UVM Layering for Protocol Modeling Using State Pattern

Tony George – Samsung Semiconductor, Inc.
Girish Gupta – Samsung Semiconductor India R&D, Bangalore
Shim Hojun - Samsung Semiconductor Co., Ltd.
Byung C. Yoo - Samsung Semiconductor Co., Ltd.
Layered Protocol and verification

• Layering protocol are similar to Open System Interconnect (OSI) models

• Each layer has its own set of functionality and features which needs to be verified

• Control of the functional flow at each layer and error injection is also part of verification
Expectations from Verification IP

Controllability
- Variety of Stimulus
- Desired Error Injection
- Dynamic Modifiable functionality

Reusability
- Horizontal and vertical reuse
- Minimum Code and avoid redundancy

Scalability
- Scalable to accommodate new features or new protocol layers

Observability
- Data and Control flow tracking
- Easy debugging
Existing Approach – Layered Sequencer

Top Virtual Sequencer

Sequences

Sequencer -> Driver -> Monitor

Pass-thru Sequences

Sequencer -> Driver -> Monitor

Physical Interface

DUT (Design Under Test)

Tests

Higher Protocol Layer Agent

Lower Protocol Layer Agent

TLM port

TLM export/imp
What is state pattern?

- The **state pattern** is a behavioral software design pattern that allows an object to alter its behavior when its internal **state** changes.

Image Source: sourcemaking.com
Proposed Approach using State Pattern

- **Top Virtual Sequencer**
- **Tests**
- **TX Sequencer**
- **RX Sequencer**
- **Higher Protocol Layer Driver**
- **Lower Protocol Layer Driver**
- **Physical Interface**
- **DUT (Design Under Test)**

Dynamically modifiable FSM’s for Layer functionality (State Pattern Design Concept)

- **TLM port**
- **TLM export/ imp**
State Pattern: State Behavior modification

State Controller
- change_state()

Scrambler State
- +Handle()

Abstract Class

Functional States

Scrambler Idle
- +Handle()

Scrambler Initialization
- +Handle()

Scrambler Operation
- +Handle()

scrambler_operation functional state is modified to scrambler_operation__m state
State Pattern : New State insertion

- State Controller
  - change_state()

Abstract Class
- Scrambler State
  - +Handle()

Functional States
- Scrambler Idle
  - +Handle()

- Scrambler Initialization
  - +Handle()

- Scrambler Operation
  - +Handle()

- Scrambler Operation_n
  - +Handle()

New state is inserted after scrambler_operation state
// Abstract class for scrambler state
class scrambler_state extends uvm_object;
`uvm_object_utils(scrambler_state)

uvm_component handle;

virtual task do_action (state_manager i_state_manager);
    //override this task
    endtask

virtual function string get_full_name();
    return handle.get_full_name();
    endfunction

declass
Code Snippet : State Controller

```python
// State Controller
class state_controller extends uvm_component;
 `uvm_component_utils(state_controller)

    //Build_phase
    scrambler_state i_state;
    i_state = idle_scrambler_state::type_id::create();
    i_state.handle = this;

    //run_phase
    i_state.do_action(this);

    virtual task change_state (string state_name);
    //Casting state handle into new state handle
    $cast(i_state,factory.create_object_by_name(state_name,
        get_full_name()))
    i_state.handle = this;
endtask
endclass
```

Building the Idle Scrambler State
Casting the state handle into new state handle
Code Snippet : State Behavior modification

```verbatim
// scrambler LFSR Initialization state
class lfsr_init_state extends scrambler_state;
  `uvm_object_utils(lfsr_init_state)

  virtual task do_action (state_manager i_state_manager);
    //Perform Scrambler LFSR Init state functionality
    ...
    // Move to next state -> MODIFIED LFSR Operation
    i_state_manager.change_state(“mod_lfsr_operation_state);
  
  endtask
endclass

Instead of moving to lfsr_operation_state, moving to modified lfsr_operation_state (Modified state)
```
Code Snippet: New State Insertion

```verilog
// scrambler LFSR Operation state
class lfsr_operation_state extends scrambler_state;
`uvm_object_utils(lfsr_operation_state)

virtual task do_action (state_manager i_state_manager);
//Perform Scrambler LFSR Operation state functionality
...
// Move to next state -> NEW LFSR Operation State
i_state_manager.change_state("new_lfsr_operation_state);
endtask

endclass
```

Instead of moving to lfsr_idle_state, moving to new functional state new_lfsr_operation_state (new state)
Error Injection using Lateral Sequencer

Transaction to be inserted in the normal flow

Wait for a synchronizing event to trigger
Error Injection using Callback

- Extend the base class and write logic for modification of ‘Tr’
- Just like a base class
State Pattern Vs Finite State Machine

• *The State Pattern abstract the states and decouple them from each other*
  - Example: you can easily replace one particular state with another. Yet you will not be happy rewriting all the states when it is time to add a new one and/or a new transition

• *The state machine abstracts the state diagram itself and decouples it from the transition payloads.*
  - Example: To change a particular state, you have to fix the whole diagram
### Observation & Results

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Traditional (Existing approach) VIP</th>
<th>Proposed VIP (Layered State pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs found</td>
<td>35, before design went into silicon</td>
<td>Additional 10 Major bugs and 5 minor bugs found in the design</td>
</tr>
<tr>
<td>Test Scenarios</td>
<td>250</td>
<td>Additional 40 (targeting error scenarios and exception handling)</td>
</tr>
</tbody>
</table>
Conclusion

• The motivation for this paper is to analyze and conclude on a Verification IP Architecture which provides full-fledged control without compromising on the simplicity of model development.

• Dynamically modifiable functionality of all layers along with complex test scenario generation is achieved using this methodology.

• The proposed architecture has been deployed for live verification project on UniPro and PCIe protocols.
Thank You !.