Deep Bug Hunting

- Asa Ben-Tzur, SW Group Director, Cadence
- Formal Verification Conference, Reading, UK
- 16th June 2016
Agenda

Introduction to Bug-Hunting

Y Swarm

Z Swarm

Guidepointing
About Bug-Hunting

• Classic formal focuses on exhaustively analyzing the state-space
  – Wide search, thorough analysis

• Bug-Hunting focuses on finding bugs 😊
  – Deep/sparse search on state-space (not exhaustive)
Where to use Bug-Hunting

- Most formal TBs can benefit from bug-hunting
- No need for bug-hunting on small designs or simple verification tasks
When to use Bug-Hunting

• Start with normal formal TB development
  – Develop assertions
  – Fix assertions
  – Find bugs using classic formal engines
  – Apply (simple) mutations/abstractions

• Move to bug-hunting when you:
  1. Have enough assertions (from coverage reports)
  2. Stop finding bugs - clean environment

• Explore more states with bug-hunting
  – Find bugs using semi-formal techniques/engines
  – Techniques build on top of existing formal TB
Bug-Hunting Strategies

Classic Formal
Exhaustive analyze

Y Swarm
Skip or only partially analyze areas in state space

Z Swarm
Automatically guide formal search using key states

Guidepointing
Manually guide analysis using formal and simulation

Simulation
Constrained-random walk through states

Formal

Bug-Hunting (Semi-Formal)

Simulation

Bug-Hunting workhorse!
Agenda

- Introduction to Bug-Hunting
- Y Swarm
- Z Swarm
- Guidepointing

**Y Swarm**
Skip or only partially analyze state space

| Formal | Bug-Hunting (Semi-Formal) | Simulation |
Y Swarm

• Classic Formal performs thorough state space search
• Search can be configured to skip/partially analyze state space areas
  – Y Swarm mode
  – Not exhaustive, but goes deeper
• Enable massive parallel search
  – Customers utilize 10s-1000s parallel jobs in single session

Default Search
exhaustive analyze

Y Swarm mode

Init

Init

Init

Skip areas

Partially analyze areas
Use Case 1: Stretch Proof Bound

- **Partial State Space Analysis**: Limited formal search
  - Avoids search getting stuck on specific state area by partial analysis

![Graph showing Trace Attempt (cycles) vs. Time (minutes)]

*Classic formal*

Partially colored rings illustrate space areas which are not exhaustively analyzed

© 2016 Cadence Design Systems, Inc. All rights reserved.
Use Case 2: Y Swarm

Example

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Engine</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert</td>
<td>AST_M_overflow</td>
<td>B</td>
<td>24 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_rcmd_overflow</td>
<td>B</td>
<td>24 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_overflow</td>
<td>B</td>
<td>23 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_stable</td>
<td>B</td>
<td>19 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_rcmd_flags</td>
<td>B</td>
<td>16 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_flags</td>
<td>B</td>
<td>13 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_flags_valid</td>
<td>B</td>
<td>8 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_rcmd_flags_valid</td>
<td>B</td>
<td>8 -</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_S_integrity</td>
<td>B</td>
<td>8 -</td>
</tr>
</tbody>
</table>

Y Swarm reaches a bound of 8 cycles.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Engine</th>
<th>Bound</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert</td>
<td>AST_M_overflow</td>
<td>B</td>
<td>24 -</td>
<td>7.5</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_rcmd_overflow</td>
<td>B</td>
<td>24 -</td>
<td>9.5</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_overflow</td>
<td>B</td>
<td>23 -</td>
<td>9.3</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_stable</td>
<td>B</td>
<td>19 -</td>
<td>8.0</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_rcmd_flags</td>
<td>B</td>
<td>16 -</td>
<td>9.3</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_flags</td>
<td>B</td>
<td>13 -</td>
<td>8.3</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_wcmd_flags_valid</td>
<td>B</td>
<td>8 -</td>
<td>6.6</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_M_rcmd_flags_valid</td>
<td>B</td>
<td>8 -</td>
<td>3.4</td>
</tr>
<tr>
<td>Assert</td>
<td>AST_S_integrity</td>
<td>B</td>
<td>8 - 15</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Quickly find failure using Y Swarm.
## Agenda

<table>
<thead>
<tr>
<th>Introduction to Bug-Hunting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y Swarm</td>
</tr>
<tr>
<td>Z Swarm</td>
</tr>
<tr>
<td>Guidepointing</td>
</tr>
</tbody>
</table>

### Z Swarm
Automatically guide formal search using key states

<table>
<thead>
<tr>
<th>Formal</th>
<th>Bug-Hunting (Semi-Formal)</th>
<th>Simulation</th>
</tr>
</thead>
</table>
Z Swarm

• Use key states to guide formal analysis
• Effective use requires specification of relevant **Key States**
  – Interesting states of the design, directing the tool to search specific state space areas
• Enable massive parallel search
  – Customers utilize 10s-1000s parallel jobs in single session
Key States

- Key states have to be:
  - **Close** to each other, so we can go from one state to the next
  - **Diverse**, so we can hit the same scenario in multiple ways

**Not effective**: Key states are far apart
No advantage over Classic Formal

**Very Effective**: We can jump through key states
Hit much deeper scenarios than Classic Formal
Case study 1 – directing key states toward bugs

• Customer silicon bug:
  – “FSM stuck in state FIFO_FULL”
  – “token counter continues to increment”
  – ast_BUG: assert property ( fsm==ST_FULL #1 fsm==ST_FULL |-> $stable(token_cnt) )

• Results:
  – Try 0: Wrote property and tried to run with BMC ➔ bound of 253 in 24hrs
  – Try 1: Added key states for Z Swarm ➔ fsm==FULL in 48hrs (8067 cycles)
  – Try 2: Further improved and directed key states ➔ fsm==FULL in 2hrs (8k+ cycles)

• Observations:
  – BMC/full proof engines were not effective for target design/depth
  – Generic key states can help Z Swarm to reach very deep targets
  – Directing them towards desired scenario is even more effective
Case Study 2

Options for Resolving the Problem

- Mutate the DUT
  - For example: downsize RAMs and FIFOs
- Over-constrain the DUT’s input stimulus
  - Limit how DUT inputs are driven to a subset of allowed values
- Divide and conquer DUT
  - Chop DUT into pieces
  - Prove properties about each piece separately
- Initial value abstraction
  - Reset design into a mature state (effective OR)

Go Bug Hunting

[Let the machines do the work!]

Strategy: Advance into Deep State Space by Successively Jumping to New Start States

Results

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Testcase1</th>
<th>Testcase2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testcase Name</td>
<td>“Datapath Bridge”</td>
<td>“Request Router”</td>
</tr>
<tr>
<td>Duration</td>
<td>4 weeks</td>
<td>6 weeks</td>
</tr>
<tr>
<td>Total Number of Bugs Found</td>
<td>25</td>
<td>22</td>
</tr>
<tr>
<td>Number of “Deep State-Space” Bugs Found</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Longest trace illustrating DUT bug</td>
<td>415 clock cycles</td>
<td>150 clock cycles</td>
</tr>
<tr>
<td>Extra effort for “Bug Hunting in Deep State-Space”</td>
<td>5 days (See Note 1)</td>
<td>1 day</td>
</tr>
</tbody>
</table>

NOTE 1: includes methodology development
Agenda

- Introduction to Bug-Hunting
- Y Swarm
- Z Swarm
- Guidepointing

Guidepointing
Manually guide analysis using formal and simulation

<table>
<thead>
<tr>
<th>Formal</th>
<th>Bug-Hunting (Semi-Formal)</th>
<th>Simulation</th>
</tr>
</thead>
</table>
Guidedpointing

- Use to tunnel through interesting states using different engines/strategies
  - Similar to Z Swarm, but in a more controlled way
Guidepointing

- Typical to want to verify assertions after hitting a certain scenario
- Several ways to hit scenario: Classic Formal, Z Swarm, Simulation

**Classic Formal**

**Guided Walk**

**Assertion Driven Simulation**

Use formal around target cover
Happy Hunting