

EE 709 - Testing and Verification  
Combinational Test Generation  
Project Report

Arun. C (10307938)  
Siddharth. Mohta (10307937)

Dept of Elect Engg.  
IIT-Bombay

May 2, 2012

## Project Statement

Objective of this project is to come up with a test generation program that can generate test patterns for combinational circuits.

1. Name : atpgGenerator
2. Input : Verilog Structural Netlist.
3. Output : Test Pattern file. Runtime info dumped to console.
4. Language : C++
5. Interface : Command Line.
6. OS/Platform : Linux (Any variant)

## 1 Approach and Algorithm

Basic D Algorithm is used for test generation. Only verilog built in primitives such as and, or, nand, not, nor, buf etc are allowed in the input file. The tool parses the netlist, build internal data structures, runs the algorithms and prints the resulting fault patterns to an output file. Detailed run time information is output to the console which can be captured using the unix tee command.

A “line” in this work stands for a fault line. A line is uniquely determined as component\_name+\_net\_name.

For eg: The text-line, or\_gate1 or(out1, in1, in2); in verilog netlist will be parsed as component or\_gate1, and the fault-lines or\_gate1\_out1, or\_gate1\_in1 and or\_gate1\_in2.

Separate test patterns (SA-0 and SA-1) are generated for each of these lines independently. Thus this scheme uniquely identifies the different fault lines even in case of multiple fanouts from a component. If there is only one to one connection between two components, like

```
inv1 not(w1, w2);  
inv2 not(w2, w3);
```

this also will be parsed to 4 fault-lines - inv1\_w1, inv1\_w2, inv2\_w1, inv2\_w2. Here inv1\_w2 and inv2\_w2 is the same fault line. ie the pattern generated for both the cases may be the same. However in our current work, tool treats these are different fault lines itself and generates patterns independently. This is redundancy and we have not optimized this aspect. It will be taken up as future enhancement.

The basic algorithm used is as explained as follows: <sup>1</sup>  
 There are two main functions, Justify() and Propagate(). Justify() justifies a given line with a given value.

```

Justify( l, val)
begin
  set l to val
  if l is a PI then return
  /* l is a gate output */
  c = controlling value of l
  i = inversion of l
  inval = val ^ i
  if (inval = c')
    then for every input j of l
      Justify (j, inval)
  else
    select one input ( j ) of l
    Justify (j, inval)
end
  
```

Propagate() propagates the given value to any of the PO.

```

Propagate (l, err)
/* err is D or D' */
begin
  set l to err
  if l is PO then RETURN
  k = fanout of l
  c = controlling value of k
  i = inversion of k
  for every input of j of k other than l
    Justify ( j, c' )
  Propagate ( k, err ^ i )
end
  
```

Now the algorithm traverses in the data structure as follows.

```

GenerateTP ( )
begin
  set all values to x
  Justify (l, v)
  if (v = 0) then Propagate (l, D)
  else Propagate (l, D')
end
  
```

---

<sup>1</sup>Adapted from the lecture notes of EE-709 by Prof. Virender Singh

## 2 Implementation aspects

The software is planned to 3 different sections, front end (verilog parser), data-structures and the atpg algorithm itself.

1. Verilog Parser (Front End) - Parses the verilog input file using the regular expressions available in boost library. (class VerilogLineParser). Reads the file line by line and builds the necessary datastructures.
2. Datastructure Our datastructure consists of nodes, lists and maps. Explicit usage of graphs is not done. An associative-table graph is accomplished using a set of maps (hash arrays). Each component is stored in the class VerilogNode. It holds the following information - component\_name, nodeID, level, type (AND/OR/NOT/INPUT/OUTPUT etc), cv (controlling value), inv\_parity (inversion value), num\_inputs, num\_outputs, input\_list (list of nets connected to input pins), output\_list (list of nets connected to output pins), input\_list\_value (maps input pin to value (1/0/X/D/Dbar)), output\_list\_value (maps output pin to value(1/0/X/D/Dbar).)

Each fault line is another class(class LineType) having the members - name, value, lineDirection, lineAttribute (PI/PO/INTERNAL\_WIRE), isVisited attribute.

3. Following associative tables (maps or hash arrays) are used to traverse through the graph. These are declared globally.
  - linemap : line to node.
  - line\_to\_faninline\_map : line to driver line.
  - line\_to\_fanoutline\_map : line to fanoutlines. (driven lines).
  - line\_to\_inputline\_map : o/p line to i/p lines of the component.
  - line\_to\_outputline\_map : i/p line to o/p lines of the component.
4. Other helper functions<sup>2</sup> - PrintPIPatterns() [prints the patterns at PI], EqualizeFaninFanoutNets() [equalizes the values of the fanout nets to the fanin nets], UpdateMaps() [Updates the associative-tables], UpdateLineList() [Updates the list of fault-lines if needed to the current working line] etc.

---

<sup>2</sup>Other functions used in the program are not specified. The meaning and usage can be easily figured out by looking at the code.

### 3 Compilation, Usage and Dependencies

Compile using shell script, compile.bash <sup>3</sup>

The program has a dependency on boost libraries <sup>4</sup>, for regular expression parsing. Boost libraries are going to be a part of next standard of C++ (C++-11)

The compiler used for this work is GNU-C++ compiler (g++) - version 4.5.2. OS is Linux (Ubuntu-11.10). However it should work with any other standard C++ compiler and OS, as there is no OS/compiler dependencies.

Usage:

```
unix-prompt:-$ ./atpgGenerator inputfile.v | tee inputfile.log
```

Result is inputfile.v\_fault.lst - Gives the list of fault patterns.

### 4 Programming Testing

The program has been tested for the following 4 circuits. Circuits are shown in Appendix-1 of this report. For each circuit verilog description is provided and manually checked the output patterns generated. Circuit 4 is a reconvergent circuit. Program is able to come up with test patterns for this circuit successfully.

### 5 Future Work

We need to apply the program to more circuits and figure out the bugs and potential miss-outs and errors.

Need to generate test patterns for ISCAS circuits. As such we have not attempted in the current phase because we dont have a golden response to compare with. ISCAS c880a ckt has 60 primary inputs and is impossible to check the response manually. We need a fault simulator to verify the test patterns generated.

Entire code has to be restructured and more powerful features and classes need to be used. Currently there are lot of workarounds to reach this stage.

Need to plan properly and rewrite the program.

Try alternate and better algorithms, compare efficiency, performance etc.

\*\*\*\*\*

---

<sup>3</sup>There is no need of an elaborate make program. Plain C++ compilation will do

<sup>4</sup>[www.boost.org](http://www.boost.org)

## 6 Appendix 1 - Results

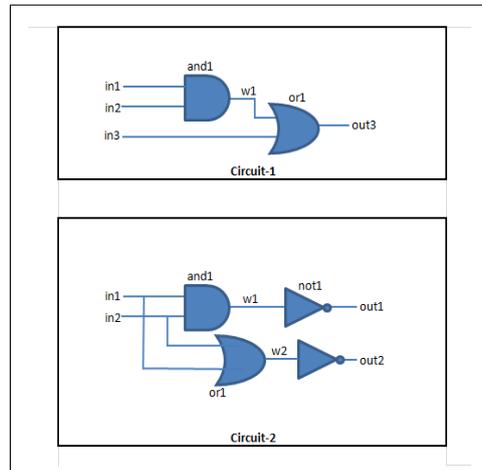


Figure 1: Circuits 1&2

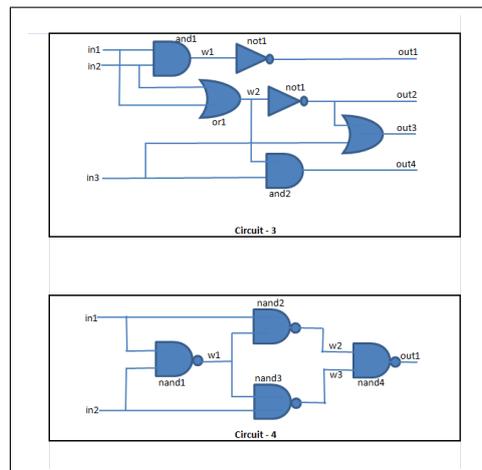


Figure 2: Circuits 3&4

The sample circuits 2, 4 are shown. (verilog code and output pattern file)<sup>5</sup>

```
// Sample ckt-2.
module combo(out1, out2, in1, in2);
    output out1;
    output out2;
    input  in1;
    input  in2;
    wire   w1, w2;

    and and1(w1, in1, in2);
    not not1(out1, w1);
    or  or1(w2, in1, in2);
    not not2(out2, w2);
endmodule // combo
```

Result:

#	FaultLine	Type	input_in2	input_in1
1	not2_w2	SA-1	0	0
2	not2_w2	SA-0	X	1
3	not2_out2	SA-1	1	X
4	not2_out2	SA-0	0	0
5	or1_in2	SA-1	0	0
6	or1_in2	SA-0	1	0
7	or1_in1	SA-1	0	0
8	or1_in1	SA-0	0	1
9	or1_w2	SA-1	0	0
10	or1_w2	SA-0	X	1
11	not1_w1	SA-1	0	X

<sup>5</sup>Repetitions in the fault lines are there. Currently ignores as explained in sec-1, Approach and Algo

12		not1_w1	SA-0		1		1	
-----								
13		not1_out1	SA-1		1		1	
-----								
14		not1_out1	SA-0		X		0	
-----								
15		and1_in2	SA-1		0		1	
-----								
16		and1_in2	SA-0		1		1	
-----								
17		and1_in1	SA-1		1		0	
-----								
18		and1_in1	SA-0		1		1	
-----								
19		and1_w1	SA-1		0		X	
-----								
20		and1_w1	SA-0		1		1	

---

```
// Sample ckt-4 - Reconvergent ckt.
module combo(out1, in1, in2);
    output out1;
    input in1;
    input in2;
    wire w1, w2, w3;

    nand nand1 (w1, in1, in2);
    nand nand2 (w2, in1, w1);
    nand nand3 (w3, w1, in2);
    nand nand4 (out1, w2, w3);
endmodule // combo
```

Result:

#		FaultLine	Type		input_in2		input_in1	
-----								
1		nand4_w3	SA-1		1		0	
-----								
2		nand4_w3	SA-0		1		1	
-----								
3		nand4_w2	SA-1		0		1	
-----								
4		nand4_w2	SA-0		0		0	
-----								

5		nand4_out1		SA-1		0		0	
6		nand4_out1		SA-0		0		1	
7		nand3_in2		SA-1		0		0	
8		nand3_in2		SA-0		1		0	
9		nand3_w1		SA-1		1		1	
10		nand3_w1		SA-0		1		0	
11		nand3_w3		SA-1		1		0	
12		nand3_w3		SA-0		1		1	
13		nand2_w1		SA-1		1		1	
14		nand2_w1		SA-0		0		1	
15		nand2_in1		SA-1		0		0	
16		nand2_in1		SA-0		0		1	
17		nand2_w2		SA-1		0		1	
18		nand2_w2		SA-0		0		0	
19		nand1_in2		SA-1		0		1	
20		nand1_in2		SA-0		1		1	
21		nand1_in1		SA-1		1		0	
22		nand1_in1		SA-0		1		1	
23		nand1_w1		SA-1		1		1	
24		nand1_w1		SA-0		1		0	