A Comprehensive UVM Verification Environment for MPEG Transport-Stream Processing

Filippo Borlenghi
Simone Borri
Gherardo Gorni

ALi Europe
Plan-les-Ouates, Switzerland

www.alitech.com

ABSTRACT

We present a comprehensive UVM testbench to verify components of an MPEG transport-stream processor, from single blocks to sub-systems. The verification environment features random-constrained generation of data inputs and device configurations, coverage information collection and assertion-based protocol checkers on communication interfaces. In our hierarchical approach, a set of dedicated classes is associated to each hardware block to define its configuration, behavioral model and scoreboard. When interconnecting multiple blocks, these classes can be easily reused and composed into their system-level equivalent. The testbench further leverages the capabilities of SystemVerilog and UVM to spread complexity across abstraction layers. For instance, hardware configurations are captured by high-level randomizable classes, translated into bit-true register settings by helper functions and applied to the device via RAL. Working with high-level configurations instead of registers significantly facilitates the modification and extension of tests, simplifies the definition of coverage items and improves the readability of the results.

Table of Contents

Table of Figures

1. Introduction

Complex SoC designs are generally composed of several hardware blocks which perform different functions and communicate through standardized interfaces to facilitate their integration into a working system. The verification of such a design has to be carried out on different levels, from the single hardware blocks to critical sub-systems composed of multiple blocks and finally to the complete system. It is therefore important to ensure that as many components of the verification environment as possible can be reused, to reduce the development effort as the verification moves towards the top level.

Many different aspects of the design have to be tested and the focus of the verification process tends to shift while moving from block to system level. To begin with, on the block level the main goal is to thoroughly verify the correct implementation of the inner functionality and signal interface. After achieving this target, different blocks can be integrated and the verification becomes more concerned with their interaction and the performance of the resulting system, to ensure for instance that the communication overhead is not an issue. Therefore, the verification environment has to include the facilities to analyse and validate such properties as latency and throughput.

Another key aspect that the testbench has to support is reconfigurability, which refers both to the configuration of the hardware under test and to the way the input data is generated. Since bugs in the process of runtime reconfiguration are fairly common, an extensive testing of this feature is very important. Furthermore, to improve coverage and increase the likelihood of catching unforeseen corner cases, input data and hardware configurations should be generated in a (constrained) random fashion.

In short, the list of requirements for the verification environment includes:

- Modular structure with reusable components;
- Golden behavioural models of the functionality of each hardware block;
- Interface protocol checking;
- Timing properties (e.g., latency and throughput) monitoring;
- Runtime reconfigurability;
- Constrained random generation of input and configuration data;
- Coverage information collection.

The Universal Verification Methodology (UVM) [1] is very suitable for developing such an environment because it combines an object-oriented approach, which facilitates modularity and reusability, with the advanced verification facilities provided by SystemVerilog, particularly with respect to constrained random data generation, coverage and assertions.

Furthermore, the UVM class library includes APIs to speed up tasks whose implementation is necessary but often tedious and error-prone. For instance, employing the UVM Register Abstraction Layer (RAL) significantly simplifies accessing registers from the testbench and performing exhaustive reset and read/write access tests, based on sequences available out-of-the-box in the UVM package [2].

This paper describes the development of a UVM verification environment according to the aforementioned requirements in the context of MPEG Transport-Stream (TS) [3] processing. To this end, the application and the hardware blocks under verification will first be introduced in Section 2. Subsequently, the UVM testbench will be described in Section 3, with details about its structure, configuration mechanisms, test sequences and solutions to check that the hardware behaves as expected. Finally, conclusions will be drawn based on this development experience.

2. Design under Verification

The design under verification (DUV) includes three basic components for MPEG TS processing, which have to be verified both individually and collectively as a processing chain:

1. *Input synchronizer* (ISync), which receives a TS stream composed of 188-byte packets on an external interface, prepends their arrival time (measured by an internal counter) to the packets and synchronizes them with the internal clock of the design.
2. *Stream merger* (SM), which receives multiple TS streams, coming from the input synchronizer or pre-stored in memory, and merges them into a single one: if required, the arrival time can be taken into account to ensure a constant delay between the input and the output of the design; otherwise, a simple round-robin policy is applied to choose among the available packets at the different input ports.
3. *Output synchronizer* (OSync), which receives the merged TS stream and synchronizes it with the clock of the external interface; this clock can be either provided as an input to the design or internally generated.

These components communicate by means of three different interfaces:

1. *Advanced Peripheral Bus* (APB) [4], which is used to access the register interface of each block for configuration and monitoring purposes.
2. *External TS interface*, which is composed of a clock, an 8-bit data input, a data valid bit and a start bit to signal the first byte of a TS packet.
3. *Internal TS interface*, which is similar to the external one except for an additional handshake signal that the receiving block can use to stop the sender.

Each block is developed separately but its interfaces are compliant with these common specifications, so that assembling the chain shown in Figure 1 is then straightforward. This is only a sub-system of a larger stream-processing design. However, its functionality is critical since it includes the external interfaces (with the corresponding synchronization parts) and it enables the system to ensure a constant input-output delay, which is a key requirement of the application and hence needs to be thoroughly verified.

To conclude this overview of the DUV, the number of TS input interfaces of the SM is configurable at design time, for a quick deployment in different application scenarios. As a consequence, the testbench must support such parameterizability, which is not necessarily straightforward when using the UVM approach.

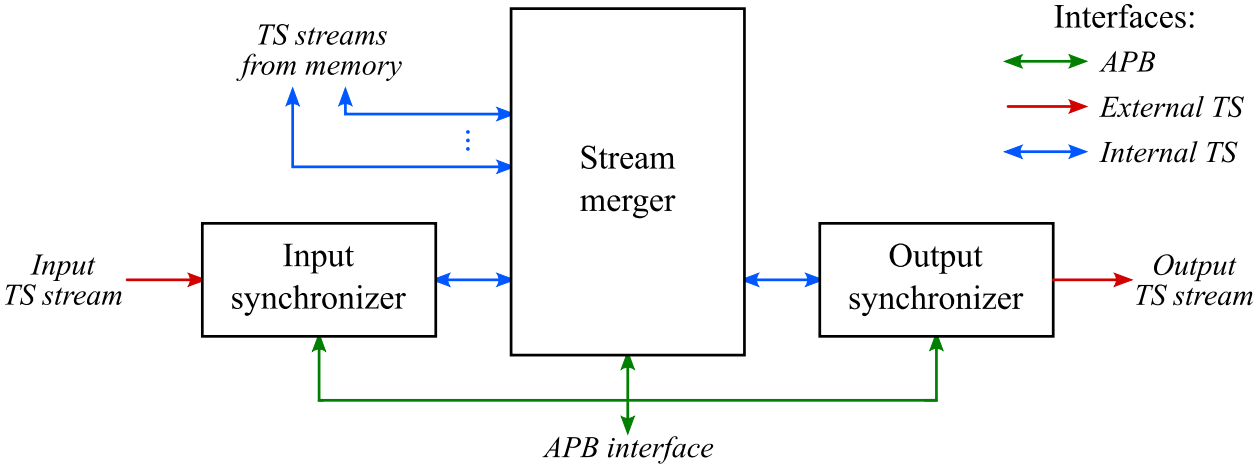

Figure 1 - Block diagram of the design under verification.

3. UVM-Based Verification Environment

a. Testbench Structure

The structure of the testbench, shown in Figure 2, follows the UVM principles. The interfaces of the DUV are connected to *agents*, which contain a monitor and/or a driver with its corresponding sequencer. An agent which can both drive and observe the interface is defined as *master*, while an agent that only includes a monitor is defined as *slave*. Agents are standard verification IP (VIP) components, which are developed once according to the interface specifications and then reused. In principle, in the use case discussed in this paper three types of agents are needed, one for each kind of interface. However, due to the strong similarities between the external and internal TS interfaces, the corresponding agents are unified in a single VIP whose actual interface is configurable in the build phase of the UVM environment. The same mechanism is used to configure whether the agent acts as a master or a slave.

Packet-based protocols, such as the MPEG TS, are particularly suited for the UVM transaction-based approach, since the agents can naturally operate on the TS packet granularity. Given the presence of several master agents, virtual sequences are used to orchestrate the transactions distributed to the different sequencers.

The testbench is designed hierarchically to verify each hardware block individually as well as the sub-system as a whole. To this end, the following set of classes is associated to each block:

- *High-level configuration*, which includes all the relevant settings of a block (e.g., the merging policy to be used by the SM or the frequency of the internally-generated clock in

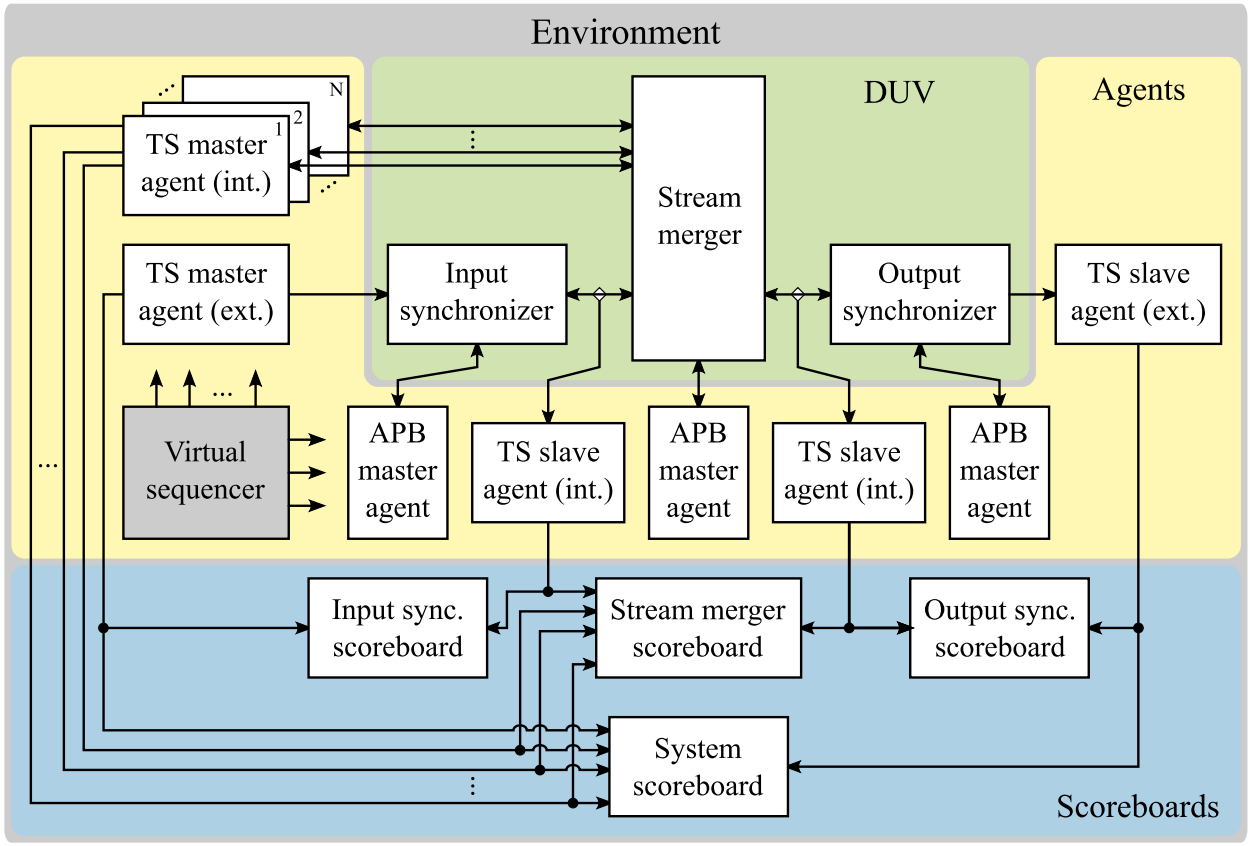

Figure 2 - UVM testbench structure.

the OSync). This class also provides helper functions to translate the high-level settings into the corresponding bit-true values of the hardware registers. By introducing this abstraction, tests can be developed independently of the actual bit values of each register, with two main advantages: first, the developer's work becomes simpler and less error-prone; second, in case of a change in the register interface only the configuration class has to be modified instead of all the tests. Moreover, this class can be extended with some predefined configurations which are used in most typical cases.

- *RAL register model* (refer to Section 3.b for a detailed description).
- *APB configuration sequences*, which make use of RAL functions (e.g., enable/disable the block, full register reconfiguration).
- *Transformer*, i.e., a "golden" behavioural (or reference) model which performs the expected input-output transformation of the data.
- *Scoreboard*, which uses the transformer class to compute the expected outputs and compares them with the actual ones observed on the hardware block. Slave agents are deployed on the internal TS interfaces between different blocks to monitor the data streams and pass the TS packets on to the scoreboard.

These classes form a package which can be reused in case the associated hardware IP is instantiated in a design. Furthermore, they can be composed into the system-level testbench. For instance, in the presented use case, the system transformer class is simply built by calling the three block-level transformers sequentially. Similarly, the system enable, disable and reconfiguration sequences are the combination of the corresponding block-level sequences. The same principle applies for the system configuration class.

As mentioned in the Section 2, a key feature of the testbench is the support for a parameterizable number of TS input interfaces in the SM. To this end, an array of TS master agents is instantiated in the environment and connected to the SM. While this operation is straightforward, supporting a parameterizable array of master agents becomes tricky in other testbench components, such as scoreboards and virtual sequences. The corresponding solutions are discussed in the sections dedicated to these components.

b. Hardware Configuration Mechanism

The first operation that needs to be performed at the beginning of a test, after reset, is the configuration of the hardware blocks. To this end, each block under consideration provides a register file accessible via a standard AMBA APB interface. These registers contain not only the settings but also status information, e.g., about internal FIFOs or memory transfers.

These register files are first modelled in the RAL Format (RALF), which is then passed as input to the Synopsys *ralgen* utility to automatically generate the SystemVerilog functions which access the registers. An exemplary RALF model of the OSync registers is given in the following.

```
block OSYNC {
  register CFG (i_osync_regif.cfg_reg) @'h0000 {
    # select internal or external clock, rising or falling edge
    field CLK_EDGE @2 { enum {FALL, RISE}; bits 1; reset 1'h0; access rw; }
    field CLK_SEL  @1 { enum {EXT,  INT }; bits 1; reset 1'h0; access rw; }
    field OSYNC_EN @0 { enum {DIS,  ENA }; bits 1; reset 1'h0; access rw; }
  }
  register INTCLK (i_osync_regif.intclk_reg) @'h0004 {
    # period of internally-generated clock
    field TCLK @0 { bits 24; reset 24'h000000; access rw; }
  }
```

```
  register ID (i_osync_regif.id_reg) @'h0008 {
    # version number of the hardware IP block
    field MAJ_REV_NUM @8 { bits 4; reset 4'h1; access ro; }
    field MIN_REV_NUM @4 { bits 4; reset 4'h0; access ro; }
    field PATCH_NUM   @0 { bits 4; reset 4'h0; access ro; }
  }
}

system osync_regmodel {
  block OSYNC (i_osync) @'h0000;
}
```

An advantage of the RAL-based approach is that it simplifies tracking late changes in the register interface, since its complete specification is contained in a single RALF file and the corresponding SystemVerilog code is automatically generated from it.
The output of *ralgen* is a class *ral_sys_osync_regmodel* which has to be instantiated in the UVM environment. This instance is then connected to the sequencer of the corresponding APB master agent through an adapter (*uvm_reg_adapter*), which converts between RAL (*uvm_reg_bus_op*) and APB (*apb_transaction* in our implementation of the APB VIP) transactions.
The HDL path of the register file can be specified to gain backdoor access to the registers. In order to maintain consistency between the actual value of the register and its mirrored version in the UVM model it is necessary to implement an explicit predictor which updates the mirrored value as soon as a transaction is reported by the monitor integrated in the APB agent. This is a straightforward operation that requires instantiating the *uvm_reg_predictor* class with the proper transaction type (*apb_transaction* in this case) and then connecting it to the APB monitor on the one hand and to the register adapter on the other hand. Using the implicit predictor does not require this extra step but does not guarantee that the mirrored version is updated immediately either, which leads to failures in register tests that check the value immediately after its modification.
The RAL infrastructure described so far provides a convenient way to access the configuration registers of the hardware, however it still requires the developer of the test to understand and access every single register (and possibly register field) to apply the desired settings. To move away from this low-level view, the first step is to define a configuration class including all the relevant fields in the register file as data members, in a readable form. Helper methods are also included to translate the values of these members into the actual bit pattern of the hardware registers, as illustrated by the following code.

```
class osync_config extends uvm_object;

  typedef enum bit {DIS=0,  ENA=1}  enable_e;
  typedef enum bit {FALL=0, RISE=1} edge_e;
  typedef enum bit {EXT=0,  INT=1}  clksrc_e;
  typedef bit [`APB_DATA_WIDTH-1:0] apb_data_t;

  // Block settings
  rand enable_e osync_en;
  rand clksrc_e clk_sel;
  rand edge_e   clk_edge;
  rand int      intclk_frequency; // in MHz

  // ...
```

```
  // Methods to retrieve the full 32-bit registers by combining
  // the individual fields

  function apb_data_t get_cfg();
    apb_data_t tmp_reg = 'b0;
    tmp_reg = tmp_reg | (  cfg_osync_en << 0
                         | cfg_clk_sel  << 1
                         | cfg_clk_edge << 2 );
    return tmp_reg;
  endfunction : get_cfg

  function apb_data_t get_intclk();
    apb_data_t tmp_reg = 'b0;
    bit[23:0] intclk_tclk = real'(`SCALING_FACTOR) / real'(intclk_frequency);
    tmp_reg = tmp_reg | intclk_tclk;
    return tmp_reg;
  endfunction : get_intclk

endclass : osync_config
```

This class can be easily extended to form a library of commonly-used configurations: in the OSync example, different configurations can be defined for different clocks (internal or external); the internal clock configuration can then be extended with a typical frequency value, as shown in the following listing.

```
// External clock configuration
class osync_config_clk_ext extends osync_config;
  // ...
  constraint clk_ext {
    clk_sel       == osync_config::EXT;
  };
endclass : osync_config_clk_ext

// Internal clock configuration
class osync_config_clk_int extends osync_config;
  // ...
  constraint clk_int {
    clk_sel       == osync_config::INT;
  };
endclass : osync_config_clk_int

// Nominal-case configuration
class osync_config_nominal extends osync_config_clk_int;
  // ...
  constraint nominal {
    osync_en         == osync_config::ENA;
    clk_edge         == osync_config::FALL;
    intclk_frequency == 10; // MHz
  };
endclass : osync_config_nominal
```

As a result, the test developer can either reuse one of the existing classes or further extend one according to his/her needs. Since the members of the configuration class are defined as random, the extending classes can simply apply proper constraints to obtain the desired configuration. On the other hand, leaving a given member unconstrained enables testing of randomly-varying

configurations. In this way, the approach of constrained random verification is seamlessly extended to the configuration of the hardware device, besides the input data.
In order to apply the configuration to the hardware block at runtime, specific sequences are provided by extending the *uvm_reg_sequence* class, mainly to enable/disable the block and to configure it. The latter reads the configuration class, calls its helper methods to convert it into the bit patterns of the concerned registers and finally writes them.
The configuration is normally set by the test, therefore a mechanism is necessary to make it available to the sequences and to the other components of the testbench. The UVM resource database, accessed by means of the *uvm_resource_db* class, is used to this end. The *uvm_resource_db* interface is preferred to the *uvm_config_db* one since it facilitates sharing the configuration objects across the hierarchy, even between components that belong to different scopes of the hierarchy, and also with sequences, which are not part of the component hierarchy. A thorough explanation of the advantages provided by the *uvm_resource_db* over the *uvm_config_db* approach can be found in [5].
The following listing shows a RAL configuration sequence which employs the *uvm_resource_db* interface to retrieve the hardware configuration.

```
class osync_ral_cfg_sequence extends uvm_reg_sequence #(apb_sequence_base);
  osync_config cfg;
  ral_sys_osync_regmodel regmodel;
  // ...

  virtual task pre_start();
    super.pre_start();
    if (!uvm_resource_db #(ral_sys_osync_regmodel)::read_by_name(
                           get_full_name(), "osync_regmodel", regmodel, this))
      `uvm_fatal("RAL_CFG", "regmodel not set");
    cfg = osync_config::type_id::create("cfg");
    // ...
  endtask

  task body();
    uvm_status_e status;
    // Retrieve block config and write it in the register file
    if (!uvm_resource_db #(osync_config)::read_by_name(
                                    get_full_name(), "osync_cfg", cfg, this))
      `uvm_fatal("RAL_CFG", "OSync config not found");
    regmodel.OSYNC.INTCLK.write(.status(status),
              .value(cfg.get_intclk()), .path(UVM_FRONTDOOR), .parent(this));
    regmodel.OSYNC.CFG.write(.status(status),
                  .value(cfg.get_cfg()), .path(UVM_FRONTDOOR), .parent(this));
  endtask // body
endclass : osync_ral_cfg_sequence
```

Further sequences are provided to cover the basic register tests, by using either the default UVM ones, such as *uvm_reg_mem_hdl_paths_seq*, *uvm_reg_hw_reset_seq* and *uvm_reg_access_seq*, or custom ones, for instance to test that the APB interface returns an error when accessing an unmapped address or when writing a read-only register. With the availability of this basic library of register-related sequences, the test developer does not normally have to implement any additional one.

c. *Test Sequences*

The presence of multiple drivers of different types requires a virtual sequence to properly orchestrate the individual sequences. Three main virtual sequences are provided:

1. *Basic* (Figure 3): it sequentially executes a reset phase, a configuration phase to set up the hardware and finally a data phase where the TS input ports are stimulated.
2. *Static reconfiguration* (Figure 4): after the initial reset phase, the configuration and data phases are alternated in a loop. This scenario tests reconfiguration without reset when the hardware is idle (i.e., no incoming data), which is particularly important since it often triggers dangerous bugs. In fact, while the basic virtual sequence with one-time configuration is mostly used in the initial verification phase, all the final tests make use of sequences with periodic reconfiguration.
3. *Dynamic reconfiguration* (Figure 5): some hardware settings can be changed on the fly, i.e., without disabling the hardware nor stopping the input stream. This feature is tested by running the configuration and data phases in parallel, with no mutual synchronization.

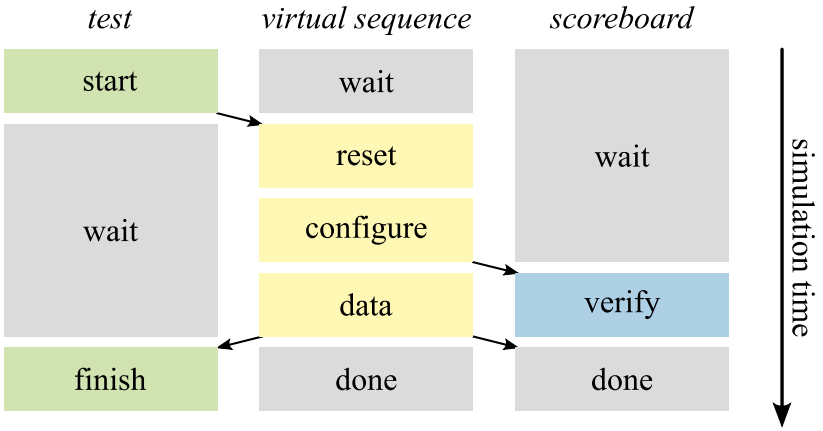

Figure 3 - Execution flow of a basic test.

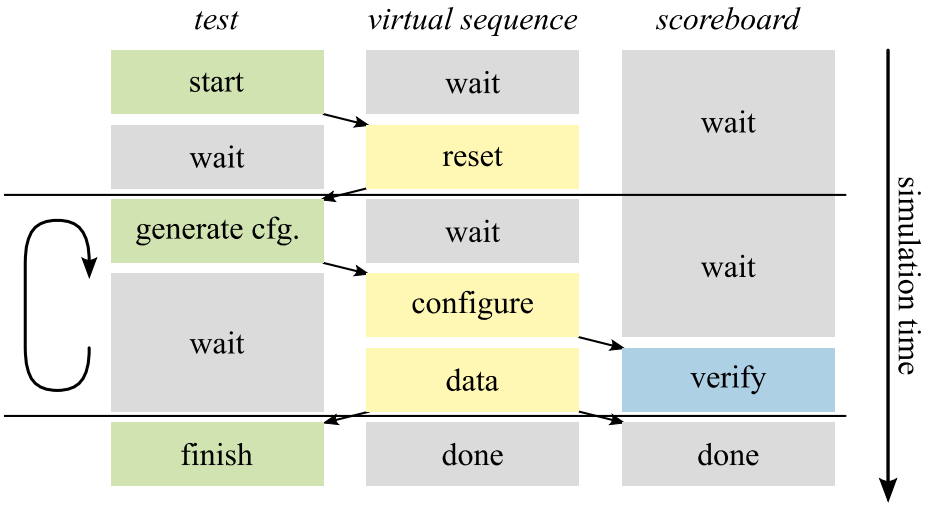

Figure 4 - Execution flow of a static reconfiguration test.

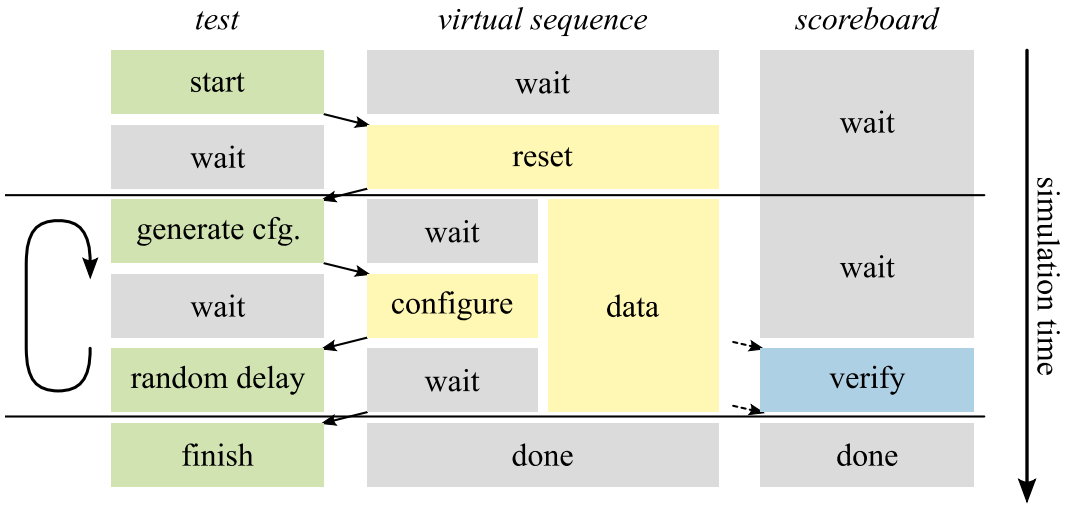

Figure 5 - Execution flow of a dynamic reconfiguration test.

While the order of execution of the individual APB (i.e., configuration) and TS (i.e., data) sequences is determined by the virtual sequences, it is the test class to control when reconfiguration happens and how the new configuration is generated. To this end, a synchronization mechanism is put in place between the virtual sequence and the test, based on events (indicated by arrows in the figures). The test class generates the new hardware configuration, makes it globally available via the UVM resource database and then triggers a configuration event. The virtual sequence receives this trigger and applies the new configuration to the hardware. Conversely, the virtual sequence signals the end of the configuration and data sequences to the test class, which can then either start a new iteration or terminate the test.

Additional synchronization is necessary between the virtual sequence and the scoreboard to make the latter aware of an ongoing reconfiguration, particularly if it is dynamic. Due to the processing latency and the non-zero time required to apply the new configuration to the hardware, the output observed by the testbench might not be "clean" for some time. Hence, the scoreboard should ignore input and output data during reconfiguration. Verification resumes with the output that corresponds to the first input arrived after the new configuration is in place.

In view of these synchronization mechanisms, each type of virtual sequence corresponds to a dedicated test class which implements the other side of the synchronization and sets the right virtual sequence via factory override. The resulting three classes are the extension of a common base test class which takes care of the instantiation and initialization of the environment with all the basic settings. New tests can be created by simply extending one of the three test classes with specific factory overrides for the hardware configuration classes and for the TS input sequences. An overview of the inheritance scheme used for the test classes is shown in Figure 6.

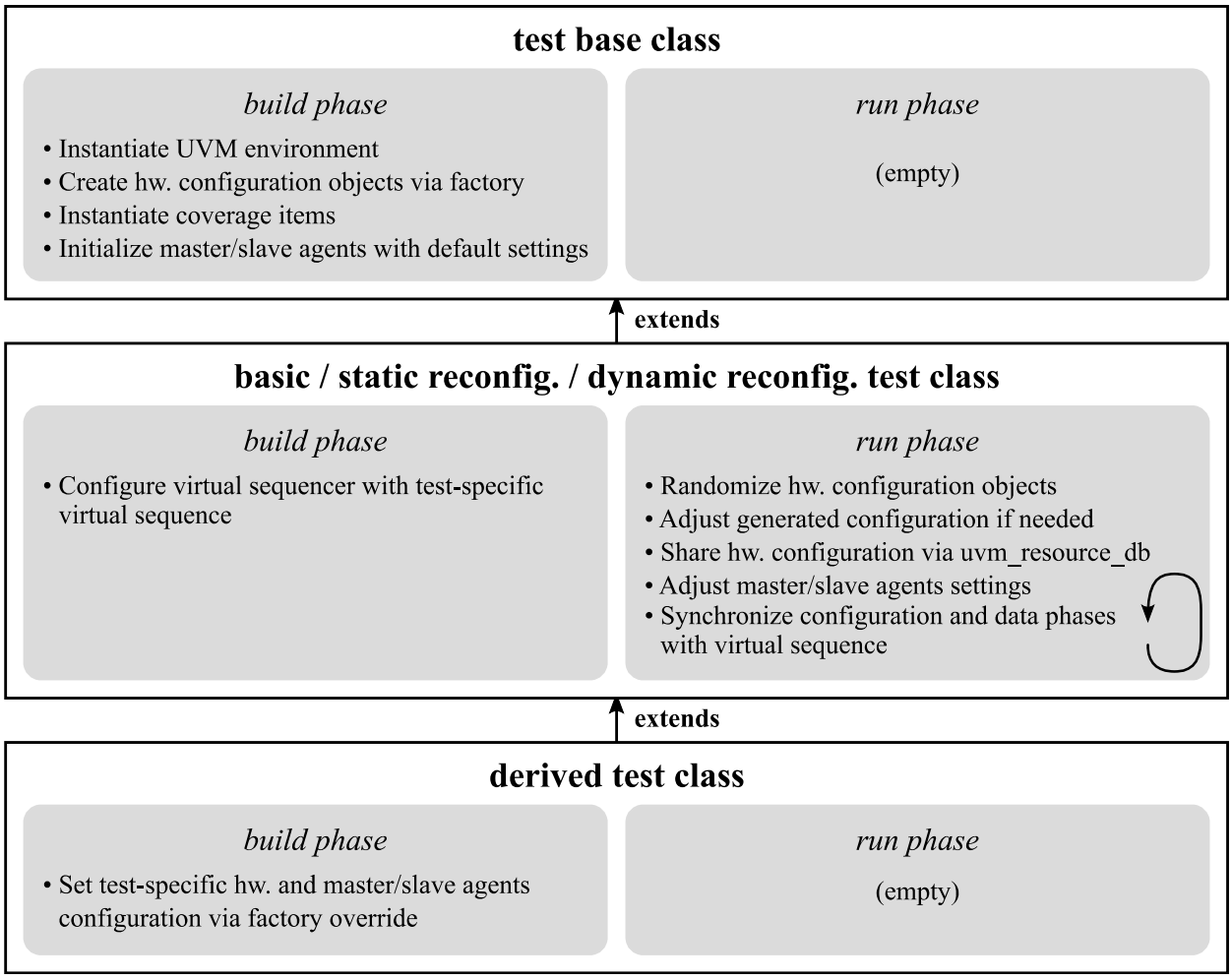

Figure 6 - Test class inheritance scheme.

An additional task of the test class is to ensure that the aggregated input bandwidth of all the TS ports does not exceed the output bandwidth. This goal is achieved by first computing the output bandwidth for the given OSync configuration and then randomly assigning a share of it to every active input port, with the obvious constraint that the total is not larger than the output bandwidth. For each input port, the bandwidth is then converted into an average delay between the incoming TS packets. In the sequences passed to the TS drivers, consecutive packets are then spaced by a random Gaussian-distributed delay centered around the average. This solution enables to validate the performance of the different blocks in a varying range of input conditions.
To conclude, a limitation that was encountered while developing virtual sequences is that using the predefined *`uvm_do_on* macro, even though convenient, hinders parameterizability. Given the array of master agents connected to the SM input ports, it would be convenient to start the corresponding array of sequencers with a parameterized "for" loop. However, since *`uvm_do_on* is a macro, the index of the sequencers cannot be a variable but has to be specified explicitely. Therefore, to maintain parameterized code, the *`uvm_do_on* macro is replaced here by calls to the *uvm_sequence_base* methods *randomize()* and *start()*, as shown by the following code.

```
class virtual_sequence_base extends uvm_sequence;
  // ...
  ts_packet_sequence sm_seq_in[`SM_N_PORTS - 1];

  function new(string name="virtual_sequence_base");
    // ...
    for (int i = 0; i < `SM_N_PORTS - 1; i++)
      sm_seq_in[i] =
         ts_packet_sequence::type_id::create($sformatf("sm_seq_in[%0d]", i));
  endfunction

  virtual task body();
    // ...
    fork
      for (int i = 0; i < `SM_N_PORTS - 1; i++) begin
        int local_i = i;
        fork
          begin
            if (!sm_seq_in[local_i].randomize())
              `uvm_error("VIRTUAL_SEQ", "Randomization failed");
            sm_seq_base[local_i].start(p_sequencer.sm_sequencer[local_i]);
          end
        join_none
      end
      wait fork;
    join
    // ...
    endtask
endclass : virtual_sequence_base
```

d. Functionality Verification

Each hardware block has a dedicated scoreboard which checks its functionality. Besides, as mentioned previously, the UVM testbench includes an additional system-level scoreboard. All of these components share the same basic structure, with “before” functions that record the inputs and an “after” function that observes and verifies the outputs. The expected values are generated by the *transformer* class, which represents a golden behavioral model of the hardware and is called within the “before” methods whenever a transaction (i.e., a TS packet) is received.

Both the SM and the system scoreboards deal with a parameterized number of TS inputs. To keep the code parameterized and enable the scoreboard to identify the TS master agent that generated an incoming transaction, each TS driver port is connected to a *uvm_subscriber* instance which explicitly calls the “before” function of the scoreboard. This function is extended with an extra input, i.e., the index of the TS driver that originated the transaction. This solution, depicted in the following, enables to store packets from different drivers in different queues.

```
class scoreboard_listener #(type scb_type, tr_type)
                                          extends uvm_subscriber #(tr_type);
  local int port_idx;
  function new(string name, uvm_component parent, int port_idx);
    super.new(name, parent);
    this.port_idx = port_idx;
  endfunction // new
  function void write(input tr_type tr);
    scb_type scb;
    $cast(scb, this.get_parent());
```

```
    scb.write_before_multi(tr, port_idx);
  endfunction
endclass : scoreboard_listener
```

```
`uvm_analysis_imp_decl(_sm_after)
class sm_scoreboard extends uvm_scoreboard;
  uvm_analysis_imp_sm_after #(ts_packet, sm_scoreboard) sm_after_export;
  scoreboard_listener #(sm_scoreboard, ts_packet)
                                          sm_before_listeners[`SM_N_PORTS];
  sm_transformer transformer;
  sm_config cfg;
  ts_packet pkt_list[`SM_N_PORTS][$];
  ts_packet new_pkt;

  // ...

  // Build phase
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sm_after_export = new("sm_after_export", this);
    for (int i = 0; i < `SM_N_PORTS; i++) begin
      sm_before_listeners[i] =
                    new($sformatf("sm_before_listeners_[%0d]", i), this, i);
    end
    transformer = sm_transformer::type_id::create("sm_transformer", this);
    cfg = sm_config::type_id::create("cfg");
  endfunction

  // Push the packets into queue
  virtual function void write_before_multi(ts_packet pkt, int id);
    if (!uvm_resource_db #(sm_config)::read_by_name(
                                      get_full_name(), "sm_cfg", cfg, this))
      `uvm_fatal("SM_SCB", "SM config not found");
    new_pkt = transformer.transform(pkt, cfg, id);
    pkt_list[id].push_back(new_pkt);
  endfunction : write_before_multi

  // Pop the packets from input queue and compare with the simulated output
  virtual function void write_sm_after(ts_packet pkt);
    for (int i = 0; i < `SM_N_PORTS; i++) begin
      // Compare pkt with the head of each queue pkt_list[i]
      // If a match is found, remove the packet from pkt_list[i]
    end
  endfunction : write_sm_after
```

When the SM is configured to use the round-robin policy with no guarantee on the input-output delay, the scoreboard cannot predict the global order of the output packets, which depends on the cycle-by-cycle operations of the hardware. However, the order for a given input port is preserved at the output. Therefore, when an output packet is received the scoreboard can search for it by comparing it with the head of each input queue.

When the merging of the input packets is done to ensure a constant input-output processing delay, the system-level scoreboard records the arrival time of packets both at the input and at the output. Based on this information, the input-output delay can be computed and checked for compliance with the maximum acceptable jitter.

e. Interface Verification

A key step in the verification of a hardware IP component concerns the interfaces, which have to comply with a common specification for a smooth integration with other blocks. First of all, the configuration interface must be tested: this task is covered by the register tests introduced in Section 3.b, which can be executed either as a stand-alone test or at the beginning of every test, as done in this work.

The second aspect that has to be verified is the actual timing of the signals at the input/output pins of the hardware block. For instance, in the internal TS interface it is important that the sender stops sending data within a given number of cycles after the receiver has signalled that it cannot accept any more data. This cycle number is part of a common specification and a failure to comply with it may cause a data loss if the receiver input buffer is full. Verifying this kind of behaviour can be done very effectively by SystemVerilog assertions: the timing specifications of the interface can typically be translated one-to-one to a set of assertions, referred to as *protocol checker*. SystemVerilog provides a very convenient mean to integrate protocol checkers in the testing environment: first, the signals that compose the interface are encapsulated by means of the SystemVerilog *interface* construct; then, the assertions are added inside this interface. In this way, protocol checking is performed seamlessly every time an interface is instantiated and connected to the I/O pins of the hardware. An example of this approach is shown by the following listing.

```
interface ts_interface( input bit clk , input logic rst_n);
  logic         ts_sync;  // TS interface clock signal
  logic [7:0]   ts_data;  // Data byte
  logic         ts_start; // Pulse to signal the beginning of a packet
  logic         ts_valid; // Asserted whenever the data is valid
  logic         ts_stop;  // Asserted by the receiver to stop the sender

  // Internal TS interface protocol checker

  // Check that data values are stable between two sync rising edges
  ts_stable_p: assert property
    (@(posedge clk)
      disable iff (~rst_n || $past(ts_sync) === 1'bx)
      (~$rose(ts_sync) && ts_valid) |->
               ($stable(ts_valid) && $stable(ts_data) && $stable(ts_start)));

  // Check that the ts_start pulse always lasts one valid ts_sync cycle
  sequence start_seq;
    @(posedge clk)
      (ts_start && ts_valid && $rose(ts_sync)) ##[1:$] $rose(ts_sync);
  endsequence
  ts_start_onecycle_p: assert property
    (@(posedge clk)
      disable iff (~rst_n || $past(ts_start) === 1'bx)
      first_match(start_seq) |-> (~ts_start || ~ts_valid));

endinterface : ts_interface
```

f. Coverage

Two different types of coverage are applied in this work. The first one aims to ensure that all the relevant configurations of the hardware blocks have been tested, while the second one targets specific combinations of signals that are identified as critical.

Information about configuration coverage can be easily obtained via RAL, by running *ralgen* with the switch "-c", which automatically generates cover groups for all the registers in the RALF model, and then enabling the coverage when instantiating the register model in the UVM environment. However, this solution results in a poor readability of the coverage report and does not enable an easy specification of cross-coverage items, necessary to ensure that certain combinations of settings have been exercised.

A better solution which overcomes these drawbacks is to specify coverage items on the high-level configuration classes introduced in Section 3.b. This approach is more flexible as it allows to define covergroups which only include the relevant settings and combinations and also bins for numerical fields that cannot be exhaustively covered (e.g., the frequency of the internally-generated clock in the OSync block). Moreover, the sampling of the configuration for coverage can be done by the testbench and restricted to when a new configuration is actually generated and applied to the hardware. This solution entails a lower overhead with respect to sampling at every clock cycle.

The following listing shows an exemplary coverage item on the OSync configuration and its instantiation and sampling in the test class.

```
covergroup osync_config_cg with function sample (osync_config osync_cfg);
  cp_clk_sel   : coverpoint osync_cfg.cfg_clk_sel;
  cp_clk_edge  : coverpoint osync_cfg.cfg_clk_edge;
  cp_intclk    : coverpoint osync_cfg.intclk_frequency {
    bins lo = {[ 1: 5]};  //  1 to  5 MHz
    bins me = {[ 6:10]};  //  6 to 10 MHz
    bins hi = {[11:15]};} // 11 to 15 MHz
  cross_cfg_int : cross cp_clk_sel, cp_clk_edge, cp_intclk;
endgroup
```

```
class test_example extends uvm_test;
  // ...
  osync_config osync_cfg;
  osync_config_cg osync_cfg_cg;

  virtual function void build_phase(uvm_phase phase);
    // ...
    osync_cfg = osync_config::type_id::create("osync_cfg");
    osync_cfg_cg = new();
  endfunction

  virtual task run_phase(uvm_phase phase);
    // ...
    repeat (`N_TIMES) begin
      // ...
      osync_cfg.randomize();
      osync_cfg_cg.sample(osync_cfg);
      // Run test
    end
    // ...
  endtask
endclass
```

Besides covering hardware configurations, it can be beneficial to make sure that certain critical combinations of signals have been tested and that no unlikely, yet potentially dangerous, corner case has been missed. For example, let us consider an arbiter for accessing a shared resource: it might be unlikely that all access requests come at the same time, but this is exactly the scenario which is most likely to trigger a bug and hence it is important to make sure it is tested. Such conditions can be identified and communicated to the test developer by the designer, who is aware of the hardware implementation details. Since their coverage typically requires access to the internal HDL signals, the corresponding coverage items are instantiated in the testbench module itself rather than in the UVM environment.

4. Conclusions

a. Results

Developing the first working prototype of the testbench required roughly two weeks of work for a developer with no previous UVM experience, other than the comprehensive SystemVerilog Verification UVM 1.1 and SystemVerilog Assertions workshops provided by Synopsys. The APB and TS VIPs were already available and fully reused for this work. The complete integration of all the more advanced features (e.g., dynamic reconfiguration, constant input-output delay verification and throttling of the input bandwidth depending on the output) continued for some more weeks in parallel with the development of more and more advanced tests.

Several bugs were found in the design, including one related to dynamic reconfiguration. Moreover, a careful optimization of the SM was performed based on the analysis of the test results, to ensure the jitter of the input-output delay respected the tight requirements. In both cases, the randomization of both the input data and the hardware configuration played a key role in highlighting critical corner cases.

The overall test plan coverage score reached a satisfactory 97.2%, by combining standard code coverage metrics (e.g., line and toggle coverage) and coverage items specifically defined on the configuration of the hardware blocks (as described in Section 3.0).

b. Lessons Learned and Recommendations

Throughout the development of the testing environment described in this paper, UVM has proven to be a very powerful approach. Even though the learning curve can be steep, once the infrastructure is in place extending it and developing new tests is relatively simple and requires a lower effort than more traditional approaches. To this end, it is of key importance to carefully and thoroughly plan the development and the structure of the testbench from the very beginning. In particular, positioning a given functionality in the right class/method can maximize its potential to be reused without having to replicate it or move it to a different place later on, which would result in a high maintenance effort. For instance, static/dynamic reconfiguration is not typically among the first features to be tested but supporting it affects the complete environment. Therefore, planning and including it right from the start, even if just by introducing placeholders in the code, is important to avoid a major rework late in the development.

Another advantage of UVM is that a whole set of components can be reused for future projects that exploit the same (or slightly modified) hardware IPs; examples are the configuration classes, the RAL models and sequences, the block-level transformers and scoreboards. Integrating these objects in a new environment is significantly easier and cleaner than porting sections of code

across monolithic testbenches which do not follow an object-oriented approach. It is therefore highly advisable to exploit the available object-oriented features as much as possible. For instance, the test class inheritance scheme described in Section 3.0 enables to write new tests in a very quick and compact way.
A careful planning and structuring may increase the initial effort to obtain a working environment but ultimately pays off when trying to fulfill the verification and coverage goals. Very helpful to this end is random-constrained verification, which goes hand in hand with the UVM methodology. For this reason, it is recommended to define as many random input and configuration parameters as possible; specific tests can then constrain them as needed.
Besides the steep learning curve, a pitfall of UVM that was observed during the development concerns maintaining code that is parameterized with respect to design-time hardware parameters. Achieving such a goal sometimes requires to drop the usage of standard UVM macros and implement custom alternative solutions. Examples can be found in the virtual sequences and scoreboards presented in this paper, which need to support the parameterized number of input ports of the DUV. Even though the parameterized alternatives are not overly complex, a better out-of-the-box support would be helpful in such scenarios.

c. Future Work

Further extensions to the work presented in this paper are possible. In particular, the integration of the testbench with a Zebu emulator is of high interest, since it offers great simulation speed-up potential against an expected low effort for adapting the UVM testbench.
Moreover, the testbench components will be available for reuse in future projects. The methodology described in this paper will be taken as reference since it proved to be effective in achieving the verification goals and enabled the development of a well-structured and extensible test environment.

Acknoledgements

We would like to thank Mike Bartley (TVS), David Long (Doulos) and Roger Ninane (Synopsys) for reviewing the draft and providing valuable feedback.

References

[1] Accellera, *Unviersal Verification Methodology*. Online: http://www.accellera.org/community/uvm

[2] D. Smith, *Easier RAL: All You Need to Know to Use the UVM Register Abstraction Layer*. SNUG Silicon Valley, 2012.

[3] ISO/IEC, *International Standard ISO/IEC 13818-1*. 2000.

[4] ARM, *AMBA Specifications*. Online: http://www.arm.com/products/system-ip/amba-specifications.php

[5] M. Glasser, *Configuration in UVM: The Missing Manual*. DVCon India, 2014.