Combinational Circuit PODEM Algorithm for Gate Substitution

David Kindel

December 14, 2014

Objective

This report, along with the accompanying code and README file, are all submissions for the ECE 5505 final project. Together, this all provides an implementation of the PODEM algorithm for gate substitution faults, written in Google's Go language. The PODEM algorithm is run as a single fault analysis tool, running at any given instance, on a single faulty gate substitution. It will not detect, or even look for, multiple faults in a single run.

Introduction

Gate substitution gave a different perspective on the PODEM algorithm. Most resources, online or elsewhere, provide descriptions of how PODEM operates for the stuck at fault case but didn't expand further than that. This project seeks to create some variation, especially since there are so many resources online. This ensures that something unique is done.

Google's Go language was chosen as the desired language to use for a couple reasons. The first reason is that the circuit file in C we were given to start with was too cluttered to make efficient sense of and had very few comments to help. I wanted a new take on what has already been provided so as I move forward, I don't hit unnecessary roadblocks. A lot of the code on building the circuit was transposed from the C file, however.

The second reason Go was chosen is that its syntax is quite C-like, obviously with some caveats. It became trivial to transpose the necessary elements from the initial file to the new Go file. Its familiar syntax, yet refreshing readability and structure, appealed to me and coding up small or large programs can be simple. Despite not being a "standard" language that one would have on any computer, install is easy on any platform.

Similarities to a Stuck-At-Fault PODEM Algorithm

There are a considerable number of similarities between my implementation of a gate substitution PODEM model and the traditional stuck-at-fault model. This section will cover just a few notable items.

5 variable algebra

A 5 variable algebra is used in this combinational PODEM implementation as well. This is because the gate substitution fault can clearly be modeled as such. To sensitize the fault, as explained in the next section, inputs need to be provided to the faulty gate that will provide a differing output. This output can described with the D algebra in the same way as a stuck-at-fault model. If the fault-free gate's output is a 0 and the faulty gate's output is a 1, the gate's output will be seen as a D'. If it's the other way around, it can be seen as a D. This was a very useful idea because many ideas and practices from a stuck-at-fault PODEM implementation could be used because of this.

Core functionality

As hit on above, the 5 variable ensured that the core functionality and the methods used would be equivalent. The methods that have been implemented and discussed in the section "A Look at the Code" are nearly equivalent to those in a stuck-at-fault implementation. They perform the same duties as they would otherwise and only have minor differences in looking at the faulty gate.

Differentiation From a Stuck-At-Fault PODEM Algorithm

There are only a few notable differences between the algorithms and most that had been encountered during development of this project dealt with setting up the algorithm before actually entering anything.

Sensitizing the fault

In order to sensitize the fault in a stuck-at-fault model, it is very simple. If the line is "supposed" to be a 1, we want to set it to a 0. If it's "supposed" to be a 0, we set it to a 1. The idea for the gate substitution is similar but at the same time, very different. If the gate is "supposed" to validate to a 1, we want to make sure it's a 0, and vice versa. This is performed by setting the inputs to the gate to certain values in order to ensure some kind of a D variable is generated. A function was necessary to find what input values would cause this effect.

Lines working with

Because of the situation in sensitizing the fault, it is clear that we need to work with multiple input lines in order to sensitize and ultimately propagate the fault. In this project, this was solved by running through the test multiple times (if necessary) for different inputs to the faulty gate that will ensure fault sensitization. This also caused objectives to be generated on input lines to the faulty gate instead of the gate itself and backtraces to ultimately happen in the same places.

Trying again after popping stack

If a simulation occurs where the implication stack has been entirely popped off and PODEM has effectively failed, other inputs may not have been tried yet. The entire PODEM algorithm is run for a different input sequence for the faulty gate. While this may, in the end, cause more duplicated work, different inputs can cause different backtraces and that could cause different primary input vectors to be found.

Everything else worked equally well for PODEM with gate substitution vs PODEM for stuck-at-faults. Backtracing, backtracking, and implications all acted the same on a given circuit.

Performance

When running for any given correctly formatted levelized circuit, the faults described in the accompanying .flt fault file are accurately simulated. If they are undetectable, a failure message is displayed and if they can be detected, they are. The c4 test files provided in this project simulated 37 faults across 9 different gates. From this test case, 32 faults were successfully detected and their test vectors were displayed. 5 faults were deemed undetectable by this version of PODEM. Therefore, the fault coverage was about 86.5%.

While I can't say for sure that this Fault Coverage is "good" or not, I do know that the faults that are undetectable by this PODEM implementation are indeed undetectable by this method, on paper or in the program.

I ran some general timing measurements on my laptop. My laptop is running Ubuntu 14.04, with an Intel Core 2 Duo processor at 2.4GHz. For a baseline, running a simple "Hello, World" program on my laptop runs is .215 seconds. To run the 37 faults from application initialization to completion, using time, it took between .31 and .35 seconds. To run a single fault on the same circuit, it took generally the same time. Again, I can't necessarily declare that this is an efficient time or not without a comparison but it certainly tells me that there is room for improvement. Despite that, though, what this has told me is that the actual PODEM algorithm takes a negligible amount of time for small circuits in Go.

What This Project Has Taught Me

This project had taught or reinforced several things for me, most notably that it's important to maintain a good understanding of the problem at hand while working.

Comparison to the D-algorithm

The biggest issue I had was that I had begun working on the project without fully understanding the PODEM algorithm. I started to, unknowingly, implement the D-algorithm. I had set up, implemented, and tested multiple functions to make my code work but it was all for the

wrong algorithm. I implemented an entire Justify function for the D-algorithm, thinking that it was essentially the same thing as the backtracing function in PODEM. It wasn't until I was about to start working on propagation of the D frontier that I realized I had been implementing the D-algorithm backwards. However upsetting this was, it cleared up a lot of confusion between the two algorithms, most specifically the difference between justify and backtracing. Backtracing is simply running a specific line's desired output back to a PI, flipping the value each time a NAND, NOR, or NOT gate is encountered. Justification, for me, was filling in every gate's possible input values as the circuit was descended. It would backtrack and pop off a new "usable" input value and try again until it either worked or was out of options. They seem like the same functions on the surface but are actually quite different when you get down to functionality. This error also caused me to understand at least a piece of why the D-algorithm was an insufficient ATPG. One error I ran into while writing the Justify function was that the circuit state could become inconsistent. This wasn't an easy thing to fix, either, and before realizing my mistake, I hadn't gotten a chance to tackle it. Because of this alone, it was clearly going to be a slower ATPG than the PODEM algorithm would be.

Fault/Circuit simulation

There were a lot of little issues I had to tackle in reaching the end goal. One of such issues was "how do I sensitize this fault?" The answer came of looping through all possible inputs and seeing what outputs different between gate types. The next question was "how do I generate every and any possible input for a gate of n inputs?" All of these small questions had to be answers in the form of functions. There are quite a few questions of this effect in implementing PODEM (or inadvertently implementing the D-algorithm).

The D algebra

The D algebra is inherently used for Stuck-At-Faults but can be just as easily applied to gate substitution faults. Once a gate has inputs that satisfy different outputs for the fault-free and faulty gate types (not counting X's), it can be shown as a D or a D', depending on if the

faulty value is a 0 or 1, respectively. The gate simulation code is used to evaluate a gate for the 5 variable algebra (0, 1, X, D, D Bar). A scanning evaluation is used on the gate to be simulation. The scan terminates when no further inputs would change the output value of the gate. The use of the D algebra really opened my eyes as to how the value D is propagated and how individual gates are calculated in an ATPG such as this.

The original project files

Rewriting the project in Go has given me a chance to really dig into the original project files, including the .lev files provided. While I had to have at least a basic understanding of the files beforehand, in recreating all the steps (aside from making simulation event driven), it validated everything I had thought. The original project written in C, while not terribly complex, had a bit of overhead to learning and very few comments included. It took a good amount of testing and verification to get an initial grasp on how the original code worked. Rewriting the code left nothing to chance. I knew how my code worked.

Specifics of the PODEM algorithm

Clearly, from just completing the PODEM algorithm, I learned specifics about how it functions.

The Go language

Before starting this project, I had a somewhat limited knowledge with Go. I knew some uses, especially in regards to concurrency, but wanted to dive a little deeper into it. This project was the perfect excuse to get some more use out of it. Go, being a relatively new language, had some holes to jump through to do certain thing. Its strict rules in syntax, however, were actually a good change to many language, such as Python or Perl. They didn't just encourage, but forced code to look a certain way, greatly enhancing readability. When all Go code is uniform, quickly getting acquainted with a colleagues code base is more trivial. Perl, for instance, has a motto that there's more than one way to do it. This, while making writeability pretty nice, might make things infinitely more difficult for anyone reading source code. Having so many variations in syntax can easily confuse developers in large or small scale applications.

A Look at the Code

In this section, I'll give an overview of the critical functions necessary for making PODEM function as it does.

sensitizedFaultList

This function build a multidimensional list of inputs to the faulty gate that will sensitize the fault. This is used mainly so we know what inputs are actually necessary in order to propagate a D or D' to the output. Each index into the array retrieves a list of specific gate inputs that will sensitize the fault. Until the fault is propagated, each input list is cycled through in order to fully test. When we try to find an objective in PODEM, we use this particular input list to see what values should trickle down to inputs.

runPodemAllFaults

This loops through all of the faults specified in the .flt file. It injects the fault before calling PODEM and then, after, restores the fault-free value, ready for the next run through. It also collects the number of successes and failures.

runPodem

runPodem() is the base PODEM algorithm method. It runs through the list of inputs, calls the xpath, objective, backtrace, and imply functions, and also determines if a test vector either passes or fails or if a backtrack is necessary. It does a lot of heavy lifting.

xpathCheck and xpathRecur

The X-Path check essentially checks if a test can still be completed, given the inputs that are currently assigned. It does this recursively, thus the xpathRecur function. There are 3 tests performed. The first is checking if the faulty gate is still an X. The second checks if the D frontier can be reach from a path of all X's from any PO. The last check verifies that, if the previous two validate, the D frontier has any next gate that evaluates to X so that the D frontier still has a chance to advance. This is described in the next point. A recursive

function was decided to make this more naturally coded. We know that we'll just descend the list and check for whatever we need to at that specific gate. This lent itself quite well to a recursive function since the same operation would be applied at different levels and the operation resulted in something similar to a tree.

getObjective

The objective function takes in the particular input list and the faulty gate number. It is used to determine where the backtracing function will begin. Using this knowledge, we know that if the faulty gate is still an X, we have to loop through the inputs to the gate to find any x and make that particular gate the current objective. Otherwise, we must make the objective a gate that evaluates to an X as a next gate on the D frontier. This means that, to advance the D frontier any further, there must be an X because we can make some further observations about the input list.

xGateFromDFrontier

This method goes and finds any gate that evaluates to x on the D frontier. This, like the xpath check is a recursive function because it fans out, finding any gate that evaluates to X on the D frontier. It first searches to the extent of the D frontier. If it finds, another D variable, it calls itself on that gate to progress. Otherwise, if it sees an X, it will return that gate along with the value that we want it to become.

backtrace

The backtrace function takes in an objective and returns a PI. The algorithm loops through inputs based on the current value and the input's controllability based on that value. If all of a gate's inputs need to be set, we want to set the hardest controllability for the gate's input. If not all do, we want to set the easiest controllability. This isn't a mandatory requirements but it potentially makes backtracing a little easier now and require fewer steps later on. Once a PI is found (that does not already equal X), it is returned. A sequential function was chosen over a recursive function in order to help myself differentiate the justify function and the backtrace

functions for myself. The justify function of the D-algorithm was setting, and working work, gates as it descended. The backtrace function was simply looking at gates and their inputs, running down the line. Since it wasn't working with gates, and because it helped with keeping things mentally straight for myself, I decided on a sequential approach to backtracing.

implyAndTest

This runs the imply function to evaluate the circuit with the given test vector at its inputs. It evaluates each gate for its fault-free input and any supposed faulty gate and marks any difference with a D or a D'. Once the entire circuit is simulated, it tests for the D frontier at the primary outputs of the gate.

Problems

As the project was being developed, numerous problems were encountered.

Initial code overly complex

As explained previously, the initial code that was provided with this project was simply too much for anything necessary for this project. I wasn't sure what would be required and not so while the new code I've written is much more simplified, and probably not as efficiently scalable, there wasn't so much overhead to work with in the new codebase.

Program Design

In starting with a new language and new code, I was given the opportunity to design modules however I wanted. Instead of separating code out into packages as I should have, I kept everything in a main package. The decision for this was based on wanting to keep all files together, in the same directory. Go restricts packages to different directories. Better code could have been formed by keeping directory and package structures and ideally, that's what I'll do in the future.

Start writing code too quickly

Because I jumped right into transposing some of the original code, it was very easy to not actually think too much about the algorithm or design my code properly. This ended with poorly designed code, as explained directly above, and I didn't necessarily even know the PODEM algorithm as well as I should have. There were many changes I could have possibly made earlier and not wasted my time in writing functions and ultimately deleting them after I realized how unnecessary they actually were.

Conclusions

The PODEM algorithm has been accurately implemented in Go for gate substitutions. The implementation is not the most robust code that has existed but it is correct and well tested. The test files I've provided along with my code easily show basic correctness, correctness with multiple inputs, handling a variety of gates, handling multiple faults in the same file, and working with large circuits. The PODEM algorithm itself became simplistic when I designed the functions around specific and distinct pieces of the code. Recursion and other standard programming practices surely helped the code become concise and ensure accuracy.

References

[1] Sachin Dhingra. *Implementation of an ATPG Using Podem Algorithm. http://www.eng.auburn.edu/ agrawvd/COURSE/E7250_05/REPORTS_PROJ/Dhingra_Podem.pdf.*

[2] M. Tahoor. *Boolean Testing Using Fault Models (D-Algorithm, PODEM). http://cdnc.itec.kit.edu/downloads/tds1_ss_2011_lecture9.pdf.*

[3] Pinaki Mazumder. `PODEM Algorithm`. *http://web.eecs.umich.edu/ mazum/F02/lectures/TG.pdf*

[4] Krish Chakrabarty. *VLSI System Testing. http://people.ee.duke.edu/ krish/teaching/ECE269/TestGeneration2.pdf*