Strategies to Maximize Reusability of UVM Test Scenarios in SoC Verification

Hyeonman Park, Namyoung Kim, Kyoungmin Lee, Hongkyu Kim, Jaein Hong, Kiseok Bae
Samsung Electronics Inc., 1-1 Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, 18488, Korea

***Abstract*-In today's large and complex System on a Chip (SoC), timely completion of functional verification is one of the most critical challenges and reducing the verification Turn-Around-Time (TAT) is very crucial for a successful SoC product development. This paper presents a new UVM code structure with three-level class inheritance. The first- and second-level class code can be completely reused without any changes and only the third level class code need to be modified according to the target SoC product, ensuring high code reusability. This paper provides an efficient flow for complex/composite scenario development that contains: i) scenario definition in the form of metadata; ii) development of an interpreter that reads metadata and automatically generates test scenarios. The proposed approach is able to shorten the scenario creation time by securing high reusability through test code structure optimization and reduces the verification TAT by achieving target coverage with a small number of test scenarios through efficient scenario generation.**

I. INTRODUCTION

The importance of SoC design verification increases with complex SoC designs. Functional verification consumes a great deal of time and manpower, resulting in contributing a major role in the overall Turn-Around Time (TAT) of SoC product development. Further, the highly competitive SoC product development in terms of time-to-market poses a greater challenge in reducing TAT compared to the earlier products. To overcome this, two approaches, i) reducing simulation time for performing test scenarios and ii) shortening test scenario development period by increasing code reusability, have been researched. This paper focuses on the second approach.

The Portable Test and Stimulus Standard (PSS) [1], a representative solution based on the second approach, has been developed and released by Accellera. PSS is a specification for expressing and creating test scenarios and stimulus for reuse in various verification platforms. Several studies [2][3] confirm the effect of code reusability in functional verification when using an EDA tool that supports PSS. However, implementing reusable code directly during SoC verification with only this kind of tools has the following limitations and they motivated the new approach which this paper proposes.

PSS-EDA tools usually focus on generating test scenarios by combining IP-level data manipulation tasks (or sequences). However, in SoC verification, the system-level control operation scenarios are of greater importance than the data manipulation scenarios of each SoC component because the majority of the design bugs occur when implementing system control logic rather than integrating well-verified IPs. For instance, many EDA tools provide ARM architecture library such as cache, DVM, and low power scenarios; the standard interface IP such as PCIe library; and system modeling libraries including memories, processors, and more. The "*solve*" engine then generates multiple scenarios based on user-defined constraints like target memory region of the specific source IP or data path reachability. This approach enables for the scenario developer to create multiple scenarios and fast coverage closure. However, the system control tasks including power and reset control are not provided by default and need to be manual coded. As a result, covering combinations of system-control logic and data manipulation paths are tedious.

In addition, PSS–EDA tools mainly emphasize code reusability across multiple execution platforms such as simulator, emulator, and post-silicon. It has the advantage to quickly secure the test scenarios for different platform after the scenarios are validated on one platform (typically simulation). On the other hand, it is more vital to secure simple test scenarios during the very early bring-up phase by reusing existing code (shift left) and extend them to complex/composite ones along with sophisticated system control operation in order to detect design errors at the early development stage and consequently shorten the verification TAT. Hence, this paper targets code/scenario reusability across multiple products which are built on similar hardware architectures within the same execution platform.

Finally, C-code based tests, the basis for PSS, are time consuming in simulation since they run on real, bulky CPUs. For this reason, replacing a real CPU with a transactor is a very common technique in SoC verification. In addition, SoC verification is mainly performed using UVM-based SystemVerilog (SV) as it supports pre-defined verification components, well-structured code, and superior control interoperation with multiple external events that are frequently used in the system control operation scenario. Accordingly, UVM-based test scenario generation flows are more valuable than C-code based tests. Recently, many PSS-EDA tools are gradually expanding support

for UVM-based test scenario generation, but cannot fully exploit the nature of UVM because they still focus more on C-based tests to reuse in multiple platforms such as emulation and silicon.

For the above reasons, this paper proposes a new approach to maximize reusability of UVM test scenarios in SoC verification. To begin with, we structurally separate all the data manipulation code that performs IP-specific operation and the system-control code such as system resets, low-power control, and general purpose I/O pad control, because the former is likely to be reused. Based on this approach we develop a new process to effectively combine them and implement complex/composite scenarios.

Section II describes the code structure that can separate data manipulation and system control based on reusability. Section III explains the process to automatically develop complex/composite scenarios based on the well-structured code. Finally, section IV and V shows achievements based on the experimental results of products across multiple generation and summarizes the proposed methods.

II. Proposed Code Structure For the New Approach

Typically test scenarios are made up of IP functional operation code and system level event/control code parts. The IP functional operation code could be the candidate for reusable portion assuming that the same IP could be integrated and retargeted to the different SoC products. However, if IP and system code is implemented within the same class, it is hard to identify and extract reusable portion of code. This paper proposes a new UVM code structure to easily separate IP specific operation part of code in the test sequence from the system control code for future reuse.

SV uses extensive Object-Oriented Programming (OOP) techniques [4] and we have designed a class layering code structure based on the role by using inheritance and polymorphism concepts of OOP for code reuse [5]. Fig. 1 depicts a three-level inheritance code structure by role definition. The first-level is a base class which is composed of all the functions of the IP, variables that can activate each function, and all the verification components required to activate the target IP (including VIP, interfaces, and so on). The second-level class inherits the first and is responsible for enabling the function of the base class. The third-level class inherits the second and connects the functions which are enabled in the second-level class to the implemented design and testbench information, e.g. target CPU, address map, register handle, and so on. This implies that the first- and second-level classes can be fully reused without any modification while the third-level class needs to be modified according to the target chip configuration.

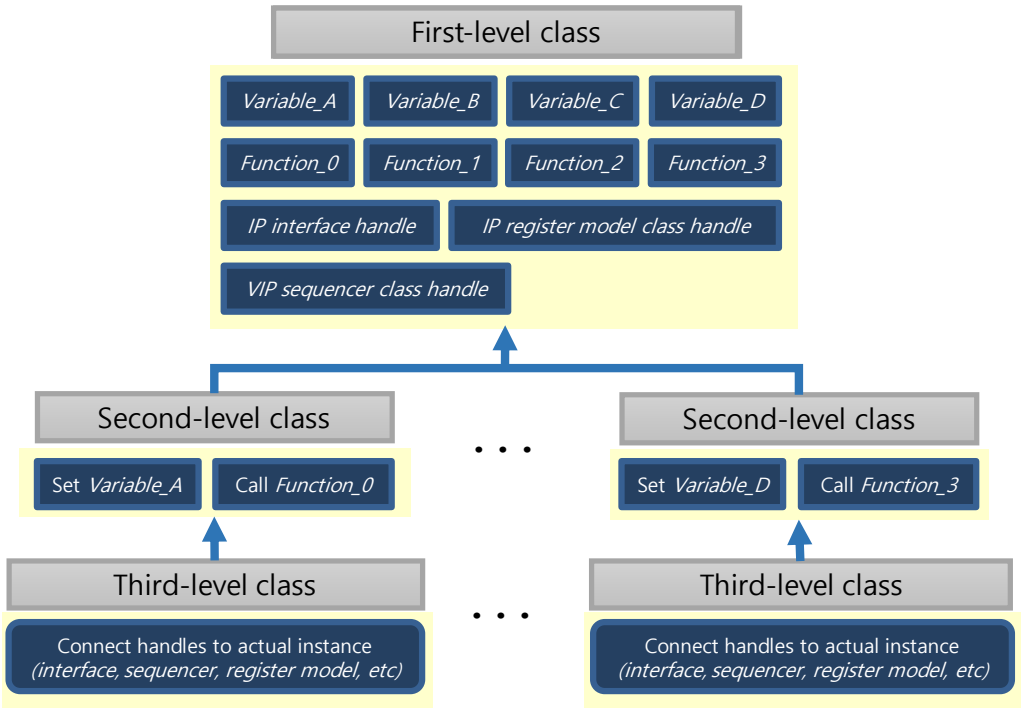


Figure 1. A three-level inheritance code structure to maximize code reusability

Fig. 2 shows implementation example code for the three-level inheritance structure. The target IP is a UART peripheral. It is assumed that the target SoC has four instances of UART and each UART has two functions like TX

(transmit) and RX (receive). As shown in Fig. 2-(a), the first-level class code implements the SV variables and the SV tasks for UART operation. An SV variable maps to an SV function/task and it features a structure that controls the variables for the specific operation. For example, enabling *m_uart_tx_func* variable executes the task *uart_tx_operation_test*. In addition, this class includes all the handles required to stimulate and monitor the UART functions. In this example, Verification IP (VIP) which acts as a counter-part of UART, an interface to monitor the values to be checked, and register model to program the UART are included. Please note that the actual control of the variables is implemented in the second-level class while the handles are mapped in the third-level class. To add new functions or change existing functions, it works only in the first class, resulting in localization of code change.

Fig. 2-(b) is the second-level class code. In this example, the classes corresponding to TX (*ip_usi_uart_tx_seq_c*) and RX (*ip_usi_uart_rx_seq_c*) UART functions are declared respectively. These classes are embodied into sequences to perform the TX/RX operation by enabling the variable declared in the first-level class. As a result, the purpose of the second-level class code is to build a sequence class library to be used to make complex and composite test scenarios. As shown in Fig. 2-(c), a third-level class code maps the sequence classes defined in the second-level code into the target product by using the macro method. As assumed, two function sequences are configured for four instances, resulting in eight operation sequences. In this class, the handles of interface, register model class, and VIP sequencer class defined in the first-level class are also connected according to the product configuration. Code changes that occur according to product configuration, such as the number of IP instances, register model, interface connected to hardware hierarchy, and so on, could be localized in the third-level class. By using this three-level inheritance code structure, the code changes for each target product can be minimized and the first- and second-level code can be fully reused without any modification.

```
10 class ip_uart_base_seq_c #(string host_cpu="") extends parameterized_base_vseq_c #(host_cpu);
11    //*** UART Feature Function Control Variable ***//
12    bit m_uart_tx_func;
13    bit m_uart_rx_func;
14
15    //** UART Interface ***//
16    virtual interface uart_intf uart_vintf;
17
18    //** UART Register Model **//
19    uart_reg_blk_c m_uart_reg_blk;
20    //*** VIP Virtual Sequencer ***//
21    vseqr_c m_vseqr_usi;
22
23    //*** UART Data Operation Test(Tx/Rx) Task ***//
24    extern task uart_tx_operation_test();
25    extern task uart_rx_operation_test();
26
27    `uvm_object_utils_begin(ip_uart_base_seq_c)
28    `uvm_object_utils_end
29
30    function new (string name = "ip_uart_base_seq_c");
31       super.new(name);
32    endfunction: new
33
34    virtual task body();
35       if(m_uart_tx_func) begin
36          uart_tx_operation_test();
37       end
38       if(m_uart_rx_func) begin
39          uart_rx_operation_test();
40       end
41    endtask: body
42 endclass: ip_uart_base_seq_c
```

(a) The first-level class of UART

```
10 class ip_uart_tx_seq_c #(string host_cpu="") extends ip_uart_base_seq_c #(host_cpu);
11    `uvm_object_utils(ip_uart_tx_seq_c)
12
13    function new (string name = "ip_uart_tx_seq_c");
14       super.new(name);
15       // Select Tx Function Sequence
16       m_uart_tx_func = 1;
17    endfunction : new
18 endclass : ip_uart_tx_seq_c
19
20 class ip_uart_rx_seq_c #(string host_cpu="") extends ip_uart_base_seq_c #(host_cpu);
21    `uvm_object_utils(ip_uart_rx_seq_c)
22
23    function new (string name = "ip_uart_rx_seq_c");
24       super.new(name);
25       // Select Rx Function Sequence
26       m_uart_rx_func = 1;
27    endfunction : new
28 endclass : ip_uart_rx_seq_c
```

(b) The second-level classes of UART

```
`define IP_UART_TEST_CLASS_MACRO(INST_NUM, FUNC_NAME, HOST_NAME) \
class uart_``INST_NUM``_ip_uart_``FUNC_NAME``_seq_c #(string host_cpu=`"HOST_NAME`") extends
ip_uart_``FUNC_NAME``_seq_c #(host_cpu); \
   \
   `uvm_object_utils(uart_``INST_NUM``_ip_uart_``FUNC_NAME``_seq_c) \
   \
   function new (string name = `"uart_``INST_NUM``_ip_uart_``FUNC_NAME``_seq_c`"); \
      super.new(name); \
      \
      if(!uvm_config_db#(virtual interface uart_intf)::get(null, "*", `"uart_``INST_NUM``_intf`",
uart_vintf)) \
         `uvm_fatal(get_full_name(), {"Interface must be set for: ", `"uart_``INST_NUM``_intf`"}) \
   endfunction : new \
   \
   virtual task pre_start(); \
      super.pre_start(); \
      \
      m_uart_reg_blk = map.uart``INST_NUM``_reg_blk; \
      m_vseqr_usi = tb.env.uart_``INTF_NUM``_vip_env.vseqr; \
   endtask : pre_start \
endclass

//IP_UART_TEST_CLASS_MACRO(instance number, function name, cpu host name)
//The number of UART instances is 4
// UART Tx Operation Test Sequences
`IP_UART_TEST_CLASS_MACRO(0, tx, a75_cpu)
`IP_UART_TEST_CLASS_MACRO(1, tx, a75_cpu)
`IP_UART_TEST_CLASS_MACRO(2, tx, a75_cpu)
`IP_UART_TEST_CLASS_MACRO(3, tx, r5p_cpu)
// UART Rx Operation Test Sequences
`IP_UART_TEST_CLASS_MACRO(0, rx, a75_cpu)
`IP_UART_TEST_CLASS_MACRO(1, rx, a75_cpu)
`IP_UART_TEST_CLASS_MACRO(2, rx, a75_cpu)
`IP_UART_TEST_CLASS_MACRO(3, rx, r5p_cpu)
```

(c) The third-level classes of UART

Figure 2. Example of the three-level inheritance code structure

As explained earlier, the SoC level test scenarios are implemented by mixing IP-specific and system-control operation. For example, the transmit and receive operation of UART should be combined with system reset in the middle of transmission and/or after the completion of the transmission. The easiest way to implement any system-control operation is to directly insert the code into each IP-specific operation code. Although it is a very straightforward technique and has advantage to early bring up the test scenarios, it has fatal disadvantages: i) as the number of verification entities increases, the same code should be implemented into each target, resulting in code size increase, and ii) all the code cannot be reused because the system-control operation code is very product-dependent. To overcome this limitation, the system-control code can be separated from the IP operation code and implemented as a form of standard library, called a "*common task*" as shown in Fig 3. Now, as each IP operation scenario calls system functions from the common task library, the system functions can be reused over different IP operation. However, this approach still has a limitation that they can be only reused for the same SoC product. They are still hardware implementation dependent and should be modified once they are used in a different product. To maximize reusability, we propose a method that uses an Application Programming Interface (API), a software programming technique [6]. By encapsulating the common tasks among different products and designing custom APIs for general-purpose system control functions, the hardware-dependent portion of code can be hiding and the same code can be reused by the different hardware platform. The scenario developers can maintain consistency when they use the system functions during implementation of complex test scenarios.

Fig. 3 presents an example of reusing two system-control functions, *system_reset1* and *system_reset2* for *N* products which has *M* IPs. Initially, each system task, *system_reset1* and *system_reset2*, are configured differently inside each test scenario. They can be converted in the form of common tasks for each product and can be reused within the same product. Finally, the two system reset common tasks are merged into a system reset API which has an argument for reset kind, *system_reset(kind)* for all *N* products. The API can be reused for all products. Of course, the real implementation of the APIs is product dependent and requires modification. Nevertheless, the system scenario developer can use the same API without specific information of the hardware implementation.

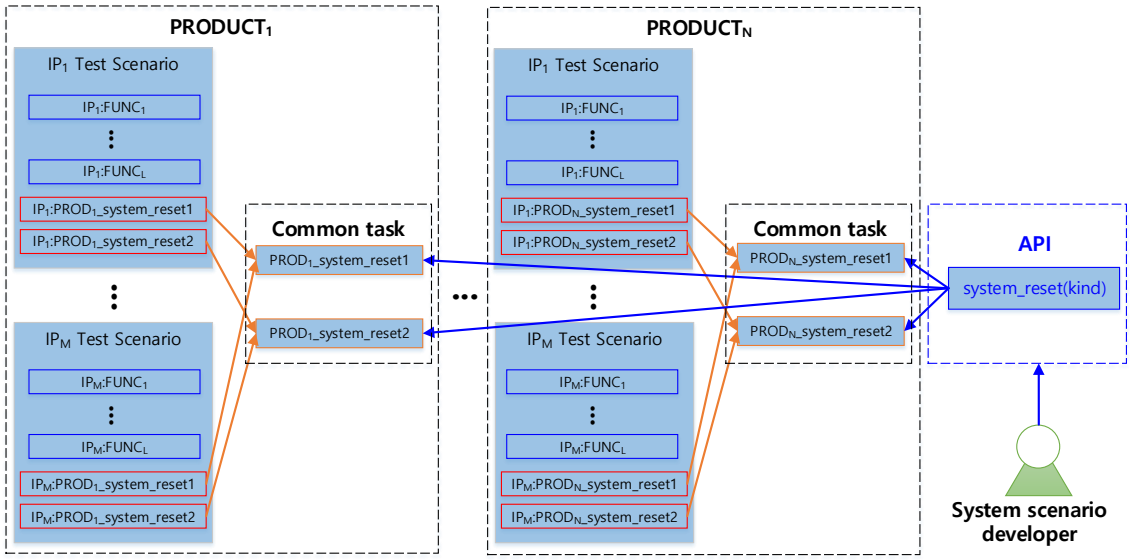


Figure 3. Code structure of system function API

Table 1 shows an example for a part of the system-control function managed using an API. Each system function can be called with a pre-defined argument. Suppose that a target SoC is equipped with a Power-on-Reset (POR) and two Watchdog-Timer resets. The hardware reset control API can be used like *control_hardware_reset ("POR"), control_hardware_reset ("WDT0"), and control_hardware_reset ("WDT1")*.

Table 1. Example of system function APIs

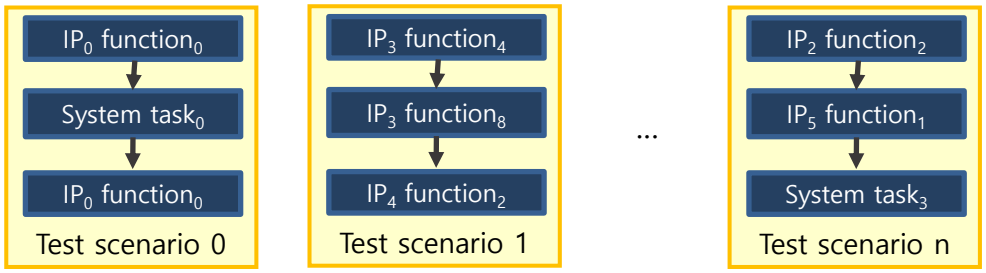
Index	Name of system function APIs	Meaning of system function
1	*control_hardware_reset(string reset_kind)*	Hardware reset control
2	*control_software_reset(string reset_kind)*	Software reset control
3	*control_voltage_domain_group(string pwr_mode)*	Voltage domain control with pre-defined system power mode
4	*control_power_domain_group(string block_name)*	Power domain control
5	*control_smmu(string page_size)*	System memory management unit control

Once the IP function sequences are implemented by using the proposed three-level classes and the APIs for system-control functions are ready, the complex system level test scenarios can be written easily by combining them together. At the same time, the major portion of test code can be reused without any modification.

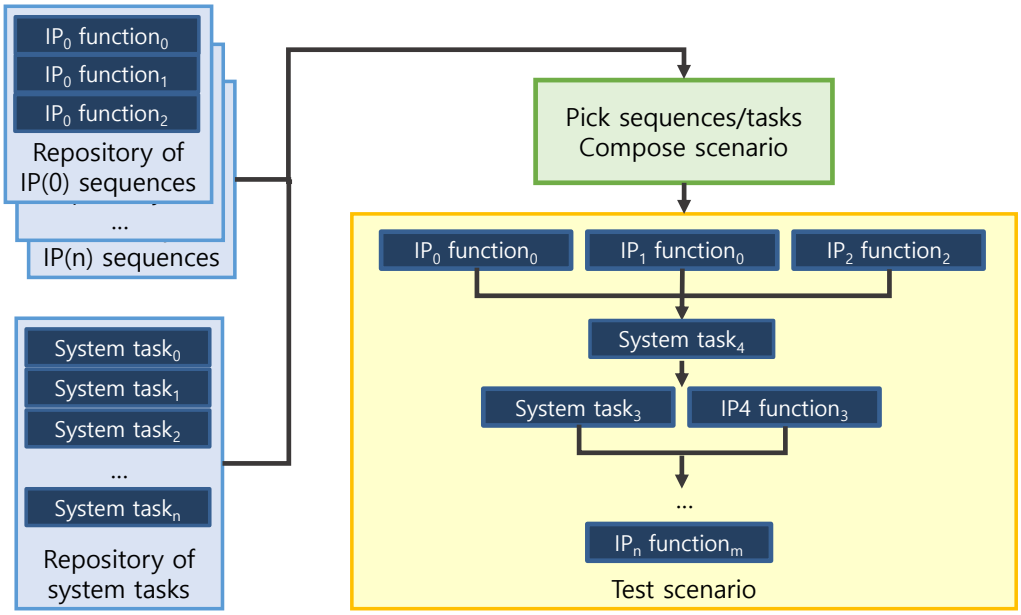
III. Increasing the Efficiency of Complex/Composite Scenario Development

Generating and performing complex/composite test scenarios can ensure better quality SoC verification and the high efficiency of developing test scenarios definitely shorten SoC verification TAT. This section describes how to increase the efficiency of scenario development. The test scenario is implemented after a set of unit sequences is ready. Unlike scenarios that consist of only one sequence, complex scenarios that consist of multiple sequences are usually developed using a framework [7]. Fig. 4-(a) shows an example of a framework of three sequences that operate in serial order. A scenario developer can freely compose these three sequences using various IP functions and system tasks depending on requirements. For instance, *Test scenario n* is composed of IP_2 $function_2$, IP_5 $function_1$, and $system\ task_3$ in Fig. 4-(a). Here, we cannot create scenarios that have more than four sequences or have more than two sequences operating at the same time. Additional scenario frameworks are required to develop a more complex test scenario and to meet the target coverage of function verification quickly. As the number of verification entities and system complexity increase, the number of scenarios also increases because a lot of combinations become possible. Or the number of test scenarios may not much increase if we can make a well-organized scenario framework that can cover many sequence combinations at once. But still, it has a limitation to reach the target coverage quickly as we have to go through these steps of configuring test scenarios, creating frameworks, and then putting sequences in them.

We propose a new scenario development flow to overcome it and further efficiently develop a diverse test scenario. The most important difference compared to a conventional flow is that scenario developers can abstract a test scenario. Once they describe a test scenario with a high level abstraction, then generating a framework and mapping a sequences happens at once. Scenario developers pick sequences from the respective repository containing IP function sequences and system APIs, and combine them depending on their requirements as shown in Fig. 4-(b). So far, they do not have to consider a form of framework at all. A framework including sequences can be generated using this scenario description they configured. In this flow, scenario development is much easier than before because even complex test scenario can be also generated if they just abstract the sequence flow what they want to cover. Furthermore, it is possible to create a much more complex/composite scenario which is composed of arbitrary sequence order.



(a) Current scenario development flow



(b) Proposed scenario development flow

Figure 4. Scenario development flow

To implement this, we define a metadata syntax and develop an interpreter. Writing a form of metadata containing the order of sequences, then the interpreter reads it and generates a framework accordingly. The metadata consists of the sequences which come from the third-level classes, system function APIs, and operators. The first and second are described in section II. Operators act as a symbol that describes the sequence order. For instance, new line means serial order, "&" is for parallel operation, and each sequence which is divided by ";" within "{}" operates sequentially. The left side of Fig. 5 shows an example of metadata to construct a complex scenario. The sequences *seq_a*, *seq_b*, *seq_d* and *seq_e* are selected from the third-level IP classes, and *task_0* and *task_2* are chosen from the system function APIs. Combining them, the scenario developer writes a form of metadata. The sequences *seq_b*, *seq_c*, and *seq_d* execute sequentially and *seq_e* and *task_1* execute at the same time in the example. Like this, metadata makes it easy to generate a framework with an arbitrary sequence order.

An interpreter that inputs the metadata automatically generates UVM test code. The interpreter creates a frame to follow the UVM syntax like a declaration of classes, tasks and variables, then generates a test scenario according to the sequence order described in metadata. The right side of Fig. 5 shows the automatically generated UVM test code by the interpreter.

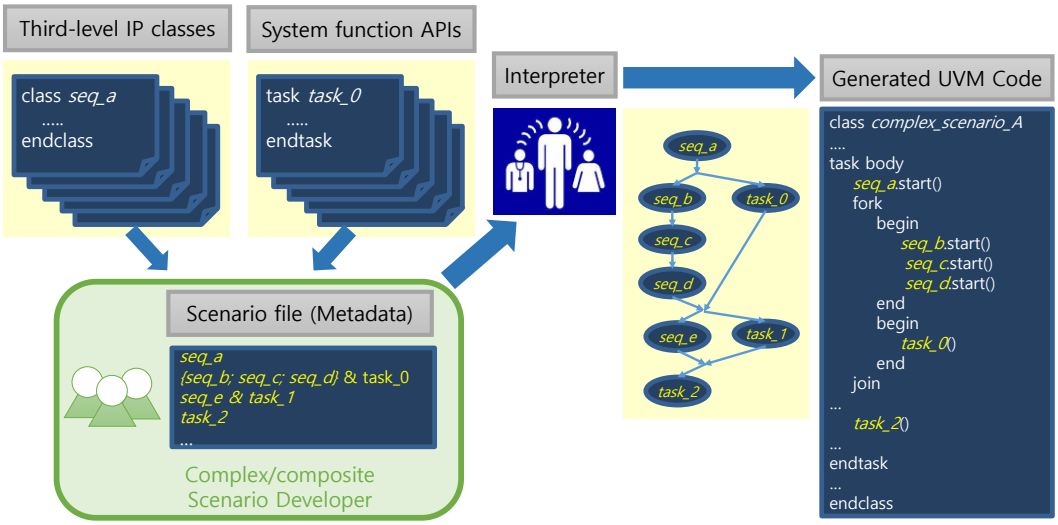


Figure 5. Using the interpreter to generate a complex/composite scenario

Fig. 6 shows an example of creating a test scenario using the third-level IP classes of UART and a set of system function APIs described in Section II. The purpose of the test is to guarantee the functionalities of i) parallel operation of all UART instances, ii) sequential operation of them, and iii) system-level and block-wise reset in the middle of UART operation. Based on the pre-defined metadata, the required functionalities and their order are easily implemented as shown in Fig. 6-(a). The interpreter automatically generates real UVM code after taking the metadata as an input. Fig. 6-(b) presents a generated UVM code. This example demonstrates that any complex test scenarios can be easily defined by using the metadata without any background knowledge on UVM language and it can be implemented automatically, resulting in fast bring-up without any error typically incurred by human coding.

```
1 uart_0_ip_uart_tx_seq_c & uart_1_ip_uart_tx_seq_c & uart_2_ip_uart_rx_seq_c
2 uart_1_ip_uart_rx_seq_c & uart_2_ip_uart_tx_seq_c & control_hardware_reset("POR")
3 uart_0_ip_uart_rx_seq_c & {control_software_reset("blk_a"); control_software_reset("blk_b")}
4 uart_0_ip_uart_rx_seq_c
5 uart_1_ip_uart_rx_seq_c
6 uart_2_ip_uart_tx_seq_c
```

(a) Example of metadata

```
14 class ssf_test_vseq_c extends ssf_base_vseq_c;
15    uart_0_ip_uart_rx_seq_c uart_0_ip_uart_rx_seq;
16    uart_0_ip_uart_tx_seq_c uart_0_ip_uart_tx_seq;
17    uart_1_ip_uart_rx_seq_c uart_1_ip_uart_rx_seq;
18    uart_1_ip_uart_tx_seq_c uart_1_ip_uart_tx_seq;
19    uart_2_ip_uart_rx_seq_c uart_2_ip_uart_rx_seq;
20    uart_2_ip_uart_tx_seq_c uart_2_ip_uart_tx_seq;
21
22    `uvm_object_utils(ssf_test_vseq_c)
23
24    function new (string name = "ssf_test_vseq_c");
25       super.new(name);
26    endfunction : new
27
28    virtual task main_vseq();
29       uart_0_ip_uart_rx_seq = uart_0_ip_uart_rx_seq_c::type_id::create("uart_0_ip_uart_rx_seq");
30       uart_0_ip_uart_tx_seq = uart_0_ip_uart_tx_seq_c::type_id::create("uart_0_ip_uart_tx_seq");
31       uart_1_ip_uart_rx_seq = uart_1_ip_uart_rx_seq_c::type_id::create("uart_1_ip_uart_rx_seq");
32       uart_1_ip_uart_tx_seq = uart_1_ip_uart_tx_seq_c::type_id::create("uart_1_ip_uart_tx_seq");
33       uart_2_ip_uart_rx_seq = uart_2_ip_uart_rx_seq_c::type_id::create("uart_2_ip_uart_rx_seq");
34       uart_2_ip_uart_tx_seq = uart_2_ip_uart_tx_seq_c::type_id::create("uart_2_ip_uart_tx_seq");
35
36       // [ssf_ui] seq_line #0 start //
37       fork
38          uart_0_ip_uart_tx_seq.start(tb.vseqr);
39          uart_1_ip_uart_tx_seq.start(tb.vseqr);
40          uart_2_ip_uart_rx_seq.start(tb.vseqr);
41       join
42
43       // [ssf_ui] seq_line #1 start //
44       fork
45          uart_1_ip_uart_rx_seq.start(tb.vseqr);
46          uart_2_ip_uart_tx_seq.start(tb.vseqr);
47          control_hardware_reset("POR");
48       join
49
50       // [ssf_ui] seq_line #2 start //
51       fork
52          uart_0_ip_uart_rx_seq.start(tb.vseqr);
53          begin
54             control_software_reset("blk_a");
55             control_software_reset("blk_b");
56          end
57       join
58
59       // [ssf_ui] seq_line #3 start //
60       uart_0_ip_uart_rx_seq.start(tb.vseqr);
61       // [ssf_ui] seq_line #4 start //
62       uart_1_ip_uart_rx_seq.start(tb.vseqr);
63       // [ssf_ui] seq_line #5 start //
64       uart_2_ip_uart_tx_seq.start(tb.vseqr);
65    endtask
66 endclass
```

(b) Example of automatically generated UVM test code

Figure 6. Example of creating test scenario using the proposed method

IV. Experimental Results

This section describes the results and effects of the proposed code structure and scenario generation flow in the previous sections. Two automotive SoC products which have been designed on the same architecture platform have been verified. The second generation product targets higher performance and is infused with increased complexity than the first generation one while having similar functionality. The first generation product was originally verified using the conventional method and the existing test scenarios have been re-organized into the proposed three-level code structure. The test scenarios of the second generation product are created by re-using the first- and second-level classes of the code. Table 2 shows the number of the UVM code lines for each level of classes constituting an example block, *BLOCK*$_1$. The results demonstrated that about 86% of the code in average belong to the reusable first- and second class. The test scenarios can be implemented after modifying the remaining 14% of code according to the target SoC configuration.

Table 2. Amount of test scenario code for five IPs in BLOCK$_1$

$BLOCK_1$	Number of code lines			Reuse rate
	① First-level class	② Second-level class	③ Third-level class	(①+②)/(①+②+③)
IP_1	1,922	147	215	91%
IP_2	508	117	139	82%
IP_3	603	72	258	72%
IP_4	547	69	155	80%
IP_5	793	76	32	96%
Total	4,373	481	799	86%

After the third-level classes for IP specific operation and the common APIs for various system tasks are ready, a set of complex/composite test scenarios have been automatically generated by using the proposed scenario generation flow. To check the efficiency of the proposed approach, we compared the number of test scenarios to cover the target blocks ($BLOCK_1$~$BLOCK_4$) which are integrated in two products as shown in Table 3. The test scenarios for the first generation product have been implemented based on the conventional pre-defined framework. Those for the second generation product have been generated after defining a set of arbitrary sequence framework using the meta-data. The results show that the same coverage can be reached with only 13% of total number of test scenarios and 87% of test scenarios can be reduced.

Table 3. Number of scenarios to test four blocks of two generation products

	Number of test scenarios		Scenario reduction rate
	① First-generation product	② Second-generation product	(①-②)/①
$BLOCK_1$	286	38	87%
$BLOCK_2$	744	116	84%
$BLOCK_3$	897	93	90%
$BLOCK_4$	1,157	145	87%
Total	3,084	392	87%

As a result, all UVM code making up the total test scenarios can be classified as three categories: i) reusable class of code (the first- and second-class of code); ii) non-reusable class of code (the third-class of code in the repository); and iii) the test scenario class which are automatically generated by the interpreter. Table 4 presents the portion of each code category. The results show that around 88% of total code can be fully reused without any modification and automatically generated. By editing the remaining 14% of code lines, all the test scenario writing step can be completed and this can dramatically increase the productivity and efficiency of the verification process.

Table 4. Code amount of test scenarios for four blocks of the second generation product

	Number of code lines	Portion
Reusable	19,842	44%
Manually coded	5,266	12%
Auto generated	19,954	44%
Total	45,062	

In conclusion, it is possible to secure many IP functional operation test scenarios at the very early SoC design phase and increase the maturity of the design within a short period of verification process. The diverse test scenarios combining lots of sequences/tasks also help to find any complex design bugs and complete the whole verification process within the given verification time budget.

V. SUMMARY

This paper intends to reduce verification time by optimizing the UVM test code structure and improving the composite/complex scenario efficiency. We propose a three-level inheritance code structure to maximize code reusability and a new scenario development flow that contains the scenario definition in the form of metadata and an interpreter to automatically generate test scenarios using the defined metadata. Experimental results demonstrate 86% code reusability when applied to the two generation products and show a decrease in the total number of

scenarios up to 87%, while maintaining the same verification quality compared to the product that did not use the proposed scenario development flow.

ACKNOWLEDGMENT

We would like to thank Mijung Noh, Corporate Vice-President, Samsung Electronics, for the support and encouragement in publication of our result. We would also like to thank Conextt Inc., Silverchips Inc., and Wizonetech Inc. for co-working with us for the implementation of IP scenarios based on the proposed three-level code structure.

REFERENCES

[1] Accellera, Portable Test and Stimulus, Version 1.0, (June 2018); https://accellera.org/downloads/standards/portable-stimulus
[2] Dayoung Kim, Jaehun Lee, and Daeseo Cha, "Post-Silicon Performance Validation Using PSS", DVCon US 2020.
[3] Suresh Vasu, Nithin Venkatesh, and Joydeep Maitra, "Media Performance Validation in Emulation and Post Silicon Using Portable Stimulus Standard", DVCon US 2021.
[4] SystemVerilog, https://en.wikipedia.org/wiki/SystemVerilog
[5] Inheritance (object-oriented programming), https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
[6] API, https://en.wikipedia.org/wiki/API
[7] Framework, https://en.wikipedia.org/wiki/Software_framework