

Combinational ATPG Using The Four-state Logic

Kalyana R Kantipudi

Dept. of Electrical & Computer Engineering

Auburn University

VLSI Testing
Spring'05

Table of Contents:

- Abstract.....1
- Introduction.....1
- Main Idea.....2
- Steps
 - Parsing and Levelizing.....3
 - Tokenizing.....4
 - Generating active combinations.....5
 - Testing for fault activation.....5
 - Fault effect propagation.....5
 - Line justification.....5
- Example.....6
- Lessons Learned.....7
- Conclusions and future improvements.....7
- Results.....8

Combinational ATPG Algorithm Using the Four-state logic

Abstract: In this new approach, the input vector will be fully specified and the vector will be modified to detect the specified fault. A library is created by tokenizing the entire circuit. It contains a list of essential inputs required to activate each fault site. A default input pattern will be given and the circuit is tested whether the fault is activated or not. If the fault is not activated, a combination generator generates an activating combination for the fault at the fault-site, which is fed to the circuit and the output verified. If the fault effect is propagated to a primary output and the line justifications do not have to change any of the essential inputs, then a test vector is found for the fault. If not propagated, the ATPG gets one more active combination and tries to propagate it. If all active combinations are tested and the fault effect cannot propagate to the output, then the fault can be considered redundant and so untestable.

Introduction: Every circuit in an IC has to be tested. Whether it may be of Medium Scale Integration or of Ultra Large Scale Integration, it has to be fully tested for all the faults in it. But to test a circuit with 'n' inputs, it is not a feasible idea to apply all the 2^n input vectors. So an ATPG is needed to generate a list of input vectors or patterns which are enough to fully test the circuit. The aim of an ATPG should be to generate the smallest set of input vectors as quickly as possible. In an ATPG, faults are modeled as stuck-at faults(s-a-1 or s-a-0). An efficient ATPG algorithm makes use of the functional and structural redundancies in the circuit to collapse all the faults into a small group of faults. So that it becomes easy to generate test vectors for the circuit.

Various Algorithms are designed to test stuck-at faults. Some uses Binary Decision Diagrams and others use implication graphs. The D-Algorithm proposed by Roth uses the topological search method. Later, Göel improved it in his classic algorithm-PODEM (Path Oriented DEcision Making), which employs implicit enumeration technique. The algorithm is improved in FAN and TOPS, which made use of various concepts like Immediate Implications, unique sensitization, headlines, multiple-backtrace etc., To reduce the backtrace conflicts, some latest combinational circuit ATPG algorithms use adaptive backtrace approach. To cope with the large size of the VLSI circuits, they dynamically partition the circuit. The Equivalent STate hashing(EST) algorithm makes the pattern generation faster by reducing the search space. These algorithms tend to optimize the generation of test patterns by reducing the test pattern generation time without compromising with the fault coverage.

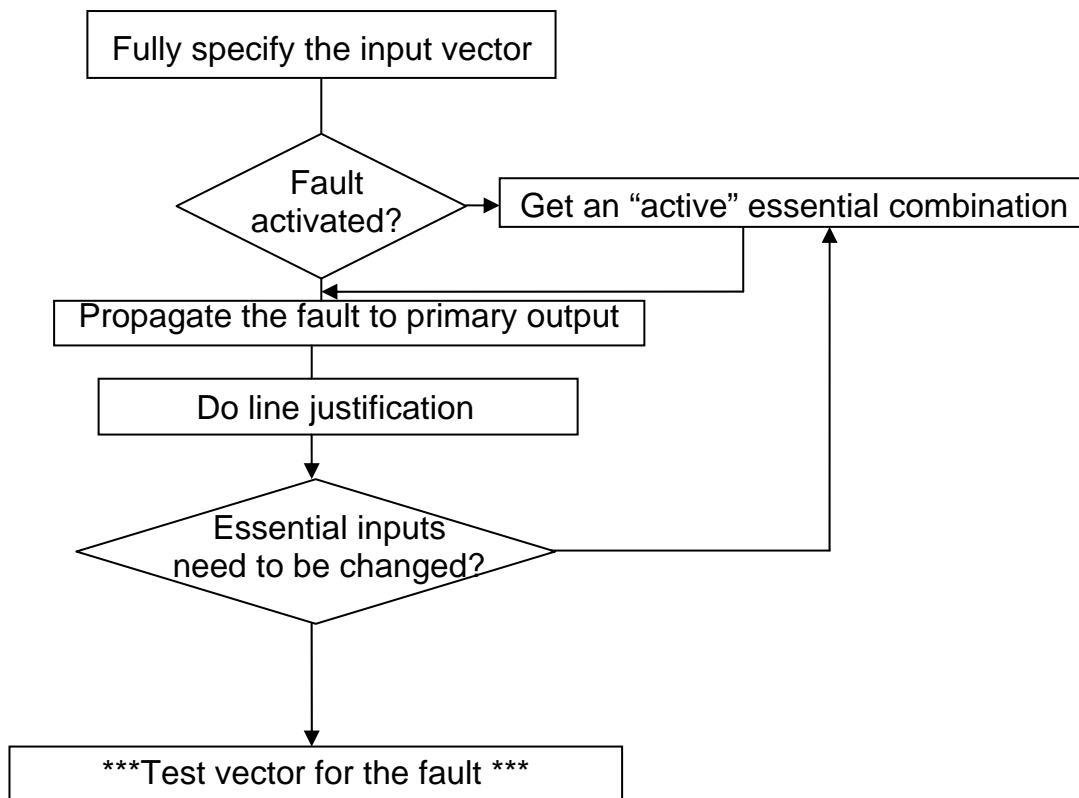
They tend to ignore one of the basic aims of an ATPG, producing a minimum set of test vectors. Every algorithm uses backtracing to activate the stuck-at fault and all backtracing approaches make use of the testability measures (controllability and observability) of the circuit to get to the primary inputs. Here there can be a problem; there will be some independent faults which may have some concurrent test vectors. The backtracing approach produces only one test vector for each fault. If the test vector which is in common to those independent faults can be found, then the number of test vectors can be greatly reduce.

The idea is not dealt in this project. The present ATPG generates a test vector for a specified fault (if improved, can generate a set of test vectors for the fault). In this new approach, the input vector will be fully specified and the vector will be modified to detect the fault at the fault site. A library will be created by tokenizing the entire circuit. The library contains a list of essential inputs required to activate each fault site. When ever a test vector is required for a fault, a default input pattern of all ones or all zeros will be given to the circuit. In fact, giving non-controlling inputs will be more advantageous that giving input vectors randomly*. By giving non-controlling values as primary inputs, one can be sure that by changing one input to the primary gate the output of it will change. Once a primary input vector is given, it is tested

whether the fault is activated or not. If the fault is not activated, a combination generator generates an activating combination for the fault at the fault-site. Now the modified input vector is fed to the circuit and the output is verified. If the activated fault is propagated to a primary output and the line justifications do not have to change any of the essential inputs, then a test vector for the fault is found. If the output is not propagated to a primary output, the ATPG gets one more active combination and tries to propagate it. If all active combinations are tested and the fault effect cannot be propagated to the output, then the fault can be considered redundant or untestable. The approach can be further extended to generate a list of test vectors for a particular fault.

Main Idea:

The main idea is as specified as above, the fully specified input vector is given to the circuit and it is checked whether the fault is activated or not. If the fault is not activated, then an active combination is given and it is checked whether the fault propagation causes the essential inputs to change. If so, another active combination is fed to the circuit and the process is repeated until the fault is propagated to the output or until all active combinations are tried. If the fault effect cannot be propagated to the primary outputs, then the fault is redundant and it is untestable.



* I found this interesting idea in John Sunwoo's project presentation.

Steps:

- Parsing the circuit.
- Levelizing the circuit.(Calculating the testability measures of each node)
- Tokenizing the circuit.
- Generating active combinations for the fault.
- Testing if the fault is activated or not.
- Fault effect propagation.
- Line justification.

Step 1 & 2:

Parsing and levelizing the circuit:

The circuit is, parsed and levelized using Hitec. The circuit is fed to the Hitec test generator, just to parse and levelize the circuit and the output of it is read into the c-program.

The input file:

```
INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)
```

```
OUTPUT(22)
OUTPUT(23)
```

```
10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

The levelized file:

```
14
10
1 1 0 0 1 6 4 ; 0 0
2 1 0 0 1 8 5 ; 0 0
3 1 0 0 2 7 6 4 ; 0 0
4 1 0 0 1 7 4 ; 0 0
5 1 0 0 1 9 6 ; 0 0
6 7 5 2 1 3 1 3 1 1 0 3 ; 1 1
7 7 5 2 3 4 3 4 2 8 9 3 ; 2 2
8 7 1 0 2 7 2 7 2 2 1 1 1 0 2 ; 4 2
9 7 1 0 2 7 5 7 5 1 1 1 3 ; 3 1
10 7 1 5 2 8 6 8 6 1 1 2 0 ; 4 2
11 7 1 5 2 8 9 8 9 1 1 3 0 ; 4 4
12 2 2 0 1 1 0 1 0 0 0 0 4 2
13 2 2 0 1 1 1 1 1 0 0 0 4 4
```

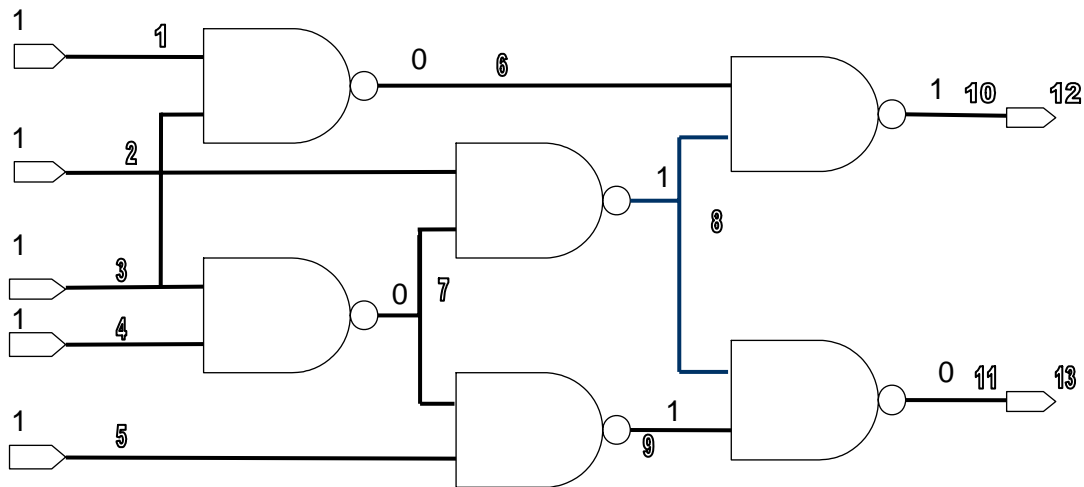
The levelized file contains the entire information of the circuit. It gives the node number, gate type, level of the gate and also a list of all the inputs & successors of each node. It also gives the controlabilities (cc0 & cc1) & observabilities (c0) of each node in the circuit.

Note: 'Hitec' uses the SCOAP testability measurement technique for calculating the testability measures.

Step 3:

Tokenizing the circuit:

The tokenizing algorithm is used to tokenize the circuit. The primary input's node-number is passed through its successor nodes and the value is listed in those nodes. The tokenizing algorithm is used to generate the list of essential inputs of each node. The list is essential while generating active combinations to activate a particular fault.



The Tokenized node list:

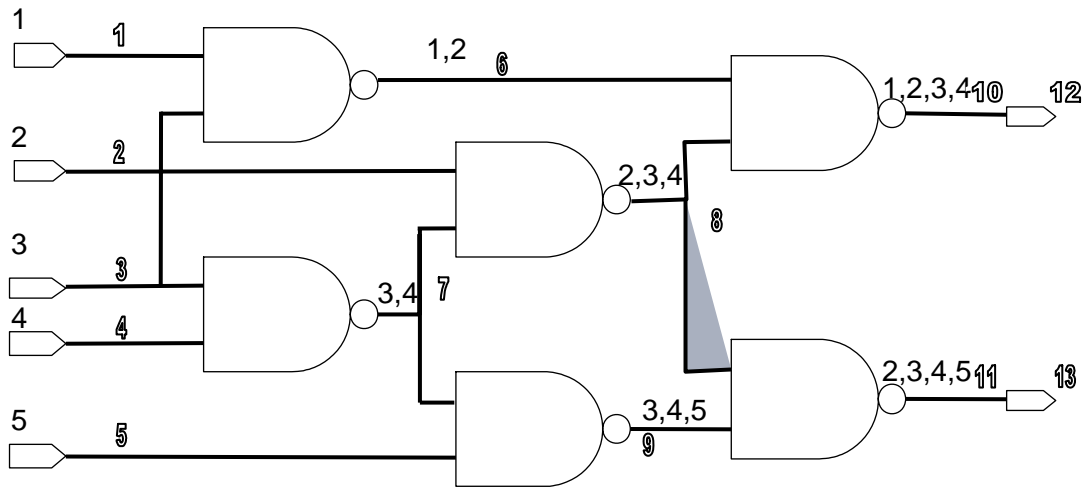
```

1 1
2 2
3 3
4 4
5 5
6 1 2
7 3 4
8 2 3 4
9 3 4 5
10 1 2 3 4
11 2 3 4 5
12 1 2 3 4
13 2 3 4 5

```

For wider circuits, it will be advantageous. But, for deeper circuits, most of the nodes will be having all the tokens(PIs). Then the tokenized node list cannot be of much use while generating active combinations.

Tokenized Circuit:



Step 4:

Generating active combinations:

Using the tokenized node list, active combinations for the required fault are generated. The generated combinations are applied when they are required. This function takes in the node list and the node structure. It then generates and stores the active combinations of essential inputs.

Step 5:

Testing if the fault is activated or not:

When the fault is specified, first an input sequence is given to the circuit. Then the circuit is tested to find if the fault is activated or not. The function is also used while generating active combinations for the specified fault.

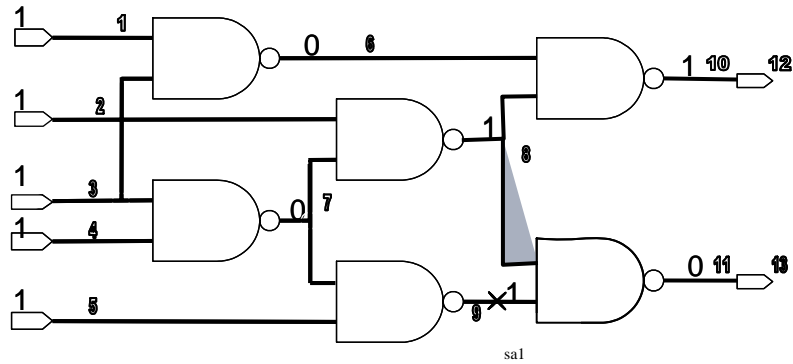
Step 6 & 7:

Fault effect propagation & Line Justification:

The activated fault's output is propagated to a primary output by passing through its successor gates and the assigned successor gate output is justified back at the primary inputs as per the 'gate' justification requirements.

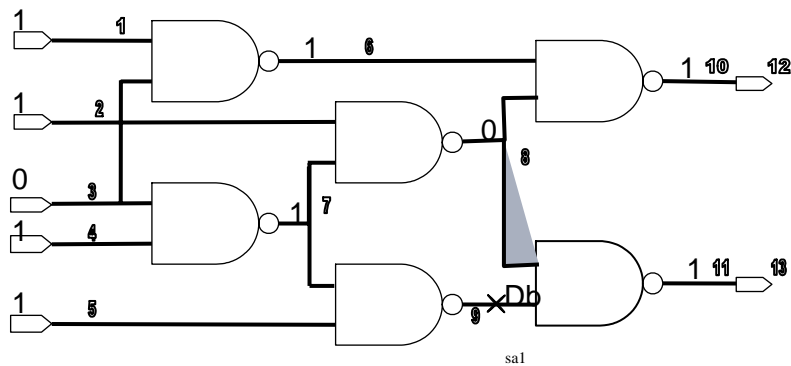
Finding a test vector for the s-a-1 fault on the node 9:

Initial input vector assignment:



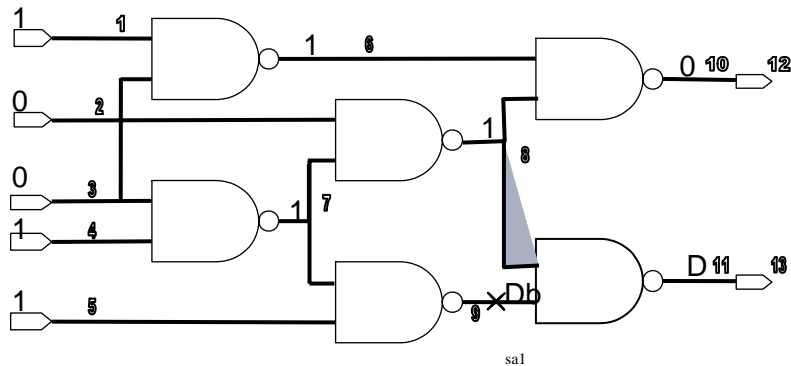
So, a 0 is needed at node 9.

Assign input 3 to 0.



Fault propagated to the output.

The intermediate gate assignments are justified at the primary inputs.



The line justifications do not have to change the essential primary inputs. So, a test vector is found.

Lessons Learned:

It is not easy to design an ATPG in a new way. It needs a comprehensive understanding of the previous work done on them and also needs a clear understanding of all the drawbacks and advantages of each algorithm.

Conclusions and future improvements:

The method specified above will be quite useful for normal circuits. But, for circuits with large number of inputs, backtracing to primary inputs will be a better option. Initially thought that the “essential inputs” idea may be useful in finding all the test vectors for a particular fault. But, it is found that it still cannot generate all the test vectors. This is due to the lack of control over the line justification. We have to take what it gives. So for the conclusion, this method can be used to detect a particular fault and due to its independence, it can be used to generate a list of test vectors for each fault site. If we have control over the line justification, we can generate all the test vectors for a particular fault. These can be used to generate the minimum set of test vectors for any given combinational circuit.

A coarse way of finding the (minimum) set:

- Equivalent collapse the faults
- For each fault (from each equiv. group), generate all (possible) test vectors.
- Combine all the vectors and find the most repeated ones.[“static compaction”]
- List the top ‘n’ and apply them to the circuit. If a particular equivalent set of faults are not detected, include their test vector in the vector list.
- (Try to get the last vectors in the list, out of the list.)

The ATPG generates test vector for the fault specified:

(Fault effect of fsa0 represented as -5 (D) and fault effect of sal as 5 (\overline{D}))

```
cc finprop.c
a.out 9 5 (← Node 9 stuck at 1)
```

The number of nodes are: 13j= 14

```
The value of node 1 of gatetype 1 is 1
The value of node 2 of gatetype 1 is 1
The value of node 3 of gatetype 1 is 1
The value of node 4 of gatetype 1 is 1
The value of node 5 of gatetype 1 is 1
The value of node 6 of gatetype 7 is 0
The value of node 7 of gatetype 7 is 0
The value of node 8 of gatetype 7 is 1
The value of node 9 of gatetype 7 is 1
The value of node 10 of gatetype 7 is 1
The value of node 11 of gatetype 7 is 0
The value of node 12 of gatetype 2 is 1
The value of node 13 of gatetype 2 is 0site = 9 value = 5
```

```
Calling getvec
n before = 13
len = 13
```

} Fault Activated ?

```
The value : 5
2
4
34
45
234
345
234
345
n= 13
1 1
2 2
3 3
4 4
5 5
6 1 2
7 3 4
8 2 3 4
9 3 4 5
10 1 2 3 4
11 2 3 4 5
12 1 2 3 4
13 2 3 4 5

value = 5 and val== 0 n1=3 site=9

num[i] = 0
num[i] = 3
num[i] = 4
nodenum: 3 nodevalue 1
```

} Fault Activation:

Fault Activation:
The essential primary inputs are changed and checked whether the fault is activated or not.
Finally, an active essential combination is generated.

n = 13

n after = 13

```
The value of node 1 of gatetype 1 is 1
The value of node 2 of gatetype 1 is 1
The value of node 3 of gatetype 1 is 0
The value of node 4 of gatetype 1 is 1
The value of node 5 of gatetype 1 is 1
The value of node 6 of gatetype 7 is 1
The value of node 7 of gatetype 7 is 1
The value of node 8 of gatetype 7 is 0
The value of node 9 of gatetype 7 is 5
The value of node 10 of gatetype 7 is 1
The value of node 11 of gatetype 7 is 1
The value of node 12 of gatetype 2 is 1
The value of node 13 of gatetype 2 is 1
```

← Fault activated

```
node in WHILE: 9
node in WHILE: 8
node to be jstified: 8
node which entered justify: 8 tobe jstfied to 1
node[8].incc0[1] = 0 ??
node which entered justify: 2 tobe jstfied to 0
PI which entered justify: 2
PI which entered justify: 2The input 2 is justified to 0
node[11].outc0[0] = 13
```

Line justification:

The node 8 has to be 1.
So, the PI node 2 is justified to 0.

```
n = 13 site = 9
The value of node 1 of gatetype 1 is 1
The value of node 2 of gatetype 1 is 0
The value of node 3 of gatetype 1 is 0
The value of node 4 of gatetype 1 is 1
The value of node 5 of gatetype 1 is 1
The value of node 6 of gatetype 7 is 1
The value of node 7 of gatetype 7 is 1
The value of node 8 of gatetype 7 is 1
The value of node 9 of gatetype 7 is 5
The value of node 10 of gatetype 7 is 0
The value of node 11 of gatetype 7 is -5
The value of node 12 of gatetype 2 is 0
The value of node 13 of gatetype 2 is -5
```

FINAL OUTPUT

The test vector is {1 0 0 1 1 }