ECE 5505 Final Project

LOSAT - Logic Simulation based Sequential ATPG

Anikate Kaw and Victor Gan

INTRODUCTION

ATPG, or Automatic Test Pattern Generation, is one of the most essential topics in integrated circuit testing. However, the complexity of modern IC has grown extremely high and made it infeasible for people to calculate it by hand. In order to test the circuit within finite and reasonable time, we need software-based automatic test pattern generators.

In this project, we built a Test Pattern Generator for Sequential Circuit based on our own Logic Simulation program. We used the test generating algorithm published by Sheng and Hsiao in 2002[1].

In this paper, they introduced three techniques to generate test sets using only logic simulator:

- state partitioning
- using GA to maximize the number of state reached in each partition
- reset signal masking[2]

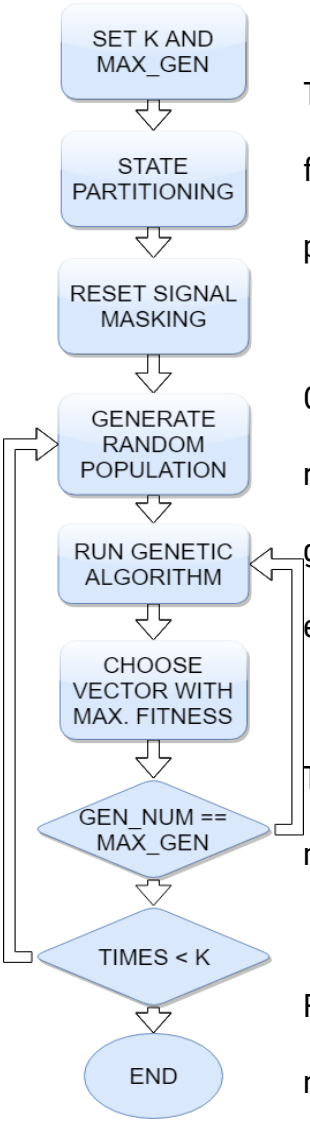
The proposed generator's fault coverage is better or comparable to several complex fault simulation based algorithms[3-5].

OBJECTIVES

Generate Test Vectors for the given circuit using Logic Simulation based Test Pattern Generator.

FLOWCHART & HIGH-LEVEL DESCRIPTION

Fig. 1 gives a flowchart of a high-level abstraction of our implementation. An in-detail description of each of these steps is present in the next section.



This algorithm begins with partitioning the flip-flops into 5 groups. This is followed by reset-signal masking to compute the probability of each primary input to be masked to 0 and 1.

Once we calculate these probabilities, we run the Genetic Algorithm on a random population. Uniform crossover and mutation are being used to generate the population of the next generation and the fittest individual of each generation is added to the final test set.

The Genetic Algorithm is repeated a fixed number of times to obtain as many vectors as desired by the user.

Finally, the final test vector set is fed to a fault simulator to find out the number of faults detected and calculate the fault coverage.

Fig. 1 High-level flowchart

TECHNIQUES & IMPLEMENTATION

In this section, we will be explaining each stage of the flowchart.

1. STATE PARTITIONING

The primary motivation behind this implementation is to reach as many states as possible for a given sequential circuit.

However, indiscriminate adding of states may not always lead to the best results as a lot of flip-flops might be much easier to toggle compared to others. Setting/Resetting of such flip-flops might give us new states but these states might not play a huge role in fault-detection.

We are interested in reaching the harder to reach states. To accomplish this we partition the states which helps us weed out the noise from the flip-flops which can easily set as well as reset.

We have recorded observations for controllability-based partitioning technique in this implementation:

- **Controllability-based Partitioning**

  In this method of partitioning, we divide flip-flops according to the ease with which they can reach 0 or 1.

  To do this we carry out logic simulation with N randomly generated test vectors and count the number of times each flip flop reached 0 and 1.

  Let the number of times the i^{th} flip-flop reached 0 and 1 be N^0_i and N^1_i respectively.

  Following this bias, represented by ff_{bias} , is calculated for each flip-flop where,

  $$ff_{bias} = (| N_i^0 - N_i^1|) / N$$

A bias value closer to 1 denotes that this flip-flop can be easily set to one state but cannot be set to the other. Similarly, a bias value closer to 0 tells us that this flip-flop can easily reach both states.

Depending on the range in which a specific bias value falls, flip-flops are divided into different partitions.

In our project we have divided the flip-flops into 5 partitions with range as {0, 0.2, 0.4, 0.6, 0.8, 1.0}.

2. GENETIC ALGORITHM

Genetic Algorithm is probably the most important part of this implementation. It computes a new population of vectors from an existing one and is instrumental in determining the best vector in a given population.

The Genetic Algorithm starts from a population of random vectors.

This random population. For each individual vector in this population, a fitness is calculated.

- **Fitness**

  Each partition *j* is assigned a weight W_j. A state table is maintained for each partition to keep track of the times a particular state is reached in each partition.

  $$\text{Fitness} = \Sigma W_j.U_j$$

  U_j = 10, when any state in a partition is reached for the first time

  After this, U_j = 1/k where k is the number of times this state has been reached.

NOTE: W_j plays a huge role in making sure that a large number of states are reached in the partition with the high bias flip-flops.

To make sure this is accomplished, we assign a large weight to the partition which has high bias flip-flops. In our implementation the weights assigned to each partition are as follows:-

- {0 - 0.2} : W = 2
- {0.2 - 0.4} : W = 4
- {0.4 - 0.6} : W = 8
- {0.6 - 0.8} : W = 16
- {0.8 - 1.0} : W = 40

Note that the weight assigned to the last partition is almost 20 times that of the first one.

Once fitness is calculated for every vector in the population, we choose the vector with the highest fitness and add it to our final test set. In our implementation, we have only incremented the state tables for the vector which is added to this set.

To compute the next population, we carry out uniform crossover followed by mutation(1%).

- **Uniform Crossover**

  Two parents, parent1 and parent2, are selected from the current population and a random mask(m) is generated. For every i^{th} bit of the mask:

  1. If the bit is 0, i^{th} bit of parent1 is assigned to child 1 and the i^{th} bit of parent2 is assigned to child2.
  2. If the bit is 1, i^{th} bit of parent1 is assigned to child 2 and the i^{th} bit of parent2 is assigned to child1.

- **Mutation**

  In our implementation, to carry out mutation we randomly select 1% of the bits from child1 and child2, both generated after the uniform crossover between parent1 and parent2, and flip them.

We repeat these two steps until the size of our next population is lesser than the size of our current population.

This genetic algorithm is repeated for a fixed number of times, say GEN_MAX, following which a new random population is generated and the entire process repeated for a fixed number of times.

The selection of GEN_MAX plays a key role in deciding the quality of the vectors and the execution time. If the GEN_MAX is too large, after a certain number of iterations the states reached by the vector selected will be the same and the fitness will plateau. This will lead to a large execution time. On the other hand, if the value of GEN_MAX is too small, there will be a lot of vectors which will never be generated as the population will be randomized again. This can lead to a huge loss in the number of faults detected by our final test set.

4. RESET SIGNAL MASKING

The goal is to generate a set of vectors that can traverse as many states as possible. Imagine that you have a PI which is a reset or partial reset signal. Every time it is triggered, it would pull the state back to an easy to reach state, and largely reduces the chance we can visit a new one. We need to mask these reset signals to increase the total number of state reached and lower the fault coverage. In order to do so, we used the method introduced by S. Sheng at el.

First, we need to calculate Reset Power 0(1) for every PIs. By setting the given PI to 0(1) and others to X, and with the initial state of FFs are all X, we apply the vector to the circuit for one time frame and observe if there is any FF be set to a 0 or 1. By definition[2], the *Reset Power 0(1) for a given PI is the number of FFs that can be reset to a specified value by applying a 0(1) at this PI and X at all other PIs, with the initial state of circuit being all X.*

Second, we calculate ResetMaskto1(0) Probability. The original equation of the ResetMaskto1 is the ResetPower0 of the given PI times k and over the total number of FFs, where k is used to scale the low ResetPower0 of partial reset signals. One can calculate the ResetMaskto0 Probability conversely by using ResetPower1. But there would be a problem that if we apply the same value of k to the equation. The probability of global reset signals would exceed one, while others remain small. Therefore, instead of applying a constant k, we take root square of the probability twice to curve the numbers.

Then, we apply the probability of mutation to the GA program after crossover and the original mutation process.

4. SEQUENTIAL LOGIC SIMULATOR

The logic simulator we used is based on our project I of this semester - combinational logic simulator with distinguished-Xs. The program was originally written by Dr. Shiao and modified by ourselves to add the X distinguishing feature. We knew that this program can simulate sequential circuits as well. But since we couldn't 100% understand how it handles flip-flops between each

time frames plus it changes the algorithm of X distinguishing when we did it as a combinational simulator, we rewrote part of the program to make it fit with sequential circuits.

We store the value of FFs into three new arrays: value1_ffs, value2_ffs, and id_unknown_ffs. These values represent the state of the FFs at the end of every time frames, while the original value1, value2, and id_unknown are the state at the beginning. If the evaluated value of FFs are stored regularly like other types of gate, their successors would reference the wrong value which it should not use them until the next time frame.

Thus, before we evaluate the whole circuit in each time frame, we must first apply stored FF values back to original value variables, and put their successors on the event wheel if the value changes. After that, we apply vector on primary inputs, and finally, simulate the circuit.

In addition, we modified the *retrieveEvent()* method in gateLevelCkt. The original version scans the level from the lowest to the maximum until all events in each level are executed. It works perfectly in combinational circuits. However, in sequential circuits, the successor of a gate can have a smaller level. For example, the input of an FF may be at the end of the circuit, while the FF itself has a very low level. Circuit s27 perfectly exemplify this situation. It could cause a problem that although an FF was put on the event wheel, it still remains unevaluated, because the *retrieveEvent()* function has gone to a higher level. To overcome this problem, we modify the function so that it can go back and check the lower levels.

RESULT

Test Bench & Environment

We used the selected circuits of iscas89 to test the performance of the test pattern generator. For the sake of time, we can only generate 20,000 vectors, while in the paper, they generated 100,000 vectors.

1. 500 gens, 40 times, without reset signal masking

Circuit	s298	s344	s400	s1198	s1488
#Faults detected	260	329	57	1238	1181
#Total Faults	308	342	428	1242	1486

2. 250 gens, 80 times, without reset signal masking

Circuit	s298	s344	s400	s1198	s1488
#Faults detected	263	329	58	1238	1155
#Total Faults	308	342	428	1242	1486

3. 500 gens, 40 times, reset signal masking

Circuit	s298	s344	s400	s1198	s1488
#Faults detected	263	329	NA	1238	NA
#Faults detected w/o RSM	260	329	57	1238	1181

REFERENCE

1. Shuo Sheng and M. S. Hsiao, "Efficient sequential test generation based on logic simulation," in IEEE Design & Test of Computers, vol. 19, no. 5, pp. 56-64, Sept.-Oct. 2002. doi: 10.1109/MDT.2002.1033793
2. Shuo Sheng, K. Takayama and M. S. Hsiao, "Effective safety property checking using simulation-based sequential ATPG," Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324), New Orleans, LA, USA, 2002, pp. 813-818. doi: 10.1109/DAC.2002.1012734
3. T.M. Niermann, J.H. Patel, "HITEC: A Test Generation Package for Sequential Circuits", Proc. European Design Automation Conf., pp. 214-218, 1991.
4. M.S. Hsiao, E.M. Rudnick, J.H. Patel, "Sequential Circuit Test Generation Using Dynamic State Traversal", Proc. European Design and Test Conf., pp. 22-28, 1996.
5. R. Guo, I. Pomeranz, S.M. Reddy, "A Fault Simulation Based Test Pattern Generator for Synchronous Sequential Circuits", Proc. VLSI Test Symp., pp. 260-267, 1999.