ECE 586 – Fault Detection in Digital Circuits
Lecture 18 ATPG for SSFs IV

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

March 23, 2015
This lecture: 6.2.1, 6.2.2, 6.2.3, 6.4
Next lecture: 9.1, 9.2
Overview

Fault-Independent TG

Random TG

ATPG Systems
TG in a Decision Tree

- Each node represents a sub-problem to be solved.
  - Justify and Propagate.
- Each branch represents a solution.
  - Single solution – easy to handle.
  - Multiple solution – need to memorize and try all in the worse case.
  - No solution – we made a bad choice, need backtracking.
Maximum Implications Principle

- TG still runs for exponential time in the worse case though we may greatly speed-up it for practical cases.

- *Maximum implications principle:* perform as many implications as possible so that less (bad) decisions need to be made.

- The TG algorithm will be able to run very efficiently in practice for *most* cases.
The Implication Process

- **Inputs**
  - The current decision tree.
  - A node and a branch of it to try next.

- **Outputs**
  - Generate a new node by solving all sub-problems that have a single solution, including the newly generated sub-problems.
  - Update the queue of unsolved sub-problems accordingly.

- The queue of unsolved sub-problems can be further decomposed into two queues.
  - The $D$-frontier: gate whose output is $x$ but have $D$ or $\overline{D}$ on its inputs.
  - The $J$-frontier: gate whose output is known but not implied by any input.
The Implication Algorithm

- Use a queue named *assignment queue* to maintain all newly generated sub-problems.
- For each iteration, we extract a sub-problem from the queue and try to solve it.
  - Return FAILURE if there is no solution.
  - Solve it if there is a single solution – this may generate more sub-problems that go to the assignment queue.
  - If there are multiple solutions, put the sub-problem into either the $D$-frontier or the $J$-frontier.
- If the assignment queue is empty and the $D$-frontier has a single element, we may apply unique $D$-drive to restart implication.
The \textit{D-Algorithm} (Abramovici et al., 1990)

\begin{verbatim}
D-alg()
begin
  if \textit{Imply\_and\_check()} = FAILURE then return FAILURE
  if (error not at PO) then
    begin
      if D-frontier = \O then return FAILURE
      repeat
        begin
          select an untried gate (G) from D-frontier
          c = controlling value of G
          assign \overline{c} to every input of G with value \(x\)
          if D-alg() = SUCCESS then return SUCCESS
        end
        until all gates from D-frontier have been tried
      return FAILURE
    end
  /* error propagated to a PO */
  if J-frontier = \O then return SUCCESS
  select a gate (G) from the J-frontier
  c = controlling value of G
  repeat
    begin
      select an input (j) of G with value \(x\)
      assign \(c\) to j
      if D-alg() = SUCCESS then return SUCCESS
      assign \overline{c} to j /* reverse decision */
    end
    until all inputs of G are specified
  return FAILURE
end
\end{verbatim}

(Abramovici et al., 1990)
Example: Step 1

▶ The $D$-frontier will contain three elements after the first implication.

(Fig. 6.24, Abramovici et al., 1990)
Example: Step 2

Try fault propagation through $i$ via recursion.

No conflict after implication.

(Fig. 6.24, Abramovici et al., 1990)
Example: Step 3

- Try fault propagation through $n$ via recursion.
  - Conflict found at $k$.  

(Fig. 6.24, Abramovici et al., 1990)
Example: Step 4

Backtracking to Step 2 with $n$ removed from the $D$-frontier.
Example: Step 5

- Try fault propagation through $k$ via recursion.
- No conflict after implication.

(Fig. 6.24, Abramovici et al., 1990)
Example: Step 6

▶ Try fault propagation through $n$ via recursion.
▶ Conflict found at $m$.

(Fig. 6.24, Abramovici et al., 1990)
Example: Step 7

Backtracking to Step 5 with $n$ removed from the $D$-frontier.

(Fig. 6.24, Abramovici et al., 1990)
Example: Step 8

▶ Try fault propagation through $m$ via recursion.
▶ Implication found $\overline{D}$ at PO. DONE.

(Fig. 6.24, Abramovici et al., 1990)
Outline

Overview

Fault-Independent TG

Random TG

ATPG Systems
Motivation

- Fault-oriented algorithms could be very costly.
  - Need to traverse the decision tree.
  - Exponential time in the worst-case.
- Fault-independent TG
  - Derive a set of tests that detect a large set of SSFs.
  - Without targeting individual faults.
Motivation

- Fault-oriented algorithms could be very costly.
  - Need to traverse the decision tree.
  - Exponential time in the worst-case.
- Fault-independent TG
  - Derive a set of tests that detect a large set of SSFs.
  - Without targeting individual faults.
Critical paths for a test can be identified easily by critical path tracing.

- Half of the SSFs along a critical path can be detected by $t$.
- So it is desirable to generate a test with long critical paths.
- Algorithmic idea: starting from PO, justify critical values recursively.
Critical-Path TG

- Critical paths for a test can be identified easily by critical path tracing.
- Half of the SSFs along a critical path can be detected by $t$.
- So it is desirable to generate a test with long critical paths.
- Algorithmic idea: starting from PO, justify critical values recursively.
Critical paths for a test can be identified easily by critical path tracing.

Half of the SSFs along a critical path can be detected by $t$.

So it is desirable to generate a test with long critical paths.

Algorithmic idea: starting from PO, justify critical values recursively.
Critical paths for a test can be identified easily by critical path tracing.

Half of the SSFs along a critical path can be detected by $t$.

So it is desirable to generate a test with long critical paths.

Algorithmic idea: starting from PO, justify critical values recursively.
Figure 6.45 Example of critical-path TG

(Abramovici et al., 1990)
Discussions

- A critical output may result from multiple possible critical inputs.
- Similar to fault-oriented TG, we may use a decision tree to explore such possibilities, or solutions, systematically.
- However, each solution may or may not lead to a test.
  - Each solution leads to a test in fanout-free circuits.
  - In circuits with fanouts, conflicts may arise.
- Since our goal is to find multiple test vectors, we may continue the traversal even if a test is found.
  - Unlike fault-oriented TG where we stop after finding a test vector.
A critical output may result from multiple possible critical inputs.

Similar to fault-oriented TG, we may use a decision tree to explore such possibilities, or solutions, systematically.

However, each solution may or may not lead to a test.

Each solution leads to a test in fanout-free circuits.

In circuits with fanouts, conflicts may arise.

Since our goal is to find multiple test vectors, we may continue the traversal even if a test is found.

Unlike fault-oriented TG where we stop after finding a test vector.
A critical output may result from multiple possible critical inputs.

Similar to fault-oriented TG, we may use a decision tree to explore such possibilities, or solutions, systematically.

However, each solution may or may not lead to a test.

- Each solution leads to a test in fanout-free circuits.
- In circuits with fanouts, conflicts may arise.

Since our goal is to find multiple test vectors, we may continue the traversal even if a test is found.

- Unlike fault-oriented TG where we stop after finding a test vector.
A critical output may result from multiple possible critical inputs.

Similar to fault-oriented TG, we may use a decision tree to explore such possibilities, or solutions, systematically.

However, each solution may or may not lead to a test.

- Each solution leads to a test in fanout-free circuits.
- In circuits with fanouts, conflicts may arise.

Since our goal is to find *multiple* test vectors, we may continue the traversal even if a test is found.

- Unlike fault-oriented TG where we stop after finding a test vector.
Outline

Overview

Fault-Independent TG

Random TG

ATPG Systems
Quality Measures

- Consider $N$ randomly generated test vectors.
  - We assume input vectors are uniformly distributed and independently generated – a test vector may appear twice.
- Testing quality: $t_N$.
  - Probability of the test vectors detect all detectable SSFs.
- $N$-step detection probability of fault $f$: $d_N^f$.
  - Probability of the test vectors detect $f$.
- Detection quality: $d_N = \min_{f \in SSF} d_N^f$.
- $t_N \leq d_N^f$ for any fault $f$, so $t_N \leq d_N$. 
Consider $N$ randomly generated test vectors.
- We assume input vectors are uniformly distributed and independently generated – a test vector may appear twice.

Testing quality: $t_N$.
- Probability of the test vectors detect all detectable SSFs.
- $N$-step detection probability of fault $f$: $d^f_N$.
- Probability of the test vectors detect $f$.

Detection quality: $d_N = \min_{f \in \text{SSF}} d^f_N$.
- $t_N \leq d^f_N$ for any fault $f$, so $t_N \leq d_N$. 
Consider \( N \) randomly generated test vectors.
- We assume input vectors are uniformly distributed and independently generated – a test vector may appear twice.

**Testing quality:** \( t_N \).
- Probability of the test vectors detect all detectable SSFs.

\( N \)-step detection probability of fault \( f \): \( d^f_N \).
- Probability of the test vectors detect \( f \).

**Detection quality:** \( d_N = \min_{f \in \text{SSF}} d^f_N \).
- \( t_N \leq d^f_N \) for any fault \( f \), so \( t_N \leq d_N \).
Consider $N$ randomly generated test vectors.
   - We assume input vectors are uniformly distributed and independently generated – a test vector may appear twice.

Testing quality: $t_N$.
   - Probability of the test vectors detect all detectable SSFs.

$N$-step detection probability of fault $f$: $d^f_N$.
   - Probability of the test vectors detect $f$.

Detection quality: $d_N = \min_{f \in SSF} d^f_N$.
   - $t_N \leq d^f_N$ for any fault $f$, so $t_N \leq d_N$. 
Consider $N$ randomly generated test vectors. 

- We assume input vectors are uniformly distributed and independently generated – a test vector may appear twice.

Testing quality: $t_N$.

- Probability of the test vectors detect all detectable SSFs.

$N$-step detection probability of fault $f$: $d^f_N$.

- Probability of the test vectors detect $f$.

Detection quality: $d_N = \min_{f \in SSF} d^f_N$.

- $t_N \leq d^f_N$ for any fault $f$, so $t_N \leq d_N$. 
Suppose we want to achieve $d_N > c$ for a given $c$.

- What $N$ should be for $n$ bits of input?

- For a fault $f$, $d^f_1 = \frac{|T_f|}{2^n}$.
  - $T_f$ is the set of all tests detecting $f$.
  - So $1 - d^f_N = (1 - d^f_1)^N$.

- Let $d_{min} = \min_{f \in SSF} d^f_1 = \frac{\min_{f \in SSF} |T_f|}{2^n}$, then

  \[ c \leq d_N = 1 - (1 - d_{min})^N. \]

So

\[ N \geq \frac{\ln(1 - c)}{\ln(1 - d_{min})}. \]
Suppose we want to achieve $d_N > c$ for a given $c$.

What $N$ should be for $n$ bits of input?

For a fault $f$, $d_1^f = \frac{|T_f|}{2^n}$.

$T_f$ is the set of all tests detecting $f$.

So $1 - d_N^f = (1 - d_1^f)^N$.

Let $d_{min} = \min_{f \in SSF} d_1^f = \frac{\min_{f \in SSF} |T_f|}{2^n}$, then

$$c \leq d_N = 1 - (1 - d_{min})^N.$$ 

So

$$N \geq \frac{\ln(1 - c)}{\ln(1 - d_{min})}.$$
Suppose we want to achieve \( d_N > c \) for a given \( c \).

- What \( N \) should be for \( n \) bits of input?

For a fault \( f \), \( d_1^f = \frac{|T_f|}{2^n} \).

- \( T_f \) is the set of all tests detecting \( f \).
- So \( 1 - d_N^f = (1 - d_1^f)^N \).

Let \( d_{\text{min}} = \min_{f \in \text{SSF}} d_1^f = \frac{\min_{f \in \text{SSF}} |T_f|}{2^n} \), then

\[
c \leq d_N = 1 - (1 - d_{\text{min}})^N.
\]

So

\[
N \geq \frac{\ln(1 - c)}{\ln(1 - d_{\text{min}})}.
\]
What if we want to achieve $t_N > c$?

\[ N \geq \frac{\ln(1-c) - \ln(k)}{\ln(1-d_{min})}. \]

- For $k$ be the number of faults whose detection probability is in $[d_{min}, 2d_{min}]$.

How large is $d_{min}$?

\[ d_{min} > \frac{1}{2^{n_{max}}}. \]

- $n_{max}$ is the largest number of PIs feeding a PO in the circuits.
- Though this lower-bound is usually too conservative.
What if we want to achieve $t_N > c$?

$$N \geq \frac{\ln(1-c) - \ln(k)}{\ln(1-d_{min})}.$$  

For $k$ be the number of faults whose detection probability is in $[d_{min}, 2d_{min}]$.

How large is $d_{min}$?

$$d_{min} > \frac{1}{2^{n_{max}}}.$$  

$n_{max}$ is the largest number of PIs feeding a PO in the circuits.

Though this lower-bound is usually too conservative.
What if we want to achieve $t_N > c$?

$N \geq \frac{\ln(1-c) - \ln(k)}{\ln(1-d_{min})}$.

For $k$ be the number of faults whose detection probability is in $[d_{min}, 2d_{min}]$.

How large is $d_{min}$?

$d_{min} > \frac{1}{2^n_{max}}$.

$n_{max}$ is the largest number of PIs feeding a PO in the circuits.

Though this lower-bound is usually too conservative.
What if we want to achieve $t_N > c$?

$n \geq \frac{\ln(1-c) - \ln(k)}{\ln(1-d_{min})}$.

For $k$ be the number of faults whose detection probability is in $[d_{min}, 2d_{min}]$.

How large is $d_{min}$?

$d_{min} > \frac{1}{2^{n_{max}}}$.

$n_{max}$ is the largest number of PIs feeding a PO in the circuits.

Though this lower-bound is usually too conservative.
What if we want to achieve $t_N > c$?

$N \geq \frac{\ln(1-c) - \ln(k)}{\ln(1-d_{\text{min}})}$.

For $k$ be the number of faults whose detection probability is in $[d_{\text{min}}, 2d_{\text{min}}]$.

How large is $d_{\text{min}}$?

$d_{\text{min}} > \frac{1}{2^{n_{\text{max}}}}$.

$n_{\text{max}}$ is the largest number of PIs feeding a PO in the circuits.

Though this lower-bound is usually too conservative.
Outline

Overview

Fault-Independent TG

Random TG

ATPG Systems
Goal of ATPG Systems

- The fault coverage of the generated tests as high as possible.
- The cost of generating tests (i.e., the CPU time) as low as possible.
- The number of generated tests as small as possible.
Goal of ATPG Systems

- The fault coverage of the generated tests as high as possible.
- The cost of generating tests (i.e., the CPU time) as low as possible.
- The number of generated tests as small as possible.
Goal of ATPG Systems

- The fault coverage of the generated tests as high as possible.
- The cost of generating tests (i.e., the CPU time) as low as possible.
- The number of generated tests as small as possible.
Two-Phase ATPG System

Figure 6.65

(Abramovici et al., 1990)
Two-Phase Algorithms

```
repeat
  begin
    Generate_test(t)
    fault simulate t
    v = value(t)
    if acceptable(v) then add t to the test set
  end
until endphase1()  

repeat
  begin
    select a new target fault f
    try to generate a test (t) for f
    if successful then
      begin
        add t to the test set
        fault simulate t
        discard the faults detected by t
      end
  end
until endphase2()  
```

Figure 6.66  First phase

Figure 6.67  Second phase  

(Abramovici et al., 1990)
Test Set Compaction

Since most test vectors are partially specified, we may use a single test vector to cover many of them.

The effectiveness of compaction depends on the choice of compaction algorithm.

- Static compaction: compact after all test vectors are generated.
- Dynamic compaction: after a partially specified vector is found for a fault, extend it immediately to detect more faults.

Experimental results show dynamic compaction is better.

- Smaller test sets and faster running time.

\[
\begin{align*}
  t_1 &= 01x \\
  t_2 &= 0x1 \\
  t_3 &= 0x0 \\
  t_4 &= x01 \\
  t_{12} &= 011 \\
  t_3 &= 0x0 \\
  t_4 &= x01 \\
  t_{13} &= 010 \\
  t_{24} &= 001.
\end{align*}
\]
Test Set Compaction

Since most test vectors are partially specified, we may use a single test vector to cover many of them.

The effectiveness of compaction depends on the choice of compaction algorithm.

- Static compaction: compact after all test vectors are generated.
- Dynamic compaction: after a partially specified vector is found for a fault, extend it immediately to detect more faults.

Experimental results show dynamic compaction is better.

- Smaller test sets and faster running time.

\[ t_{12} = 011 \]
\[ t_3 = 0x0 \]
\[ t_4 = x01 \]

\[ t_1 = 01x \]
\[ t_2 = 0x1 \]
\[ t_3 = 0x0 \]
\[ t_4 = x01 \]
\[ t_{13} = 010 \]
\[ t_{24} = 001. \]
Since most test vectors are partially specified, we may use a single test vector to cover many of them.

The effectiveness of compaction depends on the choice of compaction algorithm.

- Static compaction: compact after all test vectors are generated.
- Dynamic compaction: after a partially specified vector is found for a fault, extend it immediately to detect more faults.

Experimental results show dynamic compaction is better.

- Smaller test sets and faster running time.

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

or

\begin{align*}
t_{12} &= 011 \\
t_3 &= 0x0 \\
t_4 &= x01 \\
t_{13} &= 010 \\
t_{24} &= 001.
\end{align*}
Summary

- Fault-independent and random TGs serve as low-cost alternatives to fault-oriented TG, though the quality of the test set is not as high.
- Overall, an ATPG system combines various TG techniques to achieve high coverage using a small test set with low computational cost.