Chain of Responsibility Design Pattern for scalable UVM drivers

Chandana K N

Suresh Gandhi S

accellera SYSTEMS INITIATIVE™

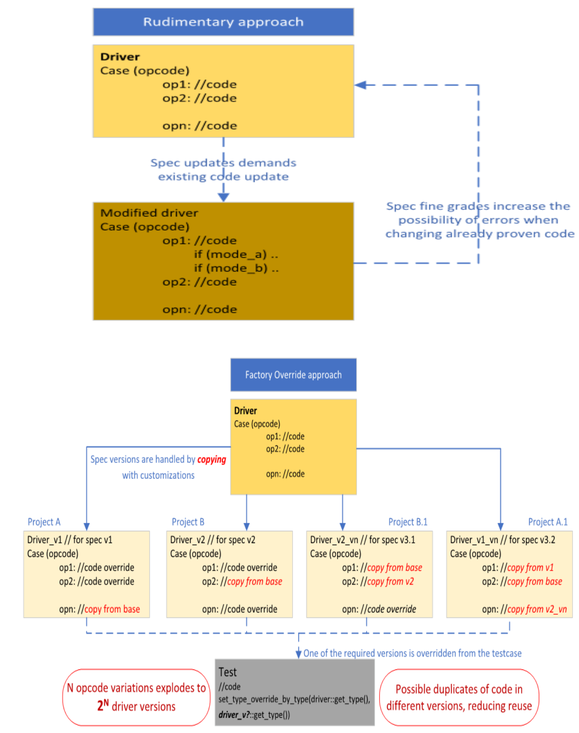
2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Problem Statement - Driver for fluctuating specifications

Specification
- Not fully defined before the IP development phase
- Evolve along with the development

IP
- Ever changing specifications
- Multiple variations before landing on the final version

Testbench
- Catering to spec and IP variations' clutters testbench
- Makes code fragile
- Maintenance time consuming



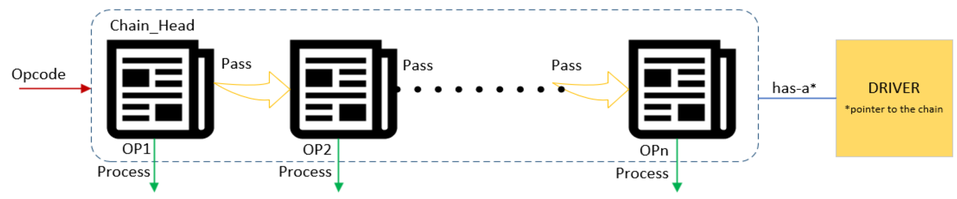
Proposed Methodology - Chain of Responsibility (CoR)

Chain of Responsibility
1. Behavioral design pattern in object-oriented design
2. Structured way to handle a sequence of operations or commands

Handlers
1. Source of command objects
2. Logic that defines the command it can process
3. Mechanism to pass down the command to the next handler in the chain

Chain
1. Series of handlers connected together
2. Connected in the testcase
3. Handle is provided in the driver

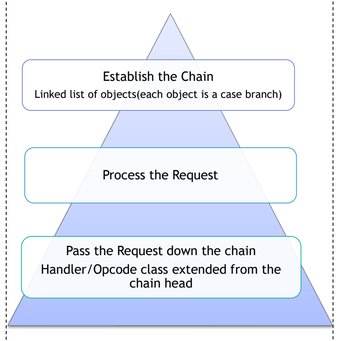
CoR based driver



Pseudocode – Converting UVM case-based driver to CoR driver

Chain head with common mechanisms as base class

```
class chain_head extends uvm_object;
  chain_head _chain; //ptr of its own type
  task set_chain(chain_head chain);
    _chain = chain;
  endtask : set_chain
  virtual task exec_cmd(xaction cmd);
    _chain.exec_cmd(cmd);
  endtask : exec_cmd
  task exec_nxt(xaction cmd);
    if (_chain != null)
      _chain.exec_cmd(cmd)
    else
      $display("End of chain: no cmd found");
  endtask : exec_nxt
endclass : chain_head
```



Handler classes extended from chain_head

```
class op1_c extends chain_head;
  virtual task exec_cmd(xaction cmd);
    if (cmd.op == op1)
      do_op1;
    else
      exec_nxt(cmd);
  endtask : exec_cmd
endclass : op1_c
class op2_c extends chain_head;
 virtual task exec_cmd(xaction cmd);
    if (cmd.op == op2)
      do_op2;
    else
      exec_nxt (cmd);
  endtask : exec_cmd
endclass : op2_c
```

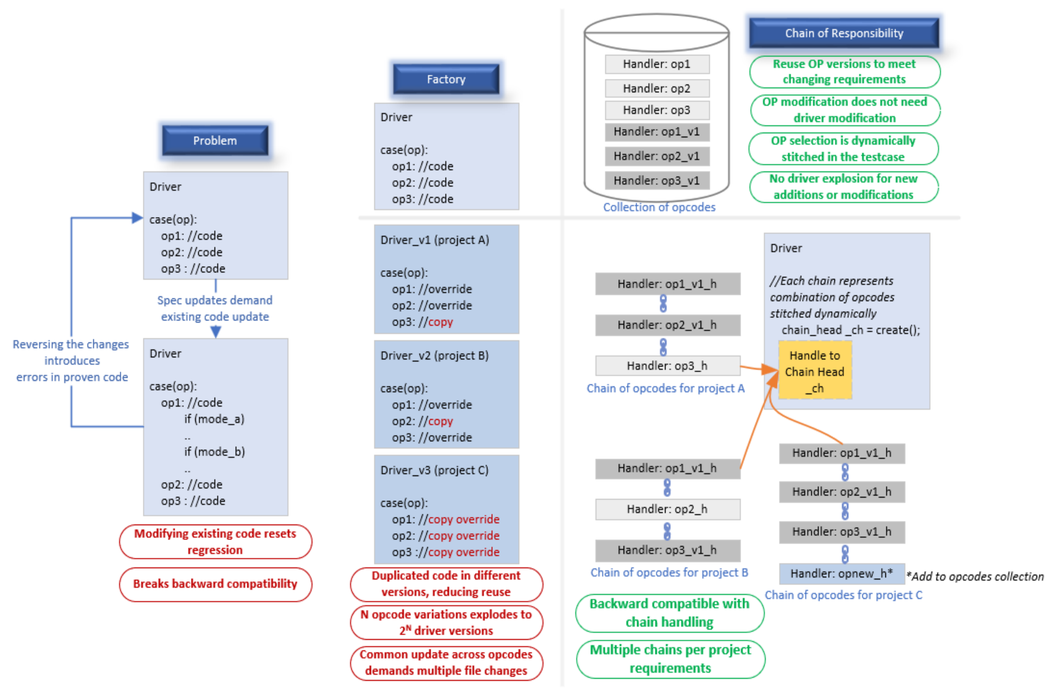
Handle to the chain head in the driver

```
class driver extends uvm_driver #(req);
  // some code
  // handle to chain_head to stitch the ops
  chain_head _chain_head;
  function construct_cor();
    _chain_head = chain_head::type_id::create("_chain_head");
  endfunction : construct_cor
  // other driver logic
endclass : driver
```

Test constructing the chain

```
virtual function construct_chain();
  op1 = op1_c::type_id::create("op1");
  op2 = op2_c::type_id::create("op2");
  .
  opn = opn_c::type_id::create("opn");
  env.agent.driver._chain_head.set_chain(op1);
  op1.set_chain.(op2);
  .
  opm.set_chain(opn);
endfunction : construct_chain
```

Implementation Details

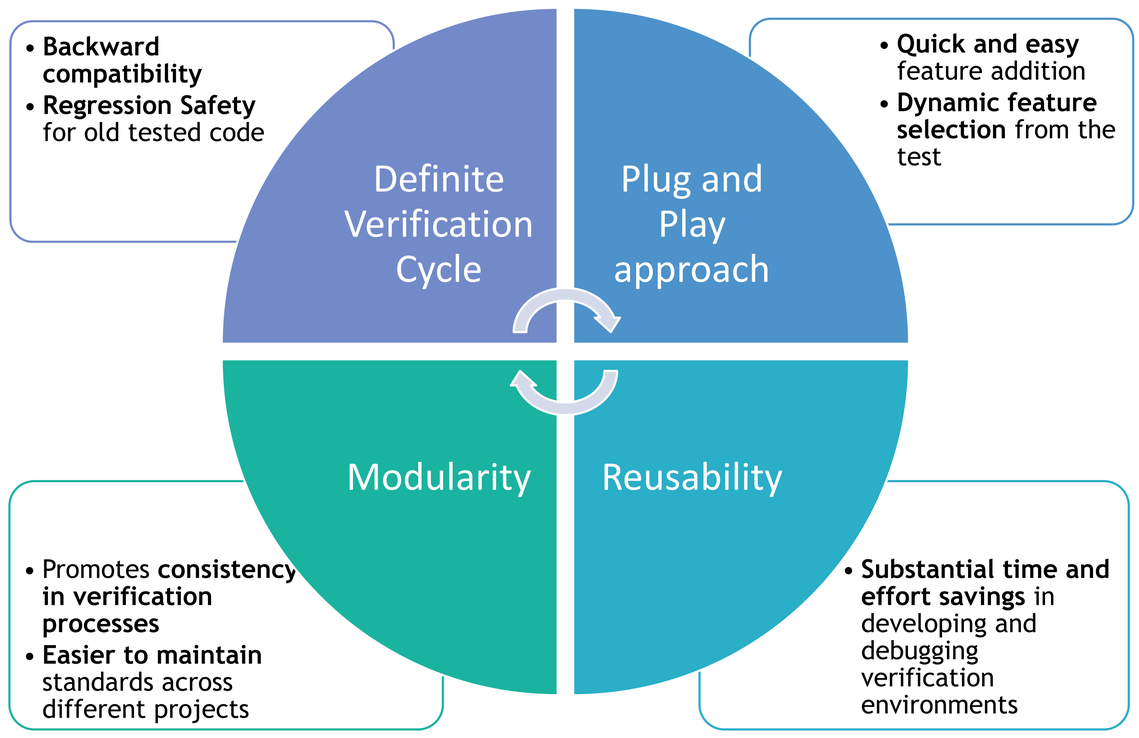


Results

- **Successfully implemented in Serial IO PHY IPs' verification**, for the past few years, demonstrating its effectiveness in real-world applications.
- First time enablement took around ~3-4 weeks of man effort
- New feature addition or enhancement in less than a day without breaking backward compatibility

Indicators	Rudimentary Approach	UVM Factory based Approach	Chain of Responsibility
Backward compatibility	✘	✓	✓
Regression Safety	✘	Depends	✓
Reusability	✘	✓	✓
Code explosion	✓	✓	✘
Code duplication	✘	✓	✘
Opcode selection per test	✘ Static case statements	✓ Override from test	✓ Chain selection from test

Conclusion



REFERENCES

[1] IEEE Standard for Universal Verification Methodology Language Reference Manual, IEEE Standard 1800.2-2017

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns Elements of Reusable Object-Oriented Software", 22 Oct 2009

[3] https://edaplayground.com/x/PE9G