Transaction Level Assertions in UVM

Steve Holloway

Dialog Semiconductor
Swindon, UK

www.diasemi.com

ABSTRACT

The UVM embodies a central concept of transaction-level modelling (TLM) to abstract the way communication in a design is modelled. A single transaction can be used to represent a complex series of signal transitions such as a bus operation or transfer of data over an external interface. Performing assertion-based checks on the order and content of these transactions offers the potential to succinctly verify the design at a higher level of abstraction. The paper describes an approach for creating assertions at the UVM transaction-level by making a bridge between dynamic and static verification code. A new UVM monitor architecture is presented which supports both assertion-based and class-based checking. Example TLM assertions are described which can verify complex sequential behaviour, express performance constraints, or check system-level interaction in a concise manner.

Table of Contents

Table of Figures

1. Introduction

The UVM methodology [1] emphasises the use of transaction-based scoreboarding, but does little to address the usage of SystemVerilog Assertions (SVA) [2] in the verification environment. In the traditional UVM approach, these two categories of checks are kept quite separate: SVA code is declarative in nature and exists in the module-based testbench (usually located in SV interfaces); UVM code is dynamic in nature and exists in a separate class-based structure which is created when *run_test()* is called. For many designs, the optimum verification approach requires the partitioning of checks into these two domains. The best choice for a given check is often not known until we are in the advanced implementation phase of the verification environment. For many control-oriented designs where signal ordering and timing are features of the verification plan, the use of SVA is often the most efficient way to code checks. For data-oriented features of the verification plan, the use of transaction-level scoreboards is usually the most appropriate approach. Sometimes a section of the verification plan will straddle control-oriented and data-oriented features. One example of this is register-mapped IO functionality. Checking these features will actually straddle both the UVM transaction-level and SVA checking domains. The approach described in this paper allows the combination of module-based SVA checks with class-based UVM checks. Transaction information is captured in SVA sequences in the DUT interface and converted into UVM transactions. This allows maximum flexibility in the choice of implementing a given check. SVA checks can be created which consider the timing, ordering and content of transaction-level information. Class-based checks can be created which consider ordering and transformation of transactions. Checks can also be created which relate transaction timing and content to "low-level" signal-level activity in the DUT.

2. Transaction Level Modelling

In Transaction-Level Modelling (TLM), the details of communication among components are kept separate from the implementation of the components or the communication architecture itself. TLM models concern themselves with the details of transactions, i.e. the data transferred in the system rather than the low-level details of the communications protocol. There are many productivity benefits to be gained from performing verification tasks at the TLM level of abstraction. Using TLM interfaces isolates each verification component from changes in other components. This allows re-use of components which have the same interface, as they effectively become "plug and play" compatible. The task of synchronising data flow in the verification environment becomes much simpler as the synchronisation details are abstracted into categories of blocking or nonblocking transactions. Simulation efficiency is increased as a consequence of raising the level of abstraction.

TLM is implemented in the UVM class library. The *uvm_transaction* class is used to carry information about a transaction which takes place between components in the system. It can be extended to contain variables, methods and constraints which facilitate modelling, generation and manipulation of transaction data. TLM interfaces in UVM define a standard API for passing transaction classes in a verification environment. A TLM port defines the methods used in a given TLM connection, and a TLM export provides their implementation. Figure 1 below illustrates two simple TLM connection styles.

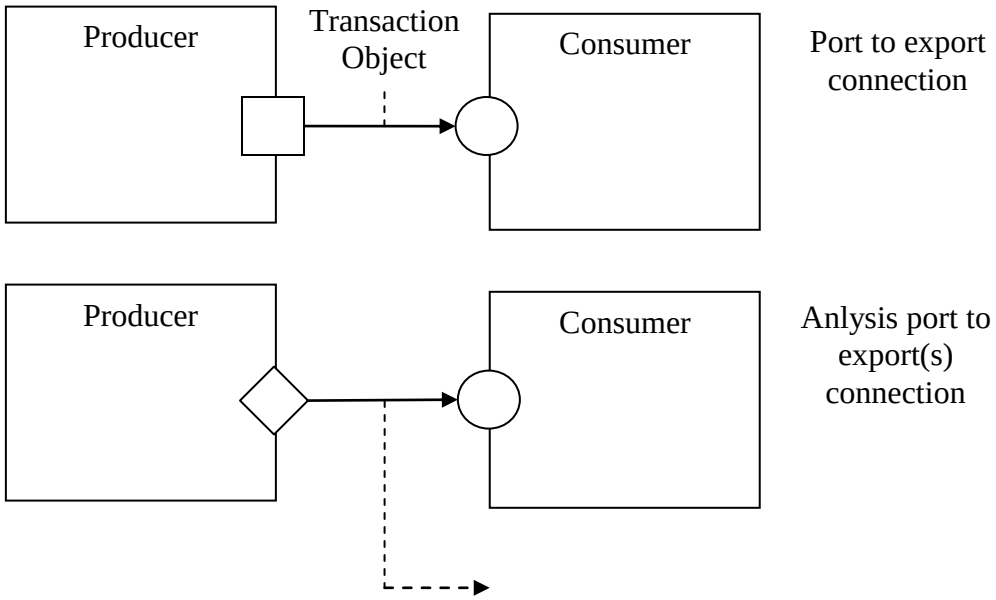

Figure 1: TLM Communication Styles

In the port to export example at the top of the Figure 1, the square box on the producer is the TLM port and the circle on the consumer is the corresponding export. Once the TLM connection has been made, a TLM API method call in the context of the producer (for example, *port.put(transaction)*) will result in the execution of the implementation in the context of the consumer. Since the mechanism of TLM communication is dealt with by method calls, connecting components is simply achieved by providing the port with a reference to the export. The second example in Figure 1 shows a one-to-many communication style. This is achieved by using an analysis port on the producer (shown by the diamond). This specialised port allows any number of analysis exports to be connected to it. It has a simple API with a single function *write()*. When this is called in the context of the producer, the implementation of *write()* is executed, in turn, in the context of each analysis export connected to it. This communication style is particularly useful for broadcasting information inside a verification component. It is the communication style used between UVM monitors and scoreboard components.

3. SystemVerilog Assertions

SystemVerilog Assertions (SVA) [3] is a declarative language designed for writing checks that form part of the verification environment. It has some key advantages over checks that are created with procedural code (e.g. in UVM class-based scoreboards):

- It is well-suited to describe temporal checks and can express sophisticated sequences
- It can be very concise compared to the equivalent procedural code
- It can perform parallel checks (multiple threads) over the same time period
- It can automatically collect functional coverage data
- Its declarative syntax precludes the verification engineer from checking the DUT against another version of the DUT coded in the testbench

There are two flavours of SVA, namely immediate assertions and concurrent assertions. Immediate assertions are evaluated immediately and have to be placed in a procedural block definition. Although they are useful constructs, they cannot express temporal sequences and will not be explored further in this paper. Concurrent assertions express temporal checks and can be built up from simpler expressions in a hierarchical manner. These can be Boolean expressions, sequential regular expressions (SERE) or property declarations. Property declarations can use an *implication* operator which allows monitoring of temporal behaviour based on satisfying a condition. The property shown below is triggered when the expression on the LHS of the implication operator "->" is satisfied and holds true if the expression on the RHS is satisfied on the same sampling event.

```
property if_x_then_y;
  @(posedge clk)
  x -> y;
endproperty: if_x_then_y
```

Sequences in SVA describe a behaviour that spans time. The simplest sequence is given by a Boolean expression (which spans 1 tick). Sequences can express complex behaviour using delay, range and repetition operators. For more detailed information see [3]. The simple sequence below is satisfied if the expressions *a*, *b*, *c* are true on successive clock cycles.

```
sequence a_then_b_then_c;
  @(posedge clk)
  a ##1 b ##1 c ##1;
endsequence: a_then_b_then_c
```

Local variables allow the modelling of data flow in a temporal property. These are dynamically created since checks can exist in parallel and overlap with other checks of the same property. The use of variables is particularly suitable for checking data in a pipelined design where the data ordering and latency is variable. The example property below shows the use of the local variable *data*. When the expression *en* is true, the variable *data* is assigned the value on the signal *input*. For the sequence expression to be satisfied, the signal *output* must be equal to the same value *data* two cycles later.

```
property pipeline;
  logic [WIDTH-1:0] data;
  @(posedge clk)
  (en, data = input) ##2
  (output == data);
endproperty: pipeline
```

Once a sequence expression has been matched, it is possible to call a subroutine. Subroutines can be either tasks, system tasks or functions. They are attached to a sequence expression in the same way that variable assignments are declared in a comma-separated list. Variable assignments can be placed in the list together with subroutine calls. The example sequence below calls the system task *$display()* at the end of a matched sequence.

```
sequence a_then_b_then_c_task;
  @(posedge clk)
  a ##1 b ##1 (c, $display("a then b then c detected"));
endsequence: a_then_b_then_c_task
```

4. Transaction Level Assertions

Assertions which operate at the TLM level offer a high level of abstraction compared to those which operate at the clock cycle level. The use of assertions at the transaction level is described in [4]. Complex temporal relationships between transactions or sequences of transactions can be efficiently checked with the capabilities offered by SVA. Instead of using a clock as the property sampling event, we can use an event based around the completion of a given transaction. We can also exploit the multiclock support in SVA to perform checks which operate at both the clock cycle level and the transaction level. This is something not easily done with a UVM scoreboard unless we introduce timing controls into the class-based code. Using sequence match items and local variables, we can declare SVA properties which perform monitoring and scoreboarding functions [5]. Due to the declarative nature of SVA, we cannot place such assertions into class-based UVM code. They must be declared in module-based code. We also cannot refer to transaction classes in our expressions or sequences as these are dynamic structures. We can however use *struct* members as expressions in properties, and this is the proposed way that we will capture transaction attributes in our module-based TLM code. An SVA-based split UVM Monitor is described in the next section which allows us to perform Transaction-Level SVA and preserve UVM compatibility.

5. SVA-based Split UVM Monitor

The proposed SVA-based monitor is split into two halves. One exists in the module-based (testbench) code, whilst the other is in the class-based code. The diagram in Figure 2 below illustrates the architecture. The module-based part is an SVA-based signal monitor placed in a SystemVerilog interface. It is concerned with identifying legal transactions which occur on the interface and translating them into a *struct* data type. The class-based part has the task of converting information in the transaction *struct* into a corresponding *uvm_transaction* object and broadcasting it to the UVM environment through an analysis port. The use of a transaction *struct* which is later converted into a *uvm_transaction* object may seem like a redundant step. However, this is necessary if we wish to declare some additional SVA properties inside the module-based code which are linked to the transaction *struct* members.

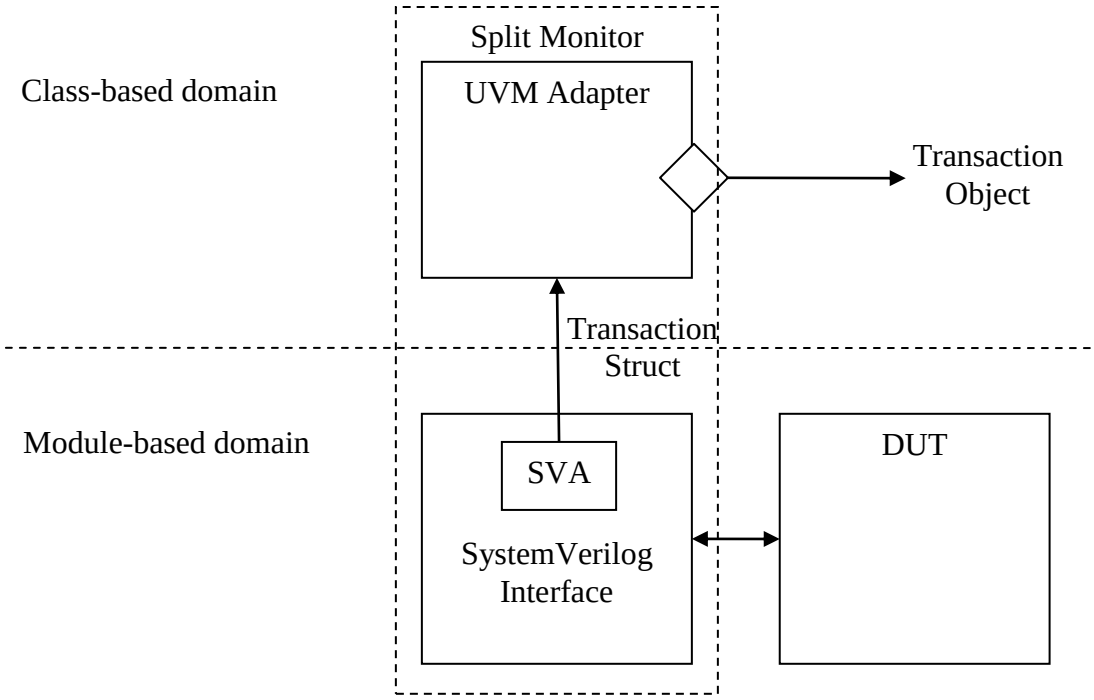

Figure 2: Split Monitor Architecture

A handle to the adapter is placed in the SV interface. This can be automatically connected to the appropriate UVM component instance as shown in a subsequent code snippet. The code snippet below shows the declaration of the interface which will contain the SVA-based monitor. The example code snippets which follow are appropriate for the APB bus.

```
interface apb_if (input pclock, input preset);

  parameter         ADDR_WIDTH = 32;
  parameter         DATA_WIDTH = 32;

  logic [ADDR_WIDTH-1:0] paddr;
  logic                  prwd;
  logic [DATA_WIDTH-1:0] pwdata;
  logic                  penable;
  logic [15:0]           psel;
  logic [DATA_WIDTH-1:0] prdata;
  logic                  pslverr;
  logic                  pready;

  // Interface transaction struct
  apb_if_tr tr;

  // Handle to the UVM adapter
  apb_adapter adapter;
 ...

endinterface : apb_if
```

In addition to the signals required to monitor the APB bus, the interface contains a declaration of a transaction struct *tr* and a handle to the UVM adapter. The timing diagrams for an APB read and write transaction are shown in Figure 3 and Figure 4 below, respectively:

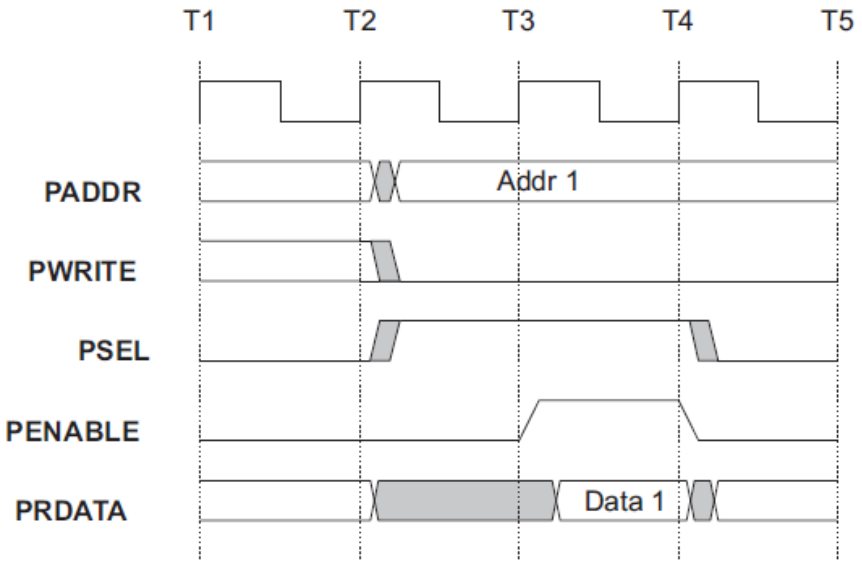

Figure 3: APB Read Timing

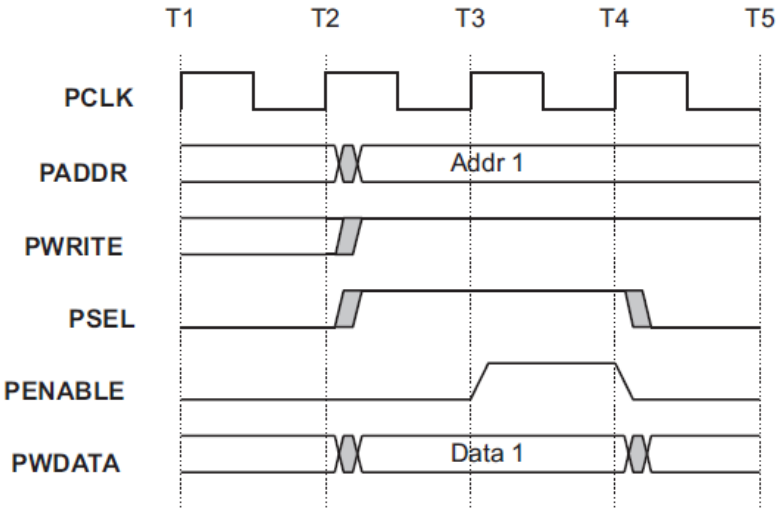

Figure 4: APB Write Timing

The example code below illustrates the SVA sequence needed to identify a valid APB transaction:

```
  sequence apb_trans_s;
    logic [ADDR_WIDTH-1:0] addr;
    logic [DATA_WIDTH-1:0] data;
    logic dir;
    time  start;
    (psel && !penable, addr = paddr, dir = prwd, start = $time)
##1
    (psel && penable && (prwd == dir) && (paddr == addr), data =
dir ? pwdata : 0) ##1
    (pready && (prwd == dir) && (paddr == addr), data = dir ?
data : prdata, write(addr, data, dir, start));
  endsequence: apb_trans_s
```

The starting time of the transaction is stored in a local variable *start*. This is used to pass the correct time-stamp to the UVM transaction later on, such that the transaction recording API can be used. Local variables are used to store the address, data and direction of the APB transaction. Note that although APB transactions are "atomic", the proposed approach is also suitable for pipelined protocols, as each sequence thread in the simulator will have its own copy of local variables used to store attributes of ongoing transactions simultaneously.

On completion of the sequence, the *write()* function is called in the context of the interface. This function, in turn, passes the transaction information to the UVM adapter.

If we employ the *cover sequence* directive for the transaction sequence *apb_trans_s* then we can enable the transaction matching SVA and also capture sequence coverage. The cover directive ensures that the sequence expression is evaluated as *strong*. Note that assertion coverage may need to be enabled in the simulator (using a vendor-specific switch) as it may not be switched on by default.

```
apb_trans_c : cover sequence(apb_trans_s);
```

In order to create TLM assertions in the testbench, we must remember that expressions in SVA sequences cannot be non-static class properties. This means we cannot write sequences or properties based on class-based transaction attributes. We can however refer to struct elements in SVA expressions. Therefore, we need to store transaction attributes in a struct located in the interface. The example below is suitable for the APB protocol:

```
typedef struct {
  logic [31:0] addr;
  logic [31:0] data;
  logic        dir;
  time         start;
  time         finish;
} apb_if_tr;
```

The *write()* function is declared in the interface to update the transaction struct elements with the transaction attributes which were collected with local variables in the SVA sequence. After this is done, it calls the *write()* method inside the UVM adapter, passing the struct as an argument.

```
function void write(logic [PADDR_WIDTH-1:0]  addr,
                    logic [PWDATA_WIDTH-1:0] data,
                    logic dir,
                    time  start);

    // Assign struct members
    tr.addr   = addr;
    tr.data   = data;
    tr.dir    = dir;
    tr.start  = start;
    tr.finish = $time;

    // Call UVM adapter write() method
    adapter.write(tr);

endfunction: write
```

In the UVM environment, we will declare an adapter as the second half of our SVA-based monitor component. This has a virtual interface handle and a TLM analysis port declared, just like the "traditional" UVM monitor. The *build_phase()* and *write()* functions are defined outside the class body declaration to clarify the following code snippets.

```
class apb_adapter extends uvm_monitor;

  uvm_analysis_port #(apb_transfer) ap;
  virtual apb_if vif;

  `uvm_component_utils(apb_adapter)

  function new (string name, uvm_component parent);
    super.new(name, parent);
    ap = new("ap", this);
  endfunction : new

  // UVM build_phase
  extern function void build_phase(uvm_phase phase);

  // Function to send transaction to ap
  extern function void write(apb_if_tr tr);

endclass: apb_adapter;
```

The *write()* method is the place where the interface transaction struct is converted into a UVM transaction object and broadcast on the analysis port, *ap*. The uvm_component methods *begin_tr()* and *end_tr()* are used to annotate the start and end times of the transaction. It can then be analysed with vendor-specific transaction recording and visualisation tools. The analysis port can be subsequently connected to UVM scoreboards or checking components in the normal way.

```
function void apb_adapter::write(apb_if_tr tr);
  // Create UVM transaction
  apb_transfer apb_tr = new;

  // Fill uvm transaction with info from struct
  apb_tr.addr      = tr.addr;
  apb_tr.data      = tr.data;
  apb_tr.direction = tr.dir ? APB_WRITE : APB_READ;

  // Set start and end times for transaction recording
  void'(begin_tr(apb_tr, .begin_time(tr.start)));
  void'(end_tr(apb_tr,   .end_time(tr.finish)));

  // Send transaction to analysis port
  ap.write(apb_tr);

endfunction: write
```

6. Automatic Connection of the UVM Adapter

Automatic connection of the handle to the adapter in the interface can be done by simply passing a handle to *this* in the build phase of the adapter to the virtual interface. The virtual interface will have been configured by a corresponding *uvm_config_db::set()* call higher up in the UVM environment or the testbench. This recipe essentially creates a "back-pointer" to the UVM adapter in the interface.

```
function void apb_adapter::build_phase(uvm_phase phase);
  super.build_phase(phase);
  // get the config setting for the virtual interface
  if (!uvm_config_db#(virtual apb_if)::get(this, "", "vif",
vif))
    `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
get_full_name(), ".vif"})
  // assign the back-pointer to the adapter in the interface
  vif.monitor = this;
endfunction : build_phase
```

Once the split monitor has been connected to the DUT in the testbench, TLM information is available in the module-based transaction struct *tr* and also in the UVM-based transaction object *apb_transfer*. The visualisation of both of these is shown in the waveform below for a stream of random APB bus transfers.

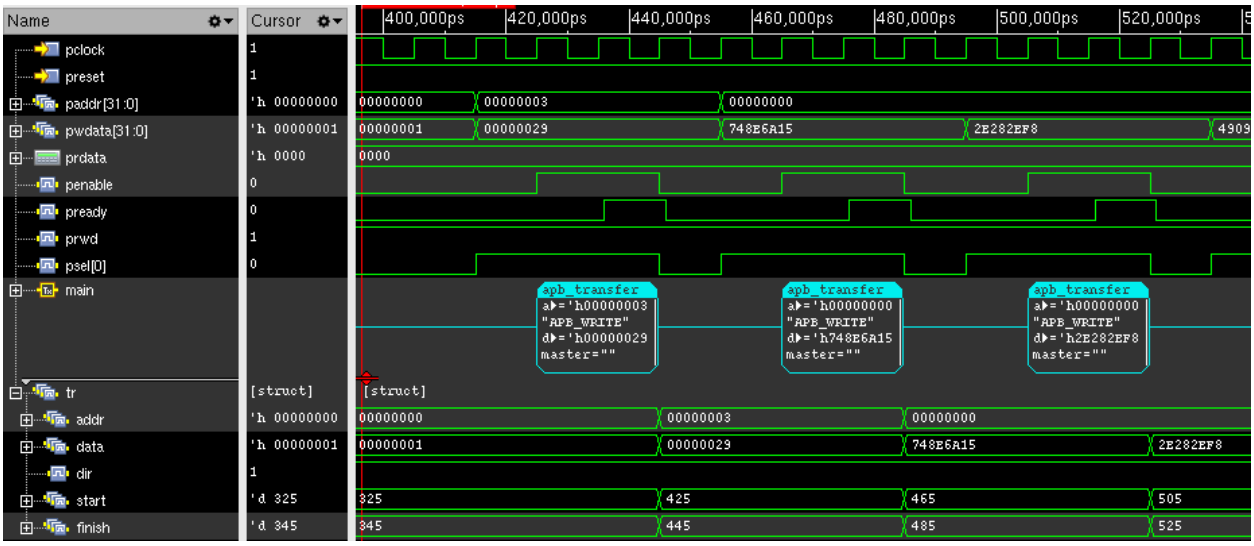

Figure 5: APB Bus Transactions

7. Example Transaction Level Assertions

Once we have established the proposed split monitor architecture, we are able to construct a transaction-based scoreboard in the UVM portion of the testbench in the normal way. However we now have the ability to create SVA checks at the transaction level in the scope of the module-based testbench. Elements of the transaction struct *tr* can be used within SVA expressions in sequences and properties. For multiple instances of the split monitor we can refer to each transaction struct *tr* using hdl paths. For example, if we have two APB interfaces on the DUT (*apbi0*, *apbi1*) then we can access each transaction struct using (*apbi0.tr*, *apbi1.tr*). Synchronisation with transactions in SVA sequences can be done using a value-change event on *tr.start*, i.e. when the start time is updated, after a valid transaction sequence is matched.

Example 1: APB access sequence

The following sequence can be used to match the occurrence of an access to a given APB address. The argument *addr* contains the address to match, and *dir* gives the direction. We can use the sequence to build more useful properties in the subsequent sections.

```
sequence apb_acc_s(addr, dir);
  @(apbi0.tr.start)
  (apbi0.tr.addr === addr)&&(apbi0.tr.dir === dir);
endsequence: apb_acc_s
```

Example 2: Register-mapped GPIO checking.

We can write a property to check a write to a register-mapped IO address. In this example, a write to the address `GPIO_ADDR results in the DUT signal *gpio* taking on the value of the APB *data* on the next positive edge of the clock *clk*. Note the use of a multi-clock property. This is required since the TLM transaction event occurs when the struct element *start* is updated, but the GPIO signal is synchronised to *clk*.

```
property chk_gpio_reg;
  logic [PWDATA_WIDTH-1:0] data;
  @(apbi0.tr.start)
  (apb_acc_s(`GPIO_ADDR, `APB_WR), data = apbi0.tr.data)##1
  |->
  @(posedge clk)
  (dut.gpio === data);
endproperty: chk_gpio_reg
```

Example 3: Read-write register checking

It is possible to create a property to check a register write/read sequence. The property antecedent is a sequence in which the register at location *addr* is written and read back an indeterminate number of transactions later. The sequence variable *data* holds the APB write value and this is checked against the readback data after a match of the antecedent sequence.

```
property chk_wr_rd(addr);
  logic [PWDATA_WIDTH-1:0] data;
  @(apbi0.tr.start)
  (apb_acc_s(addr, `APB_WR), data = apbi0.tr.data) ##[1:$]
   apb_acc_s(addr, `APB_RD)
  |->
  (apbi0.tr.data === data);
endproperty: chk_wr_rd

chk_wr_rd_reg0_a : assert property(chk_wr_rd(`REG_0_ADDR));
```

Example 4: Matching a sequence of bus transactions

It is relatively easy to create sequences to match a series of bus transaction patterns. Firstly we create a more general bus access sequence *apb_acc_s* with formal arguments for *addr*, *data*, *dir*.

```
sequence apb_acc_s(addr, data, dir);
  @(apbi0.tr.start)
  (apbi0.tr.addr === addr)&&
  (apbi0.tr.addr === data)&&
  (apbi0.tr.dir  === dir);
endsequence: apb_acc_s
```

We can use this in more complex matching sequences such as *apb_seq_reg0_reg1_reg_2_s* shown below. This will match when the registers at locations REG_0, REG_1, REG_2 have the values (0,1,2) written to them on successive transactions.

```
sequence apb_seq_reg0_reg1_reg2_s;
  apb_acc_s(`REG_0, `h00, `APB_WR) ##1
  apb_acc_s(`REG_1, `h01, `APB_WR) ##1
  apb_acc_s(`REG_2, `h02, `APB_WR);
endsequence: apb_seq_reg0_reg1_reg2_s
```

The repetition operators can also be used to match sequences of transactions such as *apb_seq_reg0_a5_x3_s* below. It matches after 3 non-consecutive writes of the value 0xa5 to the register at address `REG_0.

```
sequence apb_seq_reg0_a5_x3_s;
  apb_acc_s(`REG_0, `ha5, `APB_WR)[=3];
endsequence: apb_seq_reg0_a5_x3_s
```

Example 5: Scoreboarding between two interfaces

It is possible to create a property to perform scoreboarding between two bus interfaces. The example property below is triggered when a bus operation completes on the interface *apbi0*. The sequence variables *addr*, *data*, *dir* hold the transaction attributes. In the consequent part of the property, the sequence is synchronised to the next non-overlapping transaction on the interface *apbi1*. The sequence variables copied from *apbi0.tr* are then checked against the struct elements in *apbi1.tr* and the property will fail if there is a mismatch.

```
property chk_scbd;
  logic [PADDR_WIDTH-1:0]  addr;
  logic [PWDATA_WIDTH-1:0] data;
  logic dir;
  @(apbi0.tr.start)
  (1, addr = apbi0.tr.addr, data = apbi0.tr.data, dir =
apbi0.tr.dir) ##1
  |->
  @(apbi1.tr.start)
  (apbi1.tr.addr === addr) &&
  (apbi1.tr.data === data) &&
  (apbi1.tr.dir  === dir);
endproperty: chk_scbd
```

8. Conclusions

The concept of performing SVA assertions at the transaction level was introduced. A novel split architecture for the UVM monitor was presented which allows the creation of TLM assertions whilst retaining compatibility with standard UVM scoreboards. Several example assertions were presented which demonstrate the utility of the proposed approach. The combination of SVA and UVM based checks in a testbench provides a complementary approach to verification. The proposed architecture allows the flexibility to choose the domain in which a given check can be coded most efficiently.

9. References

[1] Universal Verification Methodology (UVM), http://www.accellera.org

[2] SystemVerilog, IEEE Std 1800-2012, http://www.ieee.org

[3] Srikanth Vijayaraghavan and Meyappan Ramanathan. A Practical Guide for SystemVerilog Assertions. Springer Science, New York, 2005

[4] N. Sudish, Raghavendra BR, H. Yagain, "An efficient method for using transaction level assertions in a class based verification environment", IEEE International Symposium on Electronic System Design, 2011, pp. 72-76

[5] Ben Cohen, "Assertions instead of FSMs/logic for Scoreboarding and Verification", *Verification Horizons*, Volume 9, Issue 1