Stony Brook University

Project 3: PODEM ATPG

Issued: 3/22/16, Due: 4/19/16 2:30PM

1. Introduction

The goal of Project 3 is to construct, optimize, and evaluate an Automatic Test Pattern Generation (ATPG) tool. This tool will perform targeted ATPG using the PODEM algorithm. When performing ATPG, you will either provide a list of single-stuck-at faults for it to use, and it will attempt to generate a test for each.

You may work alone or in a group of two. If you are looking for a partner, there is a form on Pizza you can use to post a message.

This project is divided into two parts, each of which you can read about below:

Part 1: Implement a baseline PODEM ATPG Algorithm.

Part 2: Event-driven simulation, which provides about 10x speedup to PODEM for large circuits.

You will be graded on correctness, efficiency/speed, and the quality of your code and comments.

We will again be using Piazza for discussion related to this project. I have added a "proj3" folder for us to use.

I am providing you with the following things:

- This project description
- Baseline software. This software will take care of all file input/output, and provide the basic structure for many of the functions you will need. You can find this on the lab computers at: `/home/home4/pmilder/ese549/proj3`
- API documentation for the Circuit and Gate classes, as well as documentation for the functions in main.cc. You can view this documentation as a PDF (apidoc_v3.pdf), or as a nice interactive webpage (which you can download as apidoc_v3_html.zip, unzip, and then browse). Both are available on Blackboard. (This documentation is automatically generated from the source using a tool called Doxygen.)

Please remember to work within a private work directory. Please see the Project 1 handout for directions for setting this up and copying our starting code base there. If you and your partner are familiar with Git, I can give you access to a private shared repository that you can use to share code between you.

2. Program Overview and File Formats

As in previous projects, you can compile your code by typing `make`. This builds the executable called `atpg`.

You can then run the tool with the following format:

```
./atpg [bench_file] [output_loc] [fault_file]
```

`bench_file`: the target circuit in `.bench` format

`output_loc`: location for output file

`fault_file`: faults to be considered

The system will generate a test pattern for each fault listed in `fault_file`. The test vectors will be stored in `output_loc`. (If your PODEM implementation is not able to find a test, it will output "none found" on that line.)

File Formats:

- The circuit is described in Bench format, as in Project 1 and 2. (Please refer to previous documentation if needed.)

- The fault file is a list of faults we would like to generate tests for. This is in the same format as the fault lists in Project 2. The format of this file is simple: one line gives the name of the signal where we would like the fault, and the next line gives a 0 or 1 indicating a stuck-at-0 or stuck-at-1 fault. For example:

```
A
0
B_1
1
```

  specifies two faults to simulate or generate tests for: A stuck at 0, and B_1 stuck at 1. Note that as in the second example we may target a fanout branch.

- The input vectors are provided in the same format as Project 1. Each line contains *k* characters, where *k* is the number of circuit PIs. The characters may be 0, 1, or X.

- The system's output file will give the test vector found for each fault. So, if you have 100 lines in your fault_file, then the output will be 100 lines long. If no fault is found, it will print "none found" for that line.

3. What's New in the API Code?

Nothing! Your project 2 API already has all the capabilities we will need for this project. However please note that the API documentation also now includes documentation on the functions in main.cc, to help you find your way around this file.

4. Part 1: Baseline PODEM ATPG [55 points]

In Part 1 you will implement PODEM as described in our class notes. You are provided some skeleton code with helpful comments and pseudocode. You will write the following functions in `main.cc`:

- `podemRecursion`, the main recursive algorithm. This function calls the `backtrace()`, `getObjective()`, `setValueCheckFault()`, and `simFullCircuit()` functions. The structure of this function matches the pseudocode we discussed in class, with some more details in the comments.

- `getObjective(Gate* &g, char &v, Circuit* myCircuit)`, the function that returns the PODEM algorithm's next objective gate `g` and value `v`. Note that the function takes `g` and `v` by reference, so it can simply set those to the objective Gate and value. Here, the algorithm may need to make a decision as to which gate to pick. As we talked about in class, the algorithm has some flexibility here. For now, just make the decision arbitrarily: when you need to select a gate from the D frontier, just pick the first gate. Note that if you make the decision differently, your code will still work. However, your results will not match the exact test vectors given in the reference outputs. This is because of the fact that there can be many possible correct tests for a given fault; if your algorithm chooses objectives in a way that is different from my reference solution, our codes will likely find different tests for many faults. (If you later want to optimize your code to make decisions effectively, this is fine, but first get everything working with the simpler method.)

- `backtrace(Gate* &pi, char &piVal, Gate* objGate, char objVal, Circuit* myCircuit)`, the function that determines which primary input gate (`pi`) to set to which value (`piVal`), given the supplied objective (`objGate` and `objVal`). Again, note that when you have to make a decision, just pick arbitrarily (for now). (For example, when you need to pick an input to a gate with value `LOGIC_X`, you may just pick the first `LOGIC_X` value you see.)

- `updateDFrontier(Circuit *myCircuit)`, a simple function to find the D-frontier. For Part 2, we will just implement this as a loop: look at each gate, and determine whether or not this gate belongs on the D frontier.

- This will use my reference five-value fault simulation code. If you would like to swap in your own fault simulator (from Project 2), you can replace the `simFullCircuit()` function with code that runs yours.

After this, you should test your PODEM implementation. You will find test cases in the `test/` subdirectory. For the c432 circuit, which is fairly large, you will find three sets of fault lists. The file `c432.bigfault` tries every fault in the circuit; this will probably take about 30–35 minutes to run. We also include `c432.medfault` and `c432.smallfault`. The medium one should take about a minute and the small one should take a few seconds. For each test, there is a `.refout` file. However remember: if your PODEM implementation makes choices differently than mine, you will find different (valid) tests than I do.

Additionally, you can enable the `checkTest()` function, which will use your simulator to check that the test your PODEM function found correctly detects the fault in question. It does so by taking the vector your PODEM function found, and using it to simulate the circuit with the given fault. If the system has a D or D' at its output, then your test correctly detected this fault. This function will add some overhead to your implementation; you should use this when you are working to evaluate and test your implementation. Then,

when you are confident your implementation is correct, you can disable this function again. (Please see the comments of `checkTest` to see how to do this.) Important: this function relies on your simulation code. So, if you decide to use your project 2 simulator instead of my code, you want to be very certain it isn't buggy.

5. Part 2: Event-Driven Simulation [35 points]

Event driven simulation differs from what you wrote in Projects 1 and 2. In those (regardless of whether your code was iterative or recursive), you wrote it with the assumption that all inputs were changing for each simulation. In other words, each time you simulated something, you cleared all the internal values in your circuit and computed everything again. This is efficient *if* you can make the assumption that all inputs change at once.

However, recall how the PODEM algorithm works: each time PODEM makes a decision, it changes one PI from value X to a 0 or 1. In this case, it doesn't make sense to re-simulate the entire circuit. Instead, we will only simulate things that are changed by the one changed PI. This is called event-driven simulation, in the function `eventDrivenSim(Circuit *myCircuit, queue<Gate*> q)`.

Function description: This function works by taking as input a queue of `Gate` pointers. These pointers represent new gates that must be evaluated. The function will take the first `Gate*` out of this queue, and evaluate it based on whatever its current inputs are. It then checks: did the output of this gate change? If the output has not changed, then we can stop with this gate: since it didn't change we don't have to worry about it affecting any of its fanouts. If the gate's output value does change, then we must also re-evaluate its fanout gates, by placing them in the queue.

For Project 1, this approach would not have been optimal; if your goal is to simulate the entire circuit at once, this will be very inefficient. However, when we are doing PODEM, since only one PI changes values at a time, this will save us computation.

Write this function based on the description here and in the comments. Note, that you won't need to re-write any of the actual code to simulate the gates: you can simply use function calls to my `simGate()` function.

Then, you will need to change your `podemRecursion()` function to change its old calls (to `simFullCircuit()`) with code that will use your new `eventDrivenSim()` function. (Comment out the old call so you can easily go back and forth for testing.) To begin event-driven simulation, first consider the situation: the PODEM function just changed the value on a PI gate from X to 0 or 1. Now, you want the event-driven simulator to evaluate the gates that *might be changed* based on the change on the PI. So, you will need to create a queue of Gate pointers, and then push into the queue all of the output gates of the PI.

After you have completed this code, first test your new system, as you tested Part 1. Since your new simulation method does not affect which values are set, in Part 2 your output should be exactly the same.

However, your PODEM implementation should now be *significantly* faster for large circuits. If you aren't seeing a speed improvement, then something is wrong. Evaluate how much faster it is by timing both versions of your code (the Part 2 version and the new optimized version). Finally test and time both versions on the circuit `c432.bench` with `c432.bigfault`. Your Part 1 code should take about 30–35 minutes to finish (assuming other work isn't being concurrently done on the same machine), and your Part 2 code should be significantly faster. (Mine took about 5 minutes.) Include the timing results in your report.

A quick way to figure out how long it takes to run something in Linux is:

```
time ./atpg ...params go here...
```

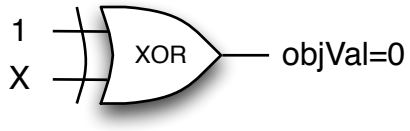
6. Ideas for Optimizations and Extensions [optional]

This section lists some ***optional*** things you could choose to try to improve your PODEM implementation. You may choose from one of my suggestions below or anything else you can think of. If you try one of these, implement it, test it, and evaluate it how well it works. Does it speed up or otherwise improve test generation? How do you know?

Some things you could try:

a. Make smarter XOR/XNOR decisions when backtracing. When we backtrace, we are not treating XOR and XNOR gates in a very sophisticated way. Other gates have a "controlling value" so it's easy to determine which input value to pick for each gate.

   However, an XOR or XNOR gate has no "controlling" value; that is, the gate's output always depends on all of the inputs. So, we can have situations like this:



   Here imagine that the XOR gate has one unknown input, and that our backtrace function is trying to figure out how to set it so that the gate's output is objVal=0. Now, if we were to approach this in the same way we approach AND/OR/NAND/NOR gates, we would simply call the XOR a non-inverting gate, and continue the backtrace. However, we can see that this is suboptimal: since the *other* input of the XOR is 1, we really know that we want the unknown input to be 1 (that is, we should increment `num_inversions` for this gate).

   If you try this optimization, first think carefully about in which situations we can improve our handling of XOR and XNOR gates.

b. Improve D-frontier management. When we wrote the `updateDFrontier()` function, we did this in an inefficient way: every time we want to know what gates are on the D-frontier, we cleared the entire list and re-examined all of the gates.

   For this optimization, rather than using the `updateDFrontier()` function, you can extend your event-driven fault simulator to maintain this list as it evaluates

gates. Each time you set the value of a new gate, determine if the gate should be added or removed from the D-frontier.

c. Reducing the size of your test set. When we run PODEM, we are running the algorithm one time for each fault we would like to detect. However, in reality, many of the tests you generate will detect multiple faults. For this optimization, your goal is to reduce the number of unique tests you need to detect all the faults in the fault file.

   Two things to try here are:
   1. After you run PODEM for a particular fault f0, then run fault simulation to check to see if the test is able to detect any of the other faults on the list (which have not yet been detected).
   2. Then, another optimization is to take advantage of don't cares in the test vectors generated by PODEM. For large circuits, often PODEM finds a vector with many X values on the inputs. Before you do the fault simulation to try this test on other faults, you can randomly replace the X input values with 0s or 1s. With this change you will still be able to detect the initial fault you were targeting, but you may be able to find other faults you can detect with the same test.

   For this optimization, you should measure its effect in terms of the number of unique test vectors needed.

d. Calculate SCOAP metrics and use them to make better decisions in `getObjective()` and `backtrace()`. As you recall, SCOAP metrics measure how hard it is to control or observe a value. In Part 1, we made a few decisions arbitrarily: we have to pick a gate from the D-frontier in `getObjective()`, and when backtracing we have to pick which gate inputs to trade our signal through. In both of these cases, SCOAP metrics can be used to make smarter decisions. For example, when you are picking a gate from the D-frontier, you can pick the gate with the lowest combinational observability cost.

   If you would like to try this optimization, you will need to add some code to `ClassGate`. If you're interested in this one come talk to me.

e. Any other optimizations you can think of. If you have an idea of something else you'd like to try for this part, please come talk to me.

This section is optional.

7. Code and Code Structure

All of your code (except possibly some of the optional extensions) will be located in `main.cc`. Be sure to document your code. If you add any functions to this file (which is probably a good idea), you should place them at the bottom between the two lines. Please do your best to make sure your code is structured and documented well. A portion of your grade will be based on this.

8. Report

Write a report describing, for each part: what you did, how you verified its correctness, and how you evaluated the quality of your implementation. Measure how much of an improvement was given by the event-driven simulation (Part 2).

If you tried any of the optional optimizations, include in your report what you did and evaluate how well it worked. Explain your thought process and how it translated into code. In the end, if your optimization did not improve the quality of the generator, think carefully and hypothesize about why.

Please submit your report as a PDF file on Blackboard (see below).

9. Code and Report Submission

Most likely, all of the code you write will be contained only in the `main.cc` file. (With the possible exception being the optional optimizations.) ***Please submit only the main.cc file, plus other files only if you needed to modify them for your optimizations (most likely none or just ClassGate.cc).***

You can submit your code and PDF on Blackboard. On Blackboard go to Assignments → Project 3.

10. Grading

Grades will be assigned based on the following breakdown:

Part 1: Baseline PODEM algorithm	55%
Part 2: Event-driven simulation	35%
Code and report quality and documentation	10%