ATG for SSFs in Combinatorial Circuits

Traian Pop  
trapo@ida.liu.se

Sorin Manolache  
sorma@ida.liu.se
Outline

- Introduction
- Deterministic Test Generation
  - Fault-Oriented ATG
  - Fault Independent ATG
- Random Test Generation
  - Combined Deterministic and Random Test Generation
- ATG Systems
- Conclusions
Introduction

Testing

- off-line
- edge-pin
- stored-pattern
- full comparison of the output results

General problems for TG:

- the cost of generating the test
- the quality of the generated test
- the cost of applying the test
Deterministic Test Generation

- manual / automatic
- fault-oriented / fault-independent
 Fault-Oriented ATG

- targeted at certain fault within a fault universe given by a fault model

![Diagram showing primary inputs, f s-a-v, and primary outputs]

Problems:
- fault activation
- error propagation
Fault-Oriented ATG (cont’d)

Fault activation: line-justification (recursively justifying the value of a gate output by values of the gate inputs until primary inputs are reached)

Error propagation:
- composite logic values
- reduced to a set of line-justification problems

<table>
<thead>
<tr>
<th>v/vf</th>
<th>0/0</th>
<th>0/1</th>
<th>1/0</th>
<th>1/1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

non-controlling value
Fault-Oriented ATG (cont’d)

Fanout-free circuits:

Circuits with fanout:

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 1$</td>
<td>$c = 1$</td>
</tr>
<tr>
<td></td>
<td>$d = 1$</td>
</tr>
<tr>
<td></td>
<td>$j = 0$</td>
</tr>
<tr>
<td></td>
<td>$k = 0$</td>
</tr>
<tr>
<td></td>
<td>$o = 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h = 1$</td>
<td>$a = 1$</td>
</tr>
<tr>
<td></td>
<td>$b = 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o = 1$</td>
<td>$c = 0$</td>
</tr>
<tr>
<td></td>
<td>$k = 0$</td>
</tr>
</tbody>
</table>
Fault-Oriented ATG (cont’d)

\[ \text{Solve()} \]
\[ \begin{align*}
\text{begin} \\
\text{if}(\text{Imply\_and\_check()} = \text{FAILURE}) & \text{ then return } \text{FAILURE} \\
\text{if}(\text{error at PO and all lines are justified}) & \text{ then return } \text{SUCCESS} \\
\text{if}(\text{error can’t be propagated to a PO}) & \text{ then return } \text{FAILURE} \\
\text{select an unsolved problem} \\
\text{repeat} \\
\text{begin} \\
\text{select one untried way to solve it} \\
\text{if}(\text{Solve()} \text{ = SUCCESS}) & \text{ then return } \text{SUCCESS} \\
\text{end} \\
\text{until all ways to solve it have been tried} \\
\text{return } \text{FAILURE} \\
\text{end}
\end{align*} \]
Fault-Oriented ATG (cont’d)

D-frontier:
- used during error propagation process (D-drive)
- becomes void if no error can be propagated to a PO

J-frontier:
- keeps track of unjustified lines

Minimizing the number of incorrect decisions:
- maximum implications principle
- global implications
- reversing incorrect decisions
- error-propagation look-ahead
Fault-Oriented ATG (cont’d)

Algorithms:
- the D algorithm
- the 9-V algorithm
- single-path sensitization
- PODEM
- FAN
- etc.

```
D-alg()
begin
  if(Imply_and_check() = FAILURE) then return FAILURE
  if(error not at PO) then begin
    if(D-frontier = void) then return FAILURE
    repeat
      begin
        select an untried gate (G) from D-frontier
        c = controlling value of G
        assign \( \bar{c} \) 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
  endif
  if(J-frontier = void) then return SUCCESS
  select a gate (G) from J-frontier
  c = controlling value of G
  repeat
    begin
      select an input (j) of G with value x, assign c to j
      if(Solve() = SUCCESS) then return SUCCESS
    end
  until all inputs of G are specified
  return FAILURE
end
```
Fault-Oriented ATG (cont’d)

Principles used for decisions:
- attack the most difficult problem first
- try the easiest solution first

Measures:
- controllability: the relative difficulty of setting a line to a value
- observability: the relative difficulty of propagating an error from a line to a PO

Cost functions:
- distance-based functions
- recursive cost functions
- fanout-based cost functions

\[
C_0(l) = \min \{C_0(i)\} + f_l - 1 \\
C_1(l) = \sum \{C_1(i)\} + f_l - 1
\]
Fault-Independent ATG

Goal: to produce a set of tests that detect a large set of SSFs without targeting individual faults

Critical-path TG algorithm:

- select a PO and assign it a critical value
- recursively justify any critical value on a gate output by critical values on the gate inputs

! critical values are used instead of primitive values (complete specification)
Fault-Independent ATG (cont’d)

Critical-path TG for circuits with reconvergent fanout:
- conflicts
- self-masking
- multiple-path sensitization
- overlap among PO cones

Fault-oriented TG vs. Fault-independent TG:
- fault-oriented TG needs an additional simulation step to detect SSFs
- using critical-path TG, new tests can be generated starting from already existing ones
- fault-independent algorithms cannot identify undetectable faults
Random Test Generation

- do not target a particular fault

- random test vectors are applied and one hopes to detect as many faults as possible
Comparison

- advantages:
  - low test generation cost
  - test vectors usually generated on-the-fly (no need for test vector storage)

- disadvantages:
  - long test sequence (approx. 10 times longer than deterministically generated tests)
  - high test application cost
Quality Measures

- **test quality** – $t_N$ – the probability to detect *all* possible faults after applying $N$ random tests
- **$N$-step detection probability of $f$** – $d^f_N$ – the probability to detect fault $f$ after applying $N$ random tests
- **detection quality** – $d_N$ – the probability to detect the most difficult to detect fault after applying $N$ random tests

$$d_N = \min_f d^f_N$$

$$t_N < d_N$$
- usually, one is interested in how long a test sequence should be in order to achieve a detection quality (test quality) of $c$
- simulation too expensive
- if $N$ tests detect the most difficult to detect fault with probability $c$, then they will detect another fault with a probability $\geq c$
Test Length (2)

- $d_N \geq c, \quad N = ?$
  
  $d^f_1 = |T_f| / 2^n$, for uniformly distributed input test vectors

  $T_f$ – the set of test vectors that detect the fault $f$

  $d_{\text{min}} = \min_f d^f_1$, the lowest detection probability among the SSFs in the circuit

  $1 - (1 - d_{\text{min}})^N \geq c$

  $N \geq \text{ceil}(\ln(1 - c) / \ln(1 - d_{\text{min}}))$

- $t_N \geq c, \quad N = ?$
  
  $N \geq \text{ceil}((\ln(1 - c) - \ln(k)) / \ln(1 - d_{\text{min}}))$, $k$ – number of faults with detection probability $d_{\text{min}} \leq d \leq 2d_{\text{min}}$

- $d_{\text{min}}$ has to be determined
- $d_{\text{min}} \geq 1 / 2^n \Rightarrow$ exhaustive testing
- the probability to detect $s-a-0 = \text{the probability of } G \text{ being } 1$
- there may be several paths to propagate the error
- $d_{\text{min}} \geq P(G_k = 1)$ (max?)
- $P(l = 1)$ is computed in linear time for fanout-free circuits, otherwise exponential
- probability intervals and cuts can be used instead of fixed probabilities $\Rightarrow$ linear time
Difficult Faults

- sometimes $N_{\text{max}}$ is bounded (fixed) (testing time)
- having a desired detection quality $c$, one can deduce a lower bound, $d_L$
- then the circuit has to be checked if there are faults with $d_f \leq d_L$, the difficult faults
- among checkpoint faults (checkpoint = primary input or fanout branch)
- if found, then modify the circuit in such a way that the difficult fault becomes easier to detect (design for testability)
Non-Uniformly Distributed Test Vectors

- turned out to be better for some circuits for various optimization goals (coverage, testing cost, cost of DFT modifications)
- research for computing the pdf of the test vectors
- *adaptive RTG*, monitors the test generation process, gathers statistical data about the most successful test vectors and adjusts then the pdf of future test vectors
Combined Deterministic/Random TG

- **RAPS (Random Path Sensitization)**

- **SMART (Sensitizing Method for Algorithmic Random Testing)**

  Half of generated tests will have $A = 1$
requirements:
- fault coverage $\uparrow$
- test generation cost $\downarrow$
- test set size $\downarrow$

should it collapse faults?
repeat
  Generate_test(t);
  fault simulate t
  v = value(t)
  if acceptable(v) then add t to the test
until endphase1();
  /* what about test scheduling? */
  /* redundancy elimination is important */
repeat
  select a new target fault f /* better one close to the PI */
  try to generate a new test t for f
  if successful
    add f to the test
    fault simulate f
    discard the faults detected by f
  fi
until endphase2();
Test Set Compaction

- **static compaction**
  
  01x  011  010
  0x1
  0x0  0x0  001
  x01  x01

- **dynamic compaction**
  
  - try to set the unspecified PIs such that additional faults are detected
  - biggest problem is the selection of the additional fault to be tested
Other TG Methods

- algebraic – impractical
- extensions for tristate logic – problem: how to avoid the simultaneous enabling of multiple bus drivers
- TG for module-level circuits
  - modules assumed to be fault free
  - propagation and justification procedures more complicated, derived from the module’s function
  - if the modules are not fault free, either they are tested before (design for testability) or after by replacing one module at a time with its gate level model