Assertion-Based Functional Consistency Checking Between TLM and RTL Models

Mingsong Chen
Shanghai Key Lab of Trustworthy Computing
East China Normal University

Prabhat Mishra
Computer and Information Science and Engineering
University of Florida, USA

This work was partially supported by NSF CAREER award 0746261, NSF of China 61202103, Open Project of SCERC 2012002, and SHTDP of China 2011AA010101.
Outline

- Introduction
- Related Work
- Assertion-Based Consistency Checking
  - Automatic TLM Assertion Generation
  - Refinement of TLM Assertions/Tests
  - Assertion-Based Functional Consistency Checking
- Experiments
- Conclusions
SoC Validation is becoming a major bottleneck. Up to 70% time and resources are used.
SoC Design and Validation Flow

- Specification Validation
- Specification
  - TLM
- SW/HW Partitioning
- HW
- SW
- Implementation
  - VHDL/Verilog
  - C/C++
- Implementation Validation
- Assertion/Test Reuse
Related Work

- **Transactor-based dynamic verification methods**
  - TLM tests can be used in TLM-RTL co-simulation
  - Based on event order without timing information
  - Assertions applied on TLM designs only

- **PSL-based Verification approaches**
  - Increase the design observability
  - Take advantages of formal techniques

- Few of them investigate the relations of TLM and RTL assertions
Overview of Our Framework

Basic idea: If a TLM test can exercise some TLM assertions, then its RTL counterpart can also activate the corresponding RTL assertions.
Three Issues

怎么做定义 TLM 索引表达式以观察功能场景？
- TLM 故障模型自动索引表达式生成

如何重用 TLM 验证努力？
- TLM 索引/测试细化

如何利用 TLM 和 RTL 索引表达式之间的相关性进行一致性检查？
- 索引基于一致性检查标准
Since we focus on the activation of functional scenarios, we use the following PSL statement pairs to detect whether the sequence P will happen finally.

- Prop1: assert eventually! p;
- Prop2: cover (p);

- Prop1 asserts that the sequence $p$ must “eventually hold strongly" during the simulation.
- Prop2 is used to record the assertion coverage during the simulation by using verification directive “cover”. 
We define a set of fault models. Each fault indicates a required “design behavior” which may be violated during the system design.

- **Transaction data fault model** deals with the possible value assignment for each part of a transaction data.

```cpp
// The second bit of “packet.parity” can be 1.
assert eventually! (packet.parity==2);
cover (packet.parity==2);
```

- **Transaction flow fault model** handles the controls (e.g., if-then-else) along a transaction flow.

```cpp
// The condition packet.to_chan=1 can be true.
assert eventually! (packet.to_chan==1);
cover (packet.to_chan==1);
```
Refinement of TLM Assertions/Tests

TLM design is significantly different from its RTL implementation in port definition, internal structure and timing details.

Packet Structure

<table>
<thead>
<tr>
<th>length</th>
<th>data[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>data[1]</td>
</tr>
<tr>
<td></td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>data[N]</td>
</tr>
<tr>
<td></td>
<td>parity</td>
</tr>
</tbody>
</table>

Header

PAYLOAD

Parity

DUT

channel0

vld_chan_0

read_enb_0

channel1

vld-chan_1

read_enb_1

channel2

vld-chan_2

read_enb_2
Refinement of TLM Assertions/Tests

We developed the **Assertion Refinement Specification (ARS)** which contains the rules to guide the assertion refinement. Generally an ARS contains two parts:

- **Symbol Mapping** specifies the name and type mapping between TLM variables and RTL signals.

- **Assertion Refinement Rules** specify control signals and timing information for RTL assertions.

**Symbol Mapping**

```
SYMBOL_MAPPING
  bit[1:0] addr = tmp_packet.to_chan;
  ........
END_SYMBOL_MAPPING
```

**Assertion Refinement Rules**

```
ASSERTION_SPEC
  `set_clock (posedge clock);
  ........
  `control tmp_packet.to_chan
     @ $rose(packet_valid);
  ........
END_ASSERTION_SPEC
```
Refinement of TLM Assertions/Tests

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

Clock Expression
Control Signals

RTL Assertion: Cover Property
(@(posedge clock) ($rose(packet_valid))) && (addr == 2’d1));
Refinement of TLM Assertions/Tests

TLM Test

- 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 relation
  @head   packet_valid = 0'b1;

- Test translation
  head => data;

RTL Test

read_enb_0 = 0;
read_enb_1 = 0;
read_enb_2 = 0;
packet_valid = 0;
reset = 0;
#5  reset = 1;
#20 reset = 0;

#5  packet_valid = 1;
#10 data = 8'b00001001;
#10 data = 8'b00000001;
#10 data = 8'b00000010;
#10 packet_valid = 0;
#10 packet_valid = 0;
#10  read_enb_1=1;
#40  read_enb_1=0;
$finish;

-- 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 relation
@head   packet_valid = 0'b1;

-- Test translation
head => data;

RTL Test

reset = 0;
#5  reset = 1;
#20 reset = 0;

#5  packet_valid = 1;
#10 data = 8'b00001001;
#10 data = 8'b00000001;
#10 data = 8'b00000010;
#10 packet_valid = 0;
#10  read_enb_1=1;
#40  read_enb_1=0;
$finish;
Assertion-based Functional Consistency Checking

Since an assertion activation means that a specific functional scenario is covered, the coverage of the assertions indicates the **adequacy** of the functional validation.

Given a TLM specification $T$ and its RTL implementation $R$, by applying TLM tests on $T$ and RTL tests on $R$, the assertion coverage can be calculated as:

$$
T_{\text{coverage}} = \frac{\text{# of exercised TLM assertions}}{\text{Total number of TLM assertions}}
$$

$$
R_{\text{coverage}} = \frac{\text{# of exercised RTL assertions}}{\text{Total number of RTL assertions}}
$$
Assertion-based Functional Consistency Checking

- For a TLM test and its refined RTL version, when applying them on the TLM and RTL designs
  - **Assertion consistent:** For each test, the activated TLM assertions is a **subset** of the corresponding RTL assertions.
  - **Strongly assertion consistent:** Besides assertion consistency, for each test, it requires that the activation order of assertions is the same.

\[ t \text{ and } t' \text{ are assertion consistent, but they are not strongly assertion consistent.} \]
Case Study 1: A Router Example

- The main function of the router is to parse incoming packets and send them to target slaves.
- By using our tool, 59 TLM assertions are generated.
  - 55 from data fault model
  - 4 from flow fault model
- We select 59 TLM tests from 1000 random TLM tests which can achieve 100% TLM assertion coverage.
- To improve RTL coverage, we derive 2 more directed tests (FIFO overflow + reset).
Case Study 1: A Router Example

- RTL Coverage using Synopsys VCS Discovery Visualization Environment (DVE) tool

<table>
<thead>
<tr>
<th>Module</th>
<th>Line</th>
<th>Toggle</th>
<th>FSM</th>
<th>Condition</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>fifo</td>
<td>76.6%</td>
<td>100%</td>
<td>NA</td>
<td>100%</td>
<td>NA</td>
</tr>
<tr>
<td>Port_fsm</td>
<td>95.92%</td>
<td>100%</td>
<td>87.5%</td>
<td>71.88%</td>
<td>100%</td>
</tr>
<tr>
<td>router</td>
<td>100%</td>
<td>100%</td>
<td>NA</td>
<td>NA</td>
<td>100%</td>
</tr>
</tbody>
</table>

- The **61** directed RTL tests only need **4 seconds**. Running **10000** random tests needs **1057 seconds**.
- Found **1 fatal error** in the RTL implementation.
  - Try to send the packet to the 4\(^{th}\) slave, i.e., to_chan = 3.
- After correcting the error, the TLM and RTL models are **strongly assertion equivalent**.
Case Study 2: An Alpha AXP Processor

- By using our tool, 163 TLM assertions are generated.
  - 117 from data fault model
  - 46 from flow fault model
- To achieve 100% TLM assertion coverage, 163 TLM tests are selected from 3000 random TLM tests.
Case Study 2: An Alpha AXP Processor

**RTL implementation Coverage** of 163 directed tests using Synopsys DVE tool.

<table>
<thead>
<tr>
<th>Module</th>
<th>Line</th>
<th>Toggle</th>
<th>FSM</th>
<th>Condition</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF_stage</td>
<td>100%</td>
<td>68.82%</td>
<td>NA</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>ID_stage</td>
<td>100%</td>
<td>80.00%</td>
<td>60.00%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>EX_stage</td>
<td>100%</td>
<td>52.94%</td>
<td>NA</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>MEM_stage</td>
<td>100%</td>
<td>74.19%</td>
<td>NA</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>WB_stage</td>
<td>100%</td>
<td>78.52%</td>
<td>NA</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>regfile</td>
<td>100%</td>
<td>71.29%</td>
<td>NA</td>
<td><strong>55.56%</strong></td>
<td>100%</td>
</tr>
</tbody>
</table>

- The **163** directed RTL tests only need **15 seconds**. Running 50000 random tests needs **1390 seconds**.
- The TLM and RTL models are **strongly assertion equivalent**.
Conclusion

- Raising the abstraction introduces two challenges
  - Functional inconsistency between abstraction levels
  - Increasing validation efforts

- Our work tries to reuse TLM validation effort to enable RTL validation
  - TLM assertion generation/activation
  - TLM-to-RTL assertion/test refinement
  - TLM-to-RTL functional consistency checking

- Experimental results demonstrate that our approach can improve the design quality and significantly reduce the validation effort.
Questions?

Thank you!