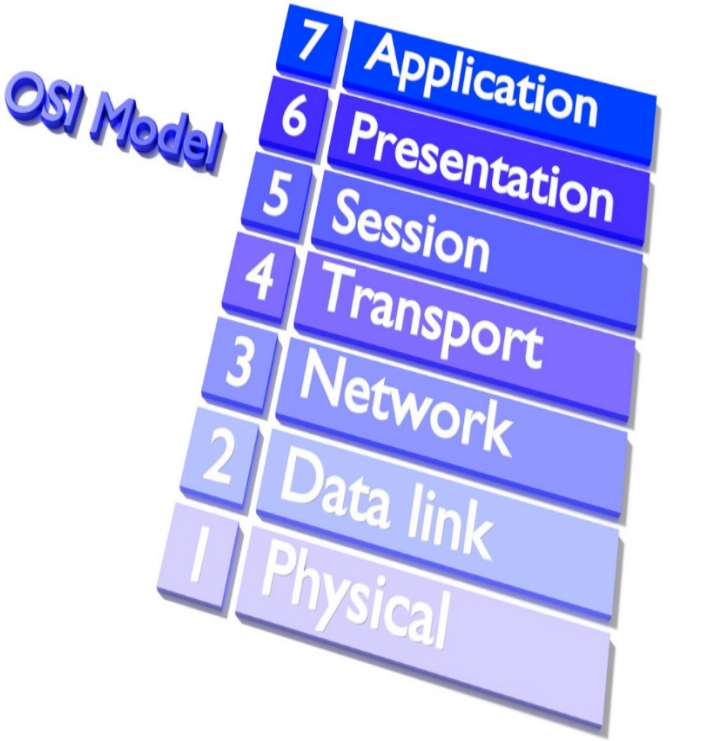
UVM Layering for Protocol Modeling Using State Pattern

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 anc 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

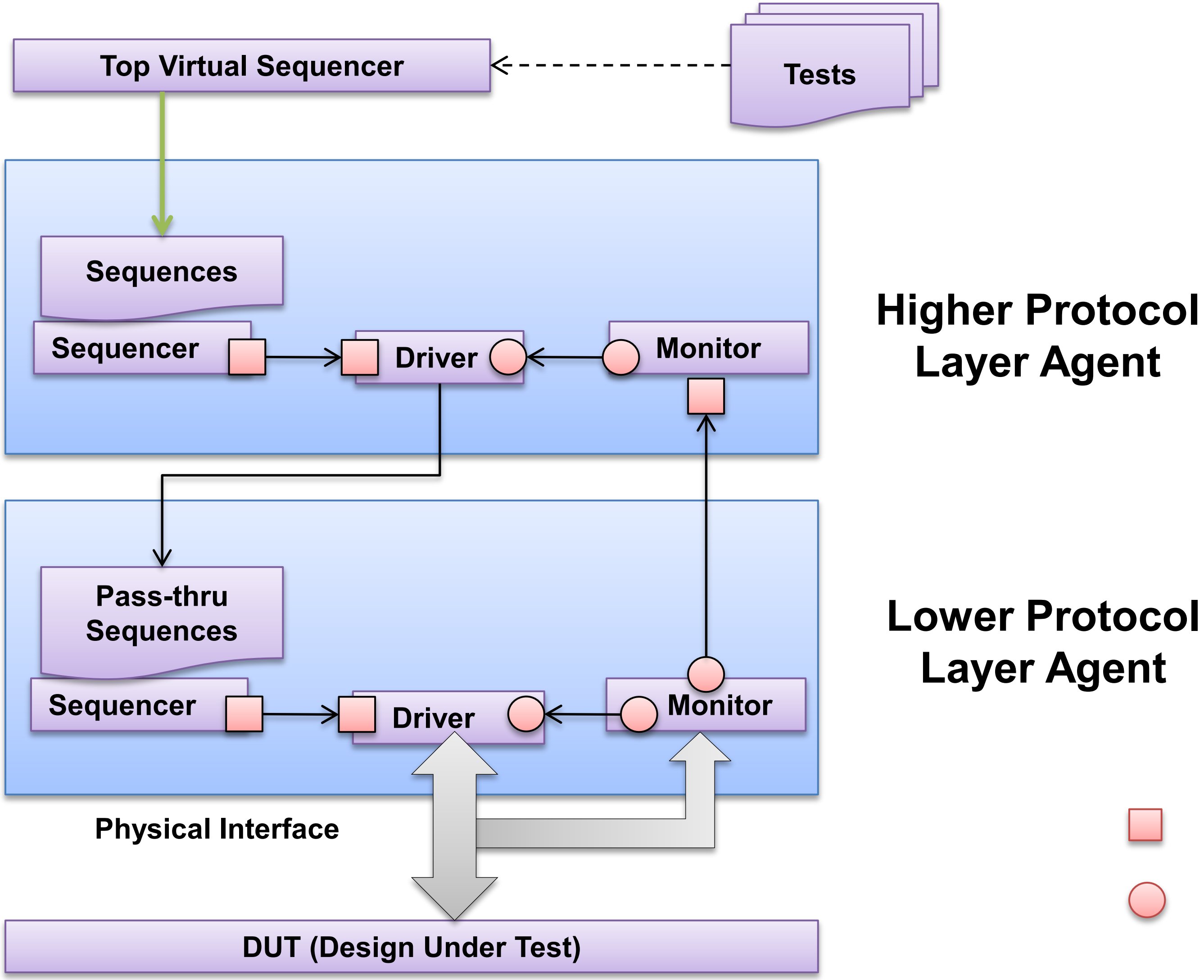
observability

- ✓ Data and Control flow tracking
- ✓ Easy debugging

Scalability

- ✓ Scalable to accommodate new features or new protocol layers

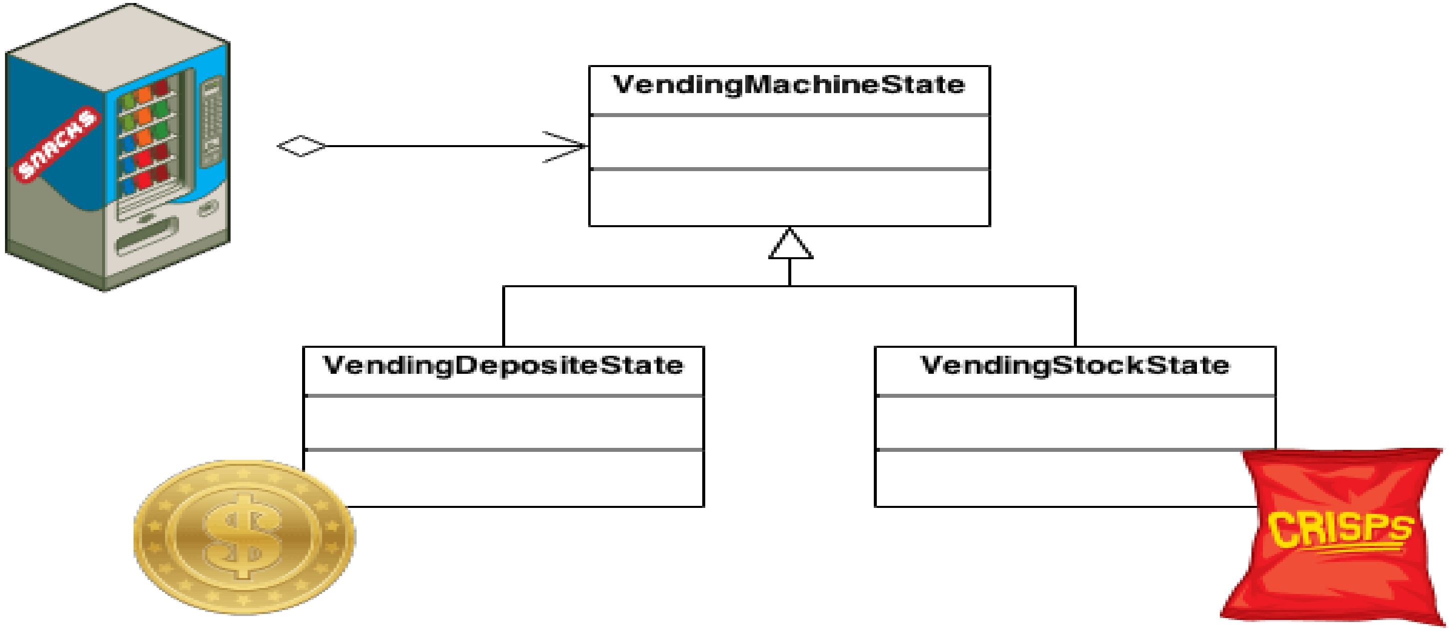
Existing Approach – Layered Sequencer



□	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.



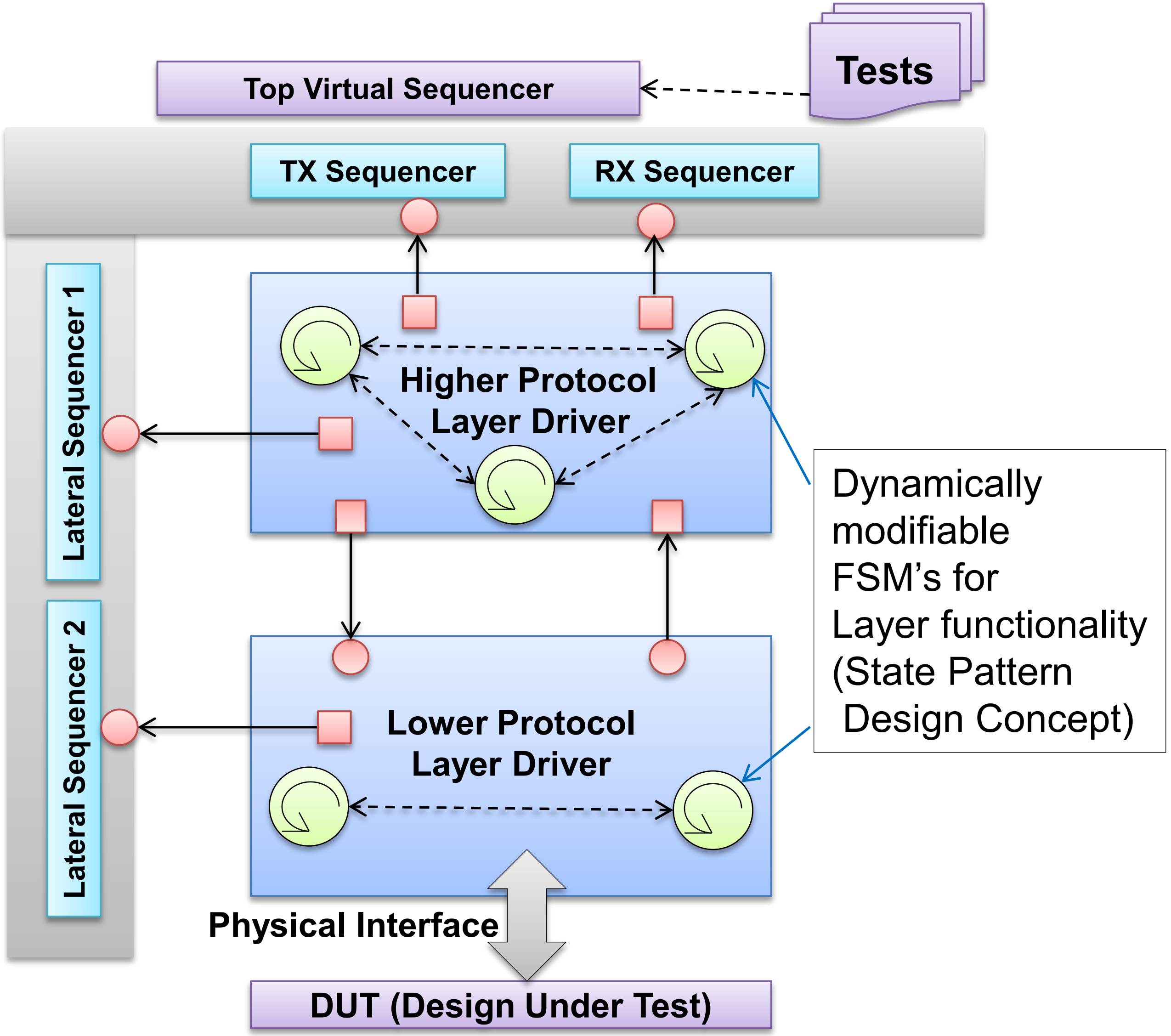
VendingMachineState

VendingDepositeState

VendingStockState

Image Source : sourcemaking.com

Proposed Approach using State Pattern

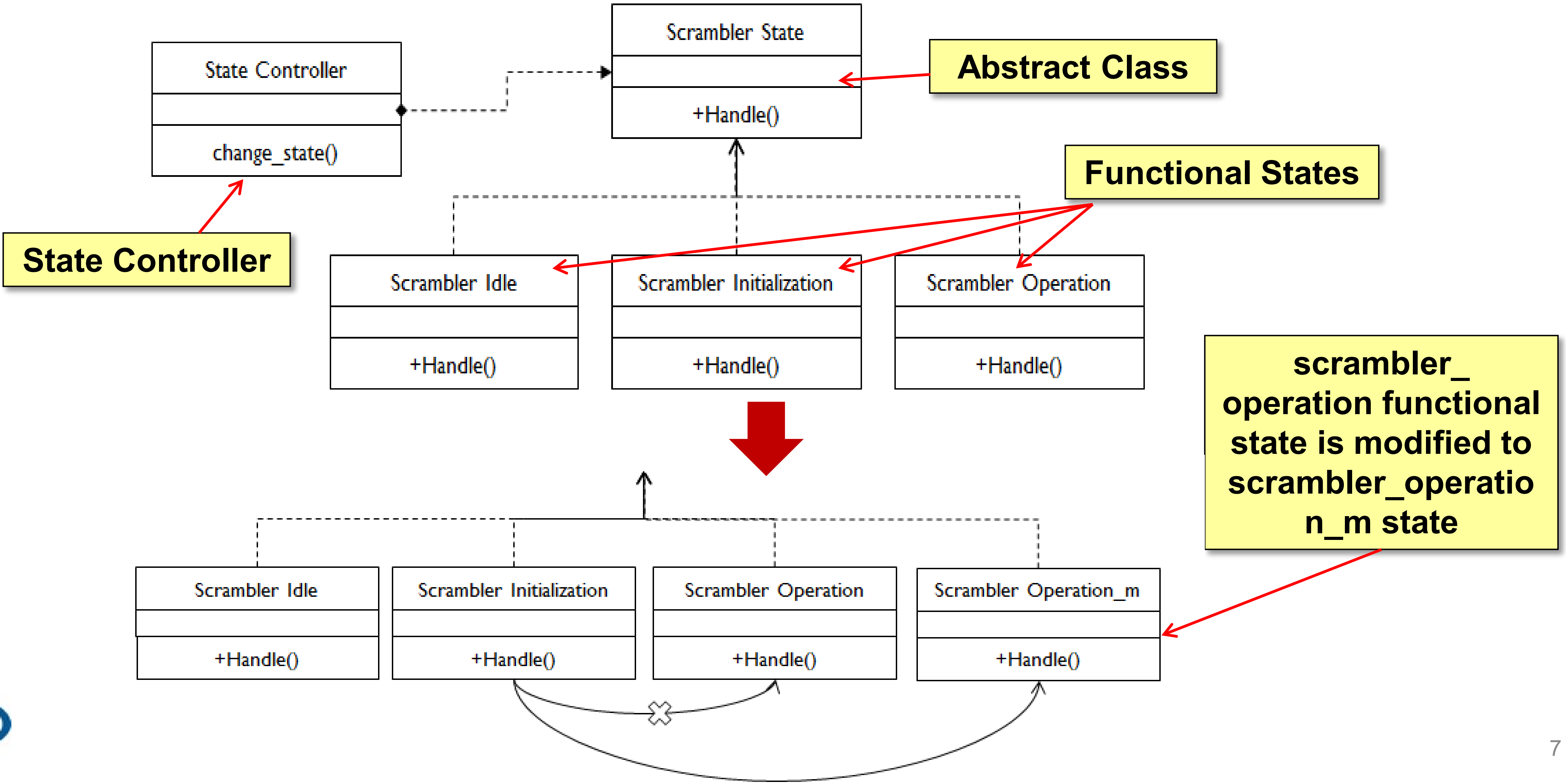


Dynamically modifiable FSM's for Layer functionality (State Pattern Design Concept)

🟥 TLM port

● TLM export/ imp

State Pattern : State Behavior modification



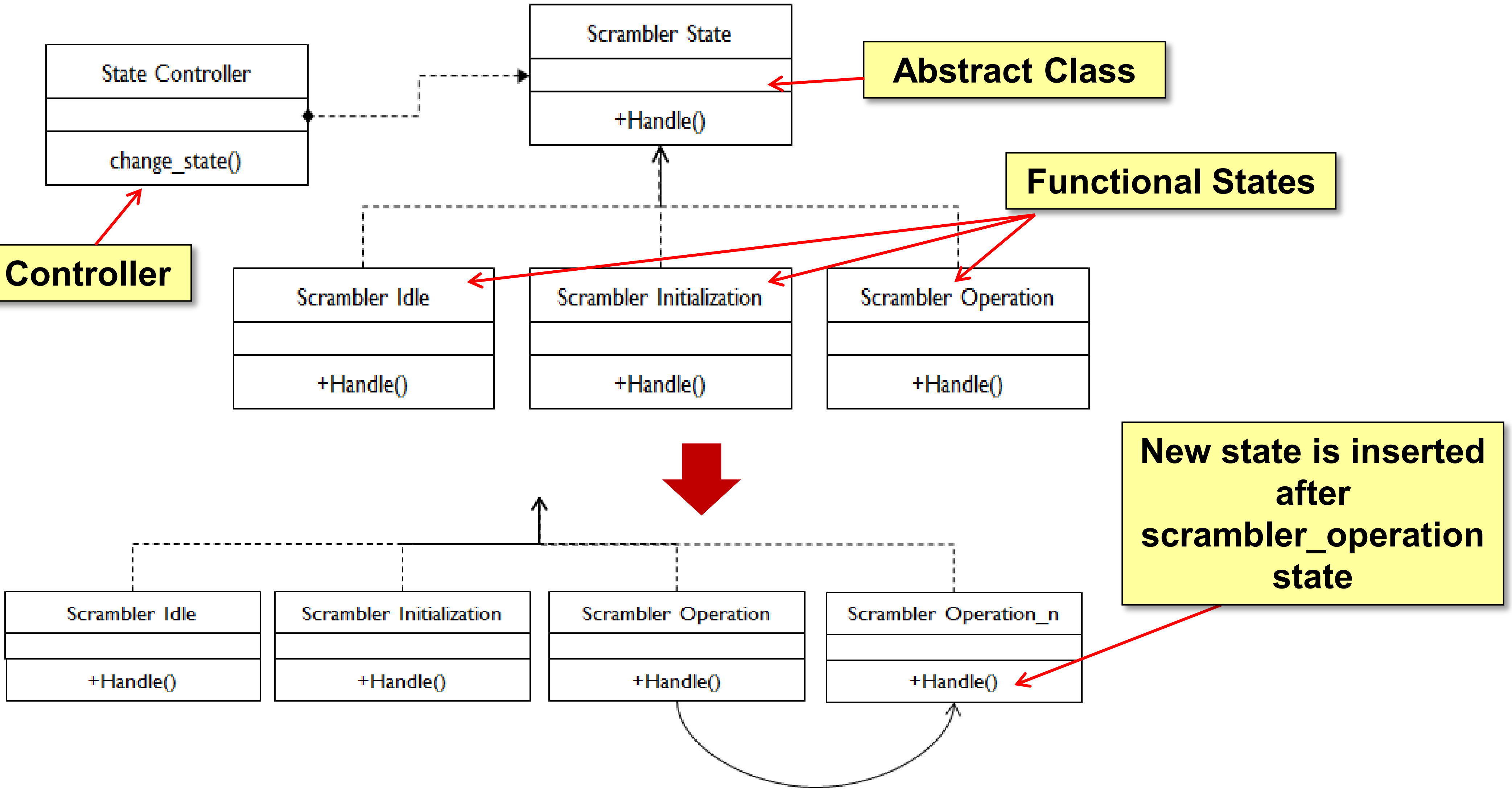
Abstract Class

Functional States

State Controller

scrambler_ operation functional state is modified to scrambler_operation_m state

State Pattern : New State insertion



Abstract Class

Functional States

State Controller

New state is inserted after scrambler_operation state

Code Snippet : Abstract Class

Task to perform the state functionality

Function to get the full name of state, to be used for state overriding

```
// 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

endclass
```

Code Snippet : State Controller

Building the Idle Scrambler State

Casting the state handle into new state handle

```
// 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
```

Code Snippet : State Behavior modification

Instead of moving to lfsr_operation_state, moving to modified lsfr_operation_state (Modified state)

```
// 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
```

Code Snippet : New State Insertion

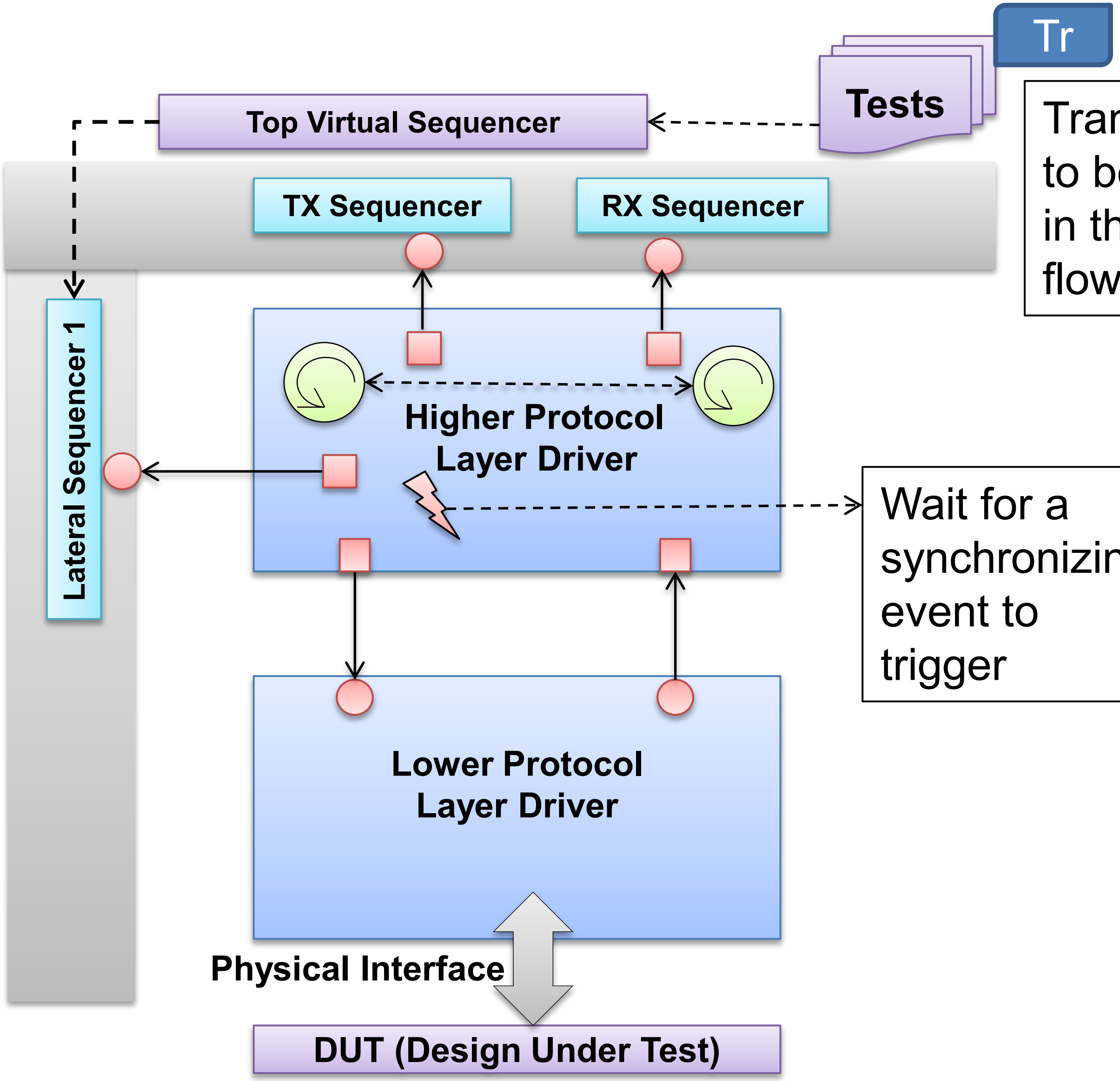
Instead of moving to lfsr_idle_state, moving to new functional state new_lsfr_operation_state (new state)

```
// 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
```

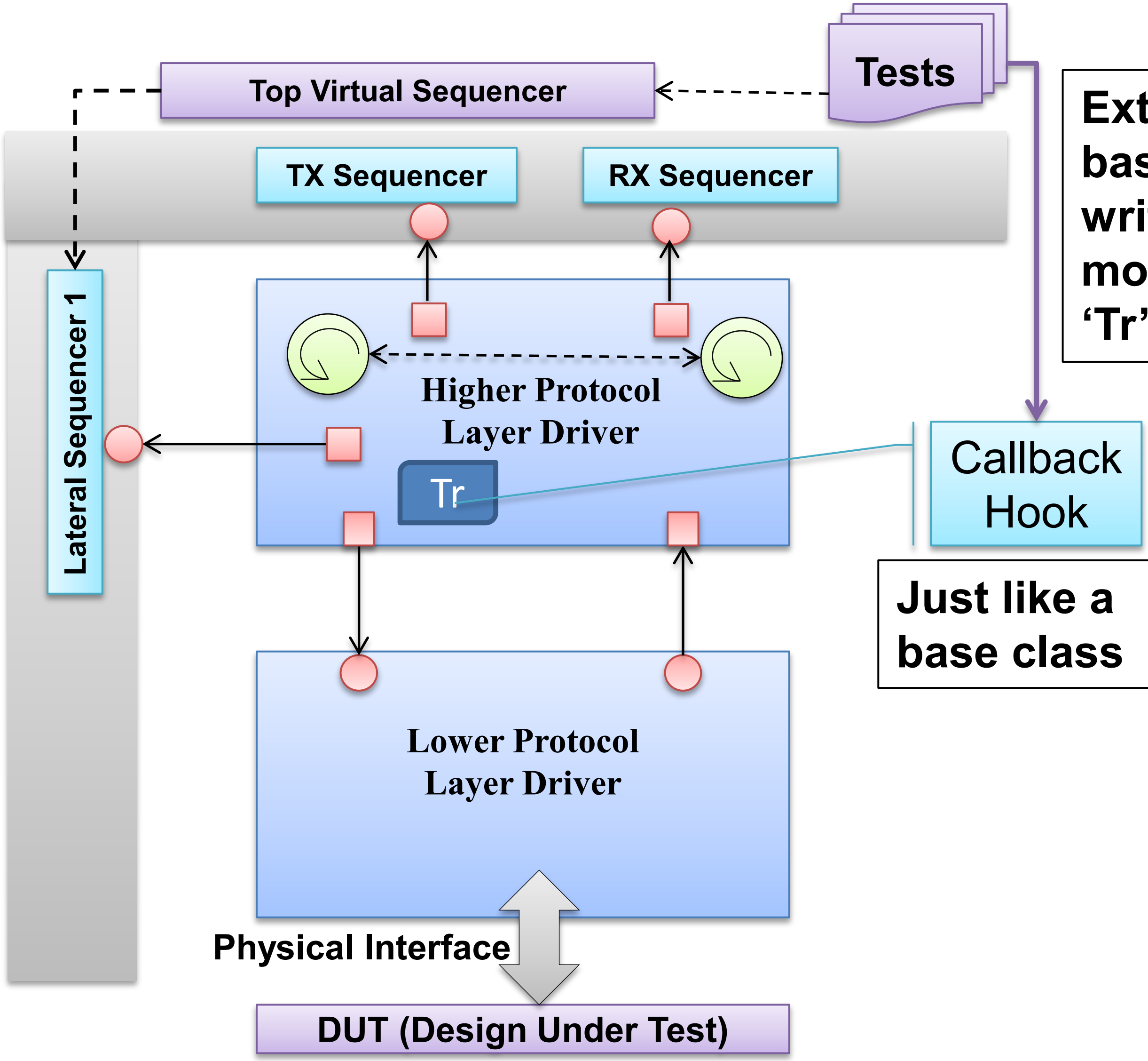
Error Injection using Lateral Sequencer



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

Parameters	*Traditional (Existing approach) VIP*	*Proposed VIP (Layered State pattern)*
Bugs found	35, before design went into silicon	Additional 10 Major bugs and 5 minor bugs found in the design
Test Scenarios	250	Additional 40 (targeting error scenarios and exception handling)

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 !.