Efficient Approaches for Functional Validation of SoC Designs Using High-Level Specifications

Mingsong Chen

Shanghai Key Laboratory of Trustworthy Computing
Software Engineering Institute, East China Normal University

October 14, 2013
Outline

- Motivation

- Research Work
  - Automatic Validation of SoC Specifications
    - Modeling of SoC specifications
    - Basic Idea of Our Approach
  - Efficient Test Generation using Learning Methods
    - Novel property clustering approaches
    - Decision ordering based learning techniques
    - Property decomposition approaches
  - Automated Reuse using Validation Refinement Techniques

- Conclusion
Functional validation is a major bottleneck during SoC development! (up to 70% of time and resources are used)
SoC Design and Validation Flow

Specification Validation

Specification
*TLM/UML*

SW/HW Partitioning

HW

SW

Implementation

VHDL/Verilog

C/C++

Implementation Validation
Challenges and Opportunities

Potential Improvements

- **Current Approach**
  \[ F_{\text{spec}} \cdot T_{\text{spec}} + (F_{\text{spec}} + F_{\text{imp}}) \cdot T_{\text{imp}} \]

- **Our Approach**
  \[ F_{\text{spec}} \cdot T_{\text{spec}} / \alpha + F_{\text{spec}} \cdot T_{\text{imp}} / \beta + F_{\text{imp}} \cdot T_{\text{imp}} \]

Minimize: \[ F_{\text{spec}} \cdot T_{\text{spec}} + (F_{\text{spec}} + F_{\text{imp}}) \cdot T_{\text{imp}} \]

Subject to:

\[ F_{\text{spec}} + F_{\text{imp}} = F_{\text{Total}} \]

\[ T_{\text{spec}} << T_{\text{imp}} \]

\[ F_{\text{spec}} > F_{\text{imp}} \]
Our Approach

**Coverage Model** (Fault Models)

**SoC Architecture** (System-level Models)

**Property Generation**

**Specification Validation**

**Assertion Generation**

**Propert Clustering**

**Test Generation**

**TLM Tests**

**RTL Tests**

**SoC Design** (RTL Implementation)

**Assertion Refinement**

**SVAs**

**Test Refinement**

**TLM Assertions**

**Automatic Validation of SoC Specifications**

**Efficient Test Generation**

**Validation Reuse**
Outline

- Motivation

- Research Work
  - Automatic Validation of SoC Specifications
    - Test Generation using Model Checking
    - Basic Idea of Our Approach
  - Efficient Test Generation using Learning Methods
    - Novel property clustering approaches
    - Decision ordering based learning techniques
    - Property decomposition approaches
  - Automated Reuse using Validation Refinement Techniques

- Conclusion
Test Generation using Model Checking

- **Model Checking (MC)**
  - SoC formal models in temporal specification language, e.g., SMV
  - Desired behaviors in temporal logic properties, e.g., LTL
  - Property falsification leads to counterexamples (tests)

- **Test Generation**
  - Generate a counterexample: sequence of variable assignments

**Problem**: Test generation is very costly or not possible in many scenarios -- in the presence of complex SoCs and/or complex properties.

**Approach**: Exploit learning to reduce validation complexity
- Reduction of TG time & memory requirements
- Enables test generation in complex scenarios
The safety property $P$ is valid up to cycle $k$ iff $\Omega(k)$ is not satisfiable.

$$\Omega(k) = I(S_0) \land \bigwedge_{i=0}^{k-1} R(S_i, S_{i+1}) \land \bigvee_{i=0}^{k} \neg P(s_i)$$

If $\Omega(k)$ is satisfiable, then we can get an assignment which can be translated to a test.
Same Property but Different Bounds

The minimal bound is $k$:

\[ \text{Save: } \Delta p_1^2 + \Delta p_1^3 + \ldots + \Delta p_1^{k-1} + \ldots + \Delta p_1^k \]

O. Strichman. Pruning Techniques for the SAT-Based Bounded Model Checking Problems. CHARME, 2001
Same Design, Different Properties

**Benefit:**

Original: **Red** + **Blue** + **Green**

Now: **Red** + (**Blue** − Δblue) + (**Green** − Δgreen)

Save: Δblue + Δgreen
Outline

- Motivation

- Research Work
  - Automatic Validation of SoC Specifications
    - Test Generation using Model Checking
    - Coverage-driven property generation
  - Efficient Test Generation using Learning Methods
    - Novel property clustering approaches
    - Decision ordering based learning techniques
    - Property decomposition approaches
  - Automated Reuse using Validation Refinement Techniques

- Conclusion
Property Clustering

- Clustering properties is to exploit the structural and behavior similarity and maximize the validation reuse

- Property clustering methods:
  - Based on structural similarity
  - Based on textual similarity
  - Based on Influence (Cone of Influence)
  - Based on CNF intersections

Outline

- Motivation
- Research Contributions
  - Automatic Validation of SoC Specifications
    - Test Generation using Model Checking
    - Coverage-driven property generation
  - Efficient Test Generation using Learning Methods
    - Novel property clustering approaches
    - Decision ordering based learning techniques
    - Property decomposition approaches
  - Automated Reuse using Validation Refinement Techniques
- Conclusion
The essence of SAT problem is to find a satisfiable assignment for a Boolean formula.

A wise decision ordering can quickly locate the true assignment.

- Bit value ordering
- Variable Ordering

Best decision: \( \neg x, \neg z \)
Two Similar SAT Problems

SAT 1

SAT 2

Ordering: a, a’, b, b’, c, c’

Ordering: a, a’, b, b’, c, c’

Without Learning, 7 conflicts in SAT2.
Learning: Bit Value Ordering

Ordering: a, a’, b, b’, c, c’

Ordering: a, a’, b’, b, c’, c

With bit value learning, 4 conflicts in SAT2.
With bit value + variable order learning, 1 conflict in SAT2.
Our method — An Example with 3 properties

<table>
<thead>
<tr>
<th>VarStat</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] V</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>[1] V</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

P1: $a=0, b=0, c=1, d=1$

<table>
<thead>
<tr>
<th>VarStat</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] V</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>[1] V</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

P2: $a=0, b=0, c=1, d=0$

<table>
<thead>
<tr>
<th>VarStat</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] V</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>[1] V</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

P3: $a=0, b=0, c=1, d=?$

- $\text{score}(a) \uparrow$, $\text{score}(a') \uparrow$
- $\text{score}(b) \uparrow$, $\text{score}(b') \uparrow$
- $\text{score}(c) \uparrow$, $\text{score}(c') \uparrow$

**Approach:** Using the statistics of the counterexamples when checking the properties in a cluster
- Count of values $\Rightarrow$ bit value ordering
- Variance of counts of two literals $\Rightarrow$ variable ordering


Case Study 1: MIPS Processor

The Architecture

MIPS Processor
- 20 nodes
- 24 edges
- 91 instructions
## Case Study 1: MIPS Processor

- For each function unit (ALU, DIV, FADD and MUL) in the pipelined processor. We generate 4 properties.

<table>
<thead>
<tr>
<th>Property (test)</th>
<th>zChaff (sec)</th>
<th>Clustering</th>
<th>Speedup (over zChaff)</th>
<th>Decision Ordering</th>
<th>Speedup (over Clustering)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>23.20</td>
<td>23.20</td>
<td>1</td>
<td>23.20</td>
<td>1</td>
</tr>
<tr>
<td>P1</td>
<td>20.73</td>
<td>2.74</td>
<td>7.57</td>
<td>0.18</td>
<td>15.22</td>
</tr>
<tr>
<td>P2</td>
<td>21.33</td>
<td>3.01</td>
<td>7.09</td>
<td>0.15</td>
<td>20.07</td>
</tr>
<tr>
<td>P3</td>
<td>18.03</td>
<td>2.70</td>
<td>6.68</td>
<td>0.29</td>
<td>9.31</td>
</tr>
<tr>
<td>DIV</td>
<td>18.78</td>
<td>18.78</td>
<td>1</td>
<td>18.78</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>23.55</td>
<td>2.72</td>
<td>8.66</td>
<td>0.13</td>
<td>20.92</td>
</tr>
<tr>
<td>P5</td>
<td>18.31</td>
<td>3.60</td>
<td>5.09</td>
<td>0.14</td>
<td>25.71</td>
</tr>
<tr>
<td>P6</td>
<td>18.11</td>
<td>3.72</td>
<td>4.87</td>
<td>0.18</td>
<td>20.67</td>
</tr>
<tr>
<td>FADD</td>
<td>22.90</td>
<td>22.90</td>
<td>1</td>
<td>22.90</td>
<td>1</td>
</tr>
<tr>
<td>P7</td>
<td>16.95</td>
<td>4.46</td>
<td>3.80</td>
<td>0.23</td>
<td>19.39</td>
</tr>
<tr>
<td>P8</td>
<td>18.89</td>
<td>2.71</td>
<td>6.97</td>
<td>0.16</td>
<td>16.94</td>
</tr>
<tr>
<td>P9</td>
<td>19.80</td>
<td>4.70</td>
<td>4.21</td>
<td>0.39</td>
<td>12.05</td>
</tr>
<tr>
<td>MUL</td>
<td>64.21</td>
<td>64.21</td>
<td>1</td>
<td>64.21</td>
<td>1</td>
</tr>
<tr>
<td>P10</td>
<td>59.15</td>
<td>3.36</td>
<td>17.60</td>
<td>0.24</td>
<td>14.00</td>
</tr>
<tr>
<td>P11</td>
<td>59.65</td>
<td>3.85</td>
<td>15.49</td>
<td>0.45</td>
<td>8.56</td>
</tr>
<tr>
<td>P12</td>
<td>73.98</td>
<td>6.28</td>
<td>11.78</td>
<td>0.18</td>
<td>34.89</td>
</tr>
</tbody>
</table>
Case Study 1: MIPS Processor

Test generation time is significantly improved
- Drastic reduction of conflict clauses
- Drastic reduction in number of implications
Case Study 2: OSES

VerifyOrd → Order → AddOrderForm → GetNewOrd → CheckLimitPrice → Market → Limit → Limit Buy → Trade_N → Update_orderDB_

UpdateM → UpdateStockHolderDB_ → UpdateStockDB → UpdateOrderDB_ → Trade_ → Trade_S → UpdateSHolderDB_ → UpdateOrderDB_ → UpdateStockDB 

End → Order Error → Get Order Result → Trade → Marker → Market → CheckLimitPrice 

Diagram showing the flow of events and processes in the OSES system.
This case study is a on-line stock exchange system. The activity diagram consists of 27 activities, 29 transitions and 18 key paths.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Size</th>
<th>zChaff</th>
<th>Clustering</th>
<th>Speedup (over zChaff)</th>
<th>Decision Ordering</th>
<th>Speedup (over Clustering)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>3</td>
<td>1.18</td>
<td>2.18</td>
<td>0.54</td>
<td>0.70</td>
<td>3.11</td>
</tr>
<tr>
<td>C2</td>
<td>4</td>
<td>14.53</td>
<td>9.53</td>
<td>1.52</td>
<td>0.78</td>
<td>12.22</td>
</tr>
<tr>
<td>C3</td>
<td>8</td>
<td>375.91</td>
<td>170.06</td>
<td>2.21</td>
<td>36.19</td>
<td>4.70</td>
</tr>
<tr>
<td>C4</td>
<td>4</td>
<td>12.98</td>
<td>8.33</td>
<td>1.56</td>
<td>1.24</td>
<td>6.72</td>
</tr>
<tr>
<td>C5</td>
<td>4</td>
<td>7.13</td>
<td>16.88</td>
<td>0.42</td>
<td>1.02</td>
<td>16.55</td>
</tr>
<tr>
<td>C6</td>
<td>8</td>
<td>720.13</td>
<td>474.68</td>
<td>1.52</td>
<td>28.60</td>
<td>16.60</td>
</tr>
<tr>
<td>C7</td>
<td>4</td>
<td>10.80</td>
<td>24.55</td>
<td>0.44</td>
<td>1.95</td>
<td>12.59</td>
</tr>
<tr>
<td>C8</td>
<td>8</td>
<td>656.95</td>
<td>321.14</td>
<td>2.05</td>
<td>77.65</td>
<td>4.14</td>
</tr>
<tr>
<td>C9</td>
<td>8</td>
<td>248.17</td>
<td>82.42</td>
<td>3.01</td>
<td>37.93</td>
<td>2.17</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>227.53</td>
<td>123.21</td>
<td>1.85</td>
<td>20.67</td>
<td>5.97</td>
</tr>
</tbody>
</table>
Outline

- Motivation
- Research Contributions

  - Automatic Validation of SoC Specifications
    - Test Generation using Model Checking
    - Basic Idea of Our Approach

  - Efficient Test Generation using Learning Methods
    - Novel property clustering approaches
    - Decision ordering based learning techniques
    - Property decomposition approaches

  - Automated Reuse using Validation Refinement Techniques

- Conclusion
Property Decomposition Techniques

Drawback: Hard to automate


Spatial Decomposition

Learning from P1 can reduce the Time(P)?
Temporal Decomposition

Cause effect relation: \( e_1 \rightarrow e_2 \quad e_3 \rightarrow e_4 \quad e_5 \rightarrow e_6 \)

Happen before relation: \( e_1 < e_3 < e_4 < e_5 < e_2 < e_6 \)
Temporal Decomposition

![Diagram with event and cause-effect relationships]

\[ \neg F(e_1) \rightarrow \neg F(e_3) \rightarrow \neg F(e_7) \rightarrow \neg F(e_9) \]
We generated 6 properties based on interaction faults on various function unit (ALU, DIV, FADD and MUL), which cannot handled by temporal decomposition.

<table>
<thead>
<tr>
<th>Property (test)</th>
<th>zChaff (sec)</th>
<th>Num. of Clusters</th>
<th>Num. of Sub-props</th>
<th>Spatial (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>127.52</td>
<td>3</td>
<td>2</td>
<td>49.41</td>
<td>2.58</td>
</tr>
<tr>
<td>P2</td>
<td>49.24</td>
<td>3</td>
<td>2</td>
<td>15.73</td>
<td>3.13</td>
</tr>
<tr>
<td>P3</td>
<td>9.18</td>
<td>2</td>
<td>1</td>
<td>4.99</td>
<td>1.84</td>
</tr>
<tr>
<td>P4</td>
<td>13.78</td>
<td>2</td>
<td>1</td>
<td>7.28</td>
<td>1.89</td>
</tr>
<tr>
<td>P5</td>
<td>31.63</td>
<td>3</td>
<td>2</td>
<td>12.74</td>
<td>2.48</td>
</tr>
<tr>
<td>P6</td>
<td>120.72</td>
<td>3</td>
<td>2</td>
<td>54.21</td>
<td>2.23</td>
</tr>
</tbody>
</table>

**Speedup:** 1.84-3.13 times
Case Study 2: OSES

This case study is a on-line stock exchange system. The activity diagram consists of 27 activities, 29 transitions and 18 key paths.

<table>
<thead>
<tr>
<th>Property</th>
<th>zChaff (sec)</th>
<th>Bound</th>
<th>Num. of Sub- properties</th>
<th>Temporal (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>25.99</td>
<td>8</td>
<td>3</td>
<td>0.78</td>
<td>33.32</td>
</tr>
<tr>
<td>P2</td>
<td>48.99</td>
<td>10</td>
<td>4</td>
<td>2.69</td>
<td>18.21</td>
</tr>
<tr>
<td>P3</td>
<td>39.67</td>
<td>11</td>
<td>5</td>
<td>3.45</td>
<td>11.50</td>
</tr>
<tr>
<td>P4</td>
<td>247.26</td>
<td>11</td>
<td>5</td>
<td>22.46</td>
<td>11.01</td>
</tr>
<tr>
<td>P5</td>
<td>160.73</td>
<td>11</td>
<td>5</td>
<td>15.68</td>
<td>10.25</td>
</tr>
<tr>
<td>P6</td>
<td>97.54</td>
<td>11</td>
<td>4</td>
<td>1.56</td>
<td>62.53</td>
</tr>
<tr>
<td>P7</td>
<td>31.39</td>
<td>10</td>
<td>4</td>
<td>12.31</td>
<td>2.55</td>
</tr>
<tr>
<td>P8</td>
<td>161.74</td>
<td>11</td>
<td>4</td>
<td>12.62</td>
<td>12.82</td>
</tr>
<tr>
<td>P9</td>
<td>142.91</td>
<td>10</td>
<td>4</td>
<td>17.57</td>
<td>8.13</td>
</tr>
<tr>
<td>P10</td>
<td>33.77</td>
<td>10</td>
<td>4</td>
<td>1.76</td>
<td>19.19</td>
</tr>
</tbody>
</table>

**Speedup: 3-63 times**
Outline

- Motivation
- Research Contributions
  - Automatic Validation of SoC Specifications
    - Test Generation using Model Checking
    - Basic Idea of Our Approach
  - Efficient Test Generation using Learning Methods
    - Novel property clustering approaches
    - Decision ordering based learning techniques
    - Property decomposition approaches
  - Automated Reuse using Validation Refinement Techniques
- Conclusion
SoC Design and Validation Flow

Specification Validation

Specification

TLM/UML

VHDL/Verilog

C/C++

Implementation

SW/HW Partitioning

HW

SW

Test/Assertion Reuse

Implementation Validation
Overview of Our Prototype Tool

Formal Model Generation
- SystemC TLM Specification
  - TLM2SMV
  - Fault Models
- SMV Specification
  - Model Checker (SMV)
  - Properties

TLM Processing
- TLM Tests
  - TLM-Test-Gen
  - TLM Assertions

RTL Processing
- Validation Refinement Specification
  - TLM2RTL
- Manual
- Automatic
  - Simulator
  - RTL Implementation

Transformation of a TLM Test to RTL Test

**Test Refinement Specification for Router**

-- Port definition

```
input.data (10) [7:0] data;
```

-- TLM and RTL name binding

```
bit[7:0] head =
{packet_data.payload_size[7:2],
 packet_data.to_chan[1:0]};
```

-- Timing sequence

```
head => data;
```

**RTL Testcase**

```
read_enb_0 = 0;
read_enb_1 = 0;
read_enb_2 = 0;
packet_valid = 0;
reset = 0;
reset = 1;
reset = 0;
packet_valid = 1;
data = 8'b00001001;
data = 8'b00000001;
data = 8'b00000010;
data = 8'b00001010;
read_enb_1 = 1;
read_enb_1 = 0;
$finish;
```

**TLM Testcase**

```
p->to_chan=1;
p->payload_sz=2;
p->payload[0]=1;
p->payload[1]=2;
p->parity=10;
```

**Data Composition**

- Name Transformation
- Use of half clock
- Use of four clocks
Transformation of a TLM Assertion to RTL Assertion

TLM Assertion:
Cover (tmp_packet.to_chan == 1);

Clock Expression
Control Signals
Name mapping

RTL Assertion: Cover Property
(@(posedge clock) ($rose(write_enb[1])) && (data_o_fsm == 2’d1));
Assertion-based Equivalence Checking

Validation Refinement Case Studies

An industry router example

- Generate 108 TLM tests, and 108 RTL corresponding tests
- RTL implementation Coverage

<table>
<thead>
<tr>
<th>Source</th>
<th>Condition</th>
<th>FSM State(Transition)</th>
<th>Toggle</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.5%</td>
<td>76.6%</td>
<td>100% (100%)</td>
<td>76.6%</td>
<td>73.6%</td>
</tr>
</tbody>
</table>

- Found 2 fatal errors in the RTL implementation.
- Assertion-based equivalent.

Alpha AXP Processor

- Generate 212 TLM tests for transaction flow and data.
- After removing redundant tests: 112 TLM tests → 112 RTL tests
- The number of final required tests depend on the length of each test

<table>
<thead>
<tr>
<th>Source</th>
<th>Condition</th>
<th>FSM</th>
<th>Toggle</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.9%</td>
<td>97.0%</td>
<td>NA</td>
<td>70.2%</td>
<td>86.3%</td>
</tr>
</tbody>
</table>

- Assertion-based equivalent.
Conclusion

- Validation is a major bottleneck in SoC design methodology
- Our research focuses on three important challenges
  1. How tests can be automatically generated?
     - Formal modeling of SoC specifications
     - Coverage-directed test generation techniques
  2. How to reduce overall validation effort?
     - Novel property clustering methods
     - Efficient decision ordering approaches
     - Design and Property decomposition techniques
  3. How to reuse validation effort between abstraction levels?
     - Rule-based test translation methods
     - Assertion-based equivalence checking
- Successfully applied on both hardware and software designs
  - Drastical reduction in overall validation effort (10X~100X)
Thank you !