sequence in a chromosome is 20