1 D-algorithm

Roth (1966) proposed a D-algebra and a deterministic ATPG algorithm. It applys 5-value operation and then implies and justifies the values of nodes in circuits to check whether it exists a test case for one single stuck at fault. In high level, the circuit is reduced to smaller circuit by checking each fault propagation path when error is not at PO(Primary Output) or checking each fault generation path after error propagates to a PO. During this process, the checked circuit path can be ignored so that the circuit size need considering is reduced. Therefore, the bounding condition includes emptying D-frontier when propogating fault, emptying J-frontier after propagating fault and the fail of implyandcheck. Here are some definitions for D algorithm in one specific step. Forward Implication: Results in logic gate inputs that are significantly labeled so that output is uniquely determined. Backward Implication: Unique determination of all gate inputs when the gate output and some of the inputs are given. ImplyAndCheck(): It is a function carries out all implications: forward and backward; it also checks for consistency and inconsistency. Consistency can also represents conflict and means an implied value different from assigned value. The algorithm fails if inconsistency exists. It can be implemented using a push down stack. Considering the time and space complexity, the worst case is considering all the internal node, which is 2^n, where n is the total number of internal nodes.
The following suedo code for D-algorithm is the basic one and this brute-force method could be improved a lot by considering heuristics.

Algorithm 1: D-Algorithm: Recursive Method

```
1  Dalg()                                                    // function name
2  if implyAndCheck() = FAILURE then
3      return FAILURE                                        // base case
4  if error not at PO then
5      if D − Frontier = Φ then
6          return FAILURE                                    // small base case
7      repeat
8          select an untried gate(G) from D-frontier
9          c = controlling value of G
10         assign cbar to every input of G with value x      // assign value
            prepare for forward implication
11
12         if Dalg() = SUCCESS then
13             return SUCCESS
14     until all gates from D-Frontier have been tried
       /* check every path to propagate fault and if there is no path that
          can generate good test pattern, then D algorithm fails          */
15     return FAILURE
   /* error propagated to a PO                                            */
16 if J − Frontier = Φ then
17     return SUCCESS                                        // small base case
18 select a gate(G) from the J-frontier
19 c = controlling value of G
20 repeat
21     begin
22         select an input(j) of G with value x
23         assign c to j        // assign value prepare for backward implication
24
25         if Dalg() = SUCCESS then
26             return SUCCESS
27         assign cbar to j    // assign value prepare for backward implication
28 until all inputs of G are specified
   /* try every possible value for J-frontier and if there is no solutions
      satisfying all J-frontiers, then D algorithm fails              */
29 return FAILURE
```

Here are the illustrations for Recursive D algorithm.

The recursive suedo code for D-algorithm is the concise and the following is the iterative method.

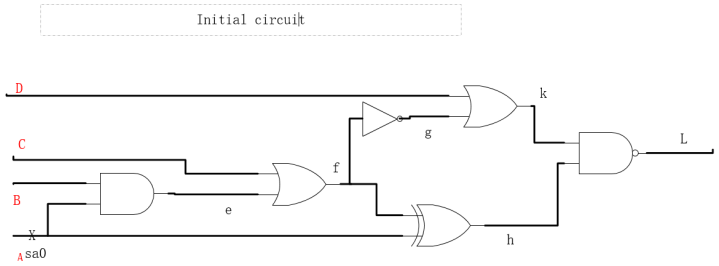

Figure 1: The initial circuit with stuck at fault

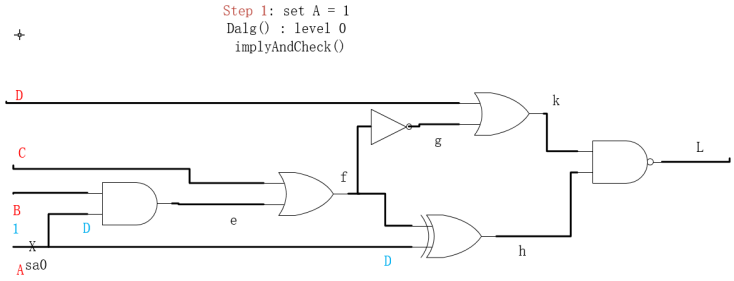

Figure 2: D-algorithm, Recursive, Step 1

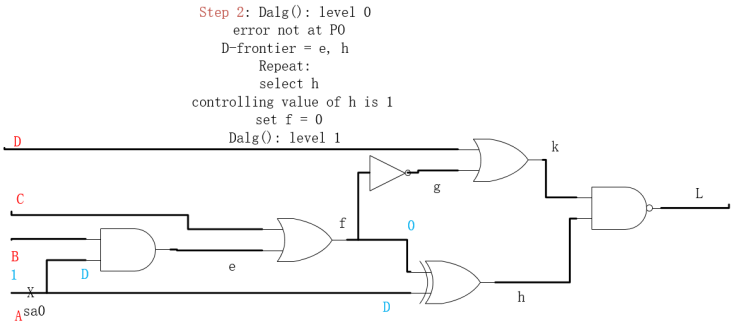

Figure 3: D-algorithm, Recursive, Step 2

Step 3: Dalg(): level 1
implyAndCheck() = true : stack{g, h, f}
error at PO

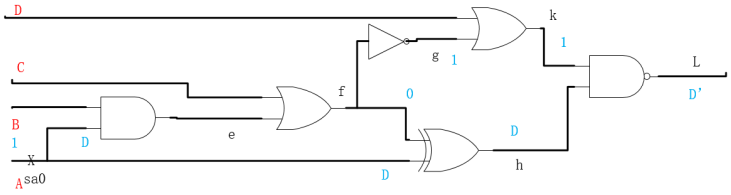

Figure 4: D-algorithm, Recursive, Step 3

Step 4: Dalg(): level 1
J-frontier = f
select f
controlling value of f = 0
Repeat: all inputs{C, e}
select C
set C = 0
Dalg():level 2

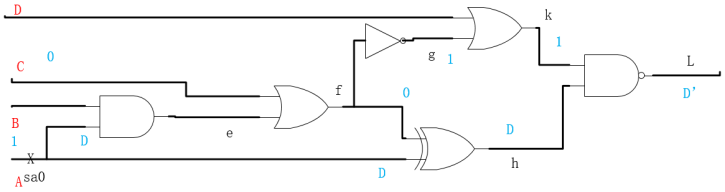

Figure 5: D-algorithm, Recursive, Step 4

Step 5: Dalg(): level 2
implyAndCheck() = true
error at PO
J-frontier = f
select f
controlling value of f = 0
Repeat: all inputs{e}
select e
set e = 0
Dalg():level 3

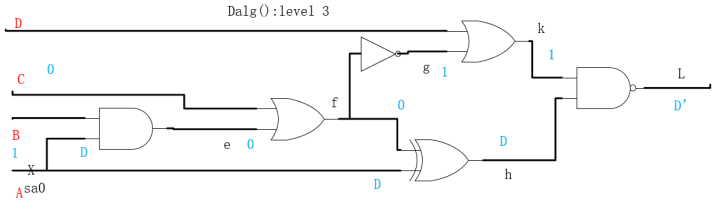

Figure 6: D-algorithm, Recursive, Step 5

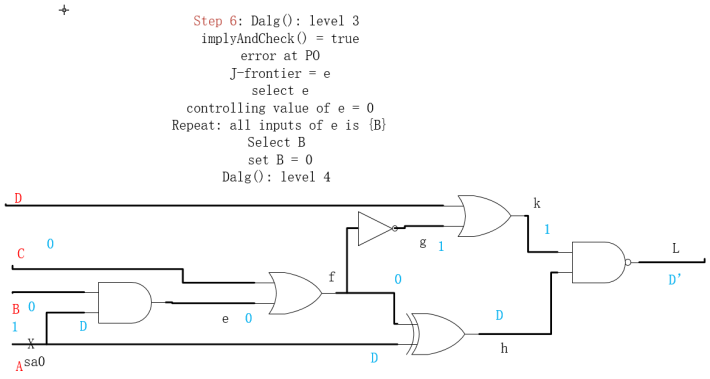

Figure 7: D-algorithm, Recursive, Step 6

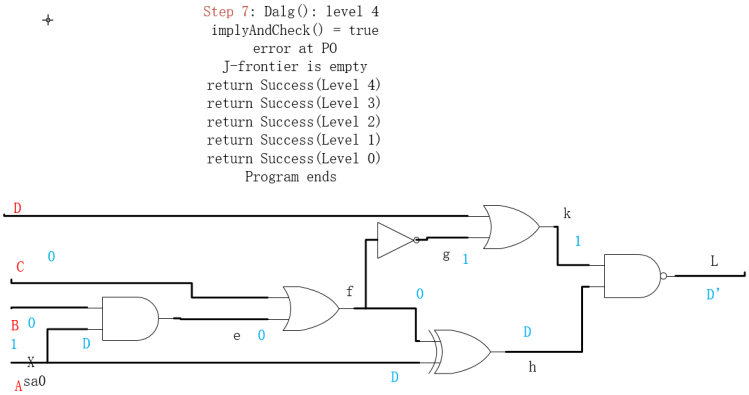

Figure 8: D-algorithm, Recursive, Step 7

Algorithm 2: D-Algorithm: Iterative Method - Top Level

```
/* D-Algorithm - Top Level                                          */
number all circuit circuit line in increasing level order from PIs to POs
select a primary D-cube of the fault to be the test cube
put logic outputs with inputs labeled as D(Dbar) onto the D frontier
D_drive()
Consistency()
return()
```

Algorithm 3: D-Algorithm: Iterative Method - D_Drive()

```
D_drive()                                                  // function name
while untried fault effects on D-frontier do
    select next untried D-fronter gate for propagation
    while untried fault effect fanouts exist do
        select next untried fault effect fanout
        generate next untried propagation D-cube
        D-intersect selected cube with test cube
        if intersection fails or is undefined then
            continue
        if all propagation D-cube tried & failed then
            break
        if intersection succeeded then
            add propagation D-cube to test cube - recreate D-frontier
            find all forward and backward implications of assignment
            save D-frontier, algorithm state, test cube, fanouts, fault
            break()
        else
            if intersection fails & D and Dbar in test cube then
                Backtrack()
            else
                if intersection fails then
                    break()
if all fault effects unpropagatable then
    Backtrack()
```

Algorithm 4: D-Algorithm: Iterative Method - Consistency()

```
/* D-Algorithm - Consistency()                                    */
g = coordinates of test cube with 1's & 0's
if g is only PIs then
    fault tesable & stop
for each unjustified signal in g do
    select highest # unjustified signal z in g, not a PI
    if inputs to gate z are both D and Dbar then
        break()
    while untried singular covers of gate z do
        select next untried singular cover
        if no more singular covers then
            if no more stack choices then
                fault untestable & stop
            else if untrued alternatives in Consistency then
                pop implication stack - try alternative assignment
            else
                backtrack()
                D_drive()
        if singular cover D_intersects with z then
            delete z from g
            add inputs to singular cover to g
            find all forward and backward implication of new assignment
            break
        if intersection fails then
            mark singular cover as failed
```

Algorithm 5: D-Algorithm: Iterative Method - Backtrack()

```
/* D-Algorithm - Backtrack()                                      */
if PO exists with fault effect then
    Consistency()
else
    pop prior implication stack setting to try alternate assignment
if no untried choices in implication stack then
    fault untestable & stop
else
    return
```

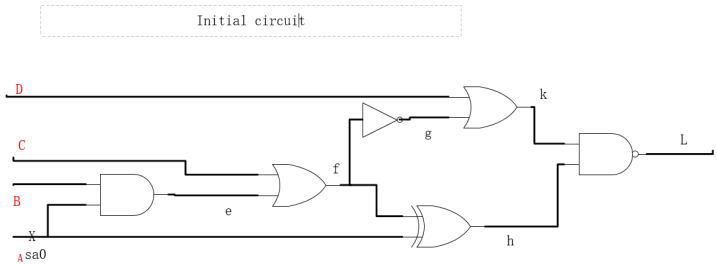

Figure 9: The initial circuit with stuck at fault

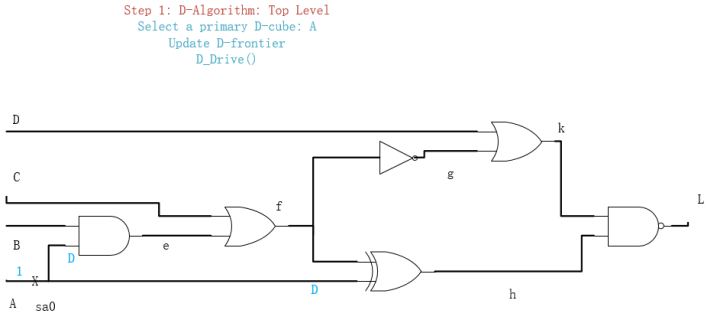

Figure 10: D-algorithm, iterative, Step 1

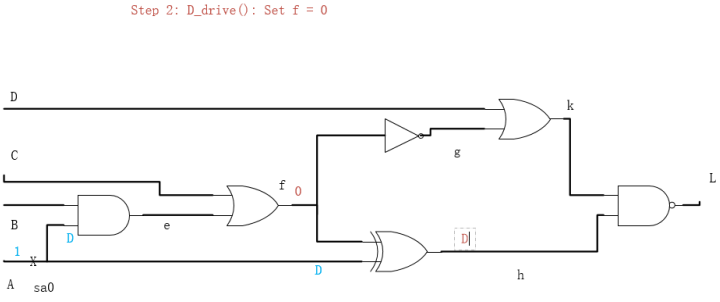

Figure 11: D-algorithm, iterative, Step 2

Step 3: D_drive(): Set k = 0

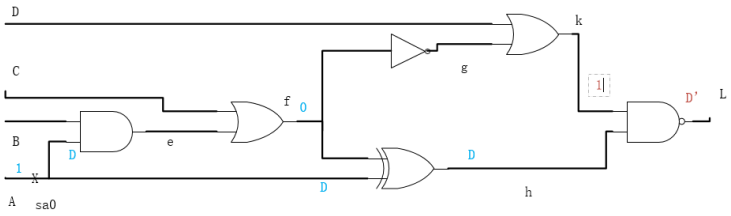

Figure 12: D-algorithm, iterative, Step 3

Step 4:Consistency(): Set g = 0

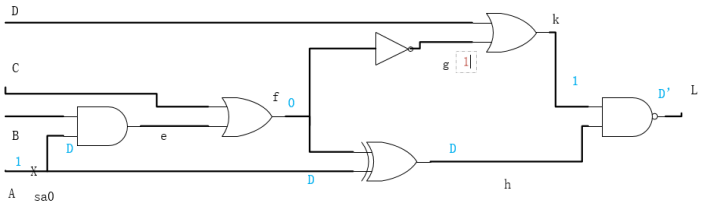

Figure 13: D-algorithm, iterative, Step 4

Step 5:Consistency(): Set f = 0, already set

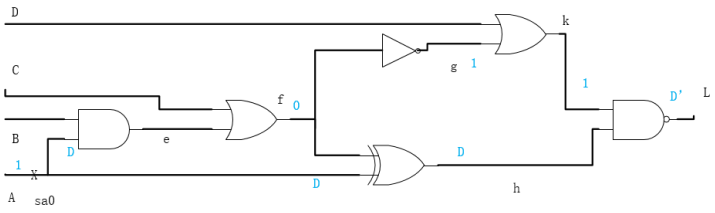

Figure 14: D-algorithm, iterative, Step 5

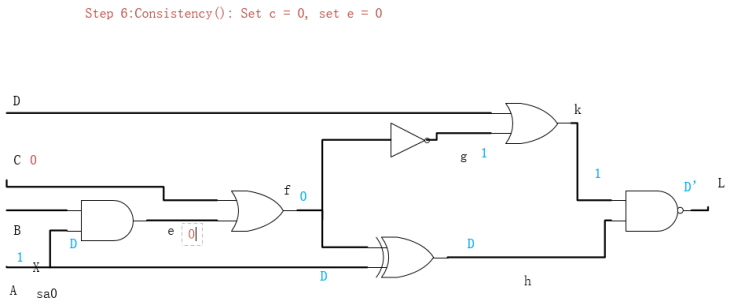

Figure 15: D-algorithm, Recursive, Step 6

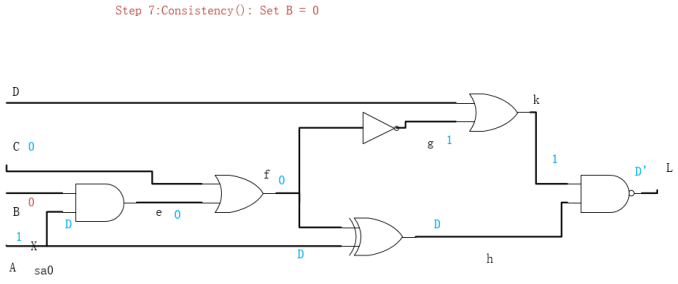

Figure 16: D-algorithm, Recursive, Step 7

2 Podem

IBM introduced semiconductor DRAM memory into its mainframes in late 1970's. Due to error correction and translation in the memory circuit, D-algorithm has limitations since it cannot test circuits with too undirected search or large XOR-gate trees, and it requires to set all external inputs to define the outputs. Therefore a new algorithm named Podem is introduced in 1981 by P.Goel. The procedure is similar but it employs a direct search techniques. although the space complexity is similar, time complexity is reduced a lot since it only makes decision at PIs. In worst case, the time complexity is 2^n, where n represents the total number of PIs. Another advantage is that no justification requires since all values are assigned to PI directly then forward implication. In high level, there are four step of PODEM algorithm.
1.Objective(): pick an objective to set a node to a value.
2.Backtrace(): backtrace to a PI and set it to a value taht will help meet the objective
3.implication(): simulatie the network to calculate the effect of fixing the value of PI. If there is no possibility of sensitizing a path to a PO, then retry by reversing the value of PI that was set in step2 and simulate again.
4.D-frontier(): update D-frontier and return to step1, stop if the D-frontier reaches a PO.

Algorithm 6: Podem: Objective()

```
   /* Determining an objective                                          */
   /* We want to test line l s-a-v or propagate a fault to a primary output.
      */
   /* There are only two objective that can result.                     */
   /* In both cases, the objective is to justify a line and return a line and
      value                                                             */
1  if line l is an x then
2      return(l, v')
                                  // objective is justifying fault site to v'
3  select gate G from D-frontier
4  select input j from G having value x
5  c = controlling value of G
6  return(j, c')          // objective is propagating fault along D-frontier to
     primary output
```

Algorithm 7: Podem: Backtrace(k, v_k)

```
   /* Given an objective, we must find a primary input to control and meet
      it.                                                                  */
   /* Backtrace is to step backward through gate until a PI is found.  It
      only track of inversions and finally it return a primary input and the
      value to set it to.                                                  */
1  v = v_k
                                                  /* while we are not at PO */
2  while k is a gate output do
3  |  i = inversion of k
4  |  select a input of k with value x
5  |  v = v ⊕ i
   |          /* determine what the input should be based on the inversion */
6  |  k = j                                                  // step backward
7  return (k, v)        // All this does is find a PI and a value to set it to
```

Algorithm 8: Podem: High Level - Podem()

```
   /* 4 Steps                                                              */
1  if error at primary output then
2  |  return SUCCESS                                              // you win
   /* test not possible:  D-frontier is empty or expected to become empty */
3  if test not possible then
4  |  return FAIL
5  k, v_k = Objective()        // pick a gate from D-frontier or activate fault
6  (j, v_j) = Backtrace(k, v_k)                                  // find input
7  imply(j, v_j)
         /* success-error is at PO, or we got another gate from D-fontier and
    (recurse, recurse, ...)  got error to PO */
8  if Podem() == SUCCESS then
9  |  return SUCCESS
10 imply(j, v'_j)        // forward implication fails, assign complement to input
11 if Podem() == SUCCESS then
12 |  return SUCCESS
13 imply(j, x)                // another failure, recover it to original state
14 return FAIL
```

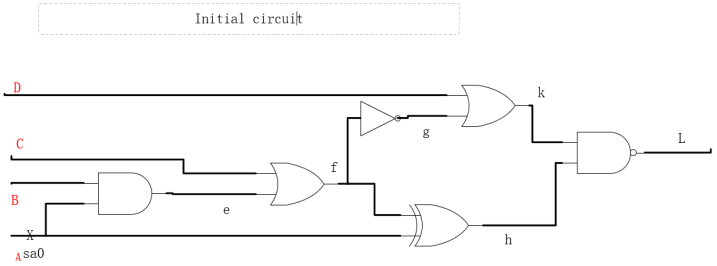

Figure 17: The initial circuit with stuck at fault

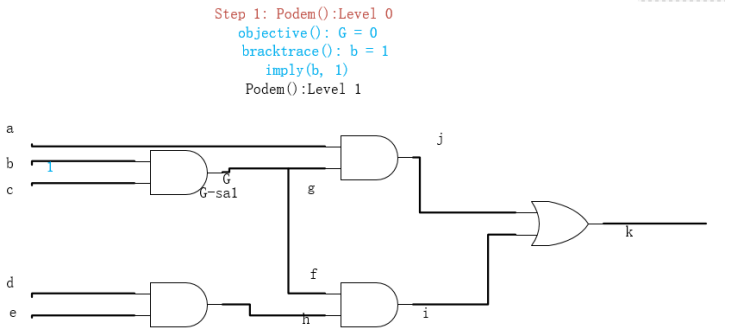

Figure 18: Podem, Step 1

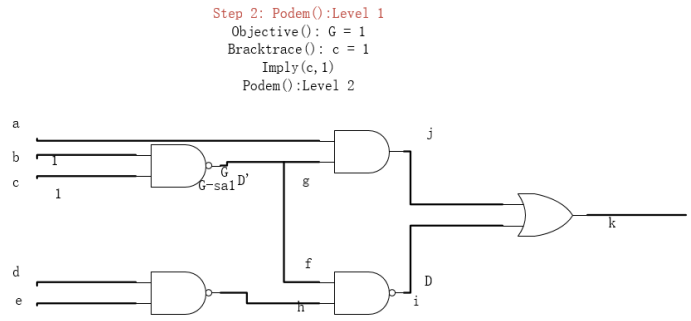

Figure 19: Podem, Step 2

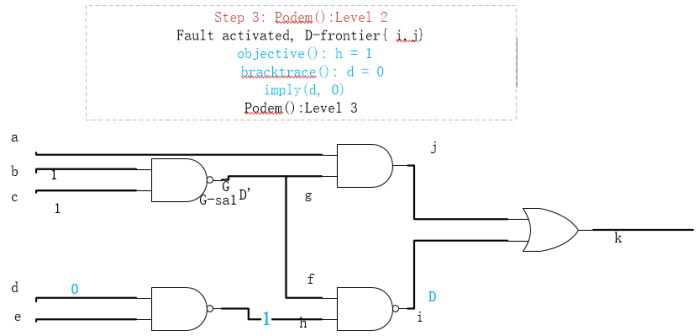

Figure 20: Podem, Step 3

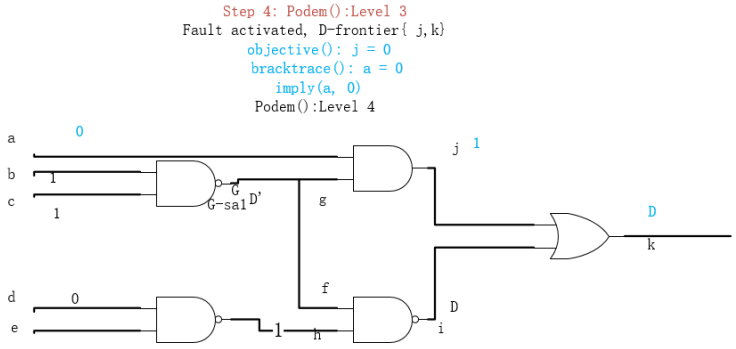

Figure 21: Podem, Step 4

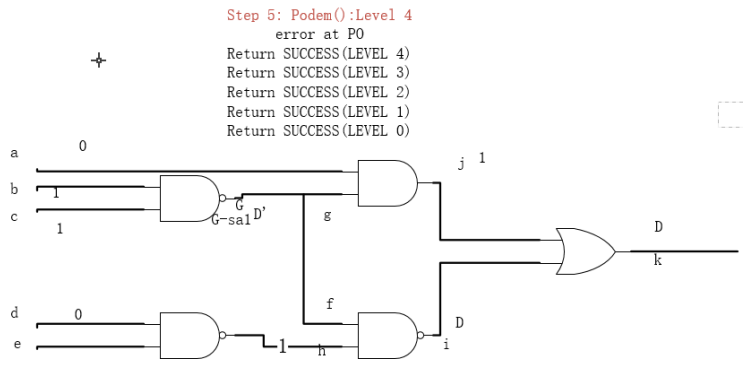

Figure 22: Podem, Step 5

3 Testability Measures

References:

- Z. Navvabi, chapter 5.2 Controllability and Observability
- Z. Navvabi, 5.3.1.2 Detection Probability
- Z. Navvabi, 7.2 Testability Insertion
- S. Gupta: chapter 4.5.3
- Goldstein LH (1979) Controllability/observability analysis of digital circuits. IEEE Trans Circuits Syst CAS-26(9):685–693.
- Agrawal VD, Mercer MR (1982) Testability measures – what do they tell us? in Proceedings of the International Test Conference.

RESPONSIBILITIES:

- Saeed: NVIDIA Paper
- Jiayi: SCOAP, and later STAFAN
- Han, circuit repo report
- Yang: parallelizing co/ob calculation
- this paper is very old, but can be read at some point: V. D. Agrawal and M. R. Mercer, "Testability Measures-What Do They Tell Us?" ITS-1982,
- Savir J (1983) Good controllability and good observability do not guarantee good testability.
- Brglez F (1984) On testability analysis of combinational networks.
- Grason J (1979) TMEAS – A testability measurement. Program. DAC 1979

The purpose of this project is to apply machine learning techniques to helping reducing time complexity in circuit test. For example, the test point insertion is SAT problem, with the help of GCN and enough training data, the possibility of a node as a test point could be predicted. Some techniques of reducing complexity of this np-hard problem are also explored. To do so, controllability and observability could be two main features besides other basic features for a node in the circuit. Two algorithms named SCOAP and STAFAN related to obtaining controllability and observability has been examined. The result of SCOAP has some bias and STAFAN is much more accurate. However, some step of STAFAN requires exhaustive test and machine learning strategy could also been applied here, so that the team could make use of SCOAP result to reduce the STAFAN complexity.

note: here we need to explain that some problems are some problems are NP-complete. and list these problems, including testability, etc.

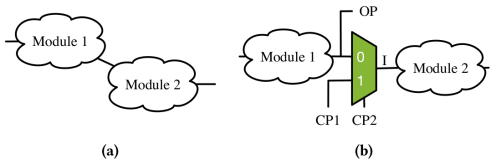

Figure 23: (a) Original circuit. Module 1 is unobservable. Module 2 is uncontrollable; (b) Insert test points to the circuit. Set (CP1, CP2) to (0, 1) and (1, 1) will set line I to 0 and 1, respectively; Set CP2 to 0 is the normal operation mode.

3.1 Observability and Controllability

While there are many heuristic approaches are introduced as observability and controllability, one can have a brute force definition for these terms. In [**?**], initial controllability measure is defined to have a range between 0 to 1 and the controllability of node A in a given logic circuit is correlated with the percentage of nodes in the circuit that must be set to specified logical values in order to justify a logic value on the node A. The observability is correlated with the percentage of nodes in the circuit that must be set to specified logical values in order to justify a logic value on the node and to observe the node value at the output.

Note: Saeed believes the second definition may not be accurate, as observation should be independent from controllability of node A.

Note: we must mention that all these measurements are in fault-free circuits.

3.2 Test Point Insertion

Test point insertion (TPI) is a broadly used approach in *design-for-testability* (DFT) to modify a circuit and improve its testability, which involves adding extra *control points* (CPs) or *observation points* (OPs) to the circuit.

An example of TPI is given in Figure 23 CPs can be used for setting signal lines to desired logic values, while OPs are added as scan cells to make a node observable. There are several issues that needed to be considered when performing TPI.

While many polynomial time heuristics were suggested for TPI problem, *such as B. Krishnamurthy, "A dynamic programming approach to the test point insertion problem," in Proc. DAC, 1987, pp. 695–705.*, the optimal test point placement problem is **NP-complete**. Therefore, there numerous TPI methods have been proposed to investigate the efficiency and performance of TPI.

Inserting test points may degrade the performance of a design in terms of area, power and timing. The ultimate goal of TPI is to achieve high fault coverage with less performance degradation. In this regard, we can consider CPs insertion, OPs insertion, inserting both CPs and OPs. The approach investi-

gated in NVIDIA paper is generic and can be applied to both CPs insertion and OPs insertion.

3.3 NVIDIA Paper

Problem definition (Observation Points Insertion): Given a set of netlists with all the nodes labeled as either difficult-to-observe or easy-to- observe. The objective is to train a classifier and adopt it to find a set of locations where the observation points should be inserted, which can maximize fault coverage and minimize observation points number and test pattern number. The feature for each node is considered as [LL, C0, C1, O], i.e. level of the node and SCOAP measurements. Every node also has a binary label. "0" (negative) means easy-to-observe and "1" (positive) means difficult-to-observe. Labels can be obtained *from commercial DFT tools.* **Note:** Saeed sent an email to the author of this work about labeling using commercial DFT tools. **Note:** complete the report by copy/paste section 4 of paper, iterative OP insertion.

3.4 SCOAP

For SCOAP algorithm, each node has a four dimensional features named LL, C0, C1, O. LL is defined as the logic gate level; C0 and C1 (controllability) are defined as the minimum number of nodes that must be set in order to produce either a zero or a one; O, observability, is defined as the minimum number of nodes which must be set for a fault to propagate from its source to a circuit output. for a gate type G, the input is A and B, and the output is C.

p;lease mention that we only focus on combination controllability here and sequential values are also defined in SCOAP.

Initialization: controllability (combination, represented with CC) of all primary input nodes are assigned to 1 and the observability of all primary outputs are assigned to 0 as well.

$CC0(PI) = 0$
$CC0(PI) = 0$
$O(PO) = 0$

for different gate type, the calculation of controllability are listed below.

- XOR gate:
  $C0(C) = min(C1(A) + C0(B), C0(A) + C1(B)) + 1$
  $C1(C) = min(C0(A) + C0(B), C1(A) + C1(B)) + 1$

- OR gate:
  $C0(C) = C0(A) + C0(B) + 1$
  $C1(C) = min(C1(A) + C1(B)) + 1$

- AND gate:
  $C0(C) = min(C0(A) + C0(B)) + 1$
  $C1(C) = C10(A) + C1(B) + 1$

- NOT gate:
  $C1(C) = C0(A)$
  $C0(C) = C1(A)$

for a gate type G, the input is A and B, and the output is C. for different gate type, the calculation of observability are listed below, and the in contrast with controllability starting from primary input, this calculation starts from primary output.

- XOR gate:
  $O(B) = min(C1(A), C0(A)) + O(C) + 1$
  $O(A) = min(C1(B), C0(B)) + O(C) + 1$

- OR gate:
  $O(A) = C0(B) + O(C) + 1$
  $O(B) = C0(A) + O(C) + 1$

- AND gate:
  $O(A) = C1(B) + O(C) + 1$
  $O(B) = C1(A) + O(C) + 1$

- NOT gate:
  $C1(C) = C0(A)$
  $O(A) = O(C)$

The SCOAP testability measures analyze the circuit at the gate-level; the values they generate are just estimates because of the simplifying assumptions made by its algorithm. These assumptions will produce some erroneous values.

SCOAP does not directly produce fault-specific data, but we can generate the specific testability data. For example, if we want to find the whether a s-a-0 fault at a fault site P can be detected or not, we can find its testability by:

$$Testability(p, sa0) = CC1(P) + CO(P) \tag{1}$$

3.5 STAFAN

From the paper itself: *STAtistical Fault ANalysis* (STAFAN) is proposed as an alternative to fault simulation of digital circuits. In this analysis, controllabilities and observabilities of circuit nodes are defined as **probabilities** which are **estimated from signal statistics** obtained from **fault-free simulation**. The computed probabilities are used to derive unbiased estimates of fault detection probabilities and overall fault coverage for the given set of input vectors. The computational complexity added to a fault-free simulator by STAFAN grows only linearly with the number of circuit nodes.

Note: what paper is claiming should be discussed with Jingwen and Shubo. In a recent work [3] fault detection probabilities were computed for random stimuli. In general, upper and lower bounds for the detection probabilities could be obtained. These bounds could be narrowed down at the cost of greater amount

of computation time. However, the method could handle only combinational circuits. In contrast, the present method computes the detection probabilities for the given stimuli (vector set) with which the fault-free simulation is carried out.

The improvement of STAFAN is that it normalizes the controllability and observability into the whole circuit as a probability, instead of just the total node number in SCOAP. Runing it exhaustively for each combination of test pattern in logic simulation, STAFAN provides a much more appropriate controllability and observability to detect a fault. Here is new definition of controllibility and observability in STAFAN. Assuming there is a node n in the circuit,

- C1(n) is one-controllability representing the probability of node n having a value 1 on a randomly selected vector.
  $C1(n)$ = total number of 1 applying all test patterns / all test patterns
- C0(n) is zero-controllability representing the probability of node n having a value 0 on a randomly selected vector. $C0(n)$ = total number of 0 applying all test patterns / all test patterns
- B1(n) is one-observability of a node n. It is defined as the probability of observing the node at a primary output when its value is 1. In other words, it gives the probability of sensitizing a path from this node to primary output. Similar to SCOAP, the observability also depends on controllability.
- B0(n) is zero observability of a node n, path sensitization probability form node n to primary output with node value 0.
- S(n) is sensitization probability. It is defined as the total sensitization path from node n to its output gate k divided by the total number of test pattern.This value also depends on differnet gate type. For instance, for AND and NAND gate as output gate k of node n, the path will only be sensitized for node n when output gate k's other input except node n itself are all one. For gate type OR and NOR, the other input values must be zero.

Besides the four basic features discussed above, to computation the observability, another features named sensitization probability is also required. like before to get C1 and C0, here a sensitization counter can be used during logic simulation. the counter will be incremented when a path from a the node to output gate has been detected. therefore The calculation of observability also depends on different gate type. Like before in the SCOAP, here all primary output will be initialized as 1 first and all other nodes will be calculated in a reversed order of level. for a gate type G, the input is A and B, and the output is C, here is the equation:

- XOR gate:
  $B1(A) = B1(C)$

$B0(A) = B0(C)$
$B1(B) = B1(C)$
$B0(B) = B0(C)$

- OR gate:
  $B0(A) = B0(C) * C0(C)/C0(A)$
  $B1(A) = B1(C) * (S(A) - C0(C))/C1(A)$
  $B0(B) = B0(C) * C0(C)/C0(B)$
  $B1(B) = B1(C) * (S(B) - C0(C))/C1(B)$

- AND gate:
  $B0(A) = B0(C) * (S(A) - C1(C))/C0(A)$
  $B1(A) = B1(C) * C1(C)/C1(A)$
  $B0(B) = B0(C) * (S(B) - C1(C))/C0(B)$
  $B1(B) = B1(C) * C1(C)/C1(B)$

- NOT gate:
  $B1(A) = B0(C)$
  $B0(A) = B1(C)$

- Some heuristics are defined for this, here is a list with references: 1. SCOAP 2. STAFAN 3. BLAH 4. FOO

- STAFAN

- Machine Learning approaches and graph neural network