Georgia Institute of Technology

[ECE6140] Digital System Tests

Project 3 – PODEM: A Test Generator

Lingjun Zhu

GTID # 903433936

Dec. 7, 2018

1. Data Structure

In this project, we implemented a test generator which can generate a test set for a circuit to detect a given set of faults, based on the Path-Oriented Decisions Making (PODEM) method. First of all, we need to modify and improve the data structure designed in Project 1 and 2 to facilitate the test generation. The updated data structure is shown in Figure 1.1.

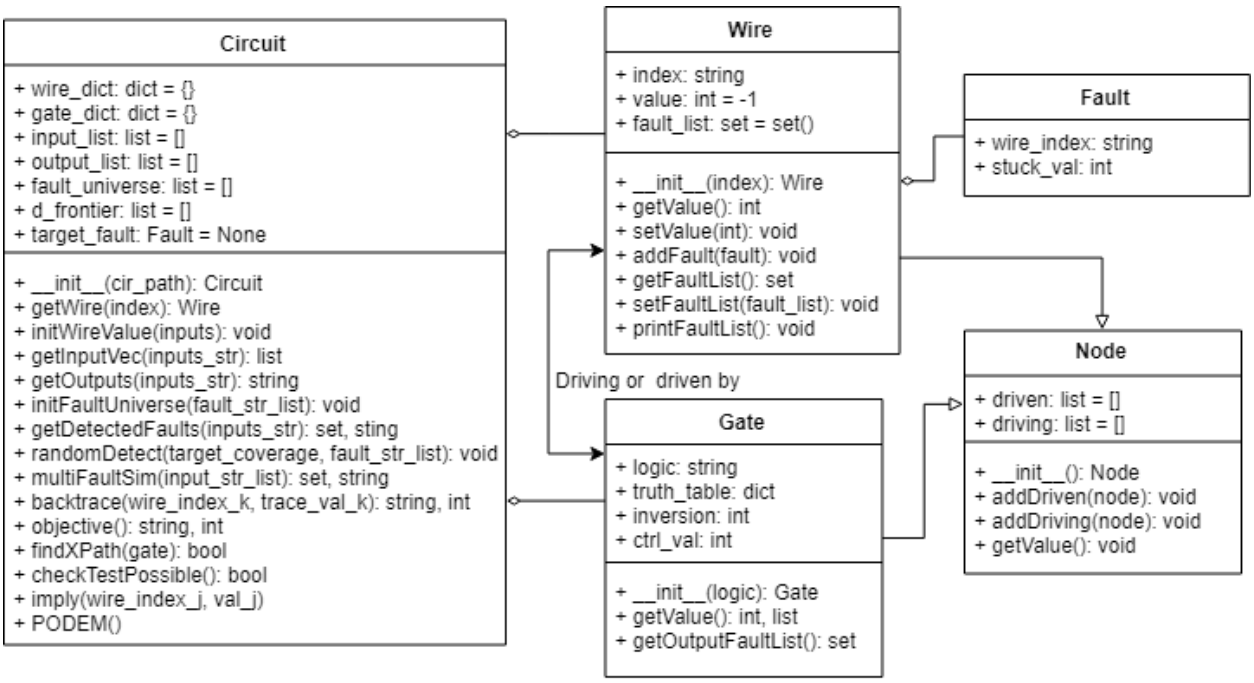

Figure 1.1 The class diagram of the test generator

The **Node, Fault, Wire, Gate** are almost the same as those in the fault simulator program. But we add some new logic values: we are using the extra integer values (i.e. -1, 2, 3) to represent the unknown value x and the errors $D, \overline{D}$. In addition, we define the rules of Boolean algebra between these values, e.g. $D + \overline{D} = 1, D \cdot 0 = 0, \overline{D} + x = x$, and so on. And we have updated the truth tables of gates correspondingly. Based on these values and operations, we can use the wire and gate model designed previously to solve the line justification and error propagation problems, which are essential in test generation.

Table 1.1 The mapping of symbols and logic values

Symbol	Integer value	Meaning
x	-1	unknown value
0	0	logic 0
1	1	logic 1
D	2	represents error, v/v_f=1/0
D_bar	3	represents error, v/v_f=0/1

In the **Circuit** class, we have added several new functions. For example, *getInputVec()* can convert the input string to corresponding test vector, considering the extra logic value mapping. What's more, we have designed the *findXPath()* and *checkTestPossible()* function to check whether the test generation is possible. The *backtrace()* and *objective()* functions are parts of the PODEM method and we will discuss them in detail in the following section.

2. Algorithm

First, we implement the PODEM method as it is described in [1]. The flow is shown as Figure 2.1.

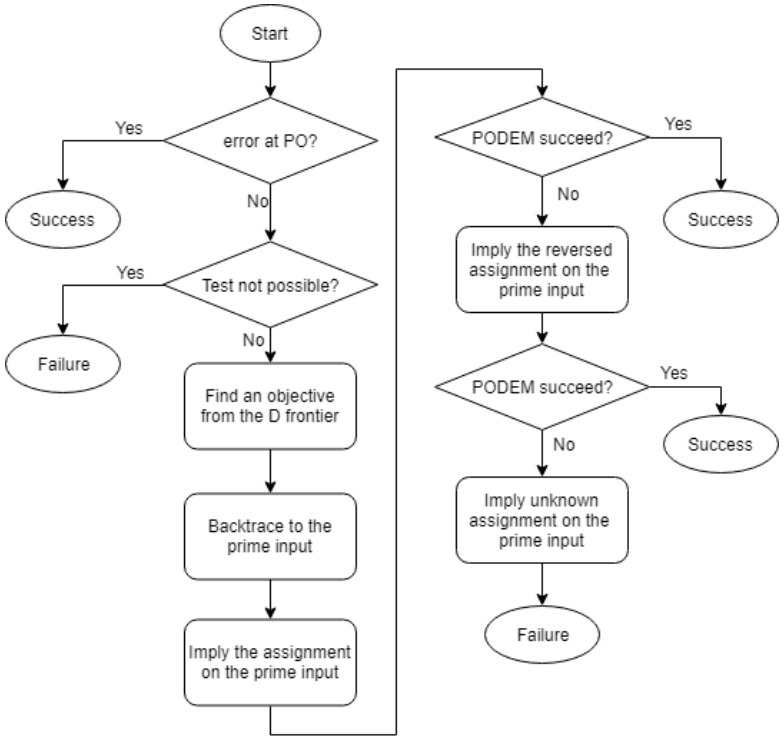

Figure 2.1 The flowchart of the PODEM subroutine

Besides, we design the *checkTestPossible()*, *findXPath()*, and *imply()* subroutines separately, all based on recursion. For each pair of wire and value to imply, the subroutine recalculates output of the gate it drives, add or remove the gate from the D frontier, and imply the output wire and value recursively.

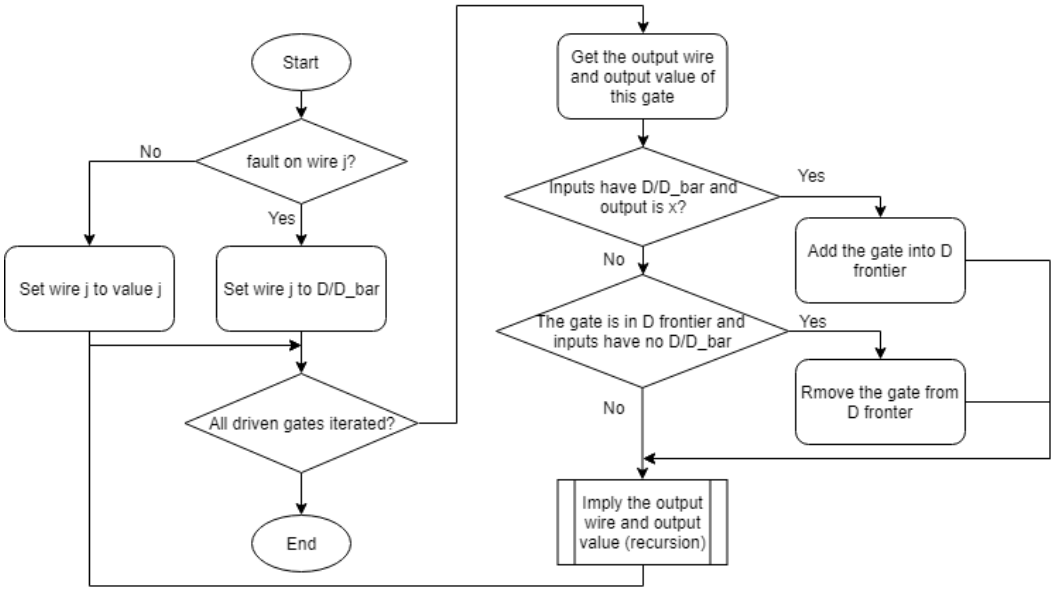

Figure 2.2 The flowchart of the imply subroutine

3. Results

The generated test set for the given circuits and faults are shown as Table 3.1. The order of bits in the test vectors are corresponding to the order of the input pins in the netlist file.

Table 3.1 The generated test for the given circuits

Circuit	Fault	Test Vector
s27	16 s-a-0	1001000
	10 s-a-1	1000100
	12 s-a-0	1000010
	18 s-a-1	
s298_f2	70 s-a-1	00101000000000000
	73 s-a-0	11100000000000000
	26 s-a-1	01010100000000000
	92 s-a-0	
s344_f2	166 s-a-0	10000000000000000000000
	71 s-a-1	11100000000000000000000
	16 s-a-0	01000000000110000000000
	91 s-a-1	00000000000000100000000
s349_f2	25 s-a-1	01000000000100100000000
	51 s-a-0	00000010000000000000000
	105 s-a-1	
	7 s-a-0	

In order to verify the results, we have fed the generated test vectors to the deductive fault simulator and checked the faults it detected. For example, the simulation results of s249_f2 is shown in Figure 3.1.

```
***** Fault Simulation ******
Netlist Path: circuits/s349f_2.txt
Input File Path: inputs/s349f_2_inputs.txt
Output File Path: outputs/s349f_2_outputs.txt
Fault File Path: faults/s349_f2_faults.txt
Read the list of fault to be simulated...
Reading completed, 4 faults read
Input:
['01000000000100100000000', '00000010000000000000000']
Output:
Num of detected faults: 4
7 stuck at 0
25 stuck at 1
51 stuck at 0
105 stuck at 1
```

Figure 3.1 The simulation results of circuit s249_f2

Obviously, the 4 target faults are all detected by the 2 test vectors, which suggests our test generator works well.

4. Conclusion

Based on the previous projects, we have implemented the PODEM test generator and verified it with the deductive fault simulator. The working principle of the test generator is not very complex, but it covers many problems in the domain of digital test, including line justification problem, global implication problem, error-propagation look-ahead problem and so on. I think it is very interesting and beneficial to dig into this topic.

References

[1] Abramovici, Miron, Melvin A. Breuer, and Arthur D. Friedman. Digital systems testing and testable design. Vol. 2. New York: Computer science press, 1990.