A Systematic Formal Reuse Methodology: From Blocks to SoC Systems

Hao Chen, Yi Sun, Ang Li, Dorry Cao
NVM Solutions Group
Intel Corporation
Agenda

• Introduction
• Reusable FV Testbench Structure
• Integrating FV Collaterals into Emulation
• Real-World Results: a SSD Controller SoC
• Conclusion
SoC Verification – Industry Trends

• SoC verification is full of hard problems:
  • HW/SW interactions at SoC top level
  • 3rd party IP integration at subsystem level
  • Exhaustive corner cases at block level
  • ...

• These drive the continuous evolution of verification strategies/methodologies

[Image: Percentage of Project Time Spent in Verification]

Our Hybrid Verification Strategy

- Goal: To extend our capability to verify larger and more complex SoCs
  - Block-level: Simulation + Formal
  - Cluster-level: Simulation
  - SoC-level/System-level: Simulation + Emulation
Challenges

• Hybrid strategy  Maximize ROI in verification quality

• Challenges:
  – Verification gaps on boundaries
  – Duplicate effort
How to Close the Gap Between Schedule and Productivity?

Solution:

A Systematic formal reuse methodology to enable **code reuse** and **cross proof** from FV and improve interoperability between FV and other platforms
Agenda

• Introduction
• Reusable FV Testbench Structure
• Integrating FV Collaterals into Emulation
• Real-World Results: a SSD Controller SoC
• Conclusion
Principles for Formal Reuse

• **Modularity**
  – Modularized components
  – Easy to reuse in different platforms and projects

• **Usability**
  – Simple and clean interfaces
  – Easy to be integrated by other verification environments

• **Consistency**
  – Follows a consistent, easy-to-communicate form
Complete Formal Testbench

- A complete formal testbench consists of
  1. A Formal Verification Component (FVC)
  2. A formal testbench environment (ENV)
An FVC provides:

- Simple and clean interfaces

Formal Verification Component (FVC)
Formal Verification Component (FVC)

- An FVC provides:
  - Coverage Sample Interface
  - Free Variable Control Interface
  - DUT Ports
  - Interface protocol modeling and checking

- Checking Model
  - SVA Asserts
  - Prediction Models
  - Scoreboard

- End-to-End Constraints
  - SVA Assumes

- FVC Configuration
  - FVC Active/Passive

- FVC Package
  - SV Data Structs
  - SV Functions

- Coverage Model
  - SVA covers
  - Covergroups

- Intf A FBM
  - Intf Asserts
  - Intf Covers

- Intf B FBM
  - Intf Asserts
  - Intf Covers

- Intf C FBM
  - VIP

- DUT Ports
Formal Verification Component (FVC)

• An FVC provides:
  • End-to-end checkers
  • End-to-end constraints
An FVC provides:

- **Coverage Sample Interface**
- **Free Variable Control Interface**
- **DUT Ports**

**Intf A FBM**
- Intf Asserts
- Intf Covers

**Intf B FBM**
- Intf Asserts
- Intf Covers

**Intf C FBM**
- VIP

**FVC Package**
- SV Data Structs
- SV Functions

**Coverage Model**
- SVA covers
- Covergroups

**Checking Model**
- SVA Assumes
- SVA Assumes
- Prediction Models
- Scoreboard

**End-to-End Constraints**
- SVA Assumes

**FVC Configuration**
- FVC Active/Passive

**Functional coverage**

---

13
An FVC provides:

- Coverage Sample Interface
- Free Variable Control Interface
- DUT Ports
- Intf A FBM (Intf Asserts, Intf Covers)
- Intf B FBM (Intf Asserts, Intf Covers)
- Intf C FBM (VIP)

FVC Configuration
- FVC Active/Passive

Checking Model
- SVA Asserts
- Prediction Models
- Scoreboard

End-to-End Constraints
- SVA Assumes

FVC Package
- SV Data Structs
- SV Functions

Coverage Model
- SVA covers
- Covergroups

A package with reusable data structs, functions, properties, etc.
Formal Verification Component (FVC)

- An FVC provides:

- **Active/Passive configuration**

- **Coverage Sample Interface**

- **Free Variable Control Interface**

- **Coverage Model**
  - SVA covers
  - Covergroups

- **Intf A FBM**
  - Intf Asserts
  - Intf Covers

- **Intf B FBM**
  - Intf Asserts
  - Intf Covers

- **Intf C FBM**
  - VIP

- **DUT Ports**

- **Checking Model**
  - SVA Assumes
  - Prediction Models
  - Scoreboard

- **End-to-End Constraints**
  - SVA Assumes

- **FVC Configuration**
  - FVC Active/Passive
FVC Interfaces

- **DUT Ports**
  - Identical to design interfaces
  - Connect FVC to its DUT

- **Coverage Sample Interface**
  - A verification interface that enables coverage reuse
  - Propagate important block-level events to upper layer testbench

- **Free Variable Control Interface**
  - A verification interface that controls FV free variables
  - Free variables are commonly used in FV
  - Extra driver logic in simulation or emulation is needed
Free Variable Control Interface

```vhdl
interface blockA_free_var_ctrl_if
#(
    parameter bit FVC_ACTIVE = 1,
    parameter int NUM_CLIENTS = 8,
    parameter int NUM_CLIENTS_WIDTH = 3
);

logic [NUM_CLIENTS_WIDTH-1:0] client_idx;

if (FVC_ACTIVE == 1) begin
    client_idx_stable: assume property (
        @ (posedge clk) disable iff (!rst_n)
        1 |-> ##1 ($stable(client_idx) && (client_idx < NUM_CLIENTS)))
    end
endinterface
```

- When `FVC_ACTIVE = 1`: this assume is in effect
- When `FVC_ACTIVE = 0`: this assume is disabled
Formal Bus Model (FBM)

- Self-contained and reusable Assertion Based Verification IPs (ABVIPs)
- Bi-directional assertions and covers
- Cross proof at the interface
End-to-End FV Properties

- SVA assumes across multiple interfaces
- End-to-end modeling and SVA asserts
- Free variables in the verif interface
Formal Coverage Model

- Measure verification completeness
- Sampled when corresponding checkers trigger
- Certain events propagated for reuse
Agenda

• Introduction
• Reusable FV Testbench Structure
• Integrating FV Collaterals into Emulation
• Real-World Results: a SSD Controller SoC
• Conclusion
Transactor-Based Emulation

- Emulation enables system-level HW/SW co-verification
- Our transactor-based emulation platform contains:
  - A SystemVerilog testbench synthesized together with the SoC DUT
  - System-level tests in C++ language
  - Transactors that communicate information between the emulator and its host
1. Bind each FVC (with $FVC_{ACTIVE} = 0$) to its DUT
2. Connect each FVC’s coverage sample interface
3. Use a transactor to drive free variables
Agenda

- Introduction
- Reusable FV Testbench Structure
- Integrating FV Collaterals into Emulation
- Real-World Results: a SSD Controller SoC
- Conclusion
## Proof of Concept (PoC) Results

- **PoC Experiment:** Reusing two block FVCs in a SoC emulation platform

### Quantitative Metrics

<table>
<thead>
<tr>
<th></th>
<th>Return on Investment (RoI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block_A</td>
</tr>
<tr>
<td>Line of FV code reused</td>
<td>5392</td>
</tr>
<tr>
<td># Assert/assume props reused</td>
<td>assert: 632, assume: 293</td>
</tr>
<tr>
<td># Failed assert/assume props in emulation</td>
<td>assert: 10, assume: 2</td>
</tr>
<tr>
<td># Block-level cover points reused</td>
<td>82</td>
</tr>
<tr>
<td># System-level inter-block covers added</td>
<td>18</td>
</tr>
<tr>
<td>DUT gate count (percentage)</td>
<td>0.8%</td>
</tr>
<tr>
<td>Cost: FVC gate count (percentage)</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

- **Overall reuse statistics:**
  - ~7000 LoC
  - ~1000 Props
- **Quality improvement (Block-level)**
- **Efficiency improvement (System-level)**
- **Manageable cost**
FVC Quality Improvement:
An Over-constraint Case

• Very important to prove assumptions made at block boundaries
  – Pay attention to your clocks and resets!

```
Timer_decr: assume property (  
  (sync_dn != `0 |-> ##1 sync_dn == $past(sync_dn)-1); 
```

![Diagram showing clock and reset signals with timing notations and block A connections.](image-url)
FVC Quality Improvement: FV Model Synthesized as Intended?

- **Assertion** `Instr_ptr_update`
  - passed in formal
  - failed in emulation

- Root cause: inferred latch!

- Pay attention to tool warnings!

```verilog
Instr_ptr_update: assert property (  
  (decode_fetch |-> ##1 instr_ptr ==  
  $past(instr_ptr_exp));
...

always_comb begin: model_ptr_gen  
  if(!rst_n)  
    instr_ptr_exp = 0;
  else if(decode_fetch) begin  
    if(condition_1)  
      instr_ptr_exp = '1;  
    else if(condition_2)  
      instr_ptr_exp = pending_ptr;  
    else if(condition_3)  
      instr_ptr_exp = fetch_address[13:0];  
  end
end
```
Emulation Efficiency Improvement: A System-level Use Case Cover

- FVCs enable use case coverage:
  - Important block events
  - Inter-block event sequences
  - System-level use cases
Emulation Efficiency Improvement: Debbuggability

• Embedded FVC checkers can greatly reduce debug effort!

Case 1: Reproduce A Post-Silicon Bug in Emulation
  ▪ Created an assertion for the RTL bug
  ▪ Reproduced system failure with the target assertion failed
  ▪ Rapid root cause analysis in Block_B

Case 2: Root Cause An Emulator Tool Bug
  ▪ Random system hanging due to a tool bug
  ▪ Root caused by an assertion in Block_A within a few days
  ▪ Saved weeks of debug time
Agenda

• Introduction
• Reusable FV Testbench Structure
• Integrating FV Collaterals into Emulation
• Real-World Results: a SSD Controller SoC
• Conclusion
Conclusion

• Modern complex SoCs require a greater rigorous verification signoff methodology
  – Simulation
  – Formal
  – Emulation

• Our systematic formal reuse methodology helps to increase productivity
  – Efficiency: avoiding duplicate effort
  – Quality: cross-proof between different platforms
Future Work

• Guidelines on how to create system-level use case covers
• Emulation-friendly scoreboard
• Building FVCs for more IPs/blocks
References


