ESE-549 Advanced VLSI System Testing
Spring 2018

Project 3: PODEM ATPG

Due Date: 4/1/18 5:00PM

Sahil Jindal 111471177
Khushboo Kumari 111634864

Contents

1 Part 1 - Baseline PODEM ATPG

The Part-1 of the project required us to implement the baseline PODEM Algorithm for Automatic Test Pattern Generation(ATPG). Here for the purpose of simulating the circuit we used the *simFullCircuit* method which start the simulation for each input vector afresh. This makes this approach least effective as during the course of our algorithm it make many decisions and its often changing only one signal but every time we wipe the previous results of our circuit and start from beginning thus loosing much useful information and redoing much of the same work.

1.1 Approach

The basic ideas that we followed while implementing the PODEM Algorithm are listed bellow.

- In order to implement the baseline PODEM Algorithm we started with the *podemRecursion* function. This is the main recursive function from where the execution begins.
- When the *podemRecursion* function starts execution it checks if there is a LOGIC_D or LOGIC_DBAR on PI if there is we are successful and the algorithm returns.
- The next call *podemRecursion* function makes is to the *getObjective* function.
- In the *getObjective* function our initial motive is to activate the fault. So the first thing we check is if the fault is activated. If it is we move forward else we set the objective as the stuck at gate and the value as the opposite of the stuck-at value.
- Even before we check if the fault is active we check if the fault site is LOGIC_ONE or LOGIC_ZERO. If this is the case that means we have failed to activate the fault and our algorithm fails.
- The other way our *getObjective* function can return failure is if there is nothing on D-Frontier. So next we populate the D-Frontier. For this we call the function *updateDFrontier*. Here we just iterate over all the gates and check if the gate has a LOGIC_X on its output and a LOGIC_D or LOGIC_DBAR on its inputs.
- After the *updateDFrontier* operation if our D-Frontier is empty we return failure else we select the first gate in our D-Frontier and selelct the first input of this gate which has a value as LOGIC_X as our objective and set the non-controlling value as our objective value. While doing this we treat XOR as OR and XNOR as NOR.
- Once we have our Objective gate and value we pass this to our *backtrace* function which traverses back till a PI and chooses a value for the PI which it feels would set our Objective gate to the objective value that *getObjective* returned.
- The *backtrace* function simply selects the first gate in its inputs list which has a value LOGIC_X till it is at a PI Input. It also counts the number of inverting gates between objective and the target gate, i.e, the PI. If the number of inversions is positive we set the PI value as objective value else NOT of the objective value.
- Next our *podemRecursion* sets the PI value as the required value and calls the *simFullCircuit* method.
- *simFullCircuit* function is our project-2 implementation. The only change we make here is to set a LOGIC_D or LOGIC_DBAR on the PI's if applicable.

- We then call the *podemRecursion* function again. If our algorithm succeeds we get a new objective value and the algorithm continues else if it returns failure we try the complementory value of the one that our backtrace method returned and again call the *simFullCircuit* method and our *podemRecursion* function. If the recursion succeeds we return success to level above. If this value also fails then we return failure to the level above but before that we set the value back to LOGIC_X and run the *simFullCircuit* so that the level above has the same state from which it started and it can try different values.

1.2 Verification

In order to validate the results from our ATPG Algorithm we created a method called *validateResultsFromATPG*. This is nothing but our Project 2 where we pass a list of all the test vectors that our algorithm detected and the original fault list and then check the number of unique tests it found. We then compare this result with the total detectable faults in the circuit.

Also in order to validate my results against the reference outputs given in the handout we followed the same steps as described in the handout. We checked the difference using between my outputs and reference outputs using the *diff* command and it showed no differences. That is how we made sure our implementation was correct. The timings are without our *validateResultsFromATPG* method.

1.3 Performance

Part-1 performs the worst due to obvious reasons. We are using the *simFullCircuit* at each steps which does not make use of any previous information that we stored and starts the full circuit simulation from scratch. The algorithm is such that we only make one change at a time and we can easily make use of this information which is the aim of part-2. Also we do not reduce the test vector size. Same test can detect multiple defects but we through all of the defects no matter what. This makes part-1 really slow. The timing and test vector information is listed in the table below.

Table 1: Timing and Test Vector Size Part 1

Circuit	**Fault**	**Time**	**# of Vectors**
c17.bench	c17.fault	0.003s	34
ex1.bench	ex1.fault	0.003s	28
ex2.bench	ex2.fault	0.004s	50
c432.bech	c432.smallfault	0.011s	10
c432.bech	c432.medfault	0.038s	50
c432.bench	c432.bigfault	14:29.47	864
c499.bench	c499.fault	0.009s	900

2 Part 2 - Event Driven Simulation

As discussed in the baseline ATPG we used the *simFullCircuit* method and that makes our algorithm slow. We are not making use of the fact that there is only one change at a time in our circuit and we can use that fact to check what all gate outputs can this change affect. This is called event driven simulation and can make our algorithm perform much better.

2.1 Approach

The only change we make in this part is to use *eventDrivenSim* instead of the *simFullCircuit* function. The major points considered in the implementation are listed below.

- In our *podemRecursion* function after we get the PI gate and value from the backtrace function instead of calling the *simFullCircuit* function we use the *eventDrivenSim* function.
- We build a queue of type Gate* where we push the PI gate with the value that we require and call the *eventDrivenSim* function.
- In the *eventDrivenSim* function we do kind of a Breadth First Search(BFS). We start with the PI and check all the gates that this PI feeds into. If the change affects the output gate we change the value and add the gate to our queue. Else we just skip the output gate since we are sure that this change will not impact anything in the circuit that it feeds into. We keep looping till there is no gate on our queue which means that we have taken care of all the gates this change could affect.

This makes our algorithm very fast as we only look at the gates that change because of the change in PI value and we don't have to go through the whole circuit.

2.2 Verification

Here since we change only the way we simulate the circuit there should be no change in the outputs so our verification strategy is the same. We validated the outputs against the reference outputs provided in the circuit and also use our *validateResultsFromATPG* method. There is no difference between our output file and the refout file.

2.3 Performance

We see a significant improvement over the timings in our part 2 over part 1. For the biggest circuit i.e., the c432 with bigfault file we cut down the running time to almost one third. For the test vector size we see no improvement since we only change the way we simulate the circuit and there is no change to any other strategy.The timing and test vector information for different test benches is listed in the table below.

Table 2: Timing and Test Vector Size Part 2

Circuit	**Fault**	**Time**	**# of Vectors**
c17.bench	c17.fault	0.000s	34
ex1.bench	ex1.fault	0.003s	28
ex2.bench	ex2.fault	0.004s	50
c432.bech	c432.smallfault	0.011s	10
c432.bech	c432.medfault	0.036s	50
c432.bench	c432.bigfault	4:01.40	864
c499.bench	c499.fault	0.009s	900

3 Part 3 - Equivalence Fault Collapsing

In order to improve the performance of our Algorithm we can make use of the fact that a single test vector can make detect multiple defects. In part 3 we try to do that using Equivalence

Fault Collapsing. A small test vector size not only improves our algorithm runtime it also saves time on the actual hardware testing. The major points considered during the implementation were

- We make no changes to the podem recursion approach from part 2. We use the same function as in part 2 with event driven simulation.
- The change we make is to fault list. We try and make it small and we use of the FaultEquiv Class provided in the handout.
- FaultEquiv starts with each test as a separate node and then we go through the circuit and set equivalence relations at which point it starts merging the nodes and gives us a collapsed test vector at the end of it.
- To set the equivalence relations we created the *setAllEquivalentNodes* method which takes in the circuit PO's and fault equivalence graph as the input.
- In the *setAllEquivalentNodes* method we go through all the gates in a Depth First Search(DFS) strategy and set all the equivalence relations for each gate using the *setEquivForGate* method.
- For XOR and XNOR gates we don't have any equivalence relations and FANOUT gate is not a real gate. We just created it to distinguish different fanout branches. So we don't call the *setEquivForGate* for these gates.
- For the *setEquivForGate* method we refferd to the table provided in lecture notes to set the relations.

3.1 Verification

Our main function writes the equivalence relations that we established into a separate output file. Along with the reference output file for ATPG we are also given the reference output files for equivalence. So In order to check if our algorithm set the equivalence correctly we use the diff of reference file and our output and found both to be same. That is how we made sure we were doing the right thing.

Now we will have a shorter ATPG output and we cannot validate our output just by checking the difference file so we use our *validateResultsFromATPG* method. We observed that we were able to detect all the detectable faults with the test vectors that we got from out ATPG output thus making sure it was working correctly.

3.2 Performance

Part 3 is an improvement over part 2 where we use the same PODEM logic but make use of the fact that a single test vector can detect multiple defects. We reduce the test vector size which not only reduces the algorithm running time but would also decrease the actual testing time for a circuit. **The later improvement according to me would make a larger impact as running PODEM is a one time thing but we need to test all our hardware circuits and it can greatly benefit from a smaller test vector size**. The timing and reduced test vector sizes are listed in the table below

Table 3: Timing and Test Vector Size Part 3

Circuit	**Fault**	**Time**	**# of Orig. Vectors**	**# of Reduc. Vectors**
c17.bench	c17.fault	0.003s	34	22
ex1.bench	ex1.fault	0.002s	28	16
ex2.bench	ex2.fault	0.003s	50	37
c432.bech	c432.smallfault	0.017s	10	10
c432.bech	c432.medfault	0.039s	50	47
c432.bench	c432.bigfault	1:29.20	864	524
c499.bench	c499.fault	0.008s	900	750

We see a slight increase in the runtime of smaller circuits which is because of the fact that we now need to calculate the equivalence before we run our PODEM and the decrease in test vector size is not that significant. Although for the bigfault for c432 circuit we see a significant performance improvement where reduce the runtime from 4 minutes to 1 minute 29 seconds.

4 Part 4- Mode 4 Runtime Improvement

The aim of part 4 is 2 folds. First is to reduce the runtime of our algorithm, which we discuss in this section, and second is to reduce the test set size which we will discuss in the next.

To improve the runtime of our algorithm we had many options but we tried the following three

1. **Improve D-frontier management** - Everytime our algorithm needs to find an objective we call the *updateDFrontier* method that goes through all the gates in circuit to find the gates on DFrontier. In this approach we use our *eventDrivenSim* method to maintain this list which while evaluating the circuit output checks if a particular gate should be on the DFrontier or removed from the DFrontier.
2. **Use the SCOAP matrix** - We calculate the SCOAP matrix for the circuit and instead of picking the first gate from our DFrontier or while backtracking we make smart decisions using SCOAP Values.
   Note: This did not produce desired results so it is commented in the code
3. **Select the Gate from DFrontier Randomly** - Instead of picking the first gate from our DFrontier we pick the gate randomly.

4.1 Approach

For part 4 we made changes to our *eventDrivenSim* method and also to our *getObjective* and *backtrace* method. It also required changes to ClassGate.h and ClassGate.cc file. The major points considered during the implementation were

- For improving the DFrontier management instead of calling the *updateDFrontier* method we use our *eventDrivenSim* method to maintain this DFrontier.
- When we check the changes to a particular gate because of the change in a PI value we maintain the previous value and compare it with the value after simulation. If the gate now has a D or DBAR and the output is still X we push the gate to the DFrontier. If outputValue is now a D or DBAR we check our DFrontier and remove this gate from DFrontier if it was there.

- For calculating the SCOAP values we added CC0, CC1 and CO values to our gate class and write setters and getters for them.
- We call the *setSCOAPValues* method which calls the *setControlability* and *setObservability* so set the required values.
- The *setControlability* and *setObservability* methods work much like the set gate depth logic in our Project 1.
- In the *setControlability* method we start from PO's and go back to PI's in a DFS strategy and set the CC0 and CC1 values for the intermediate gates using the tables from class notes.
- The *setObservability* method also works in the same way. It starts from PO's and go back to PI's in a DFS strategy and set the CO values for the intermediate gates using the tables from class notes.
- While selecting the gate in *getObjective* method instead of picking the first gate we get the gate with minimum observability using *getGateWithMinObserv* method.
- While selecting the gate from inputs in our *getObjective* function to set them to the non controlling value we choose the gate with maximum controlability. The rational behind this is that we anyway have to set the non controlling values so if the algorithm fails we make the algorithm fail earlier by trying the hardest values.
- We created a function *randNum* to get a random number between two given inputs which we use to select the gate from DFrontier.

4.2 Verification

Here we have not changed the fault list in any way it is similar to the list in part 3 so we use the same testing strategy. We use our *validateResultsFromATPG* method. We observed that we were able to detect all the detectable faults with the test vectors that we got from out ATPG output thus making sure it was working correctly.

In order to verify that our controlability and observability values are correct we print the values using *printSCOAPValues* method and verified it for homework 1 and homework 4 circuit(for which we created a test bench).

4.3 Performance

We will discuss the improvements from the 3 approaches separately.

4.3.1 Improve D-frontier management

Using the D-Frontier management approach we were able to reduce the runtime of our algorithm. For the biggest test vector we were able to cut the runtime to almost half as compared to part 3. We are not going through all the gates in our circuit to find the gates on DFrontier instead we maintain the list while setting the gate values as discussed in the approach. n greatly benefit from a smaller test vector size. The timing and reduced test vector sizes are listed in the table below

Table 4: Timing and Test Vector Size Part 4 without SCOAP

Circuit	**Fault**	**Time**	**# of Orig. Vectors**	**# of Reduc. Vectors**
c17.bench	c17.fault	0.001s	34	22
ex1.bench	ex1.fault	0.001s	28	16
ex2.bench	ex2.fault	0.001s	50	37
c432.bech	c432.smallfault	0.001s	10	10
c432.bech	c432.medfault	0.002s	50	47
c432.bench	c432.bigfault	0:41.64	864	524
c499.bench	c499.fault	0.008s	900	750

4.3.2 SCOAP Matrix

For SCOAP Matrix with improved D-Frontier management we got an increase in runtime instead of an improvement. This is because we need to set the SCOAP values first and the circuit is not big enough for SCOAP to show improvement. It may yield better results for bigger circuits. The timing and reduced test vector sizes are listed in the table below

Table 5: Timing and Test Vector Size Part 4 with SCOAP

Circuit	**Fault**	**Time**	**# of Orig. Vectors**	**# of Reduc. Vectors**
c17.bench	c17.fault	0.006s	34	22
ex1.bench	ex1.fault	0.001s	28	16
ex2.bench	ex2.fault	0.002s	50	37
c432.bech	c432.smallfault	0.003s	10	10
c432.bech	c432.medfault	0.005s	50	47
c432.bench	c432.bigfault	0:42.64	864	524
c499.bench	c499.fault	0.009s	900	750

Since it has a negative effect we have commented the SCOAP value part in our code. To test that we can uncomment the setSCOAPValues function and set the mode == 4 where we have put it as 9.

4.3.3 Randomly Selecting the Gate From DFrontier

This Improvement had neither a negative effect nor a positive effect on the runtime for the algorithm was the same.

5 Part 4 - Mode 5 - Test Set Size

In this section we discuss the second part of part 4 i.e, to reduce the test vector size. As discussed earlier for practical purposes this matters the most as a smaller test vector means we can test our circuit much faster.

There were many options to reduce the test vector size but we chose the following three

1. **Fault Dominance** - We use the dominance relations to reduce the test vector size. If we find a test for a dominating test all dominated tests can be marked as found.
2. **One test vector for many tests** For each test vector that we found we checked if that could detect any of the undetected faults. If it could we marked those and the faults dominated by that defect as found.

3. **Making use of the X's** Our ATPG algorithm return many X values we randomly put a 0 or 1 in these positions and then repeated the 2nd step to see if we could reduce the test set size even further.

5.1 Approach

In order to make use of the Dominance relations we need to set the dominance first. Also there is a change in the way we call the *podemRecursion* function since for making use of dominance we need a dynamic fault list. The approach is discussed below.

- To create the fault equivalence graph with dominant relations, while setting the equivalence if it is mode 5 we also set the dominance relations according to the table in our lecture notes.
- For mode less than 5 while calling the *podemRecursion* function we iterate over collapsed fault list or the original fault list and call the podemRecurssion for each defect
- In mode 5 we have a tree structure because of the dominance relations and we need to start from the leaf nodes and if leafs are not detectable we go up the tree finding a test vector for other nodes. If we find test for a node all the parent nodes can be marked as done.
- To achieve this we created a method called *runPODEMForNode* which traverses the nodes in a Depth First Search(DFS) manner and checks for a test vector for the leaf nodes. If it successful in finding one we mark the parent nodes as done else we keep going up the tree in recursive calls.
- We use an unordered_set to keep track of all the nodes for which we were able to find a fault. We call the *podemRecursion* only if the node is not in the unordered set.
- In the same function whenever we find a test we iterate over all the defects that have still not been detected and run full circuit simulation for these. We then check the output and if we get a D or DBAR on a PO we mark the defects and its dominated nodes as done.
- Also for the test vector we use the *randNum* function and set the X values as 0 or 1.

5.2 Verification

In this section we reduce the test set size significantly and validation of the results becomes important. In the same way as we verified the results for part 4 we call out *validateResultsFromATPG* method which gives us the number of uniquely detected faults. We checked the count and verified that it is equal to the number of detectable faults for the circuit. This is how we verify that our results are correct.

Dominance relations are also printed out in our output file and we validated our algorithm results for the 2 homework test benches and found them to be as expected.

5.3 Performance

Algortihm performs best in mode 5. It gives us the smallest test set size and has the least runtime. The performance enhancements for the 3 methods are discussed in the following sections.

5.3.1 Fault Dominance

By using the fault dominance relations we were able to reduce the test set size although not by a lot. For our biggest test set of c432 which has 864 tests we were able to bring it down to 449 tests. The following table better descibes our results.

Table 6: Timing and Test Vector Size Mode 5 with Dominance

Circuit	**Fault**	**Time**	**# of Orig. Vectors**	**# of Reduc. Vectors**
c17.bench	c17.fault	0.004s	34	16
ex1.bench	ex1.fault	0.001s	28	11
ex2.bench	ex2.fault	0.002s	50	33
c432.bench	c432.smallfault	0.001s	10	10
c432.bench	c432.medfault	0.002s	50	47
c432.bench	c432.bigfault	0:35.57	864	449
c499.bench	c499.fault	0.007s	900	698

5.3.2 One test vector for many tests

By running the circuit simulation for all the remaining defects and striking out the defects that were detected in this process we were able to reduce the test vector set significantly. For c432 with big fault list we reduced the test set from 864 to 123 tests. The results for other test benches are highlighted in the following table

Table 7: Timing and Test Vector Size in Mode 5 with Dominance and checking for other defects

Circuit	**Fault**	**Time**	**# of Orig. Vectors**	**# of Reduc. Vectors**
c17.bench	c17.fault	0.002s	34	10
ex1.bench	ex1.fault	0.003s	28	8
ex2.bench	ex2.fault	0.001s	50	13
c432.bench	c432.smallfault	0.003s	10	7
c432.bench	c432.medfault	0.005s	50	20
c432.bench	c432.bigfault	0:34.37	864	123
c499.bench	c499.fault	0.004s	900	75

We see a slight improvement in the bigfault runtime but a slight increase in timing for others. This is because the computation for smaller test benches increases but the bench size does not decrease in the same proportion to make an impact.

5.3.3 Making use of the X's

Setting the X's to 0 and 1 randomly further decreased out test set size. The test set for c432 bigfault is reduces to 71 tests in this case. The results are highlighted in the table below. Here we are using all the three optimizations.

Table 8: Timing and Test Vector Size in Mode 5 with Dominance and checking for other defects

Circuit	**Fault**	**Time**	**# of Orig. Vectors**	**# of Reduc. Vectors**
c17.bench	c17.fault	0.003s	34	7
ex1.bench	ex1.fault	0.001s	28	8
ex2.bench	ex2.fault	0.002s	50	12
c432.bench	c432.smallfault	0.003s	10	7
c432.bench	c432.medfault	0.001s	50	19
c432.bench	c432.bigfault	0:34.51	864	71
c499.bench	c499.fault	0.002s	900	66

6 Conclusion

Starting with the implementation of a baseline PODEM ATPG we able to incrementally improve the performance of our algorithm both in terms of runtime and the test set size. Both these factors are critical in real life. We need out test vector to be as small as possible since we will be testing all of our produced chips and a smaller test set would mean less time and more money and we want our algorithm to do this in as little time as possible.

The following bar charts for c432 bigfault show us how things improved with different modifications in term of both time and test vector size.

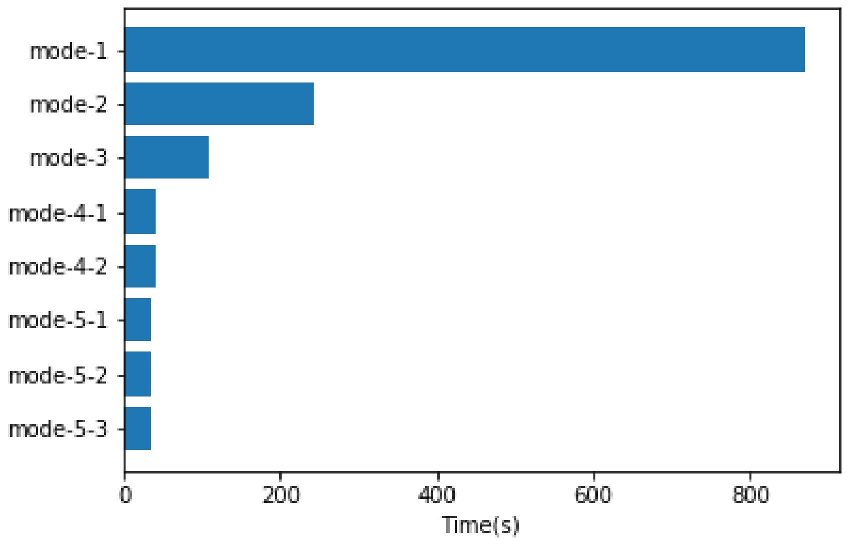

Figure 1: Run time for different modes

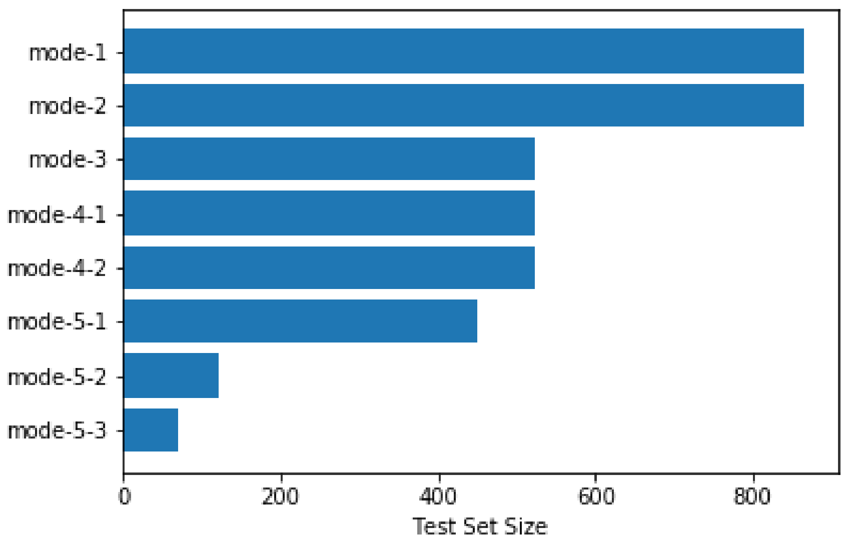

Figure 2: Test set size for different modes