2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Vertical Reuse of Reference Models in UVM

Joachim Geishauser, NXP Deutschland GmbH, Munich, Germany (*joachim.geishauser@nxp.com*)

Ciro Ceissler, NXP Deutschland GmbH, Munich, Germany (*ciro.ceissler@nxp.com*)

Leo Tran, NXP Deutschland GmbH, Munich, Germany (*leo.tran@nxp.com*)

***Abstract*—These days the complexity of designs is growing faster than ever. This leads to the requirement of dividing and conquering the verification task in order to cope with the complexity. Besides formal verification of the individual blocks, random verification is often still required to cover scenarios that cannot be covered formally due to complexity. For the random verification a reference model is required in order to check if the response of the design is in line with the specification. The Universal Verification Methodology (UVM) provides concepts to build up a testbench hierarchical to reuse verification environments in a bottom-up divide and conquer verification flow. The vertical reuse of reference models, however, is not really documented in UVM. This paper shows an approach on how the vertical reuse of reference model can be achieved. (Style: Abstract)**

Keywords—SystemVerilog, UVM, Refernce Model, Vertical Reuse, TLM2 Generic Payload

I. INTRODUCTION

Random verification covers the verification space best when formal verification reaches its limit. In order to check the Design Under Test (DUT), typically, a reference model in different forms is used. On the other side the designs become more and more complex which leads to the fact that one Intellectual Property (IP) must be split into several sub-IPs to cope with the complexity. This divide and conquer creates the need to somehow combine sub-IPs reference models to ensure the front-to-end functionality. This paper outlines how the different challenges have been addressed to allow a seamless verification from sub-IP to top level reuse of reference model collaterals.

As a side effect of the testbench architecture the testbench can be used with SystemC functional model instead of SystemVerilog models. This effect will be elaborated in the subsequent sections.

The rest of the paper is organized as follows. Section II provides some background on UMV TLM2, outlines important features which are made use in our approach, and introduces a method to address multiple transport ports limitations. Section III and Section IV discuss the handling of datapath and configuration in reference models. Section V presents the scoreboard building blocks and the use of reference models for verification.

II. UVM TLM2 BASICS

TLM-2.0 is a part of IEEE 1666 which defines the SystemC Transaction-Level Modeling standard [1]. As a successor of TLM1, it addresses three major limitations of the early version. First, it standardizes transaction class. TLM-2 introduces generic payloads as the standardized transaction object that significantly improves interoperability between models from different sources. In TLM-1, the lack of such a class meant each application had to define its own transaction type. Second, TLM-2 allows timing information to be passed as function arguments in both blocking and non-blocking transport interfaces, enabling more accurate performance modeling. Third, TLM-2 allows passing transaction objects as non-const reference, which improves modeling performance compared to the copy or const-reference semantics required in TLM1.

TLM-2.0 has been adopted in Universal Verification Methodology (UVM), bringing multiple benefits in terms of model integration, interoperability, and simulation performance. Its adoption is largely driven by the need to integrate SystemC-based models into UVM-based verification environments – a common scenario in SoC/IP design verification projects. Additionally, TLM-2.0 improves simulation performance by enabling efficient communication and concurrency. The following outlines the main features of UVM TLM 2.0 including interfaces, sockets, generic payload, and its extension. For deeper technical insight, see [1].

Interfaces: UVM TLM-2.0 supports two transport interfaces: blocking and non-blocking, which associate to `b_transport` task and `nb_transport_*` functions

```
task b_transport(T t, uvm_tlm_time delay);
function uvm_tlm_sync_e nb_transport_fw(T t, ref P p, input uvm_tlm_time delay);
function uvm_tlm_sync_e nb_transport_bw(T t, ref P p, input uvm_tlm_time delay);
```

Listing 1. Task and functions of UVM TLM-2.0 blocking and non-blocking transports

In blocking transport, the task suspends its execution and only returns when the transaction is complete. It is opposite to non-blocking transport in which the calling function always returns immediately and there is also an indication to the caller whether or not the transaction is complete.

One key benefit of TLM2 is that transaction (`t`), phase (`p`), and time object (`delay`) can all be modified by the target and the changes are visible to the initiator. This supports advanced test scenarios involving phases, latency tracking, and fine-grained performance measurement. Due to its non-blocking nature, `nb_transport` provides many advantages over `b_transport`. Instead of waiting for the transaction to be completed like b_transport, the caller can continue its own processing task while the transaction progresses. nb_transport, therefore, allows for concurrent execution of multiple transactions - enabling high-performance simulation and better scalability. In addition, it provides better control over timing with phases and bi-directinoal transport – supporting an accurate modeling of complex system behavior. For these reasons, non-blocking transport is adopted in our work for building vertically reusable reference models.

Sockets: UVM TLM-2.0 defines two types of sockets representing bidirectional connection between the initiator and the target components: `uvm_tlm_nb_initiator_socket` and `uvm_tlm_nb_target_socket`. In typical non-blocking TLM-2.0 setup, each connection endpoint includes one of each socket type. These sockets must be bound to the implementation of `nb_transport_bw()` and `nb_transport_fw()`. Between endpoints, passthrough sockets may be inserted to support complex topologies. Sockets are created and connected like UVM ports and exports, since they are derived from `uvm_port_base#(IF)`.

Generic payload: the `tlm_generic_payload` class encapsulates transaction data and its attributes. It is designed to be flexible and extensible, allowing users to define their own transaction types and customize the payload as needed. A generic payload transaction has 10 attributes, with the most important being the command, address, data array, data length, and response status.

Extension: For protocols that require information beyond the standard payload fields, the extension mechanism can be used. The extensions are especially useful for non-memory mapped buses or user-defined metadata. Even in simple use cases, it is considered good practice to include extensions with payload [2]. The following shows a simple use of generic payload and extensions.

```
class rm_config extends uvm_tlm_extension #(rm_config);
  rand bit setting1;
  rand bit [15:0] setting2;
  `uvm_object_utils_begin(rm_config)
    `uvm_field_int(setting1, UVM_DEFAULT)
    `uvm_field_int(setting2, UVM_DEFAULT)
  `uvm_object_utils_end
  function new(string name = "rm_config");
    super.new(name);
  endfunction : new
endclass : rm_config
// to set extension
uvm_tlm_generic_payload gp;
rm_config cfg = new("cfg1");
gp.set_extension(cfg1);
// to get extensions
$cast(cfg1, gp.get_extension(ref_configuration::ID));

```

Listing 2. Extension declaration and usages

In UVM TLM-2.0, a component is limited to having only one interface per direction. Since sockets are bound to `nb_transport_fw` and `nb_transport_bw` along with the SystemVerilog limitation of single inheritance, this effectively restricts each UVM component to a single target socket and a single initiator socket. To address this limitation, the following mechanism is implemented using macros, enabling flexibility in handling multiple sockets within a component.

```
`define NXP_UVM_TLM_NB_TARGET_SOCKET_IMP(REF,SFX) \
class nxp_uvm_tlm_nb_target_socket_``SFX #(type T=uvm_tlm_generic_payload, \
                                           type P=uvm_tlm_phase_e) \
  extends uvm_component; \
  uvm_tlm_nb_target_socket #(nxp_uvm_tlm_nb_target_socket_``SFX,T) \
```

```
6                                m_nb_target_socket; \
7    local ``REF l_parent; \
8    function new(string name, uvm_component parent); \
9      super.new (.name(name),.parent(parent)); \
10     $cast (l_parent, parent); \
11     m_nb_target_socket = new("m_nb_target_socket", this); \
12   endfunction : new \
13   function uvm_tlm_sync_e nb_transport_fw(T t, \
14                                           ref P p, \
15                                           input uvm_tlm_time delay); \
16     if (l_parent) return l_parent.nb_transport_fw_``SFX(t, p, delay); \
17   endfunction : nb_transport_fw \
18   virtual function uvm_tlm_sync_e nb_transport_bw(T t, ref P p, input uvm_tlm_time delay); \
19     return m_nb_target_socket.nb_transport_bw(t, p, delay); \
20   endfunction : nb_transport_bw \
21 endclass : nxp_uvm_tlm_nb_target_socket_``SFX
22 `define NXP_UVM_TLM_NB_TARGET_SOCKET(REF,SFX)
23
```

Listing 3. Overcoming of the port limitation in UVM TLM-2.0 with predefined macro

For an example of using the macro in reference models, please refer to Listing 4, lines 17-24. The example shows how two target sockets and two initiator sockets can be added to one class and how the two call back functions receive the generic payload on these ports.

III. DATAPATH HANDLING

Any IP exposes several orthogonal functional aspects – register setting, clock- and reset management, power states, interrupt handling, and the data path. The data path represents the flow of data coming in from one or more ingress ports, is optionally transformed, and exits through one or more egress ports. Because it embodies the IP's primary algorithmic function, it is usually the performance-critical region and the part that is replicated throughout higher-level hierarchies (sub-system, SOC). A scalable and reusable approach for data path handling is therefore essential for vertical reuse of reference models.

To ensure portability of the reference model across different abstraction levels, every data path transfer is represented using the TLM-2.0 Generic Payload (GP). The core idea is to keep data movement through various design components and abstraction levels as simple and uniform as possible. The GP's core fields (command, address, data, and length) are flexible and sufficient for tracking the flow of information across the system. As discussed earlier, TLM-2.0 GP offers key advantages in terms of interoperability and simulation performance. For any functionality beyond these core fields, a dedicated extension bundle is recommended. This bundle can carry side-band information such as source ID, QoS, clock domain, parity or ECC codes, and timing information. Keeping this metadata separate from the core payload allows a single, generic structure to support a wide variety of protocols without requiring a dedicated extended object class for each data path. A common practice is to use adapters to bridge protocol-specific data representations between components, allowing each block to communicate through the uniform GP format.

Figure. 1 shows an example of an IP subsystem. IP1 receives and transforms data from different sources and different bus protocols. The outputs of IP1 are then processed by IP2 and IP3 before writing into the memory. IP1 assembles two Sub IPs, Sub-IP1 and Sub-IP2, and optional Sub-IP3 depending on IP configuration. Sub-IP1 and Sub-IP3 share the same algorithmic data processing function but are distinguished in communication logics bound to protocol specifications of external interface 1 and external interface 2. Considering data abstraction with general payload, a single reference model can be used for these Sub-IPs.

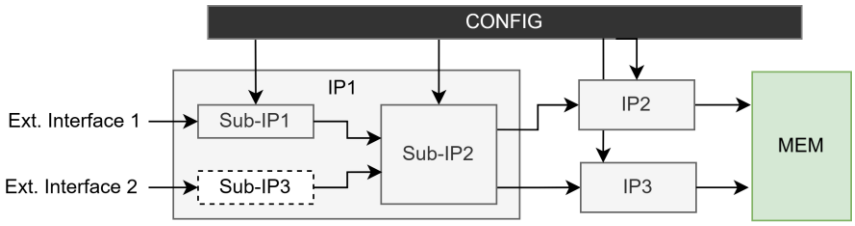


Figure 1. Example of IP subsystem

Figure 2 illustrates the data path view of IP1 reference model. The data path connectivity of reference models is established through target and initiator sockets. Ingress ports and egress ports are realized with `uvm_tlm_nb_target_socket` and `uvm_tlm_nb_initiator_socket` respectively.

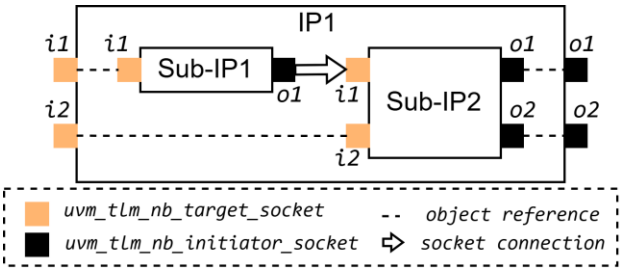


Figure 2. Datapath view of IP1 reference model

By employing the macro introduced in Listing 3, the limitation of a single socket per input or output can be addressed. Each socket instance, whether target or initiator, has its own corresponding `nb_transport_fw` and `nb_transport_bw` functions, as exemplified in the Sub-IP2 class. Within higher-level IP blocks such as IP1, subcomponents like Sub-IP1 and Sub-IP2 can be connected directly via their sockets by invoking the `connect()` function of `m_nb_initiator_socket.`

```
1  class RM_SUBIP1 extends uvm_component;
2    `uvm_component_utils(RM_SUBIP1);
3    `NXP_UVM_TLM_NB_TARGET_SOCKET(RM_SUBIP1, i1)
4    nxp_uvm_tlm_nb_target_socket_i1 #(uvm_tlm_generic_payload) i1;
5    `NXP_UVM_TLM_NB_INITIATOR_SOCKET(RM_SUBIP1, o1)
6    nxp_uvm_tlm_nb_initiator_socket_o1 #(uvm_tlm_generic_payload) o1;
7
8    function uvm_tlm_sync_e nb_transport_fw_i1 ( uvm_tlm_generic_payload payload, ref
   uvm_tlm_phase_e tlm_phase, input uvm_tlm_time tlm_delay);
9      // Data processing ...
10     void'(o1.nb_transport_fw (payload, tlm_phase, tlm_delay));
11   endfunction : nb_transport_fw_i1
12 endclass : RM_SUBIP1
13
14 class RM_SUBIP2 extends uvm_component;
15   `uvm_component_utils(RM_SUBIP2);
16     `NXP_UVM_TLM_NB_TARGET_SOCKET(RM_SUBIP2, i1)
17   nxp_uvm_tlm_nb_target_socket_i1 #(uvm_tlm_generic_payload) i1;
18   `NXP_UVM_TLM_NB_TARGET_SOCKET(RM_SUBIP2, i2)
19   nxp_uvm_tlm_nb_target_socket_i2 #(uvm_tlm_generic_payload) i2;
20   `NXP_UVM_TLM_NB_INITIATOR_SOCKET(RM_SUBIP2, o1)
21   nxp_uvm_tlm_nb_initiator_socket_o1 #(uvm_tlm_generic_payload) o1;
22   `NXP_UVM_TLM_NB_INITIATOR_SOCKET(RM_SUBIP2, o2)
23   nxp_uvm_tlm_nb_initiator_socket_o2 #(uvm_tlm_generic_payload) o2;
24   ...
25 endclass: RM_SUBIP2
26
27 class RM_IP1 extends uvm_component;
28   `uvm_component_utils(RM_IP3);
29   RM_SUBIP2 sip1;
30   RM_SUBIP2 sip2;
31   function void connect_phase(uvm_phase phase);
32     sip1.o1.m_nb_initiator_socket.connect(sip2.i1.m_nb_target_socket);
33   endfunction : connect_phase
34 endclass: RM_IP2
```

Listing 4. Implementation of data path with TLM sockets and generic payload

Beside the data checking on the data path, the timing of the data that propagates through the system may have to be checked. This depends on the specification of the IP. For this, the delay of the generic payload or an time stamp extension can be used to control and measure the data propagated to the system. The use of the TLM2 delay would be the active modeling whereas

the timestamp extension would be a passive one. The details on this are not captured in this paper as it would exceed the expected size of the paper.

The table below summarizes the objective of reference model reuse at different hierarchies and the benefit of the proposed data path handling.

Table 1. The reuse objective and benefit of data path handling across hierarchy levels.

Hierarchy Level	Re-use Objective	Benefit
IP	Golden functional prediction	Fast modeling (SystemC/System Verilog); Enable Scoreboard checking; Seamless integration into UVM Environment when RTL is in place.
Subsystem	Connect multiple IPs	NoC/Bus Adapter convert GP; functional model + Scoreboard reuse.
SOC	Early firmware and performance analysis	Entire data path execution is available. RTL drop-in with adapter change. Individually Switching between loosely timed and cycle-accurate model.

IV. CONFIGURATION

In addition to the data path, a design needs configuration for e.g., data setting, algorithmic functionality. This is typically done through a configuration interface. The configuration interface can be implemented in different ways. One way is to add a bus interface to a module to program register. The programmed register define the configuration of the module. Another approach is to have signals at the interface of a module that defines the configuration. These interface ports are then driven from another module in the design, e.g., by having a bus interface with registers capturing the configuration and providing the content via an interface to another module. As this paper deals with reference models to verify the behave of the design, the configuration information need to be populated on an abstract level to the reference model and on the other side in either driving a signal to a module or writing to a register in the System Verilog module via a bus interface to the design under test RTL.

A. Handling on the Abstract Level

The abstract configuration interface is required as some IPs will not have the registers implemented in the IP itself but rather have just ports that are fed by other IPs in the overall system. The configuration interface on the abstract level includes a UVM TLM generic payload extension to incorporate all necessary configurations for the IP. The register model abstracts register access via a defined interface to each address map. The predictor component populates an GP object with the extension, using information available through the updates on registers, and propagates them to the reference models via a TLM2 port. The advantage of having a UVM TLM2 GP port using an extension to provide this information is that IPs that have the configuration registers implemented in the IP itself will also have an TLM2 GP port that represents the register access interface of this IP.

B. Handling on the RTL Level

The RTL-level configuration interface is a sub-IP, where a standard interface is not available, and pin-level driving is necessary to propagate the information. In contrast to the configuration interface on the abstract level, the configuration on the RTL level is responsible for propagating configuration information to the RTL design.

For example, in a data path RTL design composed of multiple sub-IPs, configuration is performed via a standard interface connected to a register map. This register map translates configuration data into individual signals for each sub-IP. The configuration uses a consistent programming model - UVM Register Abstraction Layer – allowing to program the design independently sub-IP hierarchy or combination, which may vary due to design complexities and development schedules.

Each sub-IP includes UVM components to receive a configuration transaction, an UVM TLM generic payload extension, and drives accordingly to its interface. The UVM register model expects one address map for one interface, which translates the information to all sub-IPs. However, each sub-IP has a different interface for the same address map, and the UVM register model adapter allows only one UVM sequencer connection. To overcome this situation, the main interface sequencer will be extended to propagate every transaction received to each sub-IP's sequencer - listeners. Figure 3 below illustrates the components and interconnections.

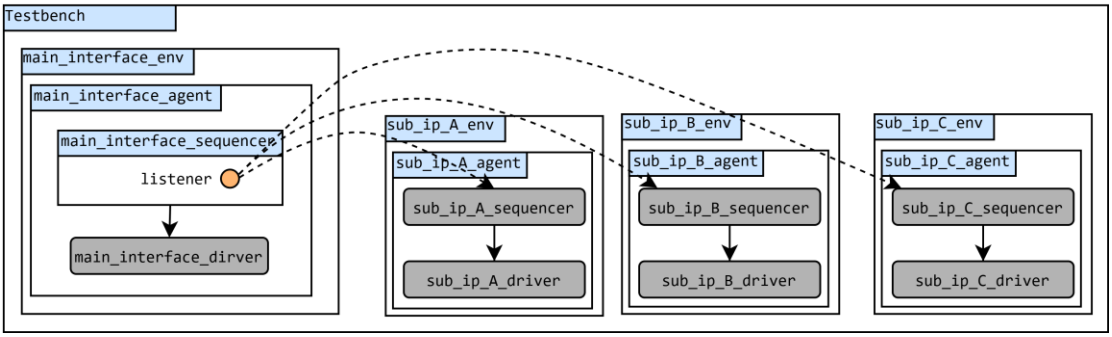


Figure 3. UVM Sequencer broadcasting a request to all listeners

The sequencer can either connect directly to a register model or a main interface sequencer with listener capabilities. In the second option, all environments are built, and the main interface sequencer connects to sub-IP sequencer pointers during the connection phase. The execution flow follows a daisy chain topology, where the main interface driver is executed first, and subsequently the sub-IPs will follow the flow. The `uvm_sequencer` implementation stores the transaction received during a call to `send_request` and during `wait_for_item_done`, sequentially executes the same transaction on all listeners. The code snippet below demonstrates the daisy-chain sequencer, excluding response handling.

```
1  class daisy_chain_sequencer extends uvm_sequencer#(uvm_tlm_gp);
2    uvm_sequencer listeners[];
3    protected uvm_sequence_item queue[$];
4    `uvm_component_utils(daisy_chain_sequencer)
5    function void send_request(uvm_sequence_base sequence_ptr,
6                               uvm_sequence_item t, bit rerandomize = 0);
7        super.send_request(sequence_ptr, t, rerandomize);
8        // add item to the queue to be consumed by `wait_for_item_done`
9        queue.push_back(t);
10   endfunction : send_request
11
12   task wait_for_item_done(uvm_sequence_base sequence_ptr, int transaction_id);
13       uvm_sequence_item t;
14       super.wait_for_item_done(sequence_ptr, transaction_id);
15       // remove item from the queue and execute on all listeners
16       t = queue.pop_front();
17       foreach (listeners[i])
18           listeners[i].execute_item(t);
19   endtask : wait_for_item_done
20 endclass : daisy_chain_sequencer
21
```

Listing 5. Implementation of daisy chain-based sequencer

V. USING THE REFERENCE MODEL TO VERIFY THE DESIGN

Using a reference model to verify a design using random sequences is not new. The approach shown in this paper does make a difference as it uses active reference models and does not require monitors on each and every interconnect of the design. The method used is to do end-to-end checking only. The active model usage has the advantage that these models can be implemented in a top-down design approach first and can be also implemented e.g., in the SystemC. Second, as these models are driven by GP, they are independent of vendor specific implementations. Third, the generic interfaces allow the generation of score board build blocks which makes the assembly of testbench fast due to a high level of code reuse.

A. Score Board Build Blocks

To maximize the reusability of reference models, a set of building blocks is introduced to facilitate connectivity and simplify output check. The building blocks are categorized into two groups: check types and connect types as listed in Table 2 and Table 3 respectively.

Table 2. NXP UVM TLM Building Block Check Types

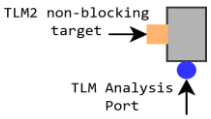
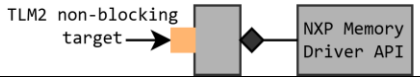
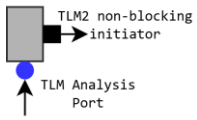
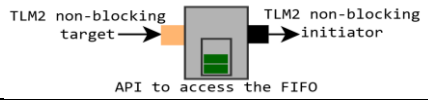
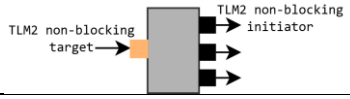
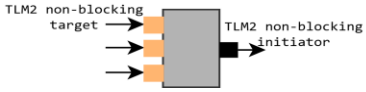
Name / Interface	Description
nxp_uvm_tlm_gp_subscriber_target 	Checks GP received via the analysis port against the GP received from the TLM2 target socket.
nxp_mem_driver_tlm2_gp_subscriber_target 	Checks GP received via TLM2 target socket in the UVM check phase against the data seen by the NXP memory driver API.

Table 3. NXP UVM TLM Building Block Connect Types

Name/ Interface	Description
nxp_uvm_tlm_gp_subscriber_initiator 	Forwards GP from analysis port to TLM2 initiator socket.
nxp_uvm_tlm2_b_target_to_nb_initiator 	Connects TLM2 blocking interface to non-blocking interface.
nxp_uvm_tlm2_nb_passthrough_fifo_target 	Pass through a TLM2 non-blocking target socket to a non-blocking initiator socket, while maintaining a reference of the transaction inside an internal FIFO.
nxp_uvm_tlm2_nb_target_broadcast 	Broadcast a transaction received from a TLM2 non-blocking target socket to multiple TLM2 non-blocking target sockets.
nxp_uvm_tlm2_nb_multi_targets_to_one_initiator 	Aggregate multiple transactions received from TLM2 non-blocking target sockets to one TLM2 non-blocking initiator socket.

B. Using the Reference Model in the Module Level

The starting point for a vertical reuse of a reference model is module level verification. Figure 4 shows the module level reference model, which is an abstract model of the IP. This means the IP does also have two data input ports and 2 output data ports as well as a configuration port.

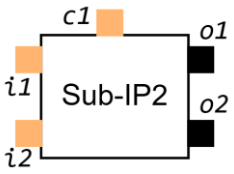


Figure 4. Sub-IP2 reference model with two data ingress ports (`i1, i2`), two data egress ports (`o1, o2`) and a configuration port (`c1`)

The testbench is built up as described in [3] which divides the reusable from the nonreusable code. In this context, the reference model is placed as a reusable component into the score board and is connected to the DUT using the build blocks described in

the previous section. The check is done by the checking blocks that compare the calculated responses from the reference model with the results generated by the DUT. The score board itself is not a reusable component in this concept.

C. Reusing a Module Level Reference Model

This section will outline how reference models can be used to verify the design. The reference models we use in this paper are active reference models. This means they provide an active response to a given input. This is different from a scoreboard, where the given input is checked by the scoreboard checking function. The checking of the reference models in this paper is done by generic checking blocks that are added to the outputs of the reference models. These checking blocks are on Table 2 with the distinct types of checking being push and pull driven. This leverages the UVM phasing to ensure correct checking. The figure below illustrates how to reuse reference models with building blocks with two different checking options.

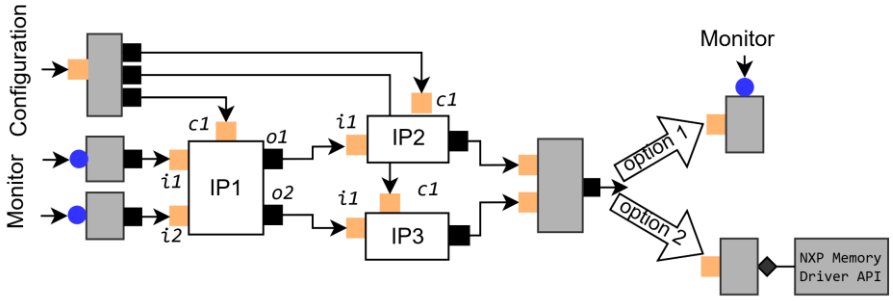


Figure 4. Reusing Reference Models with Building Blocks

Figure 4 illustrates an interconnection of multiple module level reference models and additional elements on the testbench, e.g., a monitor from an external interface and register model predictor. The reference models have configuration propagated from a register map, transactions from other models, and external interfaces. The output from the reference models that are not propagated to other ones is written to a single output port. The outputs are compared against memory interface to write transactions published, i.e., *option 1* in Figure 4. The code snippet below illustrates a scoreboard example:

```
class scoreboard extends uvm_scoreboard;
  // port input
  nxp_uvm_tlm_gp_subscriber_initiator m_interface_1_input;
  nxp_uvm_tlm_gp_subscriber_initiator m_interface_2_input;
  // configuration
  reg_predictor m_reg_predictor;
  // reference models
  rm_ip_1 m_rm_ip_1;
  rm_ip_2 m_rm_ip_2;
  rm_ip_3 m_rm_ip_3;
  // building blocks
  nxp_uvm_tlm2_nb_target_broadcast m_broadcast_config;
  nxp_uvm_tlm2_nb_targets_to_one_initiator m_aggregate_targets;
  // checking
  nxp_uvm_tlm_gp_subscriber_target m_check_output;
  ...
  virtual function void build_phase (uvm_phase phase);
    ...
    m_broadcast_config.create_broadcast_ports(3);
    m_aggregate_targets.create_target_ports(2);
  endfunction : build_phase

  virtual function void connect_phase (uvm_phase phase);
    super.connect_phase (phase);
    // interface inputs
    m_interface_1_input.m_nb_initiator_socket.connect(m_rm_ip_1.m_port_in.m_nb_target_socket);
    m_interface_2_input.m_nb_initiator_socket.connect(m_rm_ip_1.m_port_in.m_nb_target_socket);
    // configuration connections
    m_reg_predictor.config_out.m_nb_initiator_socket.connect(m_broadcast_config.socket_in);
    m_broadcast_config.socket_out[0].connect(m_rm_ip_1.config_in.m_nb_target_socket);
```

```
31      m_broadcast_config.socket_out[1].connect(m_rm_ip_2.config_in.m_nb_target_socket);
32      m_broadcast_config.socket_out[2].connect(m_rm_ip_3.config_in.m_nb_target_socket);
33      // reference models connections
34      m_rm_ip_2.m_port_out[0].m_nb_initiator_socket.
35              connect(m_rm_ip_2.m_port_in.m_nb_target_socket);
36      m_rm_ip_2.m_port_out[1].m_nb_initiator_socket.
37              connect(m_rm_ip_3.m_port_in.m_nb_target_socket);
38      // checking connections
39      m_rm_ip_2.m_port_out.m_nb_initiator_socket.connect(m_aggregate_targets.socket_in[0]);
40      m_rm_ip_3.m_port_out.m_nb_initiator_socket.connect(m_aggregate_targets.socket_in[1]);
41      m_aggregate_targets.socket_out.connect(m_check_output.m_nb_target_socket);
42    endfunction : connect_phase
43  endclass : scoreboard
```

Listing 5. UVM scoreboard

The IPs' configuration is managed by a register model predictor, knows a register model mirror on observed bus transaction. It receives registers configuration as a subscriber component and transfer via a TLM2 non-blocking initiator port. However, the port allows one-to-one connection and `nxp_uvm_tlm2_nb_target_broadcast` broadcast configuration transactions to all IPs. During the build phase, the broadcast component, called `m_broadcast_config`, is built with a pre-defined number of broadcast ports, line 19-20 Listing 5. Then, in the connect phase, the register predictor is connected directly to the broadcast component and outputs to their respective IPs, line 26-41 Listing 5.

The `m_interface_1_input` and `m_interface_2_input` represent two interfaces where they receive a GP transaction from their monitors. The `nxp_uvm_tlm_gp_subscriber_initiator` translates a transaction received to a new transaction-level interface, i.e., TLM2, via a non-blocking socket initiator. Both interfaces, after translation, connect to IP 1. The intra-connection of reference models is mapped directly because both input and output interfaces follow a TLM2 non-blocking implementation. Finally, IPs outputs are combined with `nxp_uvm_tlm2_nb_targets_to_one_initiator` to generate a single TLM2 non-blocking output port that is connected to a subscriber of type `nxp_uvm_tlm_gp_subscriber_target.` The subscriber receives a GP transaction from reference models and compares against a transaction received from a monitor attached to the output interface of RTL.

Concerning *option 2* in Figure 4, the checking is done by a building block with direct connection to the memory via NXP memory driver API [3]. Instead of receiving a transaction from the monitor and checking during the UVM run phase, this moves to the UVM check phase. The `nxp_mem_driver_tlm2_gp_subscriber_target` receives a transaction from an initiator and compares against the memory. The code snippet below shows the necessary modifications.

```
1  // checking
2  nxp_mem_driver_api m_mem_driver_api;
3  nxp_mem_driver_tlm2_gp_subscriber_target m_mem_check;
4
5  virtual function void build_phase(uvm_phase phase);
6    ...
7    m_mem_check=nxp_mem_driver_tlm2_gp_subscriber_target::type_id::create("m_mem_check", this);
8    m_mem_check.set_mem_driver_api(m_mem_driver_api);
9  endfunction : build_phase
10 virtual function void connect_phase(uvm_phase phase);
11   ...
12   m_aggregate_targets.socket_out.connect(m_mem_check.m_nb_target_socket);
13 endfunction : connect_phase
```

Listing 6. Usage of NXP memory driver API for checker

Furthermore, one can make use of the `nxp_uvm_tlm2_nb_passthrough_fifo_target` (see Table 3) to eavesdrop on communication between reference models. This is especially useful in the case that an intermediate check in addition to an end-to-end check is required. The transaction captured on the FIFO with similar behavior to the UVM analysis FIFO may be used to improve observability with additional checkers or information for debugging. Listing 7 below illustrates how to modify the scoreboard to capture the transaction and use the building block API to manage the data.

```
1  nxp_uvm_tlm2_nb_passthrough_fifo_target m_passthrough_fifo_target;
2  virtual function void connect_phase (uvm_phase phase);
3    ...
```

```
  // reference models connection
  m_rm_ip_2.m_port_out[0].m_nb_initiator_socket.connect(m_passthrough_fifo_target.socket_in);
  m_passthrough_fifo_target.socket_out.connect(m_rm_ip_2.m_port_in.m_nb_target_socket);
  ...
endfunction : connect_phase

virtual task main_phase(uvm_phase phase);
  ...
  forever begin
    m_passthrough_fifo_target.get(payload);
    ...
  end
endfunction : main_phase
```

Listing 7. UVM scoreboard

VI. CONCLUSION

The concept presented in this paper differs from the traditional UVM concept which reuses all code in the next level up with by turning off active components of the lower level. The proposed approach divides the code into reusable and non-reusable code as part of the testbench architecture and therefore saves a lot of code compilation as well as execution that will not be activated (numbers to be added later).
The other concept shown in this paper is the use of the GP with extensions instead of interface specific objects that allow the creation of standard building blocks as well as, although not shown in this paper, an easy way to use SystemC components. The use of SystemC components hereby fit to a top-down approach where the function of the system is first defined as SystemC functional blocks and then handed off to the implementation stage as executable specifications.
The GP also abstracts the interface type away from the reference model, which supports the use of functional code for IP implementation on different physical interfaces. The definition of standard extension, e.g. from ARM would further unify this concept. As of now, these user extensions carry the same content but are based on different variables and class names for different projects. Last but not least, the use of the GP extension to carry configuration data has created a high flexibility for the actual implementation of the configuration data.

VII. ACKNOWLEDGMENT

This paper is part of the IPCEI ME/CT and is funded by the European Union Next Generation EU, the German Federal Ministry for Economic Affairs and Energy, the Bavarian Ministry of Economic Affairs, Regional Development and Energy, the Free State of Saxony with the help of tax revenue based on the budget approved by the Saxon State parliament and the Free and Hanseatic City of Hamburg.

VIII. REFERENCES

[1] “IEEE Standard for Standard SystemC Language Reference Manual,” IEEE Std 1666-2024
[2] Delbergue, Guillaume, Mark Burton, Bertrand Le Gal, and Christophe Jego. "Analysis of TLM-2.0 and it’s Applicability to Non Memory Mapped Interfaces.", the Design & Verification Conference & Exhibition United State. (DVCon 2016).
[3] Geishauser et. al. , “uvm_mem – challenges of using UVM infrastructure in a hierarchical verification”. the Design & Verification Conference & Exhibition Europe (DVCon 2022)