# SMART–G: Self-adaptive Model for Automated and Rapid Test-Generation

*Author:* Uday Gurnani, Georgia Institute of Technology – Atlanta

## I. Abstract

This report presents the SMART-G or Self-adaptive Model for Automated and Rapid Test Generation approach to test vector generation for single stuck-at-faults in a combinational circuit. SMART-G is an implementation of the implicit enumeration algorithm, PODEM or Path-Oriented Decision Making (Goel et.al, 1981). SMART-G provides a rapid test generation method, making use of recursive function calls and an object-oriented protocol. SMART-G is platform-independent and can be run for any user-defined circuit layout containing one or two-input logic gates to generate tests for any stuck-at-fault in the circuit. This report runs through the three phases in the SMART-G implementation: logic simulation, fault simulation and finally the PODEM-based test generation.

## II. Introduction

The SMART-G approach is an implementation of the PODEM algorithm that uses the single stuck-at-fault model. The single stuck-at-fault model assumes that only one line in a circuit is faulty at a time, the fault is not temporary and that the functionality of the gates in the circuit is not affected by the presence of the fault. Each fault is treated to be either stuck-at-1 (connected to Vcc) or stuck-at-0 (connected to Ground). The single stuck-at-model is advantageous to use for test generation and fault detection, since it covers more than 90% of defects that can exist in CMOS circuits during manufacturing. In digital system testing, it is difficult to test for the complete function of a logic circuit, but since the single stuck-at-model can be applied at both the logic and module levels it overcomes this shortcoming.

The single stuck-at-fault model was used in the D-algorithm, a precursor to PODEM and the first method that allowed circuit tests to be generated using a computer program. However, PODEM is a lot more efficient as compared to the D-algorithm. The D-algorithm implies a test-generation technique where choices are possible at each and every internal circuit node. Therefore in the D-algorithm the number of solutions that need to be checked is exponential with the number of circuit nodes. The presence of XOR gates in circuits being tested using the D-algorithm make the searching procedure extremely cumbersome. In PODEM, on the other hand, choices are possible at each primary input, and therefore the number of solutions possible is exponential to the number of primary inputs. PODEM focuses on primary input assignment and applies values at the primary inputs as opposed to circuit nodes. Typically circuits have a lot lesser number of primary inputs than circuit nodes, making the PODEM algorithm a lot less complex and more efficient. [1]

The SMART-G method capitalizes on this efficiency embedded in the PODEM algorithm, making it possible for tests to be generated in quick time, even for complex circuits. This report focuses on the way SMART-G has been implemented in its three-phased model, the logic behind the algorithm used for each phase and finally the results obtained in each of the three phases.

---

[1] See http://www.cedcc.psu.edu/ee497i/rassp_43/ for more information on the D-algorithm and PODEM

# III. Implementation

The SMART-G approach was implemented in three phases. The three-phase approach was necessary since it provided the right foundation to build upon to be able to finally implement the PODEM algorithm. Phase 1 of the implementation consists of a basic digital logic simulator. This allows the primary inputs to be excited by user-defined values, and then propagates those values through the circuit, returning values of all circuit nodes and primary outputs. Phase 2 consists of a fault simulator that generates a certain number of random test vectors and then for each vector it simulates all faults in a circuit, one at a time, and tests for fault coverage. Phase 3 implements the PODEM algorithm making use of the digital logic simulator from Phase 1 to compute circuit node and output values. The fault simulator from Phase 2 is also used in Phase 3 to simulate faults and to crosscheck the PODEM algorithm.

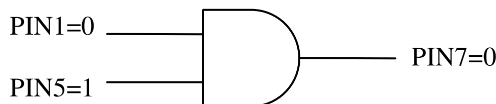
## Phase 1: Implementation of Digital Logic Simulator

## 1. Description

The primary data structure I have used to implement the simulator is a linked list *ll,* which consists of objects of the class *node*. Every object of the class *node* represents a primary digital logic gate and its corresponding inputs and outputs. Several *node* objects when linked together make up the entire digital circuit that needs to be simulated. The class *node* is used to hold the input and output pin numbers (*input1, input2, output*), input and output values (*ivalue1, ivalue2, ovalue*), the logic gate (*opcode*), and a boolean flag *eval* that indicates if both input values of the gate are available for output generation. The class *node* and linked list *ll* are defined in the C++ code as follows:

```
class node{public:
   int input1, ivalue1;
   int input2, ivalue2;
   int output, ovalue; //
   int opcode;// 1=BUF,2=INV,3=AND,4=OR,5=NAND,6=NOR,7=INPUT,8=OUTPUT
   bool eval; //eval=1 if both inputs are known
   node(){  } //Default Constructor};
#include<list>
list<node> ll;
```

For example, the gate shown below will have input1=1, ivalue1=0, input2=5, ivalue2=1, output=7, ovalue=0, opcode=3 (for AND)




Objects of the class *node* add onto the linked list *ll*, using the *ll.pushback( )* function that is defined in the <list.h> header file. Memory allocation is dynamic, such that a new *node* object is only created if there is a primary digital gate that forms part of the final circuit. Therefore once the text file (Eg. s27.txt) is read, and all *m* digital gates have been defined, the linked list would be as shown below:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Object 1 | input1 | ivalue1 | input2 | ivalue2 | output | ovalue | opcode | eval |
| Object 2 | input1 | ivalue1 | input2 | ivalue2 | output | ovalue | opcode | eval |

•
•

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Object m-1 | input1 | ivalue1 | input2 | ivalue2 | output | ovalue | opcode | eval |
| Object m | input1 | ivalue1 | input2 | ivalue2 | output | Ovalue | opcode | eval |

For traversing the linked list the program uses an iterator (*str* or *str2*) within a *for* loop, and the *ll.begin( )* and *ll.end( )* functions that return the first and last *node* objects in the linked list, respectively.

In addition to the linked list *ll* of *node* objects, the program makes use of two 2-D arrays that keep track of all the pin numbers and values for the inputs and outputs to the digital circuit under test. These are the *cinputs[m ][2]* and *coutputs[m ][2]* arrays. The first column of each of these arrays stores the pin number for the input/output and the second column stores the value for that input/output.

## 2. Simulation Algorithm

The pseudo-code for the simulation algorithm is as follows:

1. Ask user which text file he/she would like to use to extract the test circuit's parameters

2. Extract every line from the file the user selects above

3. Extract every word from every extracted line

4. For every extracted word, analyze the first word to check which logic gate it refers to

5. For INV/BUF gates define second word to be the input and output pin #s, respectively

6. For the rest of the logic gates define the second and third words to be the gate input pin numbers and the fourth word to be the output pin number

7. Assign all inputs and outputs a non-boolean default value of "2"

8. Store the input/output and logic gate information as part of a single row in a list. Each element represents a logic gate and its inputs and outputs (linked list *ll*)

9. From the text file, since the last two lines represent the entire circuit input and output pin numbers, extract this information and store it in the first column of an n x 2 matrix (array *c_inputs[][]* and *c_outputs[][]*), where n is the number of inputs/outputs.

10. Using the input pin numbers stored in the above array, run a loop to accept boolean values for each input pin from the user

11. Traverse the linked list (ll) and the array of circuit input pin numbers (*c_inputs[][]*) simultaneously and compare to find logic gates that have the same input numbers as the circuit input numbers. Since the values of these inputs are known, accordingly assign the ivalue1 and ivalue2 parameters of the logic gates

12. If both the input values of a logic gate are known, set its flag *eval* to 1 and calculate the output (*ovalue*) of that logic gate

13. Traverse the linked list once again and compare the output pin numbers of all the logic gates that have their output defined, with the input pin numbers of all logic gates. When a match is found, assign the value of that input pin (*ivalue1* or *ivalue2*) to be the value of the output pin (*ovalue*) under consideration

14. Traverse the list to check if the flag *eval* of all the logic gates is set. If all flags are not set, then go back to Step 12.

15. Since all output values are known, traverse the linked list and display only those logic gate outputs that are also the final circuit outputs (as defined in the array c_outputs[][0]).

# Phase 2: Fault Simulator

## 1. Description

The Fault Simulation program enables users to determine how many faults of the total possible faults in a given circuit are detectable, along with the corresponding fault sites, stuck-at-value and a test vector that is capable of detecting that fault. The program simplifies the process into a few clicks with minimal user feedback required. Users pick a circuit they hope to test and generate a .txt file where they specify the gate-level circuit design. For example, an AND gate with its inputs at pin 1 and pin 5, and an output at pin 7 would be represented in the .txt file as AND 1 5 7. Once the user double clicks on the main.exe application file, the program runs and displays results as it computes.

Users provide the number *'n'* of random test vectors to be generated which the program uses to determine what percentage of faults are detectable using 'n' number of random test vectors. The program output consists of both a screen progress display and an output .txt file that gives all details including what faults have been detected, their stuck at value, what random vectors were used. A screenshot of a circuit below based on *s27.txt* shows the pin number, stuck at value and the corresponding test vector. The parameter "Source pin number" tells the user from where that particular pin has fanned out. When this parameter is zero, it means that a primary output is the one under consideration. The intelligence to handle fan-outs as separate points is an important attribute of the SMART-G fault simulator.

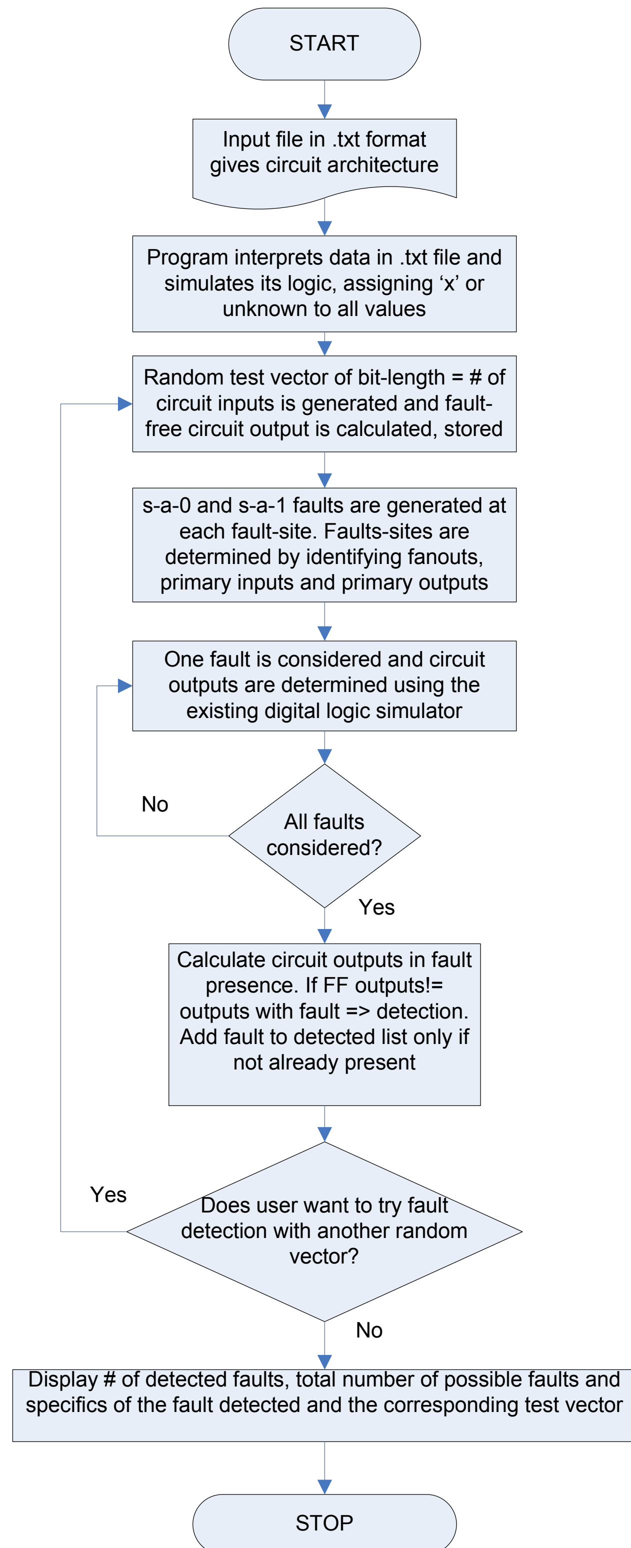
```
Detected faults are as follows:

Pin number: 12  Source pin number: 12   Stuck at 1 is detected by 1011011
Pin number: 14  Source pin number: 14   Stuck at 0 is detected by 1011011
Pin number: 17  Source pin number: 17   Stuck at 1 is detected by 1011011
Pin number: 2   Source pin number: 2    Stuck at 1 is detected by 1011011
Pin number: 10  Source pin number: 10   Stuck at 1 is detected by 1011011
Pin number: 8   Source pin number: 8    Stuck at 1 is detected by 1011011
Pin number: 19  Source pin number: 19   Stuck at 1 is detected by 1011011
Pin number: 4   Source pin number: 4    Stuck at 1 is detected by 1011011
Pin number: 7   Source pin number: 0    Stuck at 1 is detected by 1011011
Pin number: 9   Source pin number: 0    Stuck at 1 is detected by 1011011
Pin number: 11  Source pin number: 0    Stuck at 1 is detected by 1011011
Pin number: 5   Source pin number: 0    Stuck at 0 is detected by 1011011
Pin number: 21  Source pin number: 9    Stuck at 1 is detected by 1011011
Pin number: 23  Source pin number: 1    Stuck at 1 is detected by 1011011
Pin number: 25  Source pin number: 15   Stuck at 0 is detected by 1011011
Pin number: 30  Source pin number: 20   Stuck at 1 is detected by 1011011
Pin number: 9   Source pin number: 9    Stuck at 1 is detected by 1011011
Pin number: 15  Source pin number: 15   Stuck at 0 is detected by 1011011
Pin number: 20  Source pin number: 20   Stuck at 1 is detected by 1011011
Pin number: 1   Source pin number: 1    Stuck at 1 is detected by 1011011
```

Overall the ease-of-use, speed of computation and the ability to track results both through real-time progress updates and through a .txt file for subsequent analysis makes this fault simulation tool a great buy.

## 2. Flowchart for Fault Simulation, Detection Algorithm

This flowchart presents the algorithm that was designed for fault simulation and detection.



# Phase 3: Test Generation

## 1. Description

### a. Program Input/Output

The test generation phase is the final goal of the SMART-G project. The user specifies the circuit layout file to be read and faults to be detected in two text files, “input.txt” and “faults.txt” respectively. The program interprets the circuit layout specified in the circuit layout files. The input.txt file should contain a single digit that represents the circuit file required:
1 = s27.txt
2 = s344f_2.txt
3 = s349f_2.txt
4 = s298f_2.txt

Example: input.txt
1

This will retrieve the circuit layout from s27.txt

In faults.txt, the user will enter <number> space <number>. The first <number> represents the line number of the faulty line in the circuit. The second number can be 4 or 5, where 4 implies stuck at 0 fault (D=1/0) and 5 implies stuck at 1 fault (Dbar=0/1).

Example: faults.txt
1 5
12 4

This will cause the program to generate test vectors for the faults 1 s-a-1 and 12 s-a-0.
Once the program is run the output.txt file will display the test vector for the simulated fault, the primary output at which the fault is detected and the primary input/output truth table. It will also display that the PODEM algorithm has been verified. This verification is performed by running the generated test vector through a fault simulator, developed in Phase 2, to check if the D or Dbar has actually propagated to the output. More about this verification procedure is outlined in Section IV, “Test Validation Methodology.”

### b. Program objects

**i. “node” class:** Each object of this class represents a logic gate in the circuit. SMART-G reads the user-specified circuit layout file and generates an object of the “node” class for every logic gate in the circuit. Every “node” object has the following parameters associated with it:

- input/output pin numbers (input1, input2, output)
- input/output pin values (ivalue1, ivalue2, ovalue)
- logic operation (opcode)
- program-generated gate identification number assignment (gateid)
- flag representing whether gate is at d-frontier or not (dfrontier)

ii. **“faultlist” class:** Each object of this class represents a fault in the circuit. Based on the faults specified in “faults.txt,” SMART-G generates a “faultlist” object for each fault in the circuit. The object has the following parameters associated with it:
- fault line/pin number (fpin)
- stuck-at-value (fvalue)
- program-generated test vector that detects the fault (tvector[])

iii. **“cktline” class:** An object of this class represents any line in the circuit. It is primarily used by the PODEM(), Objective(), Imply() and Backtrace() functions to return a line number and its corresponding value. By default, every object of this class has an unknown value (represented by a ‘2’) and a line/pin number of ‘0’.
- line number (linenum)
- stuck-at-value (linevalue)

**c. Program’s primary functions**

i. **evalfile( ) and getvalue( ):** Reads and interprets the circuit layout file

ii. **getfaults( ):** Extracts fault information from faults.txt file and stores it in the linked list "lfault" that contains objects of the class “faultlist”

iii. **nextfault( ):** Moves the pointer to the next fault in the fault list

iv. **resetckt( ):** Resets the values of all nodes in the circuit to an unknown value

v. **displaygate( ):** Displays all circuit gates with input and output values

vi. **otptdisplay( ):** Displays truth table for all primary inputs and outputs

vii. **objective( ):** Returns the line number and the value required for the current objective (see “2. PODEM algorithm” for more information)

viii. **backtrace( ):** Returns a line number and value for a primary input, by backtracing from the node specified by objective( ) to the primary inputs (see “2. PODEM algorithm” for more information)

ix. **imply ( ):** Creates and maintains the D-frontier and performs implications from the primary input through the rest of the circuit (see “2. PODEM algorithm” for more information)

x. **PODEM( ):** Defines the PODEM algorithm as specified in “2. PODEM algorithm and flowchart”

xi. **checkPODEM():** Validates the PODEM algorithm and verifies test vector generated

## 2. PODEM Algorithm and Flowchart

### a. Pseudo-code

***PODEM( )***

```
{if (error at primary output) return SUCCESS;
if (test not possible) return FAILURE;
(k, v_k) = Objective();
(j, vj) = Backtrace (k, vk); // j is a primary input
Imply (j, v_j);
If ( PODEM ( )  =  SUCCESS)
        { return SUCCESS;}
Imply (j, v_j-bar);
If (PODEM( ) = SUCCESS)
        {return SUCCESS;}
Imply (j, x)
return FAILURE;
}
```

***Objective( )***

```
{//Target Fault is line s-a-v
if (the value of line is unknown) return (line, v-bar)
select a gate G from the D-frontier
select an input j of G with unknown value
c = controlling value of G;
return (j, c-bar)
}
```

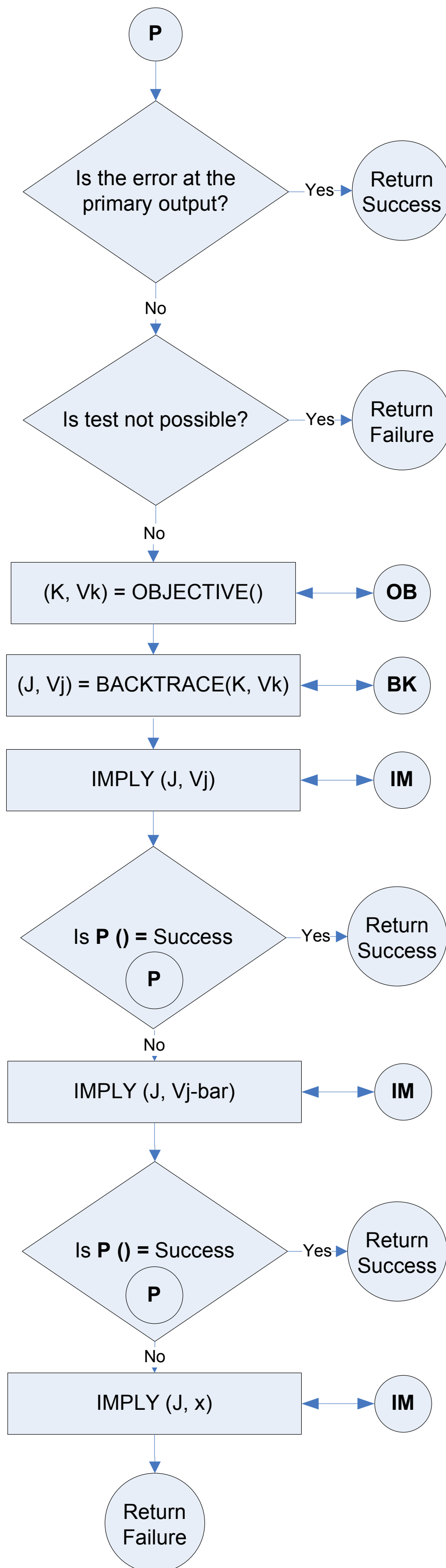
***Imply( )***

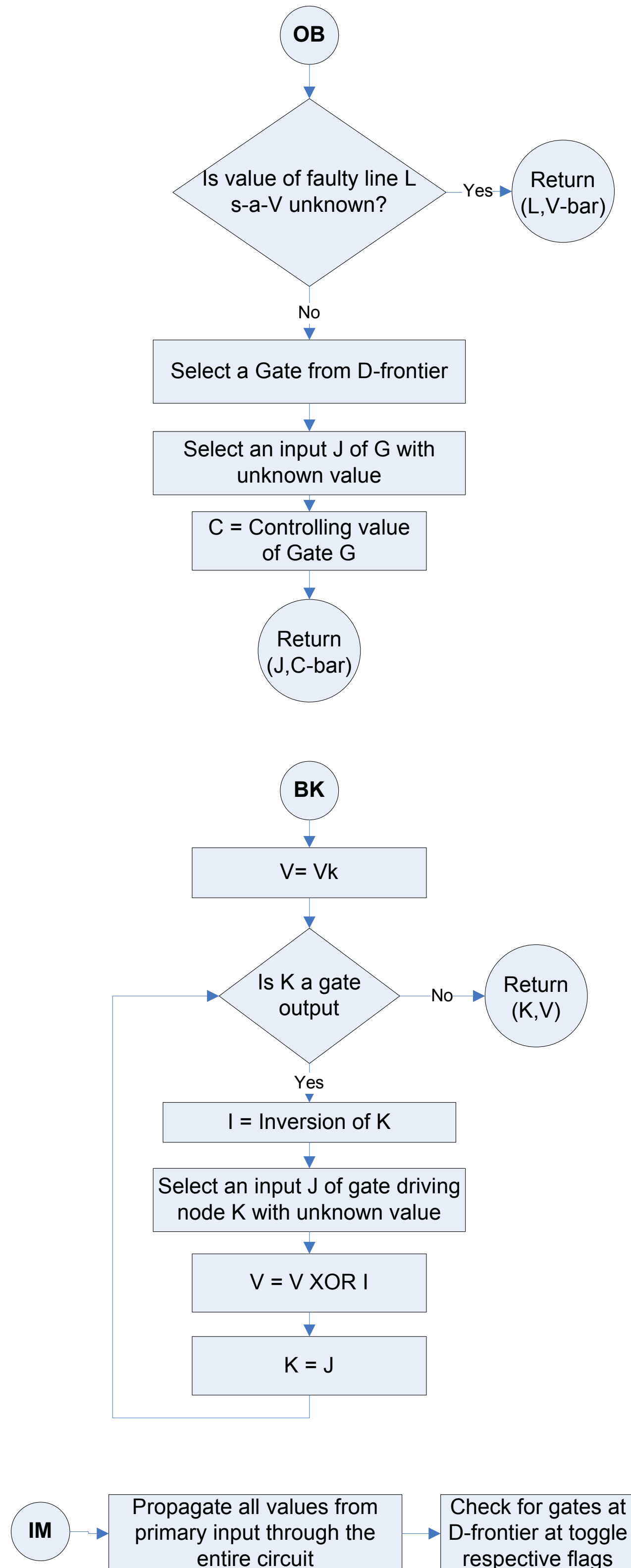
```
{Creates D or Dbar and maintains D-frontier;}
```

***Backtrace( )***

```
{v = v_k;
while (k is a gate output)
        {i = inversion of k;
        select an input j of gate driving node k with value unknown;
        v = v XOR i;
        k = j;
        }
// k is a primary input
return (k, v)
}
```

**b. Flowchart**



# IV. Results and Test Validation Methodology

## A. RESULTS

## Phase 1: Digital Logic Simulator

***1. s27.txt***

| **Inputs** (1 2 3 4 6 8 10) | **Outputs** (7 9 11 5) |
|---|---|
| 1110101 | 1001 |
| 0001010 | 0100 |
| 1010101 | 1001 |
| 0110111 | 0001 |
| 1010001 | 1001 |

***2. s298f_2.txt***

| **Inputs** (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17) | **Outputs** (18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37) |
|---|---|
| 10101010101010101 | 00000010101000111000 |
| 01011110000000111 | 00000000011000001000 |
| 11111000001111000 | 00000000001111010010 |
| 11100001110001100 | 00000000100100100101 |
| 01111011110000000 | 11111011110000101101 |

***3. s344f_2.txt***

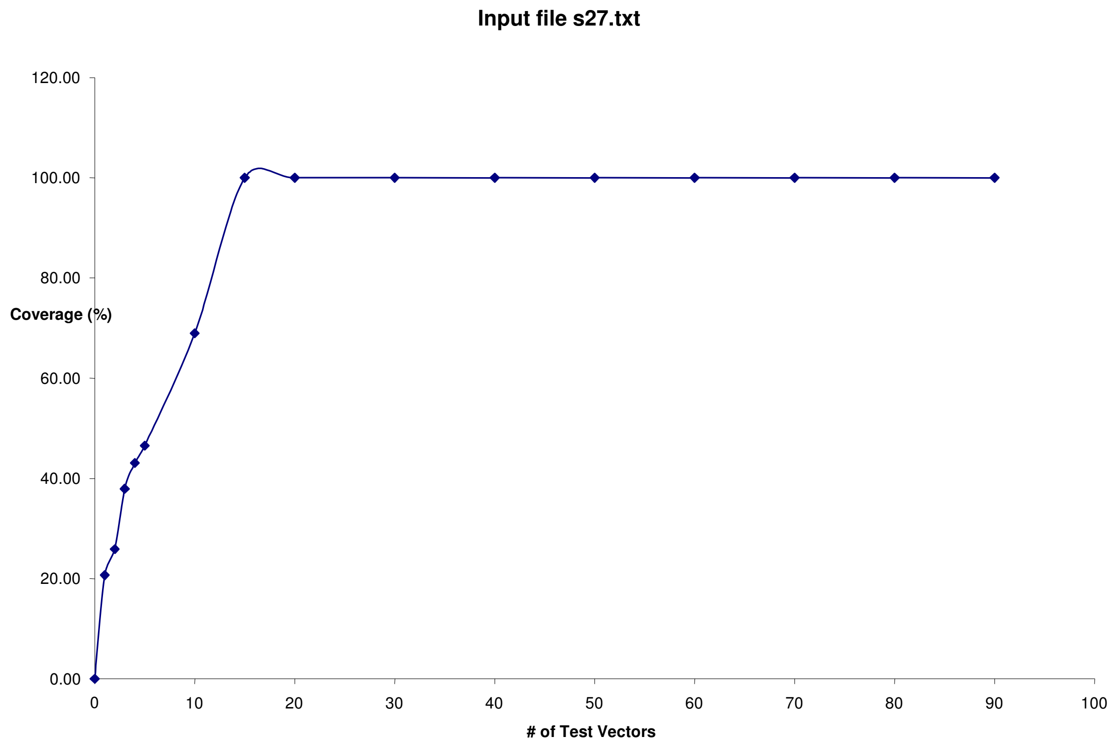
| **Inputs** (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24) | **Outputs** (25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50) |
|---|---|
| 101010101010101011111111 | 10101010101010101010101101 |
| 010111100000001110000000 | 00011110000000100001111100 |
| 111110000011110001111111 | 00011100000111011000111010 |
| 111000011100011000000000 | 00001101111001111111000010 |
| 011110111100000001111111 | 10011101111000001001000100 |

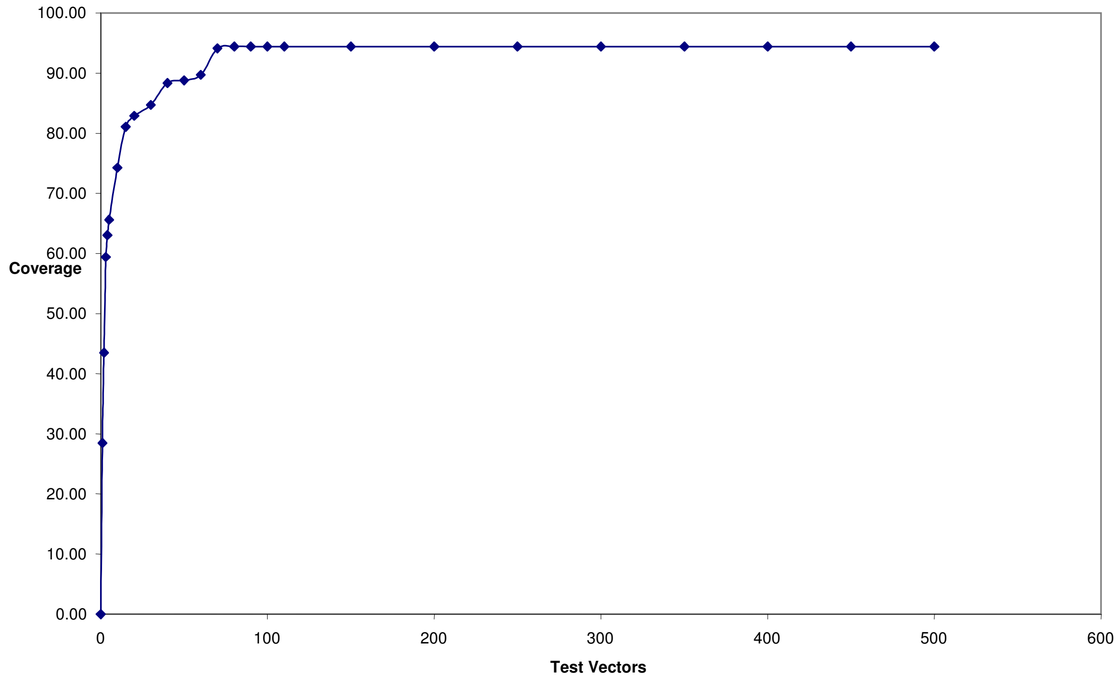
***4. s349f_2.txt***

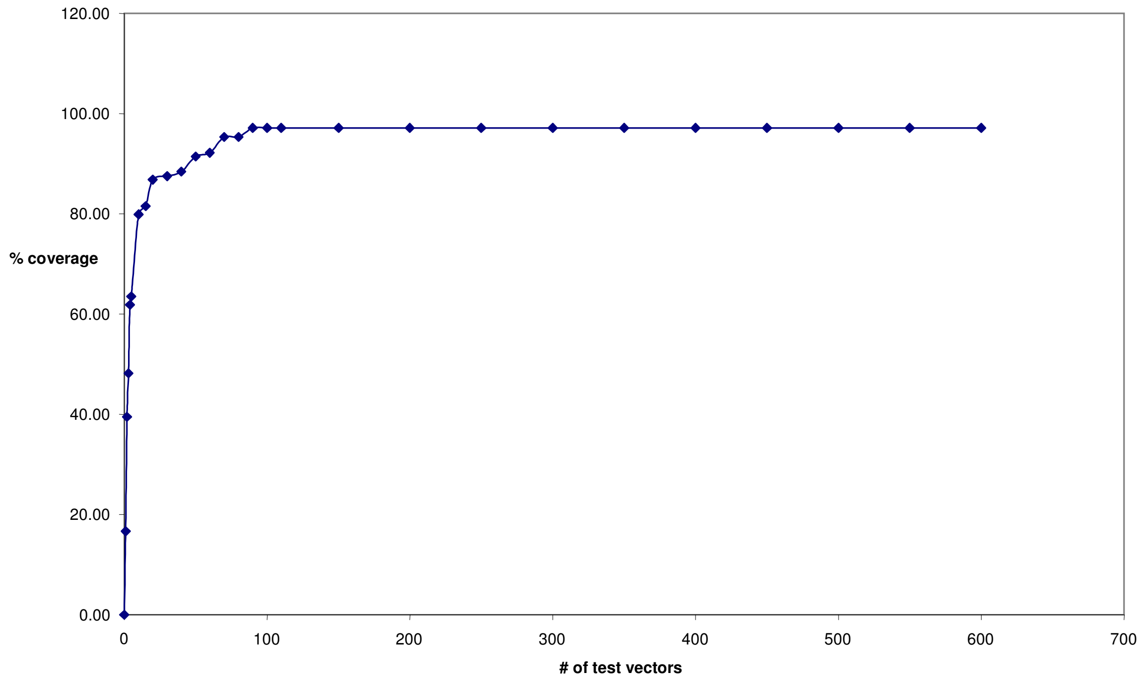
| **Inputs** (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24) | **Outputs** (25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50) |
|---|---|
| 101010101010101011111111 | 10101010101010101101010101 |
| 010111100000001110000000 | 00011110000000101011110000 |
| 111110000011110001111111 | 00011100000111010001111100 |
| 111000011100011000000000 | 00001101111001110010001111 |
| 011110111100000001111111 | 10011101111000001010000100 |

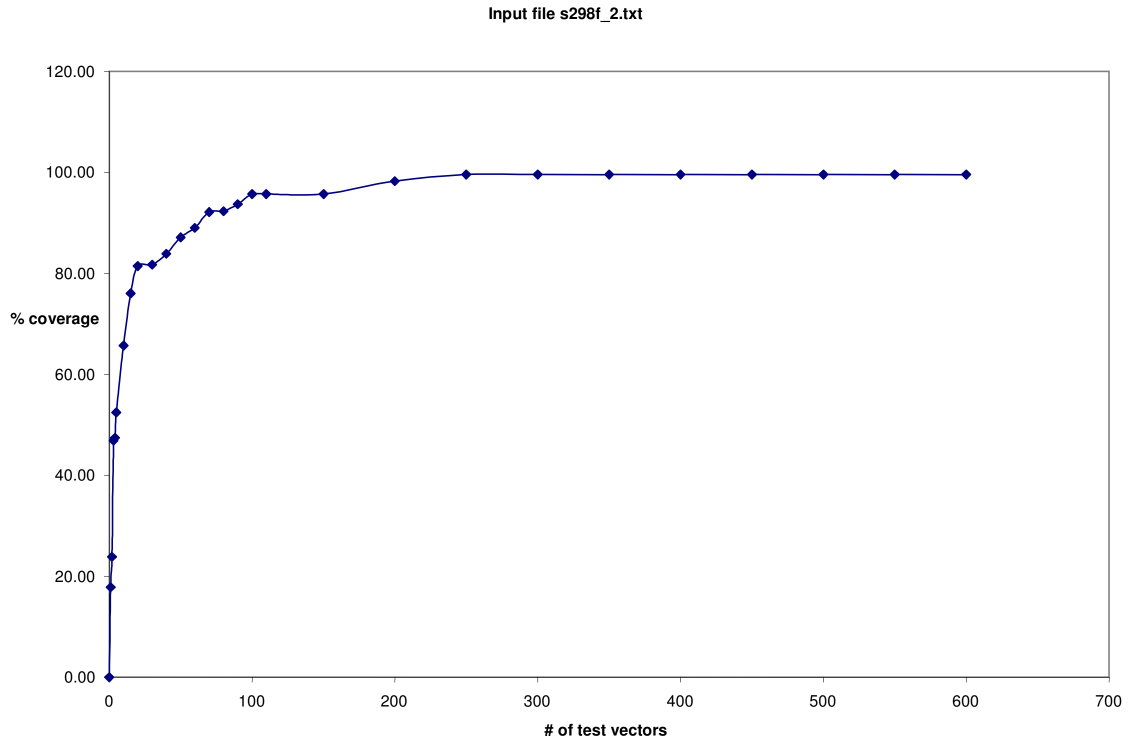
## Phase 2: Fault Simulator

The graphs below are generated for the different demo input files including s27.txt, s344_2.txt, s349_f2.txt, s298_f2.txt. The x-axis represents the number of random test vectors that were applied to the circuit. The coverage is the ration of the number of detectable faults to the total number of faults in the circuit. As we see, the more random test vectors we apply, the greater the coverage achievable, up until a maximum point, where the remaining faults in the circuit are undetectable.




Input file s344f_2.txt
Coverage
Test Vectors
100.00
90.00
80.00
70.00
60.00
50.00
40.00
30.00
20.00
10.00
0.00
0
100
200
300
400
500
600


Input file s349f_2.txt
% coverage
# of test vectors
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0
100
200
300
400
500
600
700


Input file s298f_2.txt
120.00
100.00
80.00
60.00
40.00
20.00
0.00
% coverage
0
100
200
300
400
500
600
700
# of test vectors

## Phase 3: Test Generation

For each circuit layout, tests were generated for five faults. For each circuit, I have tabulated the test vectors the SMART-G program generated for each of the five faults. I have also tabulated the line number for the primary output/s where the fault is detected. For the actual output.txt file generated for these faults please see Appendix A.

**Circuit layout file: s27.txt**

| FAULT | TEST VECTOR (1 2 3 4 6 8 10) | Detected at Primary Output |
|---|---|---|
| 16 s-a-0 | x 0 x 1 0 x 0 | 5, 9 |
| 6 s-a-0 | x 0 x 1 1 x 0 | 5, 9 |
| 7 s-a-1 | 0 x x x x x x | 7 |
| 15 s-a-0 | x 0 0 x x x 0 | 11 |
| 17 s-a-1 | 1 0 x 0 0 x 0 | 5, 7, 9 |

**Circuit layout file: s298f_2.txt**

| FAULT | TEST VECTOR (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ) | Detected at Primary Output |
|---|---|---|
| 70 s-a-1 | 0 1 x 1 x x x x x x x x x x 0 x x | 19 |
| 32 s-a-0 | x x x x x x x x 1 x x x x x x x x | 32 |
| 56 s-a-1 | 0 x 0 1 1 x x x x x 1 x x x x x x | 28 |
| 101 s-a-0 | x 1 0 1 0 1 x x x x x x 0 x x x x | 24 |
| 119 s-a-0 | x 1 0 0 1 0 x x x x x x x x x x x | 28 |

**Circuit layout file: s344f_2.txt**

| FAULT | TEST VECTOR (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24) | Detected at Primary Output |
|---|---|---|
| 166 s-a-1 | 0 1 x 0 0 x x x x x 0 0 1 x x 0 x x x x x x x x | 26 |
| 106 s-a-1 | 1 0 1 x x x x 1 x x x x x x x x x x x 0 x x x x | 32 |
| 107 s-a-0 | 0 0 0 x x x x x x x x x x x x x x x x 1 x x x x | 32 |
| 73 s-a-0 | 0 0 0 x x x x x x x x 1 x x x x x x x x x x x 0 | 36 |
| 188 s-a-0 | 1 1 x x x x x x x x x x x x x x x x x x x x x x | 50 |

**Circuit layout file: s349f_2.txt**

| FAULT | TEST VECTOR (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24) | Detected at Primary Output |
|---|---|---|
| 25 s-a-0 | 1 0 x x x x x x x x x x x x x 0 x x x x x x x x | 25 |
| 169 s-a-1 | 0 1 x x x x x x x 1 x x x x x x x x x x x x x x | 35 |
| 180 s-a-0 | 0 1 1 x x x x x x x x x x x x 0 x x x x x x x x | 25 |
| 177 s-a-1 | x 0 0 x x x x x x x x x x x x 0 x x x x x x x x | 26 |
| 73 s-a-0 | 0 0 1 x x x x x x x x x x x x x x x x x x x x x | 42 |

# 1. Algorithm Tree for 179 s-a-1 in s349f_2.txt

Objective: Returning line 179 and value 0 for gate 108

Backtrace: Returning line 3 and value 0 for gate 108

Imply: Found line 3 and assigned the value 0 to it.

Call to PODEM() returns failure

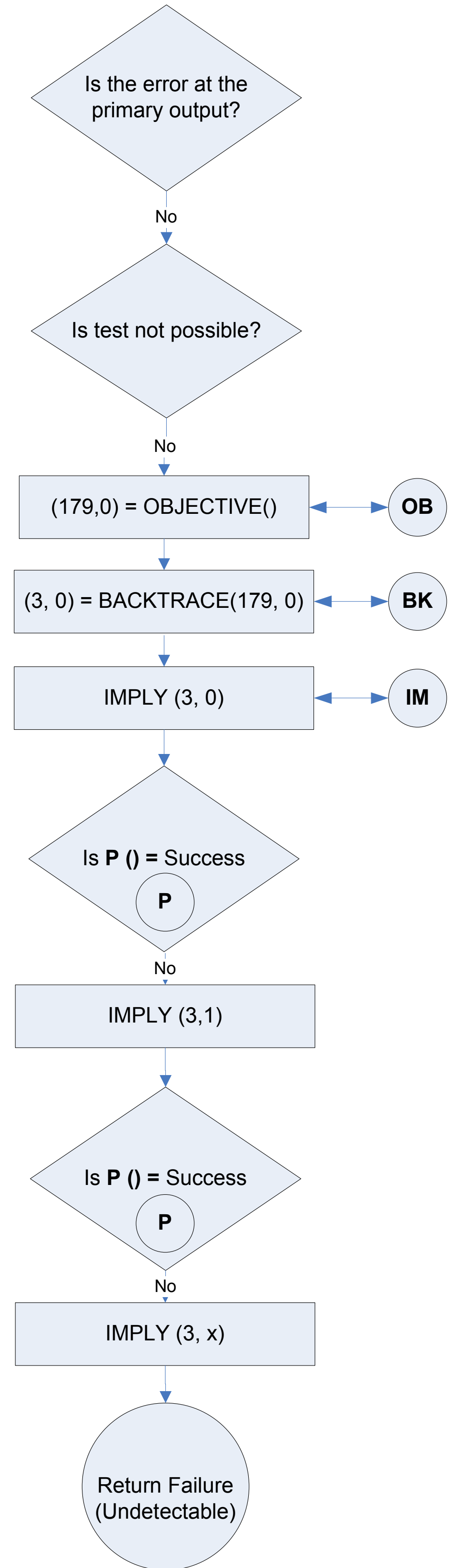
Imply: Found line 3 and assigned the value 1 to it.

Call to PODEM() returns failure

Imply: Found line 3 and assigned the value 2 to it.

Call to PODEM() returns failure

The fault is undetectable



## 2. Algorithm Tree for 166 s-a-0 in s344f_2.txt

Objective: Returning line 166 and value 1 for gate 98

Backtrace: Returning line 11 and value 0 for gate 134

Imply: Found line 11 and assigned the value 0 to it.

Objective: Returning line 166 and value 1 for gate 98

Backtrace: Returning line 12 and value 1 for gate 135

Imply: Found line 12 and assigned the value 1 to it.

Objective: Returning line 166 and value 1 for gate 98

Backtrace: Returning line 5 and value 0 for gate 69

Imply: Found line 5 and assigned the value 0 to it.

Objective: Returning line 166 and value 1 for gate 98

Backtrace: Returning line 13 and value 1 for gate 133

Imply: Found line 13 and assigned the value 1 to it.

Found a D-frontier at gate with output line 135

Objective: Returning line 167 and value 1 for gate 98

Backtrace: Returning line 4 and value 0 for gate 154

Imply: Found line 4 and assigned the value 0 to it.

Found a D-frontier at gate with output line 142

Objective: Returning line 92 and value 1 for gate 105

Backtrace: Returning line 1 and value 0 for gate 126

Imply: Found line 1 and assigned the value 0 to it.

Found a D-frontier at gate with output line 142

Objective: Returning line 92 and value 1 for gate 105

Backtrace: Returning line 2 and value 1 for gate 121

Imply: Found line 2 and assigned the value 1 to it.

Found a D-frontier at gate with output line 29

Objective: Returning line 16 and value 0 for gate 5

Backtrace: Returning line 16 and value 0 for gate 0

Imply: Found line 16 and assigned the value 0 to it.

The fault is detected

Is the error at the primary output?

No

Is test not possible?

No

(166,1) = OBJECTIVE()

(11,0) = BACKTRACE(166,1)

IMPLY (11, 0)

PODEM ()

No

(166,1) = OBJECTIVE()

(12,1) = BACKTRACE(166,1)

IMPLY (12, 1)

PODEM ()

(166,1) = OBJECTIVE()

A

A

(5,0) = BACKTRACE(166,1)

IMPLY (5, 0)

PODEM ()

(166,1) = OBJECTIVE()

(13,1) = BACKTRACE(166,1)

IMPLY (13,1) — D-frontier at gate with output 135

PODEM ()

(167,1) = OBJECTIVE()

(4, 0) = BACKTRACE(167,1)

IMPLY (4,0) — D-frontier at gate with output 142

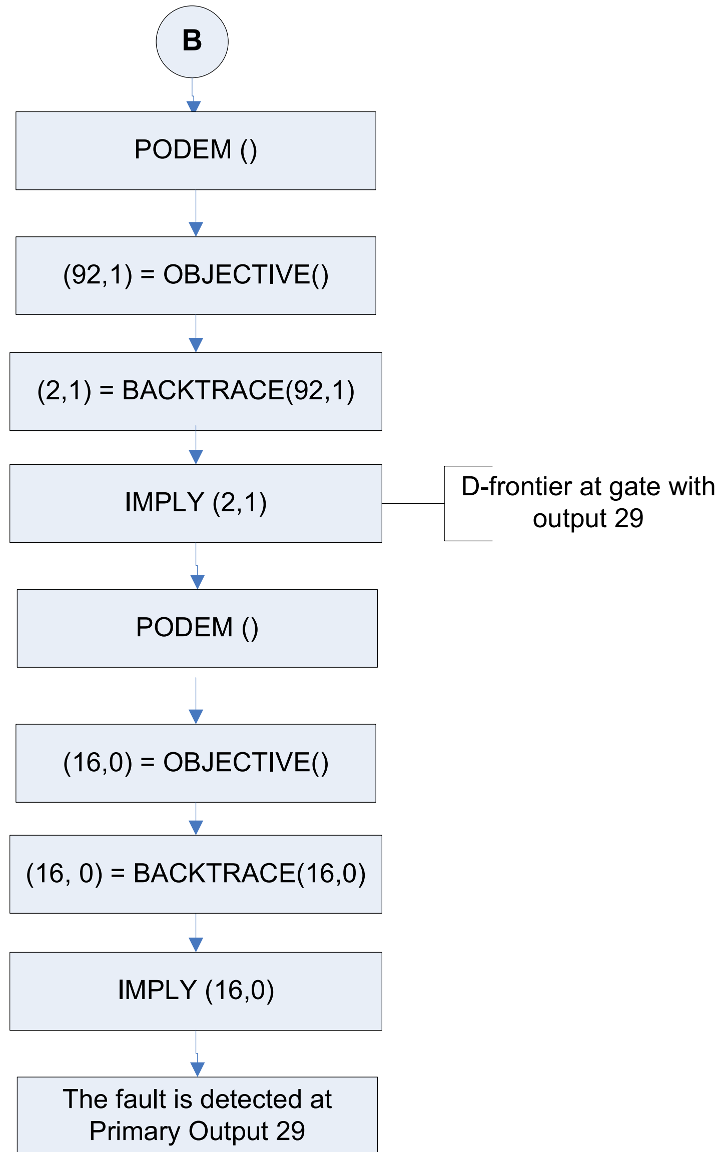
PODEM ()

(92,1) = OBJECTIVE()

(1, 0) = BACKTRACE(92,1)

IMPLY (1,0) — D-frontier at gate with output 142

B


B
PODEM ()
(92,1) = OBJECTIVE()
(2,1) = BACKTRACE(92,1)
IMPLY (2,1)
D-frontier at gate with output 29
PODEM ()
(16,0) = OBJECTIVE()
(16, 0) = BACKTRACE(16,0)
IMPLY (16,0)
The fault is detected at Primary Output 29

## B. TEST VALIDATION METHODOLOGY

SMART-G has an inbuilt error-checking system which makes sure that the PODEM algorithm has generated the correct test vector. The *checkPODEM()* function makes use of the fault simulator from Phase 2. First *checkPODEM()* resets all the circuit inputs and outputs. Next, it simulates one fault at a time and sensitizes the circuit's primary input with values from the test vector generated by the PODEM algorithm. It then propagates these primary input values throughout the circuit and computes the values of the primary outputs. Now, it compares the primary output values it generates to the primary output values of the circuit at the end of the PODEM algorithm run. If the PODEM algorithm says that the fault was detected, *checkPODEM()* also checks to see that the D or Dbar has actually reached a primary output. If these two criteria are satisfied, we know that the PODEM algorithm has run successfully.

## C. EXTENSIONS

For the second phase of the project, the fault simulator, I embedded a function that made the program capable of handling fanouts (as highlighted above). The program was able to read the circuit layout and determine where the fanouts were present. Each fanout was considered as a different line, making it possible to simulate faults separately at the fanout outputs and corresponding inputs.

For the third phase of the project, in the logic simulator in the *imply()* function I embedded some intelligence, whereby gate outputs were not calculated in a brute force fashion where the input pins are read and then the output pins are calculated. Instead, I made the program traverse the truth table to see in what cases a particular output would take a particular value. This increased the efficiency of the program and decreased execution time. This functionality along with the efficiency embedded in the PODEM algorithm makes the program run within the blink of an eye.

# V. Conclusion

From the results we see that SMART-G is thus able to make the best of the efficiency embedded in the PODEM algorithm, generating test vectors for complex circuits in rapidity. The in-built self-checking system makes sure that the test vectors generated are actually able to detect circuit faults.

The SMART-G project not only provided me with a thorough understanding of the PODEM algorithm, but also allowed me to see a real-world example of how test-generation is performed. Learning on the theoretical side through exams and assignments are important, yet without such a project, I think the class would be incomplete. The three-phased approach was a good idea since it ensured that we kept working part-by-part on the project, so that the load of an entire project at the end of the semester was avoided. I think this project should definitely be part of the curriculum for future classes too.

# Appendix A: Results from output.txt

## 1. s27.txt

---ooo--- Welcome to the SMART-G PODEM Test Generation Tool ---ooo---
by Uday Gurnani

---

Retrieving circuit structure from s27.txt.

Faults defined in faults.txt are:
Circuit line 16 stuck at 0
Circuit line 6 stuck at 0
Circuit line 7 stuck at 1
Circuit line 15 stuck at 0
Circuit line 17 stuck at 1

---

The test vector that detects the fault at 16 is:
2 0 2 1 0 2 0

Fault 16 stuck at 0 is detected at Primary Output 9

Fault 16 stuck at 0 is detected at Primary Output 5

PODEM algorithm has been verified.

---

The test vector that detects the fault at 6 is:
2 0 2 1 1 2 0

Fault 6 stuck at 0 is detected at Primary Output 9

Fault 6 stuck at 0 is detected at Primary Output 5

PODEM algorithm has been verified.

---

The test vector that detects the fault at 7 is:
0 2 2 2 2 2 2

Fault 7 stuck at 1 is detected at Primary Output 7

PODEM algorithm has been verified.

---

The test vector that detects the fault at 15 is:
2 0 0 2 2 2 0

Fault 15 stuck at 0 is detected at Primary Output 11

PODEM algorithm has been verified.

---

The test vector that detects the fault at 17 is:
1 0 2 0 0 2 0

Fault 17 stuck at 1 is detected at Primary Output 7

Fault 17 stuck at 1 is detected at Primary Output 9

Fault 17 stuck at 1 is detected at Primary Output 5

PODEM algorithm has been verified.

---

Thank you for using the SMART-G Test Generation Tool
Copyright 2007, Uday Gurnani, Georgia Tech

---

## 2. s298f_2.txt

---

---ooo--- Welcome to the SMART-G PODEM Test Generation Tool ---ooo---
by Uday Gurnani

---

Retrieving circuit structure from s298f_2.txt.

Faults defined in faults.txt are:
- Circuit line 70 stuck at 1
- Circuit line 32 stuck at 0
- Circuit line 56 stuck at 1
- Circuit line 101 stuck at 0
- Circuit line 119 stuck at 0

---

The test vector that detects the fault at 70 is:
0 1 2 1 2 2 2 2 2 2 2 2 2 2 2 0 2 2

Fault 70 stuck at 1 is detected at Primary Output 19

PODEM algorithm has been verified.

---

The test vector that detects the fault at 32 is:
2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2

Fault 32 stuck at 0 is detected at Primary Output 32

PODEM algorithm has been verified.

---

The test vector that detects the fault at 56 is:
0 2 0 1 1 2 2 2 2 2 2 1 2 2 2 2 2 2

Fault 56 stuck at 1 is detected at Primary Output 28

PODEM algorithm has been verified.

---

The test vector that detects the fault at 101 is:
2 1 0 1 0 1 2 2 2 2 2 2 2 0 2 2 2 2

Fault 101 stuck at 0 is detected at Primary Output 24

PODEM algorithm has been verified.

---

The test vector that detects the fault at 119 is:
2 1 0 0 1 0 2 2 2 2 2 2 2 2 2 2 2 2

Fault 119 stuck at 0 is detected at Primary Output 28

PODEM algorithm has been verified.

---

Thank you for using the SMART-G Test Generation Tool
Copyright 2007, Uday Gurnani, Georgia Tech

---

## 3. s344f_2.txt

---

---ooo--- Welcome to the SMART-G PODEM Test Generation Tool ---ooo---
by Uday Gurnani

---

Retrieving circuit structure from s344f_2.txt.

Faults defined in faults.txt are:
Circuit line 166 stuck at 1
Circuit line 106 stuck at 1
Circuit line 107 stuck at 0
Circuit line 73 stuck at 0
Circuit line 188 stuck at 0

---

The test vector that detects the fault at 166 is:
0 1 2 0 0 2 2 2 2 2 2 0 0 1 2 2 0 2 2 2 2 2 2 2 2

Fault 166 stuck at 1 is detected at Primary Output 26

PODEM algorithm has been verified.

---

The test vector that detects the fault at 106 is:
1 0 1 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2

Fault 106 stuck at 1 is detected at Primary Output 32

PODEM algorithm has been verified.

---

The test vector that detects the fault at 107 is:
0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2

Fault 107 stuck at 0 is detected at Primary Output 32

PODEM algorithm has been verified.

---

The test vector that detects the fault at 73 is:
0 0 0 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 0

Fault 73 stuck at 0 is detected at Primary Output 36

PODEM algorithm has been verified.

---

The test vector that detects the fault at 188 is:
1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

Fault 188 stuck at 0 is detected at Primary Output 50

PODEM algorithm has been verified.

---

Thank you for using the SMART-G Test Generation Tool
Copyright 2007, Uday Gurnani, Georgia Tech

---

## 4. s349f_2.txt

---

---ooo--- Welcome to the SMART-G PODEM Test Generation Tool ---ooo---
by Uday Gurnani

---

Retrieving circuit structure from s349f_2.txt.

Faults defined in faults.txt are:
  Circuit line 25 stuck at 0
  Circuit line 169 stuck at 1
  Circuit line 180 stuck at 0
  Circuit line 177 stuck at 1
  Circuit line 73 stuck at 0

---

The test vector that detects the fault at 25 is:
1 0 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2

Fault 25 stuck at 0 is detected at Primary Output 25

PODEM algorithm has been verified.

---

The test vector that detects the fault at 169 is:
0 1 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2

Fault 169 stuck at 1 is detected at Primary Output 35

PODEM algorithm has been verified.

---

The test vector that detects the fault at 180 is:
0 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2

Fault 180 stuck at 0 is detected at Primary Output 25

PODEM algorithm has been verified.

---

The test vector that detects the fault at 177 is:
2 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2

Fault 177 stuck at 1 is detected at Primary Output 26

PODEM algorithm has been verified.

---

The test vector that detects the fault at 73 is:
0 0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

Fault 73 stuck at 0 is detected at Primary Output 42

PODEM algorithm has been verified.

---

Thank you for using the SMART-G Test Generation Tool
Copyright 2007, Uday Gurnani, Georgia Tech

---