Facilitating Unreachable Code Diagnosis and Debugging

Hong-zu Chou and Sy-yen Kuo
National Taiwan University
Kai-hui Chang
Avery Design Systems

ASPDAC
Jan. 27, 2011
Motivation

- Code coverage is an important metric in design verification
  - Typically, it is performed using logic simulation with constrained-random testbench
    - May miss corner cases and is not accurate
  - Formal code statement reachability analysis provides proving capabilities [Chou et al., ASPDAC10]

- Code that is proven to be unreachable is called dead code
  - Dead code is usually associated with bugs
  - Diagnosing the cause of problem can be challenging
Challenges in Unreachability Diagnosis

- Engineers perform debugging using waveforms
  - Unreachability means no waveform is available
  - Engineers have to analyze nonexistent paths to identify the cause of the problem

- Automatic error diagnosis and repair methods cannot be used
  - These algorithms require the “correct values” of a signal to be known
  - Unreachability means there are no correct values to be solved for
Our Contributions

- A new symbolic simulation algorithm that can explore nonexistent paths
  - “Liberated variables” give the algorithm the freedom to explore new paths
  - Cannot be achieved using traditional synthesis-based formal methods

- Error diagnosis is then applied to analyzing the symbolic condition to execute the target code
  - Key variables that contribute to the unreachability will be identified
  - Suggested values to solve the problem is provided
Outline

- Background – error diagnosis
- Problem formulation
- Common causes of unreachability
- Unreachableability diagnosis technique
- Experimental results
- Conclusions
Background - Error Diagnosis

1. To model errors: insert MUXes into the circuit
2. To limit the number of allowed errors: use an adder and a comparator
3. Convert the circuit to CNF
4. Constrain inputs/outputs using input vectors/correct output responses

[Smith et al., ASPDAC’04]
Problem Formulation

- Given a testbench, a design, a list of liberated variables and the target unreachable code
- Find a set of variables that contribute to the unreachableability
- Also provide suggested values to solve the problem
Common Causes of Unreachability

- **Hardware bugs**
  - Conflicting conditions
  - Obsolete code

- **Design modalities**

- **Testbench errors**
  - Over-constrained rules

- **Reachability analysis limitation**
  - Insufficient sequential depth
When liberated variables are accessed, a MUX is introduced so that the variable has the freedom free to take a new value.

This gives the symbolic simulator the freedom to explore previously-impossible paths.
module example;
reg mode, clk;
reg [7:0] result, a, b;
always @(posedge clk) begin
  if (mode == 0)
    result= a + b;
  else
    result= a – b;
end
initial begin
  mode= 0;
a= $random;
b= $random;
end

1. Select liberated variables: mode, a

Unreachable code. Target of diagnosis.
module example;
reg mode, clk;
reg [7:0] result, a, b;
always @(posedge clk) begin
  if (mode == 0)
    result = a + b;
  else
    result = a - b;
end
initial begin
  mode = 0;
a = $random;
b = $random;
end

1. Select liberated variables: mode, a
2. Modify symbolic simulation algorithm for MUX insertion when liberated variables are accessed:
   mode = mode_sel ? mode_free : 0
   result = ((mode_sel ? mode_free : 0) == 0) ? (a_sel ? a_free : 0) + b:
                      (a_sel ? a_free : 0) - b
module example;
reg mode, clk;
reg [7:0] result, a, b;
always @(posedge clk) begin
  if (mode == 0)
    result = a + b;
  else
    result = a - b;
end
initial begin
  mode = 0;
  a = $random;
  b = $random;
end

1. Select liberated variables: mode, a
2. Modify symbolic simulation algorithm for MUX insertion when liberated variables are accessed
3. Since symbolic condition for entering the else branch is no longer false, symbolic simulation will execute the target code under condition: \((mode_{sel} ? mode_{free} : 0) != 0\)
4. Unreachability diagnosis is performed on the symbolic condition to identify the select signals to be asserted
Unreachability Diagnosis - Example

- Unreachability diagnosis is performed on 
  \(((\text{mode}\_\text{sel} \ ? \ \text{mode}\_\text{free} : 0) \neq 0)\)

- A SAT solver is used to find a solution that can satisfy the condition
  - In this example, \(\text{mode}\_\text{sel} = 1\) and \(\text{mode}\_\text{free} = 1\)
  - Signal mode contributes to the unreachability
    - If its value is 1, then the code can be reached

- Similar to traditional error diagnosis, cardinality constraints are necessary
  - To narrow down the problem
Implementation Insights

- Selection of liberated variables
  - Typically involves all design variables except clocks and resets
  - Hierarchical approach can be applied
    - Narrow down the problem to a few blocks first
    - Then look into the blocks
- Values returned for $V_{\text{free}}$ are suggestions on how to fix the problem
Implementation Insights

- Shared select line/free variables can improve unreachability diagnosis performance

- For code that models hardware
  - Variables updated at the same cycle can share the same select and free symbols
  - At the RTL, each variable is typically updated only once at a clock

- For testbench code
  - Select lines typically cannot be shared because a variable may be updated many times at a time step
Experimental Results

- DLX design from Bug UnderGround project in University of Michigan
  - One of the few publicly available designs that contain non-trivial dead code
  - Design contains 40 bugs, but 6 of them can never be triggered
  - Goal: diagnose the causes of the 6 dead bugs

- Two industrial designs were also used
  - A block in a multimedia chip
  - A block in a high-speed I/O interface design
Diagnosis Example

- Bug description: if write to r7 is followed by ADD with rt=r7 write to r14 occurs
- Bug triggering code:
  ```
  RDwire = ((IR4[`op]==`SW) && (IR4[rt]==5'd7) && (RDaddr5==5'd7) && (IR5[`op]==`ADD))
             ? 5’d14 : RDaddr5;
  ```
- Diagnosis result: ADD is an illegal OP code. Correct instruction: OP=`SPECIAL_OP with subtype = ADD
# DLX Result – Full Chip

<table>
<thead>
<tr>
<th>Case</th>
<th>Runtime</th>
<th>Memory (MB)</th>
<th>#Liberated variables</th>
<th>#Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug20</td>
<td>3m20s</td>
<td>637.36</td>
<td>217</td>
<td>1</td>
</tr>
<tr>
<td>Bug22</td>
<td>1m22s</td>
<td>504.95</td>
<td>216</td>
<td>7</td>
</tr>
<tr>
<td>Bug29</td>
<td>8m49s</td>
<td>601.92</td>
<td>217</td>
<td>24</td>
</tr>
<tr>
<td>Bug31</td>
<td>14m45s</td>
<td>815.19</td>
<td>216</td>
<td>5</td>
</tr>
<tr>
<td>Bug33</td>
<td>28m39s</td>
<td>1146.15</td>
<td>218</td>
<td>1</td>
</tr>
<tr>
<td>Bug34</td>
<td>28m48s</td>
<td>1146.35</td>
<td>218</td>
<td>1</td>
</tr>
<tr>
<td>CaseA</td>
<td>29m17s</td>
<td>897.93</td>
<td>215</td>
<td>8</td>
</tr>
<tr>
<td>CaseB</td>
<td>29m13s</td>
<td>887.22</td>
<td>215</td>
<td>8</td>
</tr>
</tbody>
</table>

CaseA and CaseB are unreachable due to over-constrained testbenches. All other cases use properly-constrained testbenches and unreachability is due to design bugs.
## DLX Result – Buggy Module

<table>
<thead>
<tr>
<th>Case</th>
<th>Runtime</th>
<th>Memory (MB)</th>
<th>#Liberated variables</th>
<th>#Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug20</td>
<td>45s</td>
<td>357.73</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Bug22</td>
<td>43s</td>
<td>417.61</td>
<td>67</td>
<td>3</td>
</tr>
<tr>
<td>Bug29</td>
<td>4m52s</td>
<td>549.41</td>
<td>68</td>
<td>9</td>
</tr>
<tr>
<td>Bug31</td>
<td>8m1s</td>
<td>655</td>
<td>67</td>
<td>2</td>
</tr>
<tr>
<td>Bug33</td>
<td>13m26s</td>
<td>584.22</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Bug34</td>
<td>13m24s</td>
<td>584.33</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>CaseA</td>
<td>7m14s</td>
<td>394.69</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>CaseB</td>
<td>7m31s</td>
<td>394.29</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

- Runtime is shorter and diagnosis is more accurate
- Hierarchical approach can be useful
## Industrial Case Result

<table>
<thead>
<tr>
<th>Case</th>
<th>Lines of RTL</th>
<th>Runtime</th>
<th>#Liberated variables</th>
<th>#Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>DesignA</td>
<td>5074</td>
<td>1m34s</td>
<td>236</td>
<td>36</td>
</tr>
<tr>
<td>DesignB</td>
<td>8068</td>
<td>32m5s</td>
<td>1520</td>
<td>15</td>
</tr>
</tbody>
</table>

- Diagnosis involving 2 variables were reported for DesignA
  - Without our diagnosis, 27730 combinations of variables need to be checked
  - With our diagnosis, 99.8% possible combinations can be eliminated
- Our diagnosis can narrow down the problem
Conclusions

- Unreachability diagnosis is challenging
  - No counterexamples exist for debugging
- A new symbolic simulation algorithm that can explore nonexistent execution paths
- Error diagnosis based on symbolic conditions can identify the cause of unreachability
- Experimental results show that our techniques can successfully narrow down the problem