**D-Algorithm**

- **D-Frontier**
  - All gates whose output values are X, but have D (or D) on their inputs.
- **J-Frontier (Justification Frontier)**
  - All gates whose output values are known (implied by problem requirements) but are not justified by their gate inputs.

---

**D-Algorithm**

- **Initialization:**
  - set all line values to X
  - activate the target fault by assigning logic value to that line
- 1. Propagate D to PO
- 2. Justify all values
- `Imply_and_check()` does only necessary implications, no choices.
- if `D-alg() = SUCCESS` then return SUCCESS
  - else undo assignments and its implications
D-Algorithm vs. PODEM

**D-Algorithm**
- Values assigned to internal nodes and PI's
- Decisions
  - Choose a D from D-Frontier
  - Assign a value to justify an output
- Conflict -- An implied value different from assigned value
- Other bounding condition
  - empty D-Frontier

**PODEM**
- Values assigned only to PI's
- Decision
  - Choose a PI and assign a value
- No conflicts
- Bounding conditions
  - fault not excited
  - empty D-Frontier
  - X-path check
  - lookahead

PODEM

- Improvements in the original paper were:
  - **X-path check**
    - Checks that at least one gate in D-Frontier has a path to a PO such that the node values on that path are all X's.
  - **Heuristics for selecting an objective in backtrace**
    - Among the required objectives, choose the hardest objective.
    - Among alternative objectives, choose the easiest objective.
Selection Heuristics

- Hardest to control
  - 1
  - 0

- Easiest to control
  - 0
  - 1

PODEM Algorithm

**PODEM**

- if error at PO then SUCCESS
- if test not possible then FAILURE
- $(k,v) = \text{Objective}(k,v)$ /* k=line, v=value */
- $(j, w) = \text{Backtrace}(k,v)$ /* j=PI, w=value */
- $\text{Imply}(j,w)$
- if PODEM()=SUCCESS then SUCCESS /* reverse direction */
- $\text{Imply}(j,w')$
- if PODEM()=SUCCESS THEN SUCCESS
- $\text{Imply}(j,w')$
- return FAILURE

**Objective()**
returns a gate from D-frontier

**Backtrace(l, v)**
traces from line l to a PI and counts inversions. It returns PI j and value W for j that can possibly set line l to v.
Cost Functions for Test Generation
Controllability and Observability

- Distance based.
- Fanout based.
- Probabilistic
- Many others.

\[ C_4 = f(C_1, C_2, C_3) + (3 - 1) \]

Cost Functions for Test Generation
Controllability and Observability

- Recursive (due to Rutman)

\[ f = 3 \]

\[ C_0(D) = \min\{C_0(A), C_0(B), C_0(C)\} + (f - 1) \]
\[ C_1(D) = C_1(A) + C_1(B) + C_1(C) + (f - 1) \]

\[ C_0(B) = C_1(A) \]
\[ C_1(B) = C_0(A) \]

\[ C_0(\text{PI}) = 0 \]
\[ C_1(\text{PI}) = 0 \] important: not 1

- Observability of PO is set to 0

\[ O(A) = C_1(B) + C_1(C) + O(D) \]
\[ O(D) = \min\{O(D_1), O(D_2), O(D_3)\} \]
Cost Functions for Test Generation
Controllability and Observability

- Probabilistic

\[ p = \text{probability of 1}\]
\[ 1 - p = \text{probability of 0}\]

\[ p_D = p_A \cdot p_B \cdot p_C \]
\[ O(A) = p_B \cdot p_C \cdot O(D)\]

\[ p_D = 1 - (1 - p_A)(1 - p_B)(1 - p_C)\]

\[ p_D = 1 - p_A\]

\[ C1(PI) = \frac{1}{2}\]
\[ C0(PI) = \frac{1}{2}\]
\[ O(PO) = 1\]

Example Circuit

```
+---+-----+---+---+---+
|   |     |   |   |   |
| 1 |     | 2 | 3 | 4 |
+---+-----+---+---+---+
|   |     |   |   |   |
| 5 |     | 6 | 7 | 8 |
```

Node | C0 | C1 | O
-----|----|----|----
  1  | 0  | 0  | O
  2  | 0  | 0  | O
  3  | 0  | 0  | O
  4  | 0  | 0  | O
  5  | 0  | 0  | O
  6  | 0  | 0  | O
  7  | 0  | 0  | O
  8  | 0  | 0  | O
```
FAN (Fanout-Oriented TG)

- Backtrace, rather than stopping at a PI, stops at internals: head lines and stems.
  - A head line is a line which is fed by a fanout-free region from the PI's and whose successor is fed by signals from regions containing fanout.
  - Stops at stems when multiple objectives are conflicting (described next)
  - Gives early backtrack.

Multiple objective backtrace
- Multiple objectives are taken from the J-frontier
  - Example:

```
Diagram of the circuit with multiple objectives
```

- Multiple objectives are \{a=1, b=1, c=1\} (PODEM has only one objective)
- These objectives are backtraced to a PI, head line, or stem.
**FAN (Fanout-Oriented TG)**

- Count number of requests for 1 and 0 and pick the value with highest number of requests. Then `implies()`

- Observation: FAN usually runs faster than PODEM:
  - Head lines are often too shallow to give substantial benefits.
  - Multiple backtrace is hard to evaluate.
  - FAN also did forward and backward implications

---

**Dominator and Mandatory Assignments**

- A gate is a *dominator* of a line if all paths from that line to all PO's pass through the gate.

- All off-path lines of dominator gates need to be set to non-controlling values for D/D' to propagate
  - Set of Mandatory Assignments (SMA) for a fault
Dominators and Mandatory Assignments

- Mandatory assignments are found by backward and forward implications [SOCRATES, Schulz et al., 1987] [TOPS, Kirkland and Mercer, 1987]

  - This yields a list of objectives to propagate the D forward.
  - This list is then sorted in order of hardest to easiest to control. The hardest-to-control objectives are satisfied first.

Dominators Example

- Gates 1, 3, 5, and 6 are dominators of line A.
- Lines B and C must be set to X/1.
- Lines D and E must be set to X/1 or 1/X depending on the parities of the paths.
- Lines F and G are set to X/1 by backward implication.
- Line H is set to X/1 by forward implication.
- List of objectives: (B, X/1), (C, X/1), (F, X/1), (G, X/1), (D, ?), (E, ?) (unique sensitization)
Static Learning (SOCRATES)

• Law of Contraposition: $A \Rightarrow B \text{ implies } \neg B \Rightarrow \neg A$

  $a=1 \text{ forward implies } f=1$

  $a=1 \Rightarrow f=1$ implies that we learn $f=0 \Rightarrow a=0$

• Learn and store this logic implication during a preprocessing step. Use it during ATPG to save time.

SOCRATES_learn()

for every signal $i$ do {
  assign($i$, 0)
  imply($i$)
  analyze_result($i$)
  assign($i$, 1)
  imply($i$)
  analyze_result($i$)
  assign($i$, X)
  imply($i$)
}

analyze_result($i$):

  let $i=v \Rightarrow j=v'$ and $j$ output of gate $G$. If all input of $G$ have non controlling values then it is worth learning $j=v' \Rightarrow i=v$

Dynamic Learning [Kunz et al, 1991]: apply SOCRATES dynamically (higher # of recursion levels). Can be exponential in time, always linear in space.
Dynamic Learning (Kunz et al, 1991)

Dynamic learning: Used in ATPG, logic optimization and verification

Example:

\[
\begin{align*}
\text{a} & \quad \text{b} \\
\text{e} & \quad \text{f} \\
\text{d} & \quad \text{c} \\
\text{y} & \quad \text{x}
\end{align*}
\]

\[\text{imply}(x, 0)\] gives no logic implications (i.e, learns nothing) in SOCRATES

Since \((a=0, b=0)\) is a requirement and \((b=0, f=0)\) conflict we need to assign \(x=1\) as the only viable option in the OR part of the decision tree. Therefore we learn logic implication:

\[y=1 \Rightarrow x=1 \quad \Leftrightarrow \quad x=0 \Rightarrow y=0\]
**Redundancy Identification**

- **Homework:** Fault $q$ s-a-1 is redundant (conflict in SMA)

**Multiple Fault Masking**

- **Homework:** F s-a-0 tested when $q$ s-a-1 is not present
Multiple Fault Masking

- Homework: F s-a-0 not tested (masked) when q s-a-1 is present

Random Test Generation

- Phase I: Random
  - inexpensive
  - \( t_1 \)

- Phase II: Deterministic
  - expensive
  - \( t_2 \)
  - total time = \( t_1 + t_2 \)

- Is random phase useful?

  
  
  
  100% Fault Coverage
  
  
  number of vectors

  } apply deterministic
  
  Random phase
Random Test Generation

- Some measurements show that there is very little to gain in time with two phases, if your deterministic phase is good.

```
<table>
<thead>
<tr>
<th>Deterministic phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Coverage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>100%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>number of vectors</td>
</tr>
</tbody>
</table>
```

Weighted Random ATG

- For uniform distribution of 1’s and 0’s, the probability of detection of faults F s-a-0 and G s-a-1 is 1 in a million.
- Having more 1’s than 0’s will increase the probability of detection of F s-a-0 but will decrease the probability for G s-a-1.
- Having more 0’s than 1’s will have the opposite effect.
- In general, each input may require different probability of 1’s and 0’s. [Wunderlich, DAC, 1987]
**RAPS**
Random Path Sensitization (Goel)

- Create random critical paths between PI’s and PO’s.
  - Select a random PO Z and a random value v.
  - Perform random backtrace with (Z, v) as the objective.
  - Continue selecting PO’s until all have assigned values.
  - Fill in any unknown PI values so as to maximize the number of critical paths.
- Results are better than random test generation.

**Test Set Compaction**

- Reduction of test set size while maintaining fault coverage of original test set.
  - **Static Compaction**
    - Compaction is performed after the test generation process, when a complete test set is available.
  - **Dynamic Compaction**
    - Compaction is performed as tests are being generated.
Static Compaction Techniques

- Reverse order fault simulation.
  - Remove vectors that don’t detect any faults.
- Generate partially-specified test vectors and merge compatible vectors.

\[
\begin{align*}
    t_1 &= 01x \\
    t_2 &= 0x1 \\
    t_3 &= 0x0 \\
    t_4 &= x01
\end{align*}
\]

\[
\begin{align*}
    t_{12} &= 011 \\
    t_3 &= 0x0 \\
    t_4 &= x01
\end{align*}
\]

or

\[
\begin{align*}
    t_{13} &= 010 \\
    t_{24} &= 001
\end{align*}
\]

- Heuristics are used to select vectors for merging.

Dynamic Compaction Techniques

- Obtain a partially-specified test vector for the primary target fault.
- Attempt to extend the test vector to cover secondary faults.
  - Heuristics for choosing secondary faults.
Independent Faults

- No test exists that detects two different faults in a set of independent faults.
- The size of the largest set of independent faults is a lower bound on the minimum test set size.

Fault Ordering

- Compute maximum independent fault sets.
  - Compute independent fault sets only within maximal fanout-free regions (FFR).
- Order faults such that largest sets of independent faults come first.
- Identify faults in FFR that can potentially be detected by the vector for a given fault.
Maximal Compaction

- Unspecify PI values that are not needed to detect target fault (heuristic).
  - Test vector may no longer detect target fault.

Try:

0111 → not detected
1011 → detected
1101 → detected
1110 → not detected

vector = 1xx1

Note that target fault is not detected by 1001

COMPACTEST Algorithm

- Get primary target fault from front of fault list.
- Generate test for target fault that also detects other faults in its FFR if possible.
- Maximally compact the vector.
- Get secondary target fault and attempt to extend test vector to cover it.
  - If successful, try to detect additional faults in same FFR as secondary target fault, and maximally compact newly-specified parts of test vector.
  - Else set newly-specified values back to X.
- Repeat until all inputs specified or all faults tried.
- Specify all unspecified PI’s randomly.