2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Chain of Responsibility Design Pattern for scalable UVM drivers

Chandana K N
Intel, Santa Clara, CA, US 95054

Suresh Gandhi S
Intel, Folsom, CA, US 95630

***Abstract* - Modern-day IP-development must deal with ever changing specification documents. Specifications are not fully defined before the IP development phase; they tend to evolve along with the development. IP also goes through multiple variations accordingly before landing on the final version. This causes a substantial number of changes to the Verification IP, which accommodates multiple protocols and products. Catering to these variations without breaking existing features clutters the testbench with several conditionals, making the code fragile and maintenance a time-consuming task. The objective is to ensure that new code addition never breaks the functionality of old and tested code. *Chain-of-Responsibility (CoR)* is an *Object-Oriented Design Pattern*, which can help achieve this objective. This paper discusses how *Chain-of-Responsibility* was used in IP that supported multiple protocols and products and its relative merits over traditional factory methods.**

Keywords – Changing specifications, UVM Factory, Chain of Responsibility

I. INTRODUCTION

Traditional UVM methodology[1] assumes a well-documented spec, allowing a small degree of variation. Consider the case of a driver in High-Speed Serial Interconnects (HSIO) which supports various Physical layer Modes (PCIe, USB3.2, Converged IO, DP and so on) defined by the PIPE (PHY Interface for the PCI Express* Architecture, USB, etc.) specification. In such a driver, opcodes corresponding to different operations are processed using a case statement, and each enumerated case expressions handle a specific opcode. Here's an example of how such a case statement might look within a UVM driver:

```
class driver extends uvm_driver #(req);
    // some code

    // driver core logic
    case (opcode)
    op1 : begin
            //op1 related operations
          end
    op2 : begin
            //op2 related operations
          end
    .
    .
    .
    opn: begin
           //opn related operations
         end
    endcase
endclass : driver
```

This approach is quite common in UVM drivers for handling different types of transactions. When a specification is updated to include new modes for an existing opcode, the UVM driver must be updated to handle these new modes. The rudimentary approach would involve modifying the case statement within the driver to include the additional logic for the new modes. Below is an example of how the driver's case statement might be updated to handle new modes *mode_a, mode_b* for an existing opcode OP1:

```
class driver extends uvm_driver #(req);
    // some code

    // driver core logic
    case (opcode)
    op1 : begin
            if (mode_a)
              //op1 related operations in mode_a
            else if (mode_b)
              //op1 related operations in mode_b
          end
    op2 : begin
            //op2 related operations
          end
    .
    .
    .
    opn: begin
            //opn related operations
         end
  endcase
endclass : driver
```

This approach is straightforward and works well for a small number of updates. However, as the number of modes and opcodes grows, the case statement can become unwieldy. Further, if the specifications change results in removal of *mode_a*, we may have to revert the changes which might result in the introduction of bugs. Indeed, directly modifying existing code to accommodate specification changes, such as the addition or removal of modes, can introduce several risks and challenges:

- **Breaking Backward Compatibility**: Changes to the driver may affect existing test cases or verification components that rely on the previous behavior.
- **Introduction of Bugs**: Modifying code increases the risk of introducing new bugs, especially if the changes are extensive or if the existing codebase is complex.
- **Increased Verification Lifecycle**: Each change requires re-verification to ensure that new bugs have not been introduced, and that the driver still meets the specification. Ensuring regression-safety with new changes can extend the overall verification cycle.

Regression-safety means a framework which ensures the functionality of old code need not be regressed again after new additions. In HSIO testbenches, each regression suite has a run time of up to 3-4 days. Analyzing the results, triaging the failures, and root causing the issues will consume another 3-4 days. A single regression analysis adds a delay of approximately 1-2 weeks in the best possible scenario.

In the case of the introduction of new features or opcode, the case-statements are not dynamically extensible, as in each new addition of the opcode requires new enumeration to be added along with an associated action for its case item. Hence, this solution resets the regression results and adds more time to the verification cycle based on the complexity of the project. Though callbacks can be used for minimal feature changes, factory overrides address these problems more elegantly with their own set of problems as discussed in the next section.

II. UVM FACTORY BASED APPROACH

UVM factory-based approach allows us to implement the upgrades using the benefits of polymorphism. The use of factory overrides in UVM offers a more dynamic and flexible approach to extending functionality without the need to alter existing code. This method allows for the creation of new components or the modification of behavior at runtime based on configuration or factory settings. In this approach, the upgrades are managed by extending the existing driver (driver) and adding the new changes in the extended driver (driver_v1) as shown in the code snippet below:

```
class driver_v1 extends driver #(req);
    `uvm_component_utils(driver_v1)  //factory registration

    // driver core logic
    case (opcode)
    op1 : begin
            if (mode_a)
              //op1 related operations in mode_a
            else if (mode_b)
              //op1 related operations in mode_b
          end
    op2 : begin
            //op2 related operations
          end
    .
    .
    .
    opn: begin
            //opn related operations
         end
  endcase
endclass : driver_v1

class test extends uvm_test;
    //test related code

    virtual function void build_phase(uvm_phase phase);
        //some code
        set_type_override_by_type(driver::get_type(), driver_v1::get_type());
    endfunction

endclass
```

Both old and new drivers exist in this case. If the specification changes are rolled back, this approach allows for easy reversion to previous versions by changing the factory registration to create instances of the old proven driver instead of the driver_v1. This solution is well suited for cases where multiple copies of drivers will not add overhead to the verification testbench.

For HSIO IPs, the number of variations quickly explodes and reaches insurmountable proportions. For example, say project A requires upgraded opcodes for OP1, OP2 and project B requires upgraded opcodes for OP1, OP3 –

there will be three copies of drivers with OP1, OP2, OP3 in base driver and OP1_v1, OP2_v1, OP3 for project A and OP1_v1, OP2, OP3_v1 for project B as shown in *Figure 1*. Even though OP1_v1 exists in project A, it necessitates the need to create a copy of the same code in project B. Let us say each opcode has 2 different variations and there are 10 opcodes – we may have to maintain 2^10 different code base to have regression-safety over multiple design IPs.

If a common update is required across any of the opcodes, it needs to be updated in all copies of drivers, as each operation might have certain variations compared against the base opcode's definition. Hence, this approach of creating multiple driver variants for different projects and opcode versions can lead to an explosion of code to maintain, which is neither scalable nor efficient.

Every incremental spec version demands exhaustive regression to ensure nothing breaks in several live projects. These drawbacks from the existing methods call for the testbench to be dynamic and flexible enough to support the variations along with maintaining backwards compatibility. We found an Object-Oriented Design Pattern[2] called Chain-of-Responsibility (CoR) to give dynamic flexibility for VIP drivers to capture the functionality of opcodes in the world of fluctuating specifications.

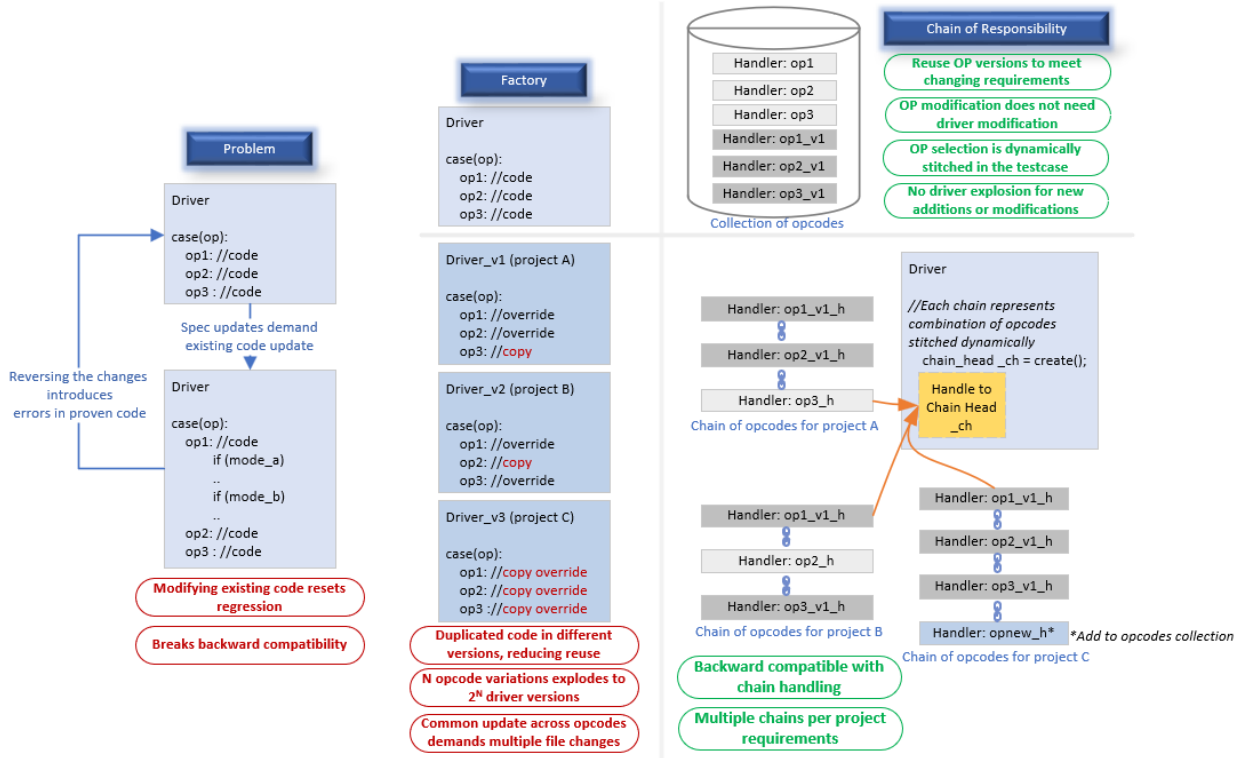

Figure 1 Comparison of Rudimentary approach, Factory, Chain of Responsibility in driver for fluctuating specifications

III. CHAIN OF RESPONSIBILITY BASED SOLUTION

The Chain of Responsibility[3] is a behavioral design pattern in object-oriented design. It provides a structured way to handle a sequence of operations or commands without hard coding the command processing within a single class or method, such as a case statement. The chain consists of a source of command objects called handlers and a series of these handlers connected in a chain as shown in *Figure 2.* Each handler contains logic that defines the

command it can process and a mechanism to pass down the command to the next handler in the chain. The command gets passed along the chain until a handler processes the command.

In a verification testbench, the driver should have the ability to process each of the commands received from the sequencer. The Case-based driver can be converted to CoR based driver by converting each of the case items into a class object called Handler (OP1, OP2 etc.) as shown in *Figure 3*. These opcode handlers are connected in a chain in the testcase, and its handle is provided in the driver as shown in *Figure 4*. Based on the opcode received, one of the handlers in the chain processes the received command. The chain can end with a default handler that either handles all remaining opcodes or provides an error message for unsupported opcodes. The novelty of the application lies in the dynamic selection of the opcodes in the testcase depending on the project or user requirements, without the need of modifying the driver.

The idea of maintaining different opcodes in a chain helps to maintain different versions of the code under each chain, there by untouching the opcode classes themselves as shown *Figure 4*. The CoR pattern enhances reusability and maintainability, as common functionality can be shared across handlers, and updates to the opcode handling logic only need to be made in one place. Additionally, this pattern supports dynamic flexibility, which is crucial for managing the ever-changing specifications in verification environments.

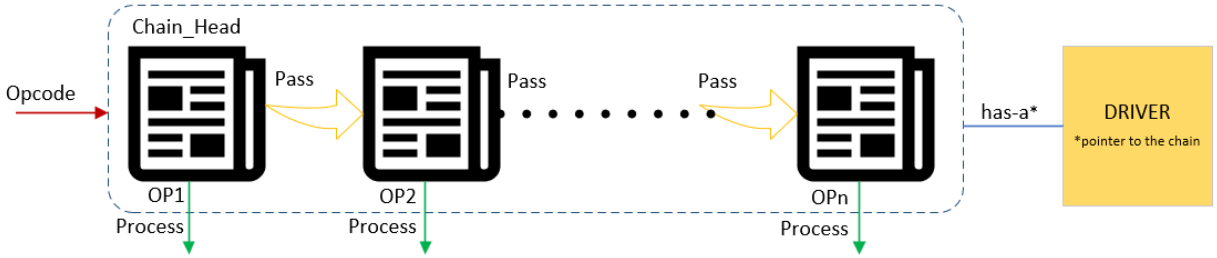

Figure 2 Chain of Responsibility in UVM Driver

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
```

1. *Establish the Chain (set_chain)*
2. *Process the Request (exec_cmd)*
3. *Pass it down the Chain (exec_nxt)*

Handler/Opcode 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
```

1. *Opcode case item split into separate class*
2. *Opcode can process the request or pass it down*

Figure 3 Conversion of rudimentary driver to COR Handlers

```
class driver extends uvm_driver #(req);
    //some code
    chain_head _chain_head;  //handle to chain_head to stitch the ops

    function construct_cor();
        _chain_head = chain_head::type_id::create("_chain_head");
    endfunction: construct_cor

    //other driver logic

endclass
```

Handle to the chain head in the driver

```
//inside test
virtual function construct_chain();

    op1 = op1_c::type_id::create("op1");
    op2 = op2_c::type_id::create("op2");


    env.agent.driver._chain_head.set_chain(op1);
    op1.set_chain(op2);
    .
    .
    .
    op2.set_chain(opn);
endfunction
```

Test constructing the chain

Figure 4 Driver referencing chain constructed from the testcase

By using the CoR pattern, duplicating code can be avoided for each opcode variation and project. Instead, a single chain of handlers is created that can be easily extended or modified without affecting the rest of the chain. For different use-cases/mode requirements the handler objects can also be changed via factory type override instead of creating a new object derived from chain_head. This way the chain construction stays the same for all variants. This approach also allows for backward compatibility, as existing handlers remain unchanged, and new handlers can be added to the chain as needed. For HSIO IP, the chains have been created based on PIPE standard versions (4, 4.1 up to 6.2), type of the PHY mode (PCIe, USB, Converged IO, DisplayPort, HDMI and so on), protocol versions (Gen1, Gen2 etc.) and so on. We have established a working example[4] that demonstrates the application of the Chain of Responsibility in a generic UVM driver for reference purposes.

Table I COMPARISON OF RUDIMENTARY APPROACH, FACTORY, CHAIN OF RESPONSIBILITY IN DRIVER FOR FLUCTUATING SPECIFICATIONS

	Rudimentary Approach	**UVM Factory based Approach**	**Chain of Responsibility**
Backward compatibility	No	Yes	Yes
Regression Safety	No	Depends	Yes
Reusability	No	Yes	Yes
Driver Code explosion	Yes	Yes	No
Code duplication	No	Yes	No
Opcode selection per test	No – Static case statements	Yes – Override from test	Yes – Chain selection from test

IV. RESULTS AND CONCLUSION

CoR ensures that the old code is untouched when new requirement is added. Thus, retaining the regression-safety of the old code and thereby a definite verification cycle for a given project. The CoR design pattern also increased the reusability of the existing opcode classes when there is a need for combination of opcodes from different versions as compared in *Figure 1*. The user can select the required versions of opcodes in the chain without the need for modifying the opcodes themselves. As compared against the factory override approach, the combination of opcodes for different versions can only be achieved by creating a new driver with opcodes copied from already existing versions.

The Chain of Responsibility has allowed us to achieve modularity in our testbench by splitting the operations into their respective class objects, thereby enhancing the readability in the driver. This setup allows users to dynamically plug and play various versions of the opcode classes as per the project customizations. The comparison of the three approaches discussed is highlighted in *Table I.*

The CoR concept can also be extended to various verification components but not limited to monitors, checkers, scoreboards which deal with processing series of requests. This paper highlights the usage of CoR design patterns

for fluctuating specifications; it can also be applied for standardized specifications to gain other benefits of this concept as mentioned in other works[567].

ACKNOWLEDGMENT

We thank Batmanaban Pourouchottaman, Pavitra Balasubramanian and Anargha Jatheendran for their expertise and assistance throughout all aspects of our study and for their help in bringing up the concept to implementation.

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://en.wikipedia.org/wiki/Chain-of-responsibility_pattern
[4] https://edaplayground.com/x/PE9G
[5] Ensilica, “Applying design patterns to accelerate development of reusable, configurable and portable UVCs”, DVCon Europe 2015
[6] Sprott, "Improve Your SystemVerilog OOP Skills", SVUG 2008
[7] D. M. Tomušilović , H. J. Arbel , “UVM Verification Environment Based on Software Design Patterns”, DVCon US 2018