Managing Complexity in Design Debugging with Sequential Abstraction and Refinement

Brian Keng and Andreas Veneris

University of Toronto

ASP-DAC 2011
Motivation

- Functional Verification and Debug are major problems
  - Exponentially more costly to find bugs in silicon
  - Functional errors responsible for over 60% of re-spins
  - Trend: Two verification engineers per single designer!
- What’s the biggest bottleneck?
  - **Debug**: Takes up to 60% of total verification time

Time Spent in Design vs. Verification:

- Design: 47%
- Verification: 53%
The Debugging Bottleneck

- Functional Debug
  - Localize errors detected during verification

- Bottleneck:
  - Manual process
  - Designs are getting bigger and more complex
  - Consumes 5-7 months of design time per cycle

- How do we address it?
  - Automation!
Automating Debug

Automated Debugging
- Automatically locate places (i.e. *suspects*) in RTL that could fix failure

Algorithms
- Simulation-based, BDD-based, SAT-based etc.

Complexity = (design size * # cycles) # errors

How can these factors be managed for:
- Larger circuits?
- Longer traces?
- Multiple Errors?
## Previous Work and Contributions

<table>
<thead>
<tr>
<th>Previous Work [Safarpour et al., TCAD09]</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstraction</strong></td>
<td></td>
</tr>
<tr>
<td>• Simulated values (Neither over/under-approximation)</td>
<td>• Simulated values to generate an under-approximate model</td>
</tr>
<tr>
<td><strong>Refinement</strong></td>
<td></td>
</tr>
<tr>
<td>• Solutions for module refinement</td>
<td>• UNSAT cores for time + module refinement</td>
</tr>
<tr>
<td><strong>Solutions</strong></td>
<td></td>
</tr>
<tr>
<td>• Over-approximation of solutions</td>
<td>• Exact solutions</td>
</tr>
<tr>
<td><strong>Error Complexity</strong></td>
<td></td>
</tr>
<tr>
<td>• Requires increased error complexity</td>
<td>• No increased error complexity</td>
</tr>
</tbody>
</table>
Outline

- **Background**
  - Automated Debugging
  - SAT-based Debugging
  - UNSAT Cores
- **Sequential Abstraction and Refinement**
- Experiments
- Conclusion
Automated Debugging

- Erroneous circuit
- Error Trace
  - Initial State
  - Primary Inputs
  - Expected Values

![Diagram of an erroneous circuit with initial state, primary inputs, and expected values. The output mismatch is indicated by a green box labeled "Error! Output Mismatch."
SAT-based Debugging

[Smith, et. al TCAD ’05]

- 1) Unroll
- 2) Error models (e.g. muxes)
- 3) Constrain initial state, inputs, expected outputs
- 4) Constrain number of errors (error cardinality, N)

\[
\begin{align*}
\{e_0=1, e_2=1, e_3=1, e_4=1\}
\end{align*}
\]
**UNSAT Cores**

- Subset of clauses that are unsatisfiable
- Proof of unsatisfiability

This path will form an UNSAT CORE
Outline

- Background
- **Sequential Abstraction and Refinement**
  - Overall Algorithm
  - Abstraction
  - Module Refinement
  - Sequential Refinement
  - Comparison to Previous Work
- Experiments
- Conclusion
Overall Algorithm

1. Generate initial abstract model
2. Solve abstract model
3. Analyze UNSAT core:
   1. Exit if UNSAT core has no abstract clauses
   2. Refine using UNSAT core, repeat step 2
Abstraction:

- Replace module constraints in SAT instance with their simulated input/output values
  - Reduce size of SAT instance (design size)
  - Smaller run-time/memory
- Abstract instance finds a subset of the suspects of the original SAT instance (Under-approximation)
  - Property holds even after refinement
  - No need to find previous found solutions
  - Incremental solving
Abstraction Example

- Replace module constraints with simulated input/output values

Trivially UNSAT
Module Refinement

- **Refinement**
  - Use UNSAT core to determine which modules to refine
  - In next iteration, do not replace module constraints with simulated values
  - Allows for refinement with the same error cardinality

- **Exit condition:**
  - When UNSAT core does not contain any abstract input/output values
  - Complete set of solutions without refining entire problem
Module Refinement Example

SAT when \{e_0=1, e_2=1, e_3=1\}

SAT when \(e_4=1\)

UNSAT Core

UNSAT Core
Sequential Refinement

- Only refine modules in time-frames that are in UNSAT core
- Allows fine-grain refinement across time
- Smaller instances vs. many iterations
- Use same exit condition as before

Refine windows

- Refine all modules around radius $r$ involved with the UNSAT core
Sequential Refinement Example

SAT when

Exit Condition:
UNSAT core does not contain abstract input/output values

UNSAT Core

UNSAT Core

UNSAT Core

SAT when \( e_4 = 1 \)

UNSAT when \( \{e_0 = 1, e_2 = 1, e_3 = 1\} \)

Exit Condition:
UNSAT core does not contain abstract input/output values
## Comparison to Previous Work

<table>
<thead>
<tr>
<th></th>
<th>Previous Work [Safapour et al.]</th>
<th>Sequential Abstraction &amp; Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>Neither</td>
<td>Under-approximation</td>
</tr>
<tr>
<td>Refinement</td>
<td>Module</td>
<td>Module/Time</td>
</tr>
<tr>
<td>Debugging Engine</td>
<td>Any</td>
<td>SAT-based</td>
</tr>
<tr>
<td>Exact Solutions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>(over-approximation)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Cardinality</td>
<td>Requires increase</td>
<td>No increase</td>
</tr>
</tbody>
</table>
Outline

- Background
- Sequential Abstraction and Refinement
- Experiments
  - Experimental Setup
  - Solved Instances
  - Number of Solutions
  - Module vs. Sequential Refinement
- Conclusion
Experimental Setup

- Pentium Core 2, 2.66 Ghz, 8 GB ram
- 10 circuits from OpenCores.org and industrial partners
- Inserted in a typical RTL error
  - Wrong assignment, missing case statement, incorrect operator, etc.
- PicoSAT v913
- Timeout: 3600 seconds
- Sequential Refinement Window: $r=20$
Solved Instances

- **SAT-based**
  - Completed: 13
  - Time-out: 2
  - Mem-out: 7

- **Suspect Refinement**
  - Completed: 5
  - Time-out: 14
  - Mem-out: 3

- **Module Refinement**
  - Completed: 13
  - Time-out: 2
  - Mem-out: 7

- **Sequential Refinement**
  - Completed: 8
  - Time-out: 5
  - Mem-out: 9

* Suspect Refinement [Safarpour et al.]
### Number of Solutions

<table>
<thead>
<tr>
<th></th>
<th>SAT-based</th>
<th>Suspect * Refinement</th>
<th>Module Refinement</th>
<th>Sequential Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>conmax1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>fdct1</td>
<td>0</td>
<td>2450</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>fpu1</td>
<td>0</td>
<td>879</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>fxu1</td>
<td>24</td>
<td>1313</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>s_comm1</td>
<td>0</td>
<td>213</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>vga1</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>14</td>
</tr>
</tbody>
</table>

- Sequential refinement returns solutions for all instances

* Suspect Refinement [Safarpour et al.]
Conclusion

- Sequential Abstraction and Refinement
  - Finds exact solutions
  - Under-approximate abstraction
  - UNSAT core based refinement
    - Module refinement
    - Sequential refinement

- Experiments
  - Returns solutions for 100% of instances compared to 41% without the technique