2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

# Strategies to Maximize Reusability of UVM Test Scenarios in SoC Verification

Namyoung Kim, Hyeonman Park, Kyoungmin Lee, Hongkyu Kim, Jaein Hong, Kiseok Bae

Samsung Electronics Inc.

accellera SYSTEMS INITIATIVE

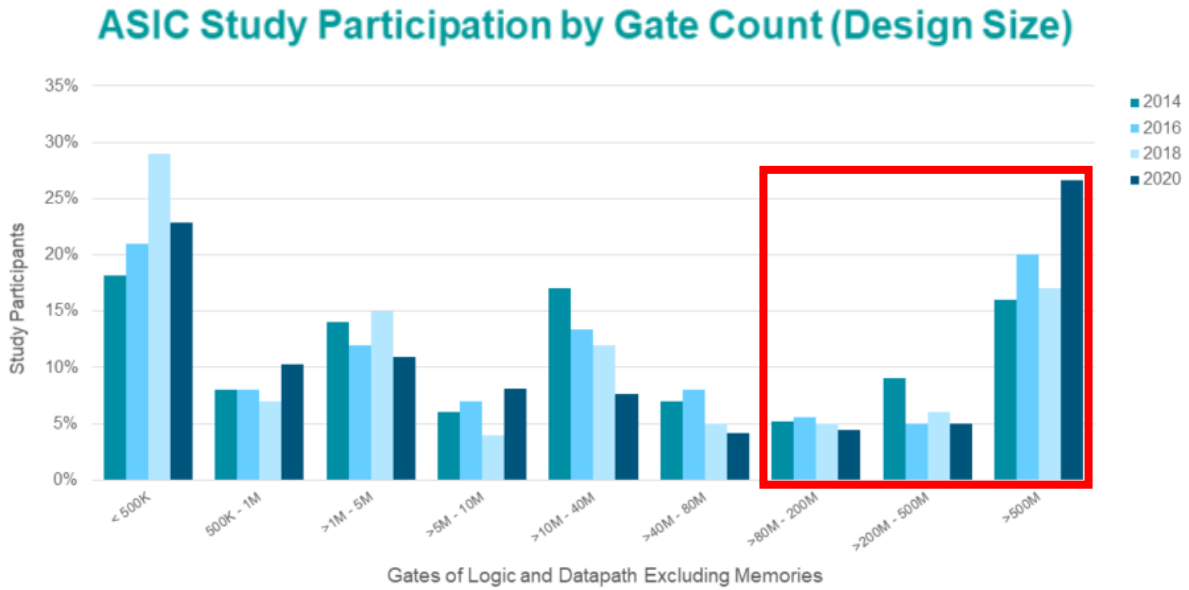
# Agenda

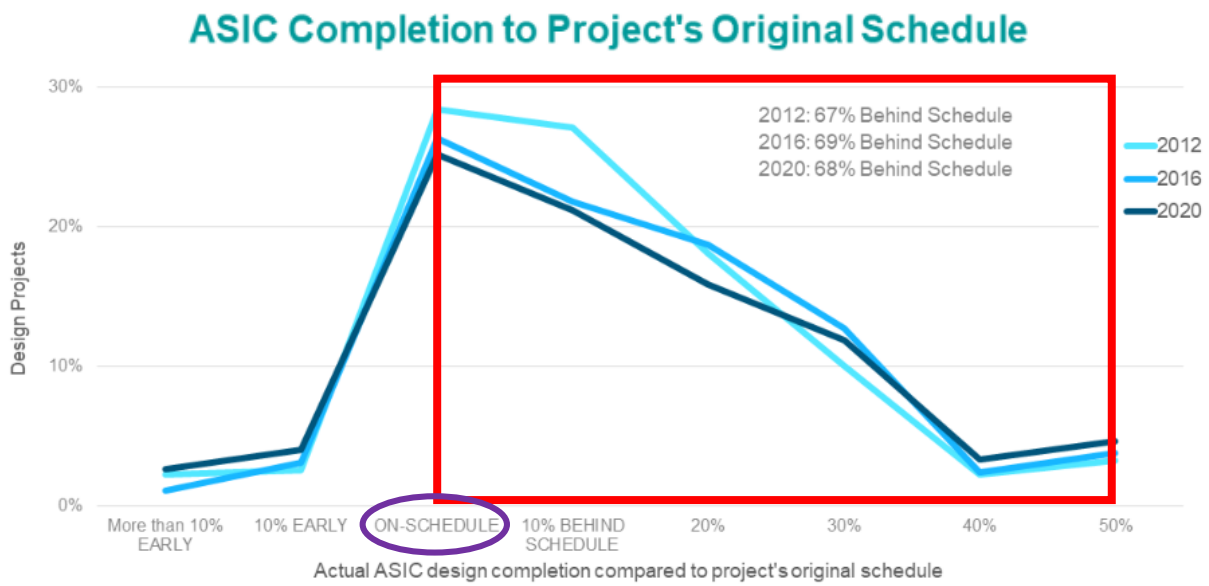
- Motivation
- Maximizing Code Reusability through Hierarchical Structure
- Increasing Efficiency of Complex/Composite Scenario Development
- Experimental Results
- Conclusion

# Motivation – Trend (1)

- Complex SoC design, time-to-market product
  - Challenge to shorten the functional verification Turn-Around Time (TAT)

*The 2020 Wilson Research Group Functional Verification Study



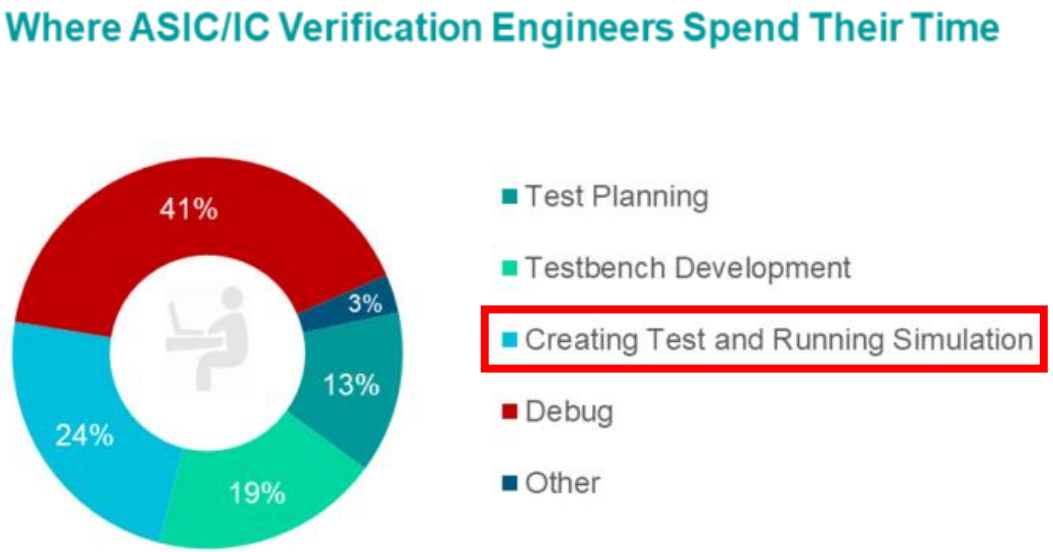



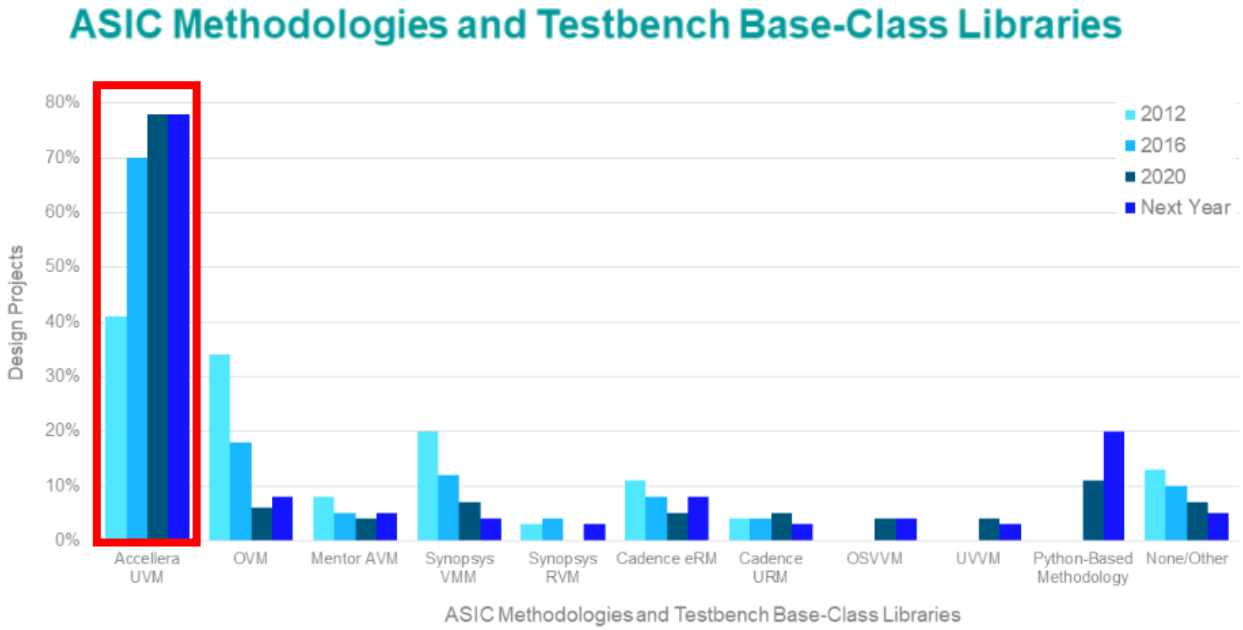

✓ **Continue to move to larger designs**

✓ **68 percent of IC/ASIC projects were behind schedule**

# Motivation – Trend (2)

- Spend a lot of time to create test scenario and run simulation
- UVM is the most commonly used verification methodology



# Motivation - Approach

- Approaches how to reduce verification TAT
  - **Shorten test scenario development period**
  - Reduce simulation time for performing test scenarios

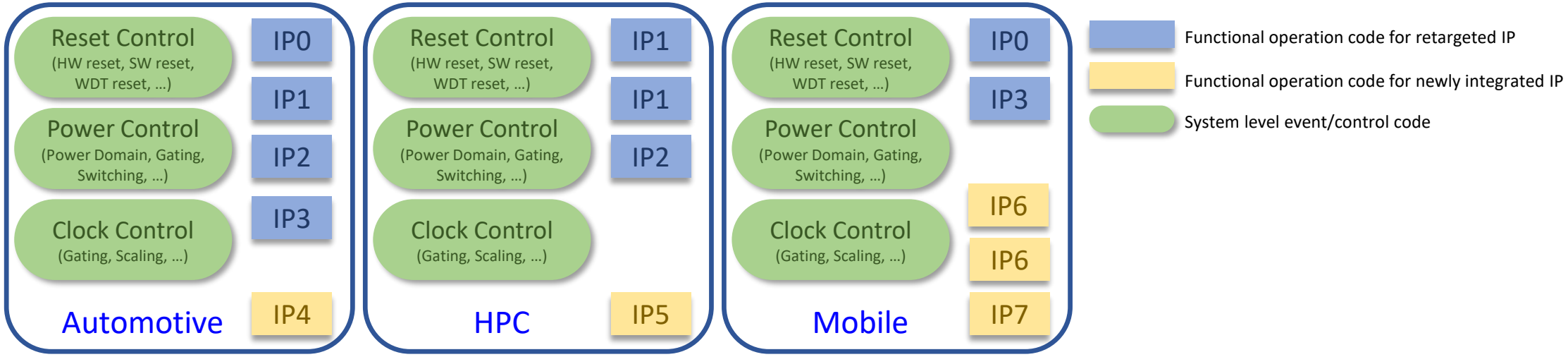
- How to shorten test scenario development period
  - Maximize test code reusability
  - Increase efficiency of scenario development
  - Language/Methodology: SystemVerilog(SV)/UVM

# Agenda

- Motivation
- Maximization of Code Reusability through Hierarchical Structure
- Increasing Efficiency of Complex/Composite Scenario Development
- Experimental Results
- Conclusion

# What is Reusable across Multiple Products

- IP functional operation code
  - Reusable when the same IP is retargeted to different SoC products
- System level event/control code
  - Reusable when the same category of system functions is defined (even for different implementations)



# Strategy to Maximize Code Reusability

- Hierarchical code structure for IP functional operation code
  - Layering IP test code based on role using inheritance concept
  - Identification and extraction of reusable code
- System API* for system level event/control code
  - Designing custom APIs for general-purpose system control functions
  - Encapsulating common system level function code among different products
  - Hiding hardware-dependent implementation code

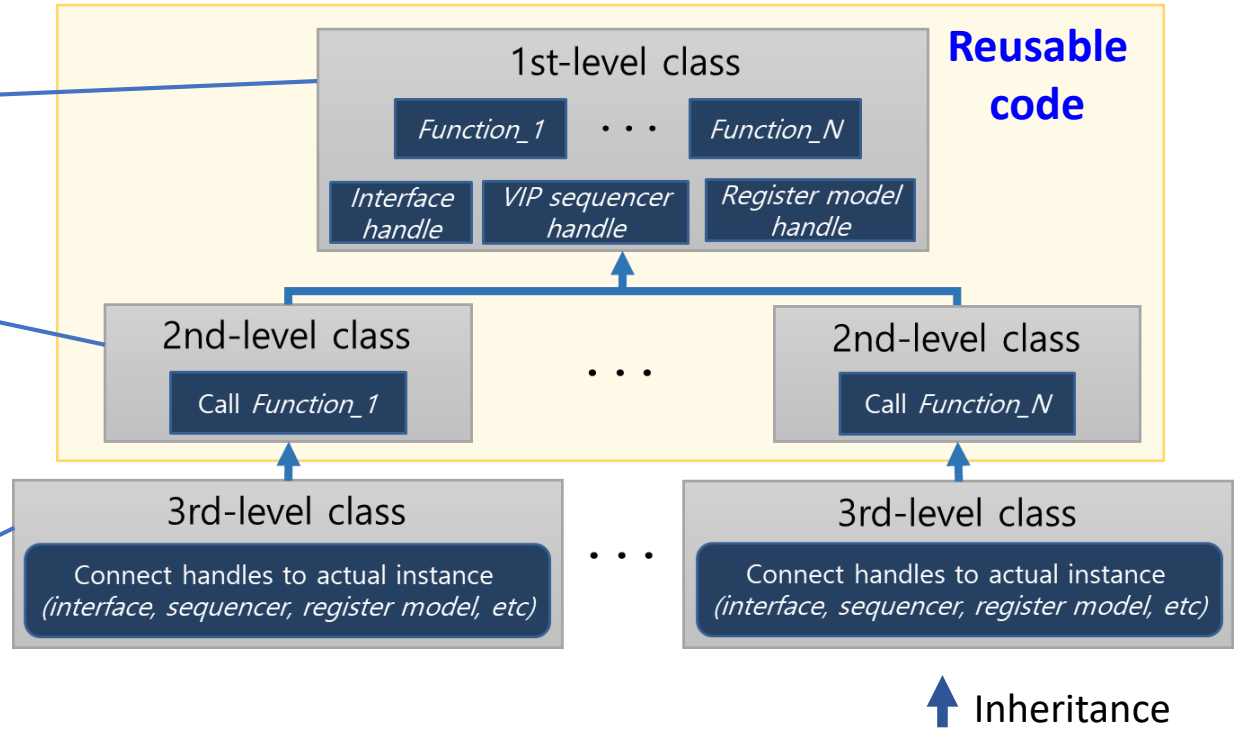
* Application Programming Interface

# Hierarchical Code Structure for IP Operation

1st layer - Embody all the IP operation code in one class using a handle rather than an instance of the required verification components

2nd layer - Define classes to perform each IP function using code from the first layer

3rd layer - Define actual IP tests by product configuration and connect each handle of the first layer to the instances of each product/testbench dependent implementation



# Example – The 1st-level Class

- Assumption
  - Target IP is UART peripheral
  - Each UART has 2 functions
  - Target SoC has 4 UART instances

Declare UART verification component handles

Embody all operation code for UART functions

```
class ip_uart_base_seq_c extends base_vseq_c;
  virtual interface uart_intf m_uart_intf;
  uart_reg_blk_c m_uart_reg_blk;
  uart_vip_vseqr_c m_uart_vip_seqr;
  …
  task do_tx_op();
    vip_rx_cfg_seq.start(m_uart_vip_seqr);
    m_uart_reg_blk.utxhn.write(status, data);
    wait(m_uart_intf.INT_UART == 1);
    …
  endtask

  task do_rx_op();
    vip_tx_cfg_seq.start(m_uart_vip_seqr);
    wait(m_uart_intf.INT_UART == 1);
    m_uart_reg_blk.urxhn.read(status, data);
    …
  endtask
endclass
```

# Example – The 2nd-level Class

Inherit the 1st-level class

Define Tx function class by calling the related operation task

```
class ip_uart_tx_seq_c extends ip_uart_base_seq_c;
  …
  virtual task body();
    do_tx_op();
  endtask
endclass

class ip_uart_rx_seq_c extends ip_uart_base_seq_c;
  …
  virtual task body();
    do_rx_op();
  endtask
endclass
```

Inherit the 1st-level class

Define Rx function class by calling the related operation task

# Example – The 3rd-level Class

Inherit the 2nd-level class

```
`define IP_UART_TEST_CLASS_MACRO(INST_NUM, FUNC_NAME) \
class uart_``INST_NUM``_``FUNC_NAME``_seq_c extends ip_uart_``FUNC_NAME``_seq_c; \
  …
  function new (string name = `"uart_``INST_NUM``_``FUNC_NAME``_seq_c`"); \
    super.new(name); \
    if(!uvm_config_db#(virtual interface uart_intf)::get(null, "*", `"uart_``INST_NUM``_intf`",   m_uart_intf)) \
      `uvm_fatal(get_full_name(), {"Failed connection 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_uart_vip_seqr = tb.env.uart_``INST_NUM``_vip_env.vseqr; \
  endtask \
endclass

`IP_UART_TEST_CLASS_MACRO(0, tx)
`IP_UART_TEST_CLASS_MACRO(0, rx)
`IP_UART_TEST_CLASS_MACRO(1, tx)
`IP_UART_TEST_CLASS_MACRO(1, rx)
`IP_UART_TEST_CLASS_MACRO(2, tx)
`IP_UART_TEST_CLASS_MACRO(2, rx)
`IP_UART_TEST_CLASS_MACRO(3, tx)
`IP_UART_TEST_CLASS_MACRO(3, rx)
```

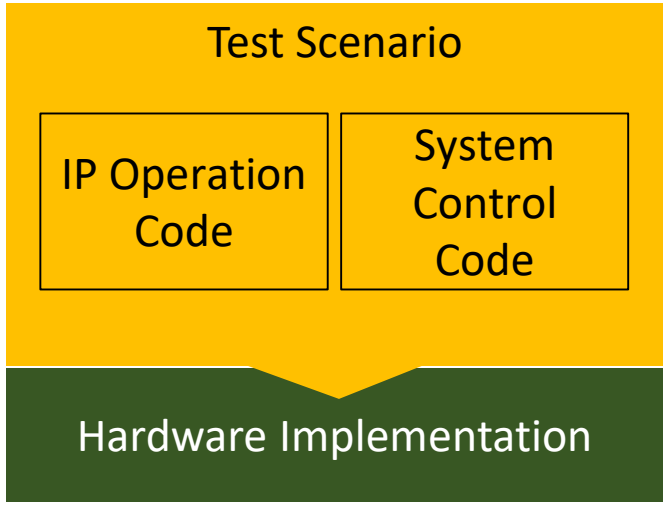
Connect interface handle to actual instance

Connect handles of register model and VIP sequencer to actual instance

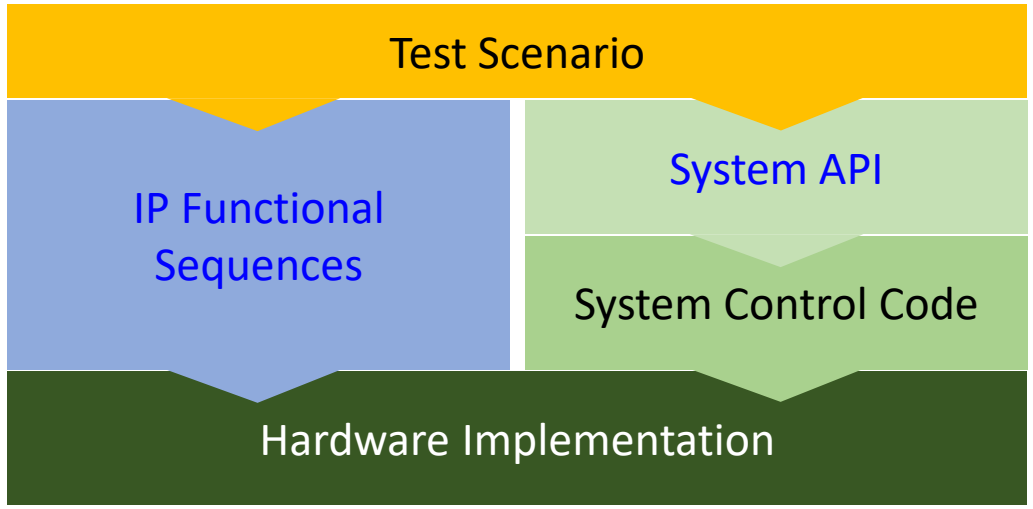
8 UART tests
for 2 functions and 4 instances

# System API for System Event/Control Code (1)

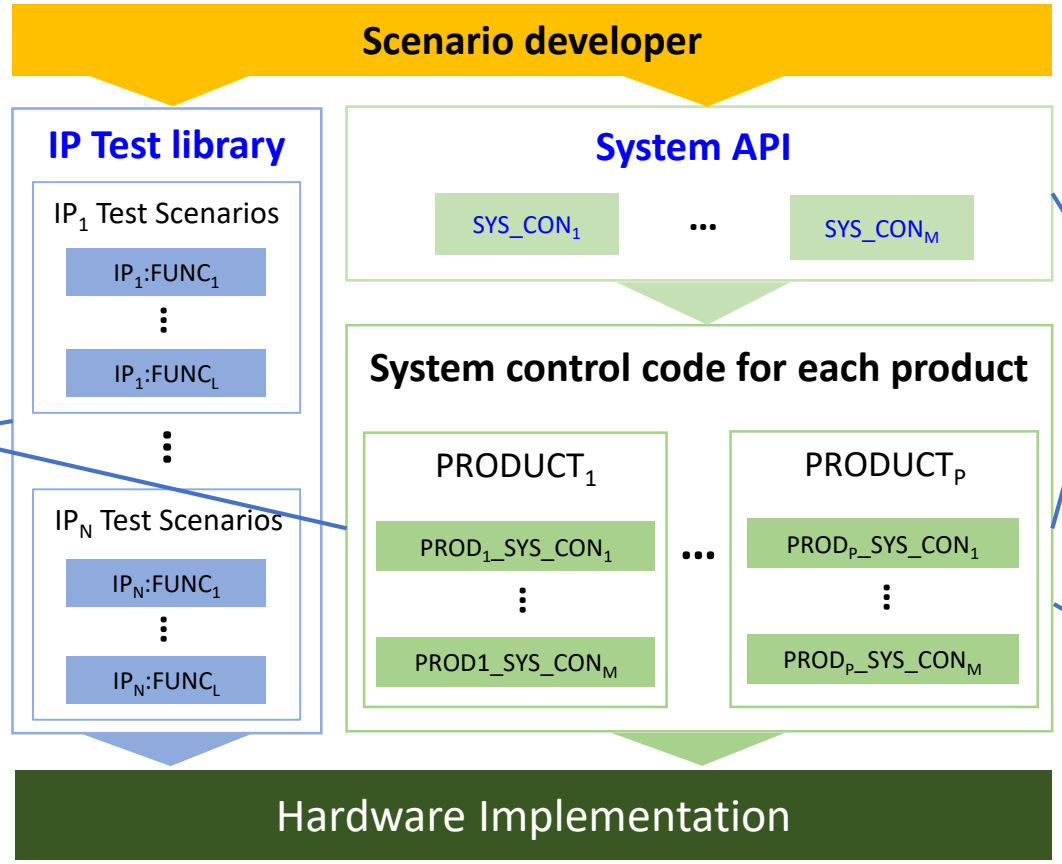
- Conventional way to implement SoC level test scenario
  - Mixture of system control operation and functional code directly in a test scenario




- Proposed way to implement SoC level test scenario
  - Separation and modularization
  - Encapsulation → API



# System API for System Event/Control Code (2)




Separate the system control and the IP operation code

Encapsulate the system control code to hide from scenario developer and design custom APIs to not rely on hardware implementation

Modularize each system control code

# Example of System API

- General-purpose system control functions
  - Candidate: Reset, Power, Clock
- Example of system-control function managed using an API

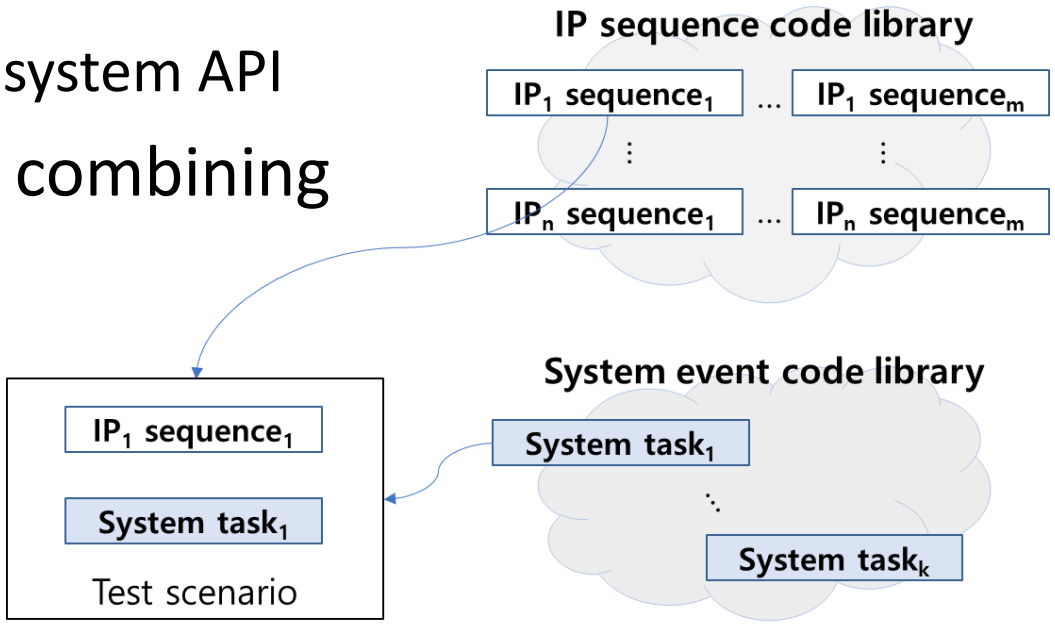
| Index | Name of system function APIs | Meaning of system function |
|---|---|---|
| 1 | *control_hardware_reset(string reset_kind)* | Hardware reset control |
| 2 | *control_software_reset(string reset_kind)* | Software reset control |
| 3 | *control_power_domain_group(string block_name)* | Power domain control |
| 4 | *control_clock_manager(string mode)* | Clock management |

# Agenda

- Motivation
- Maximization of Code Reusability through Hierarchical Structure
- Increasing Efficiency of Complex/Composite Scenario Development
- Experimental Results
- Conclusion

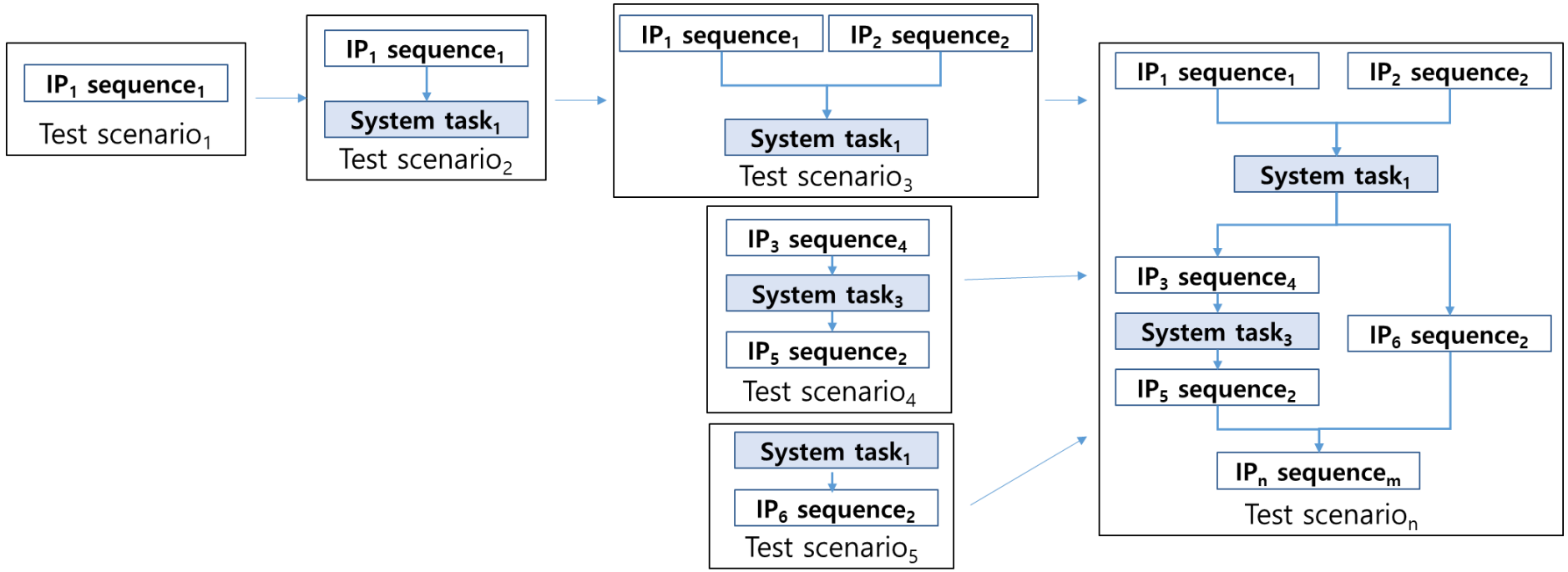
# Scenario Development for SoC Verification

- Reusable library code is ready
  - IP functional operation code library from the hierarchical class code structure
  - System event task code library from the system API
- Develop test scenarios by picking and combining sequences from libraries



# Challenge of Complex Scenario Development

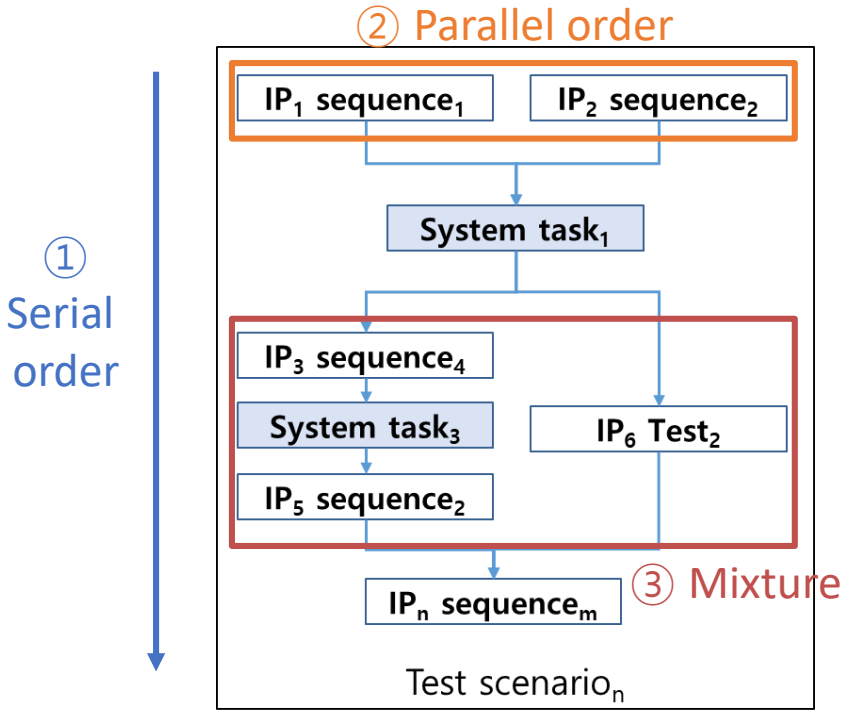
- Scenario development: Simple → Complex, Single → Multiple

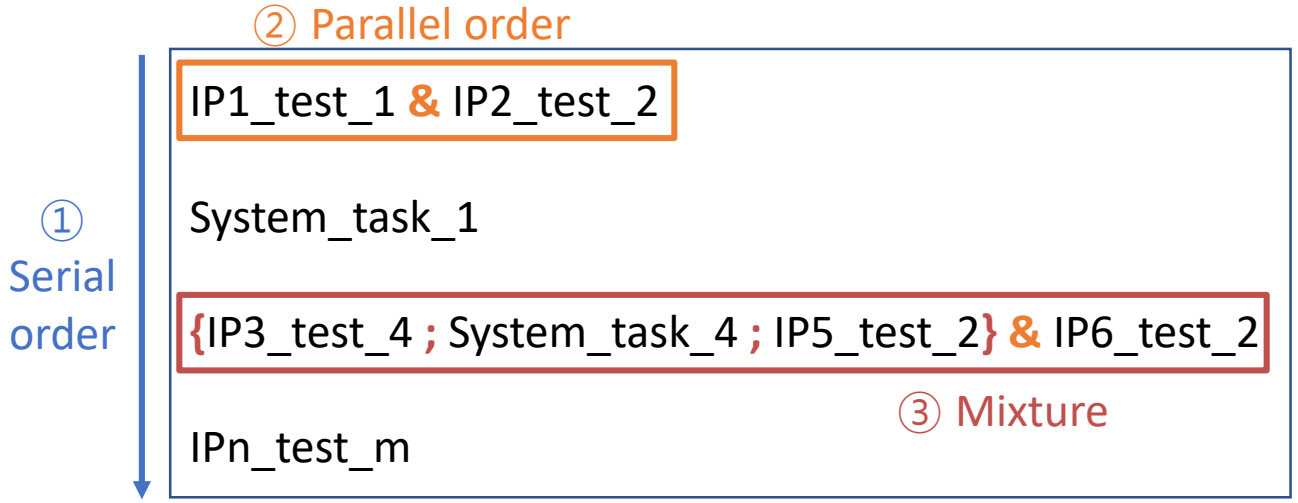



***Complexity of scenario ↑*** → ***Efficiency of scenario development ↓***

# Increasing Efficiency (1)

- Metadata for expressing sequence/task easily and intuitively



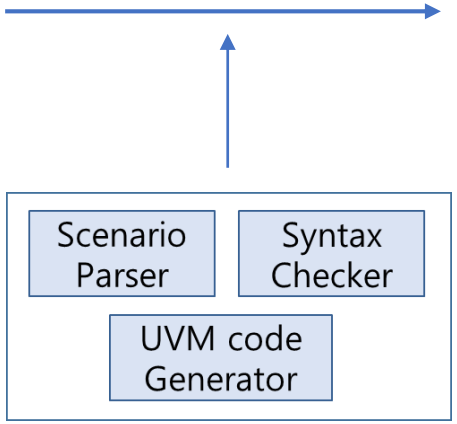

**Metadata form of test scenario**

# Increasing Efficiency (2)

- Interpreter for automatically generating UVM test code

Scenario file (Metadata)

```
IP1_test_1 & IP2_test_2
System_task_1
{IP3_test_4 ; System_task_4 ; IP5_test_2} & IP6_test_2
IPn_test_m
```




Auto generated UVM test

```
class complex_scenario_A
  …
  virtual task body
    fork
      IP1_test_1.start()
      IP2_test_2.start()
    join
    System_task1()
    …
  endtask
  …
endclass
```

# Example

```
uart_0_tx_seq_c  & uart_1_rx_seq_c & uart_2_tx_seq_c & control_hardware_reset("POR")
uart_0_rx_seq_c & {control_software_reset("blk_a"); control_software_reset("blk_b")}
uart_0_tx_seq_c
uart_1_tx_seq_c
uart_2_rx_seq_c
```

Scenario file written by metadata form

```
class ssf_test_vseq_c extends ssf_base_vseq_c;
  uart_0_rx_seq_c  uart_0_rx_seq;   uart_0_tx_seq_c  uart_0_tx_seq;
  uart_1_rx_seq_c  uart_1_rx_seq;   uart_1_tx_seq_c  uart_1_tx_seq;
  uart_2_rx_seq_c  uart_2_rx_seq;   uart_2_tx_seq_c  uart_2_tx_seq;

  virtual task body();
    uart_0_rx_seq = uart_0_rx_seq_c::type_id::create("uart_0_rx_seq");
    uart_0_tx_seq = uart_0_tx_seq_c::type_id::create("uart_0_tx_seq");
    uart_1_rx_seq = uart_1_rx_seq_c::type_id::create("uart_1_rx_seq");
    uart_1_tx_seq = uart_1_tx_seq_c::type_id::create("uart_1_tx_seq");
    uart_2_rx_seq = uart_2_rx_seq_c::type_id::create("uart_2_rx_seq");
    uart_2_tx_seq = uart_2_tx_seq_c::type_id::create("uart_2_tx_seq");
    fork
      uart_0_tx_seq.start(tb.vseqr);
      uart_1_rx_seq.start(tb.vseqr);
      uart_2_tx_seq.start(tb.vseqr);
      control_hardware_reset("POR");
    join
    fork
      uart_0_rx_seq.start(tb.vseqr);
      begin
        control_software_reset("blk_a");
        control_software_reset("blk_b");
      end
    join
    uart_0_rx_seq.start(tb.vseqr);
    uart_1_rx_seq.start(tb.vseqr);
    uart_2_tx_seq.start(tb.vseqr);
  endtask
endclass
```

Auto generated UVM code by interpreter

# Expected Benefits

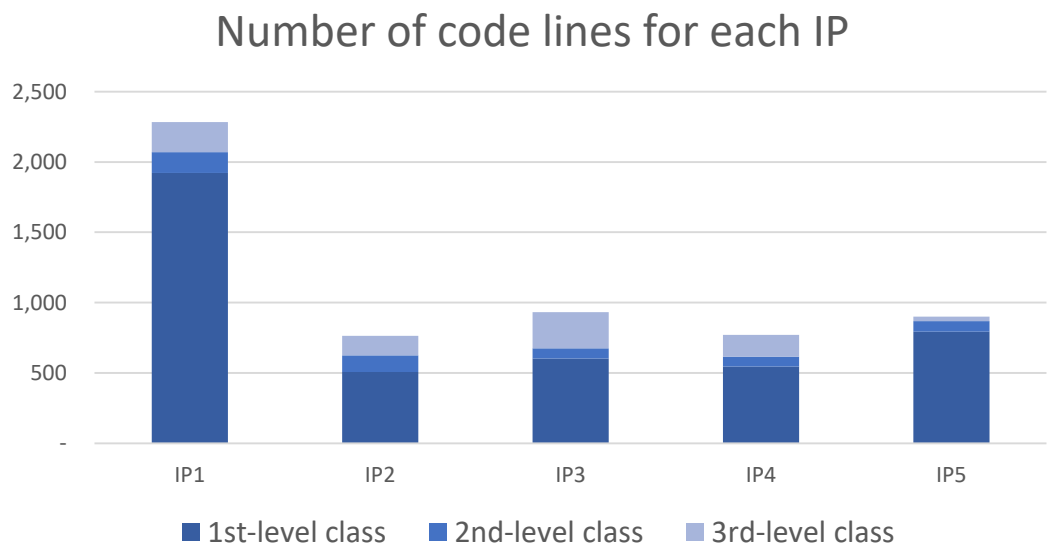
- Easy to define/implement complex test scenarios without any background knowledge of the UVM language
- Fast bring-up without errors typically incurred by human coding
- Hit the corner cases with efficient scenario configuration
  - → Achieve the target verification completeness quickly
  - → Shorten SoC verification TAT

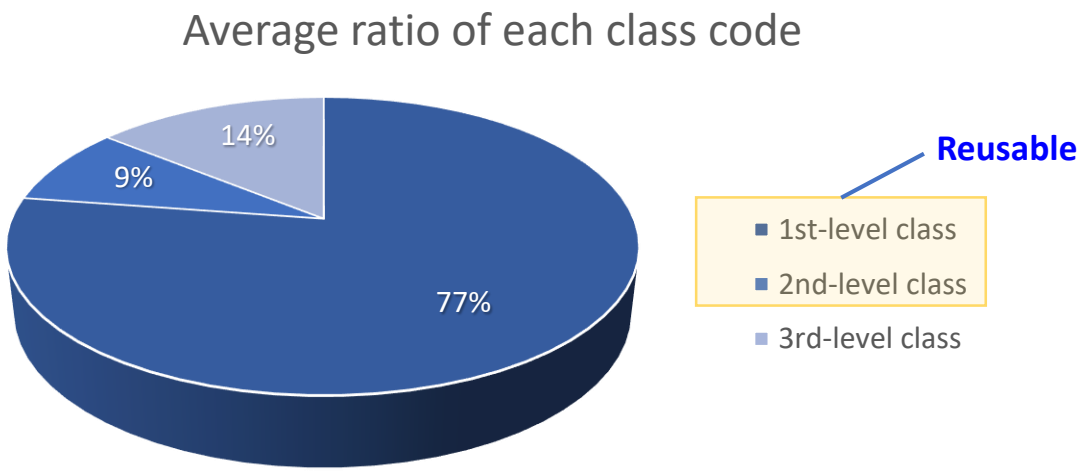
# Agenda

- Motivation
- Maximization of Code Reusability through Hierarchical Structure
- Increasing Efficiency of Complex/Composite Scenario Development
- Experimental Results
- Conclusion

# Code Reusability

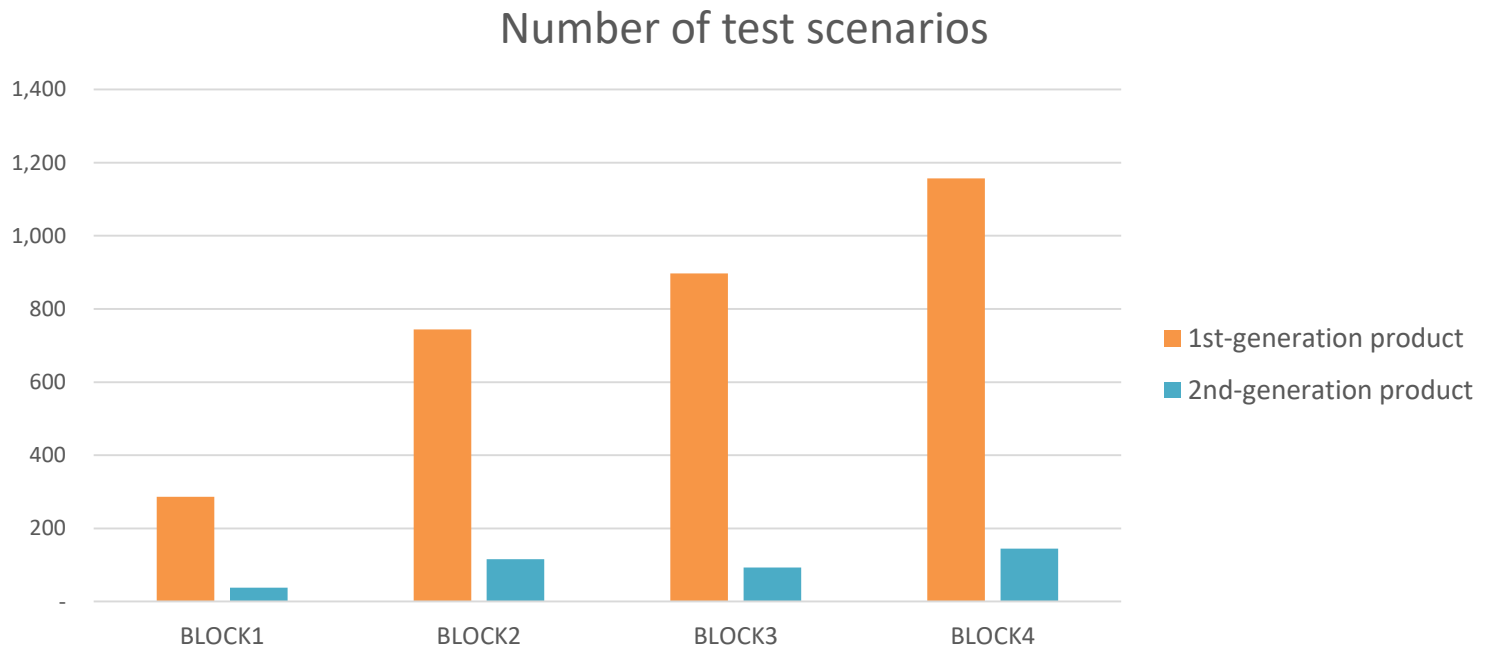
- On average, 86% of the code belong to the reusable 1st-/2nd-level class
- Can implement test scenarios after modifying the remaining 14% of code according to the target SoC configuration



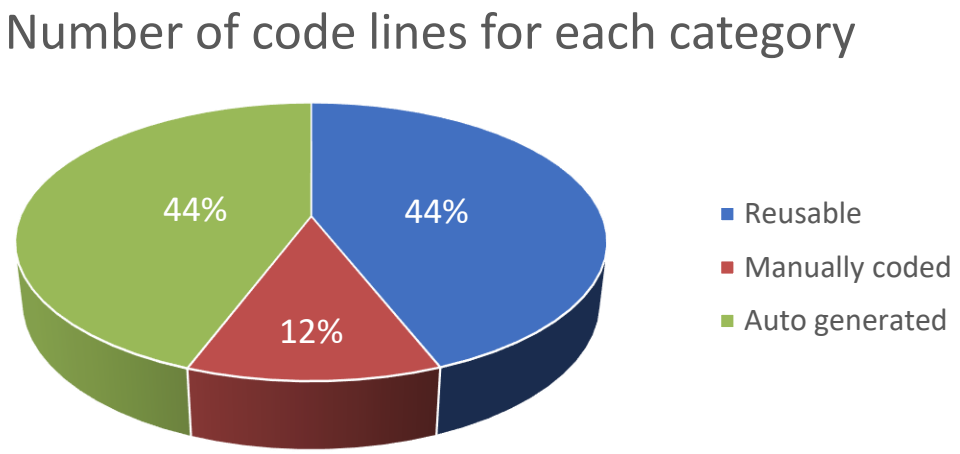
# Efficiency of Scenario Development (1)

- Reduce 87% of test scenarios
- Reach the same coverage with only 13% of total number of test scenarios



# Efficiency of Scenario Development (2)

- Fully reuse about 44% of total code without any modification
  - Reusable the first- and second-class of code
- Automatically generate about 44% of total code
  - The test scenario class which are automatically generated by the interpreter



# Agenda

- Motivation
- Maximization of Code Reusability through Hierarchical Structure
- Increasing Efficiency of Complex/Composite Scenario Development
- Experimental Results
- Conclusion

# Conclusion

- Layering code structure to maximize code reusability
  - Secure many IP functional operation test scenarios at the very early SoC design phase
- Efficient scenario development flow that contains metadata and interpreter
  - Easy to define/implement complex test scenarios combining lots of sequences/tasks from the library code
  - Find any complex design bugs and reach the target verification completeness quickly

# Questions