SNUG-2018 Austin
Voted Best Presentation
2nd Place

Sunburst Design

World Class SystemVerilog & UVM Training

UVM Analysis Port Functionality and Using Transaction Copy Commands

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
Provo, UT, USA
www.sunburst-design.com

Heath Chambers
HMC Design Verification
hmcdvi@msn.com
Albuquerque, NM, USA

ABSTRACT

There is significant confusion surrounding UVM analysis ports and similar confusion about the UVM transaction copy command. Many verification engineers who consider themselves to be UVM experts can easily spend hours debugging analysis port issues if they are unaware of important considerations related to analysis port paths.

This paper explains UVM analysis port usage and compares the functionality to subscriber satellite TV. The paper shows simplified, non-UVM, analysis port implementations to clarify how the corresponding UVM port connections work. The paper describes how the analysis port write() method efficiently calls each subscriber's write() method. Part of the explanation describes when an analysis implementation port requires the use of a transaction copy() command. The paper describes problems that arise when multiple analysis port implementations are required in the same component and how to address the problems.

The paper also describes an example of how improper handling of transactions can hide design and testbench bugs. The example shows how a bug was hidden in a scoreboard that went unnoticed for months and took hours to detect and fix once we identified that there was a problem.

Table of Contents

1. Introduction - Satellite TV Example 5
2. Observer pattern & analysis path basics 5
2.1 Observer pattern definition 5
2.2 Simple SystemVerilog analysis path examples 5
2.2.1 Scenario common files 6
2.2.2 Scenario1 - monitor1 with separate analysis_if declarations – no connect() methods 7
2.2.3 Scenario 2 - monitor2 with separate analysis_if and connect() method declarations ..9
2.2.4 Scenario 3 - monitor3 with analys_if queue and common connect() method 11
Scenario 4 – Buggy example where subscriber2 modifies the transaction 13
UVM analysis_port.write() versus analysis_imp write() method 14
3. UVM Port Fundamentals 15
3.1 UVM port connection chains 15
3.1.1 UVM Ports 15
3.1.2 UVM Exports 15
3.1.3 UVM Imps 15
3.1.4 Unfortunate port naming convention 15
3.1.5 Port-Export-Imp chains 16
3.1.6 UVM Port & Export interchangability? 17
3.1.7 UVM Port & Export usage guidelines 18
3.2 uvm_port_base connect() method 19
3.3 Scope of port discussion in this paper 19
4. UVM analysis ports, exports and imps 20
4.1 uvm_analysis_port - broadcast port 20
4.1.1 Why is the transaction copy() method so important? 20
4.2 uvm_analysis_export – transfer port 20
4.3 uvm_analysis_imp – termination port 20
4.4 port, export, imp confusion 21
4.4.1 Driver – Car / Port - Export 21
4.4.2 Export, imp confusion 21
5. Declaring and constructing ports & TLM FIFOs 22
5.1 How are analysis ports and TLM FIFOs declared? 22
5.2 Where are TLM ports and TLM FIFOs constructed? 23
5.3 uvm_analysis_imp port usage options 24

6. uvm_subscriber 25
6.1 pure virtual write function 25
6.2 multiple uvm_analysis_imp ports on the same component 25
7. SystemVerilog mailbox 25
7.1 Mailbox –vs- queue 25
8. TLM FIFOs 26
8.1 uvm_tlm_fifo_base 26
8.2 uvm_tlm_fifo 28
8.2.1 uvm_tlm_fifo new()-constructor & size 28
8.2.2 uvm_tlm_fifo put() & try_put() methods 29
8.2.3 uvm_tlm_fifo get() & try_get() methods 30
8.2.4 uvm_tlm_fifo disadvantage 30
8.3 uvm_tlm_analysis_fifo 31
8.4 Quick-summary of uvm_tlm_analysis_fifo –vs- uvm_tlm_fifo 31
9. `uvm_analysis_imp_decl(SFX) macro 32
9.1 How many ports are allowed on a scoreboard? 33
10. Example with typical analysis/copy() problems 34
11. Summary & Conclusions 34
12. Acknowledgements 35
13. References 35

Table of Figures

Figure 1 - Monitor & subscribers – Simulation output 9
Figure 2 - Monitor & subscribers – Buggy simulation output 13
Figure 3 - Common analysis port connections – recommended connections 15
Figure 4 - Analysis paths - first set of three paths 16
Figure 5 - Analysis paths - second set of paths - just one path 17
Figure 6 - NOT Recommended – analysis ports & no analysis exports (but it works!) 17
Figure 7 - NOT Recommended – analysis exports & 1 analysis port per analysis source (but it works!) 18
Figure 8 - UVM analysis ports – recommended usage block diagram 18
Figure 9 - UVM analysis exports & imps – recommended usage block diagram 18
Figure 10 - uvm_tlm_fifo_base ports 26
Figure 11 - uvm_tlm_fifo put() and get() method behavior 30
Figure 12 - uvm_tlm_analysis_fifo – most common usage 31
Figure 13 - uvm_tlm_fifo –vs- uvm_tlm_analysis_fifo usage 32

Table of Tables

Table 1 - uvm_tlm_fifo_base port names and port name aliases 27
Table 2 - uvm_tlm_fifo_base methods and usage notes 28

1. Introduction - Satellite TV Example

Those familiar with satellite TV know that programs are broadcast as scheduled and the viewer either needs to watch the program live as it is being broadcast, or they need to setup a Digital Video Recorder (DVR) to record the program for later viewing.

The satellite will broadcast the program as scheduled whether there are 1,000's of viewers or no viewers at all. The broadcast has been scheduled and it will happen on schedule.

If a viewer neglects to setup a DVR to record a desired program and the viewer turns the TV on 15 minutes after the program has started, the viewer cannot request that the satellite start over and re-broadcast the desired program from the start for these two reasons: (1) the viewer has no way to communicate to the satellite the desire to re-start a program, and (2) other viewers would object to programs being restarted while they were watching a program live.

It should also be noted that the viewer of a live broadcast cannot modify the Satellite version of the broadcast program in real time. If the viewer has the right equipment and software, the viewer might splice the recorded program or censor unwanted language and content, but those edits have to be done on the local copy of the program and not on the live satellite broadcast of the program.

The **`uvm_analysis_port`** is a broadcast port that is very analogous to this satellite TV example.

Since UVM analysis paths do not broadcast transactions over the airwaves, it is instructive to understand how classes are assembled to allow new subscribers to be added to an analysis broadcast source with little modification to the existing environment. Despite the fact that viewers of Satellite programs do not modify broadcast programs, engineers are frequently guilty of modifying the original transaction and such modifications can cause subtle problems that are difficult to debug. This paper will show many analysis features that engineers should consider when using analysis ports.

2. Observer pattern & analysis path basics

The UVM analysis path is an example of a software design pattern known as the observer pattern.

2.1 Observer pattern definition

A concise definition of the observer pattern is found in Wikipedia.

"The **observer pattern** is a software design pattern in which an object, called the **subject**, maintains a list of its dependents, called **observers**, and notifies them automatically of any state changes, usually by calling one of their methods."[3]

2.2 Simple SystemVerilog analysis path examples

We have found that most examples of analysis paths are overly complex and difficult to understand. The multi-part scenarios shown in this section are an over-simplification of the analysis path implementation in SystemVerilog and are not fully UVM compliant, but their simplicity make them easy to comprehend and being simple allows an engineer to have a conceptual understanding of how the **`uvm_analysis_port`** path works.

Four scenarios will be presented to demonstrate how analysis paths (observer patterns) work.

In all four scenarios, the transaction, **`analysis_if`** and subscribers remain unchanged. Also unchanged is the fact that the top module declares all of the subscriber handles and has to **`new()`**-construct each subscriber. The top module in each scenario will show impotant differences after the subscribers are constructed.

The scenario differences are visible in the monitors and latter part of the top modules.

In the first scenario, **monitor1** has to declare all of the subscribers and call their respective **write()** methods by explicitly referencing the handle names. The **top1** module has to copy all of the constructed subscriber handles to the **monitor1** subscriber handles, which required the **top1** module to know the internal handle names of each subscriber.

In the second scenario, **monitor2** defines **connect**[#]**()** methods for each subscriber so that the **top2** module only has to know there are **connect()** methods without being required to know the subscriber handle names.

In the first two scenarios, each subscriber required a separate declaration, separate calls to **write()** methods and the top module had to either copy each subscriber's handle name to the corresponding handle names in the monitor, or had to call a **connect()** method that was unique to each subscriber. Adding more subscribers requires lots of extra code.

In the third scenario, **monitor3** has a declared queue of **analysis_if** handles and a common **connect()** method that pushes a new handle onto the queue. The **run()** task simply uses a **foreach** loop to pull each handle off of the queue and call the **write()** method defined in each subscriber. From this point forward, whenever the **top3** module adds a new subscriber, no modifications will be necessary inside of the monitor. This is possible because each new subscriber that extends the **analysis_if**, must provide an implementation of the commonly named virtual **write()** method.

In the fourth scenario, **subscriber2** modifies the transaction values and we observe that since each subscriber has a handle to a common transaction, that **subscriber3** sees the modified transaction and not the original broadcast transaction. This demonstrates why subscribers should never modify the original transaction but should take a copy before doing any transaction modifications.

2.2.1 Scenario common files

The files in this section are common to all four of the subsequent scenarios.

The transaction (**trans1**) that is passed around these scenarios has two randomizable fields, **addr** and **data**, and both fields, through the **post_randomize()** method, are automatically printed each time the transaction is randomized. The **trans1** class is shown in Example 1.

```
class trans1;
  rand bit [7:0] addr;
  rand bit [7:0] data;

  function void post_randomize();
    $display("\nRandomized trans1 values addr=%2h  data=%2h", addr, data);
  endfunction
endclass
```

Example 1 - transaction class with built-in post_randomize() method to print randomized transaction values

A **virtual analysis_if** base class, shown in Example 2, is declared with a **pure virtual write()** method. Any class that extends the **analysis_if** class will be required to provide a **write()** method implementation. In these examples there will be three subscribers that are extensions of the **analysis_if** class. This is similar in concept to a **uvm_analysis_imp** inside of a **uvm_subscriber**. Since there is a **pure virtual write()** method defined in the **analysis_if**. Any class that extends the **analysis_if** (**uvm_subscriber**) is required to use the exact same prototype (function header) and provide the actual implementation. The implementations in this example will be simple display commands to show the transaction that was received.

```
virtual class analysis_if;
  pure virtual task write(trans1 t);
endclass
```

Example 2 - virtual analysis_if base class and pure virtual write() method definition

For these scenarios, three subscribers, shown in Example 3, Example 4 and Example 5, have been extended from the **analysis_if** virtual class. All three have a **write()** method implementation that displays which **subscriber**[#] issued the message and the current contents of the transaction. **subscriber2** in Example 4 includes **BUG** code to modify the transaction when the simulation is compiled with **+define+BUG**. The behavior of the bug is described later in this section.

```
class subscriber1 extends analysis_if;
  virtual task write(trans1 t);
    $display("subscriber1: ",
       "received    addr=%2h  data=%2h", t.addr, t.data);
  endtask
endclass
```

Example 3 - Subscriber #1 with write() method to do $display

```
class subscriber2 extends analysis_if;
  virtual task write(trans1 t);
    $display("subscriber2: ",
       "received    addr=%2h  data=%2h", t.addr, t.data);

    `ifdef BUG
    t.addr = 8'hFF;
    t.data = 8'h00;
    $display("subscriber2: ",
       "set         addr=%2h  data=%2h", t.addr, t.data);
    `endif

  endtask
endclass
```

Example 4 - Subscriber #2 with write() method to do $display – includes BUG testing code

```
class subscriber3 extends analysis_if;
  virtual task write(trans1 t);
    $display("subscriber3: ",
       "received    addr=%2h  data=%2h", t.addr, t.data);
  endtask
endclass
```

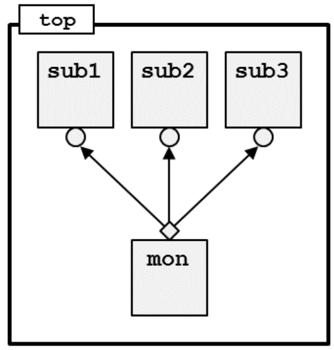
Example 5 - Subscriber #3 with write() method to do $display

2.2.2 Scenario1 - monitor1 with separate analysis_if declarations – no connect() methods

In the first scenario, **monitor1**, shown in Example 6, declares three **analysis_if** handles with handle names **ap1**, **ap2** and **ap3**. The **monitor1** class also has a **run()** task method that, when executed, will call **ap1.write()**, **ap2.write()** and **ap3.write()**.

```
class monitor1;
  analysis_if ap1;
  analysis_if ap2;
  analysis_if ap3;

  task run();
    trans1 t = new();
    repeat(5) begin
      void'(t.randomize());
      $display("monitor:   ",
       "**BROADCAST** addr=%2h  data=%2h", t.addr, t.data);
      ap1.write(t);
      ap2.write(t);
      ap3.write(t);
    end
  endtask
endclass
```



Example 6 - monitor1 with separate analysis_if declarations – no connect() methods

The **top1** module, shown in Example 7, declares and **new()**–constructs the **monitor1**, **subscriber1**, **subscriber2** and **subscriber3** class objects, then copies the subscriber handles **sub1**, **sub2** and **sub3** to the respective **analysis_if** (**ap**) handles declared in **monitor1**.

```
module top1;
  import tb_pkg::*;

  monitor1    mon;
  subscriber1 sub1;
  subscriber2 sub2;
  subscriber3 sub3;

  initial begin
    mon  = new();
    sub1 = new();
    sub2 = new();
    sub3 = new();
    mon.ap1 = sub1;
    mon.ap2 = sub2;
    mon.ap3 = sub3;
    mon.run();
  end
endmodule
```

Example 7 - top1 module with subscriber handles copied to ap handles in monitor1

When this simulation is run, the **monitor1 run()**–task calls all the **write()** methods from each **ap**[#] object. the simulation loops 5 times (**repeat(5)**), each time re-randomizing the transaction. **monitor1** then broadcasts the transaction, and each **subscriber**[#] **write()** method receives and displays the randomized transaction values. Each subscriber re-prints the current contents of the transaction since each subscriber has a handle to the same broadcast transaction.

The simulation results are shown in Figure 1.

```
Randomized trans1 values addr=f9  data=50
monitor:   **BROADCAST** addr=f9  data=50
subscriber1: received    addr=f9  data=50
subscriber2: received    addr=f9  data=50
subscriber3: received    addr=f9  data=50

Randomized trans1 values addr=e9  data=27
monitor:   **BROADCAST** addr=e9  data=27
subscriber1: received    addr=e9  data=27
subscriber2: received    addr=e9  data=27
subscriber3: received    addr=e9  data=27

Randomized trans1 values addr=1f  data=18
monitor:   **BROADCAST** addr=1f  data=18
subscriber1: received    addr=1f  data=18
subscriber2: received    addr=1f  data=18
subscriber3: received    addr=1f  data=18

Randomized trans1 values addr=8f  data=4d
monitor:   **BROADCAST** addr=8f  data=4d
subscriber1: received    addr=8f  data=4d
subscriber2: received    addr=8f  data=4d
subscriber3: received    addr=8f  data=4d

Randomized trans1 values addr=1e  data=7e
monitor:   **BROADCAST** addr=1e  data=7e
subscriber1: received    addr=1e  data=7e
subscriber2: received    addr=1e  data=7e
subscriber3: received    addr=1e  data=7e
```

Figure 1 - Monitor & subscribers – Simulation output

Scenario 1 shows that a common transaction can be broadcast to multiple observers or subscribers. One problem with this scenario is that each time a new subscriber is added to the **top1** module, another **analysis_if** port must be declared in **monitor1**, and the **run()** task must add another call to the new **analysis_if write()** method.

Of course the **top1** module would also need to declare, **new()**-construct another **subscriber** [#] and copy the constructed handle to the new **monitor1 ap** [#] handle. Each new subscriber requires two existing files to be updated.

In this scenario, **top1** needs to know the internal handle names of each **analysis_if**. It would be better to have a **connect()** method to make the connections and hide the internal **analysis_if** handle names. Scenario 2 will add the desired **connect()** method.

2.2.3 Scenario 2 - monitor2 with separate analysis_if and connect() method declarations

In the second scenario, **monitor2**, shown in Example 8, declares three **analysis_if** handles with handle names **ap1**, **ap2** and **ap3**.

monitor2 also has added a **connect()** method for each individual **analysis_if**.

```
class monitor2;
  analysis_if ap1;
  analysis_if ap2;
  analysis_if ap3;

  function void connect1 (analysis_if port);
    ap1 = port;
  endfunction

  function void connect2 (analysis_if port);
    ap2 = port;
  endfunction

  function void connect3 (analysis_if port);
    ap3 = port;
  endfunction

  task run();
    trans1 t = new();
    repeat(5) begin
      void'(t.randomize());
      $display("monitor:   ",
       "**BROADCAST** addr=%2h  data=%2h", t.addr, t.data);
      ap1.write(t);
      ap2.write(t);
      ap3.write(t);
    end
  endtask
endclass
```

Example 8 - monitor2 with separate analysis_if and connect() method declarations

It is starting to become obvious that the addition of each new **analysis_if** subscriber requires the overhead of declaring a new **analysis_if** handle, a corresponding **connect()** method and a call to the **ap**[#]**.write()** method. In scenario 3, all of these issues will be addressed.

```
module top2;
  import tb_pkg::*;

  monitor2    mon;
  subscriber1 sub1;
  subscriber2 sub2;
  subscriber3 sub3;

  initial begin
    mon  = new();
    sub1 = new();
    sub2 = new();
    sub3 = new();
    mon.connect1(sub1);
    mon.connect2(sub2);
    mon.connect3(sub3);
    mon.run();
  end
endmodule
```

Example 9 - top2 module with calls to separate connect() methods

The **top2** module, shown in Example 9, declares and **new()**–constructs the **monitor1**, **subscriber1**, **subscriber2** and **subscriber3** class objects, then calls the **monitor2 connect**[#]**()** method to copy the handles **sub1**, **sub2** and **sub3** to the respective **analysis_if** (**ap**) handles declared in **monitor2**. The **top2** module no longer needs to know the internal **ap**[#] handle names.

One problem that existed in scenario 1 still exists with this scenario. The problem is that each time a new subscriber is added to the **top2** module, another **analysis_if** port must be declared in **monitor2**, and the **run()** task must add another call to the new **analysis_if write()** method. Scenario 2 also requires the addition of a new **connect()** method for each new **analysis_if** handle.

Of course the **top2** module would also need to declare, **new()**-construct another **subscriber** [#] and connect the constructed handle to the new **monitor2 ap**[#] handle. Each new subscriber still requires two existing files to be updated.

When the simulation is run, the **monitor2 run()**–task calls all the **write()** methods from each **ap**[#] object. The simulation loops 5 times (**repeat(5)**), each time re-randomizing the transaction. **monitor2** then broadcasts the transaction, and each **subscriber**[#] **write()** method receives and displays the randomized transaction values. Each subscriber re-prints the current contents of the transaction since each subscriber has a handle to the same broadcast transaction.

The simulation results are the same as those shown in Figure 1.

2.2.4 Scenario 3 - monitor3 with analys_if queue and common connect() method

Scenario 3 solves the problems that required us to modify **monitor1** and **monitor2**. In **monitor3**, shown in Example 10, an unbounded queue of **anlysis_if** ports is declared: **analysis_if ap[$];**

In **monitor3**, each time the **connect()** method is called, a new subscriber handle is **push_back**–added to the **ap**–queue.

Also in **monitor3**, when the **run()** task is called, a **foreach**–loop calls each of the **write()** methods for the queued subscriber handles.

```
class monitor3;
  analysis_if ap[$];  // queue of analysis_if ports

  // Each call to connect will push_back another
  //     analysis_if port onto the ap-queue
  function void connect (analysis_if port);
    ap.push_back(port);
  endfunction

  task run();
    trans1 t = new();
    repeat(5) begin
      void'(t.randomize());
      $display("monitor:   ",
       "**BROADCAST** addr=%2h  data=%2h", t.addr, t.data);
       // Call the write method for each port on the ap-queue
       foreach(ap[i]) ap[i].write(t);
    end
  endtask
endclass
```

Example 10 - monitor3 with analysis queue and common connect() method

These three improvements make it possible to add more subscribers without making multiple modifications to the **monitor3** class. This is roughly how **uvm_analysis_ports** and **uvm_subscribers** work.

The **top3** module, shown in Example 11, can now call a common **connect()** method each time a new subscriber is added to the design.

```
module top3;
  import tb_pkg::*;

  monitor3    mon;
  subscriber1 sub1;
  subscriber2 sub2;
  subscriber3 sub3;

  initial begin
    mon  = new();
    sub1 = new();
    sub2 = new();
    sub3 = new();
    mon.connect(sub1);
    mon.connect(sub2);
    mon.connect(sub3);
    mon.run();
  end
endmodule
```

Example 11 - top3 module with calls to common connect() method that pushes subscriber handles onto queue

Scenario 4 – Buggy example where subscriber2 modifies the transaction

In scenario 4, the simulation was run with the **`+define+BUG`** compilation switch to force **`subscriber2`** to modify the **`addr`** and **`data`** fields of the broadcast transaction.

As can be seen in Figure 2, after **`subscriber2`** modifies the **`addr`** and **`data`** fields, **`subscriber3`** reads the fields from the referenced transaction handle and displays the updated values. **`subscriber3`** should have acted upon the original **`addr`** and **`data`** fields.

```
Randomized trans1 values addr=f9  data=50
monitor:   **BROADCAST** addr=f9  data=50
subscriber1: received    addr=f9  data=50
subscriber2: received    addr=f9  data=50
subscriber2: set         addr=ff  data=00
subscriber3: received    addr=ff  data=00

Randomized trans1 values addr=e9  data=27
monitor:   **BROADCAST** addr=e9  data=27
subscriber1: received    addr=e9  data=27
subscriber2: received    addr=e9  data=27
subscriber2: set         addr=ff  data=00
subscriber3: received    addr=ff  data=00

Randomized trans1 values addr=1f  data=18
monitor:   **BROADCAST** addr=1f  data=18
subscriber1: received    addr=1f  data=18
subscriber2: received    addr=1f  data=18
subscriber2: set         addr=ff  data=00
subscriber3: received    addr=ff  data=00

Randomized trans1 values addr=8f  data=4d
monitor:   **BROADCAST** addr=8f  data=4d
subscriber1: received    addr=8f  data=4d
subscriber2: received    addr=8f  data=4d
subscriber2: set         addr=ff  data=00
subscriber3: received    addr=ff  data=00

Randomized trans1 values addr=1e  data=7e
monitor:   **BROADCAST** addr=1e  data=7e
subscriber1: received    addr=1e  data=7e
subscriber2: received    addr=1e  data=7e
subscriber2: set         addr=ff  data=00
subscriber3: received    addr=ff  data=00
```

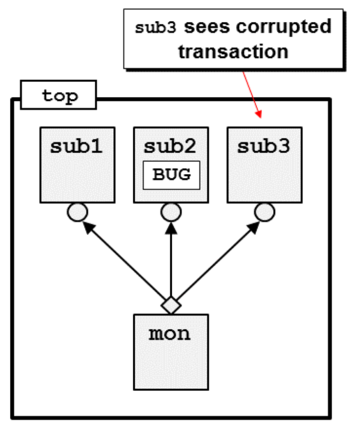


Figure 2 - Monitor & subscribers – Buggy simulation output

Broadcast transactions from an **`analysis_port`** should never be modified. This is why the transaction **`copy()`** command is so vital to a UVM testbench environment. The scoreboard predictor should make a copy of the broadcast transaction, then it reads the copied-transaction inputs to calculate the predicted output. The predicted output is then placed into the copied transaction for comparison to the actual broadcast transaction. The original transaction should never be modified.

Guideline: Modifying the fields of an **`analysis_port`** broadcast transaction should never be done.

UVM analysis_port.write() versus analysis_imp write() method

In this simplified example, each monitor had a **`run()`** method that called each of the subscriber **`write()`** methods.

In UVM the **`uvm_analysis_port`** calls an **`analysis_port.write(tr)`** method to broadcast a **`tr`** transaction. Each **`uvm_analysis_imp`** defines a **`write()`** method that is executed when the analysis port calls the **`write()`** method. There are pros and cons to this naming convention.

The ***port*****`.write()`** command actually reads each **`uvm_analysis_imp`** handle and then calls the **`write()`** method for each handle. The advantage to this approach is that an engineer only needs to remember that there is a **`write()`** command that broadcasts the transaction and a **`write()`** command at the end of each analysis path that is executed. The disadvantage to this approach is that engineers often mistakenly think the ***port*****`.write()`** command is an actual call to the **`uvm_analysis_imp`** **`write()`** method. The ***port*****`.write()`** method could have been named anything, including ***port*****`.broadcast()`** as long as the ***port*****`.command()`** itself called the respective subscriber **`write()`** methods.

In short, ***port*****`.write()`** is not the same as the subscriber **`write()`** methods. The ***port*****`.write()`** method CALLS the subscriber **`write()`** methods. The ***port*****`.write()`** command could have been named anything, but the developers of UVM decided to keep the broadcast and implementation method names the same.

3. UVM Port Fundamentals

The UVM Base Class Library (BCL) includes base port classes that are extended to define the TLM1 (Transaction Level Model 1) ports that are used in UVM verification environments.

3.1 UVM port connection chains

UVM TLM connections include chains of ***port***(s) – (***exports***(s)(optional)) – ***imp***(s)(implementations).

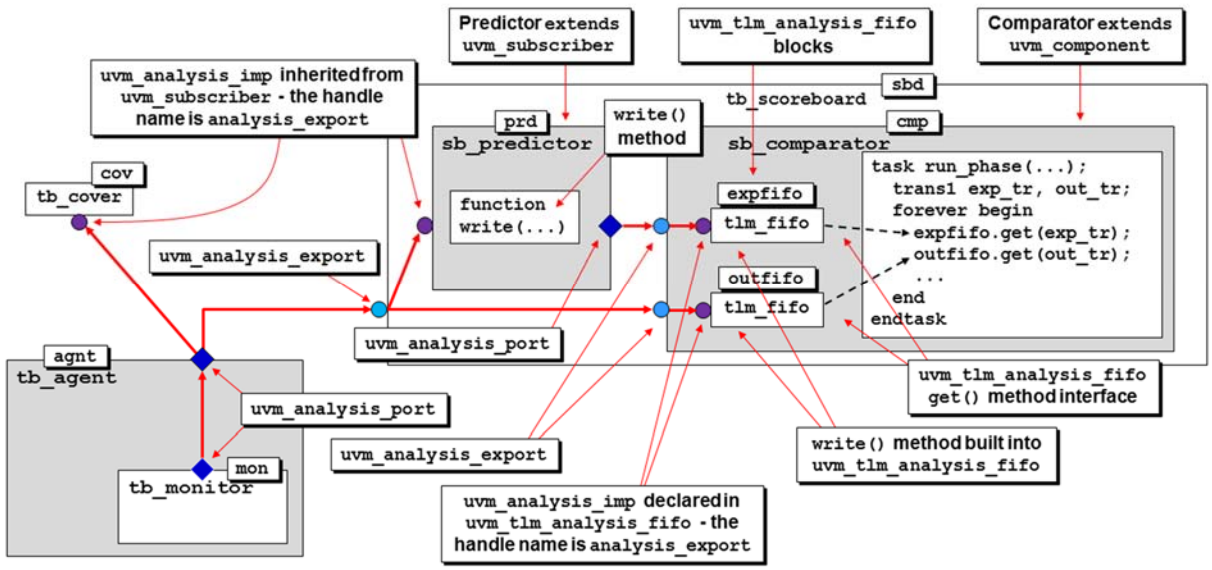


Figure 3 - Common analysis port connections – recommended connections

3.1.1 UVM Ports

UVM ***port***s initiate transaction activity and can connect to: (1) other UVM ***port***s, (2) UVM ***exports***, and (3) UVM ***imps***.

3.1.2 UVM Exports

UVM ***exports*** are basically transfer-ports that can connect to: (1) other UVM ***exports***, and (2) UVM ***imps***.

3.1.3 UVM Imps

UVM ***imps*** (implementations) terminate a chain of ***port***(s)-(***export***(s))-***imp***.

3.1.4 Unfortunate port naming convention

An unfortunate naming convention inside of UVM makes the ***imp***-connections rather confusing. UVM documentation teaches about ***ports*** connecting to ***exports*** and is somewhat vague about ***imps***. In fact the **`uvm_sequencer`** base class includes a **`seq_item_export`** handle declaration, but this so-called "export" handle is really a **`uvm_seq_item_pull_imp`** port type. Similarly, the **`uvm_subscriber`** base class, which is frequently extended to help create scoreboards and coverage collectors, has an **`analysis_export`** handle declaration, but this so-called "export" handle is really a **`uvm_analysis_imp`** port type.

3.1.5 Port-Export-Imp chains

Most ***port-export-imp*** chains only allow a single connection point. The driver-sequencer is a common example in UVM where the driver port (**seq_item_port**) connects to the sequencer ***export*** (**seq_item_export** – which is really an implementation port).

The broadcast port type in UVM is the **uvm_analysis_port**. This port type is allowed to connect to multiple ***port-export-imp*** chains, each of which must terminate with a **uvm_analysis_imp**.

Figure 3 (on page15) shows two common analysis paths that are used in a UVM testbench.

The common sets of paths are shown in Figure 4. The first analysis paths originate with a **uvm_analysis_port** on the **tb_monitor** that broadcasts to another **uvm_analysis_port** on the **tb_agent**, which then branches into two paths with the first path (labeled Path #1) terminating at the **uvm_analysis_imp** on the **tb_cover** coverage collector. The second branch from the **tb_agent** connects to the **uvm_analysis_export** on the **tb_scoreboard**. The **uvm_analysis_export** on the **tb_scoreboard** then branches into two paths with the first path (labeled path #2) terminating at a **uvm_analysis_imp** port on the **sb_predictor** (extended from the **uvm_subscriber**), and the second path (labeled path #3) connects to a **uvm_analysis_export** on the **sb_comparator**, which then terminates at a **uvm_analysis_imp** on the **uvm_tlm_analysis_fifo** with handle name **outfifo**.

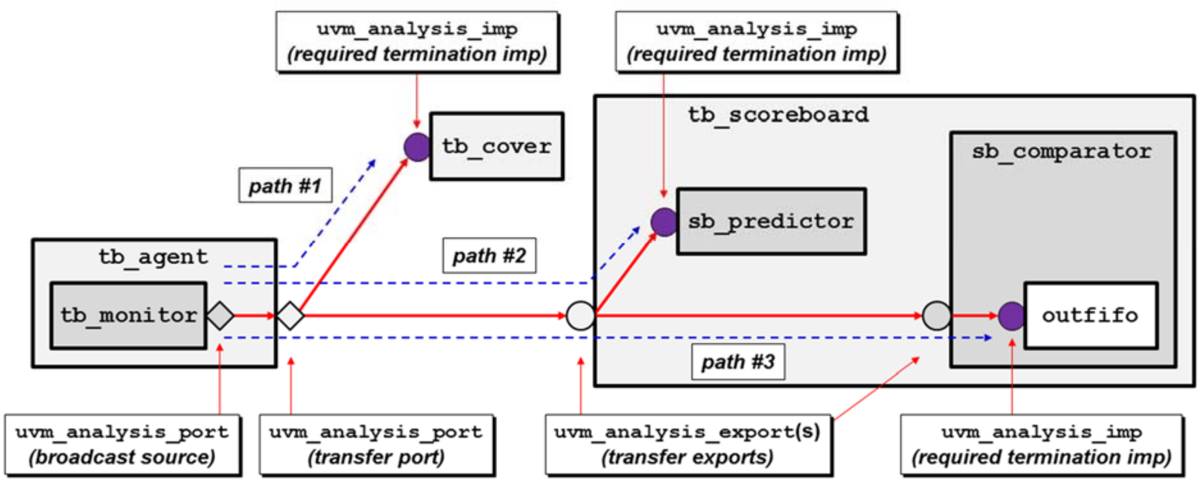


Figure 4 - Analysis paths - first set of three paths

The second analysis path is shown in Figure 5 on page 9. The analysis path starts with a **uvm_analysis_port** on the **sb_predictor** that connects to the **uvm_analysis_export** on the **sb_comparator** and terminates at a **uvm_analysis_imp** on the **uvm_tlm_analysis_fifo** with handle name **expfifo**.

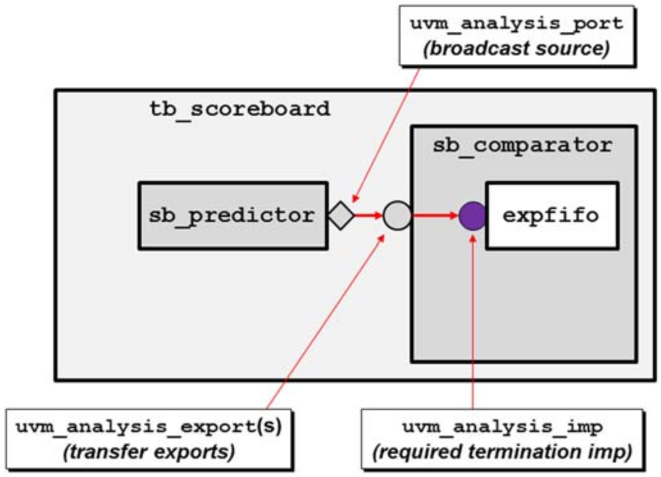


Figure 5 - Analysis paths - second set of paths - just one path

3.1.6 UVM Port & Export interchangability?

An interesting fact about analysis ***ports*** and ***exports*** is that they are largely interchangeable, except for the originating **`uvm_analysis_port`** that is responsible for broadcasting the transaction.

Replacing all of the **`uvm_analysis_export`**(s) (as shown in Figure 3) with **`uvm_analysis_port`**(s) (as shown in Figure 6) does not change the behavior of the UVM testbench. Replacing the ***exports*** with non-broadcasting ***ports*** just changes the type of transfer port. The simulation continues to run the same as it did in Figure 3.

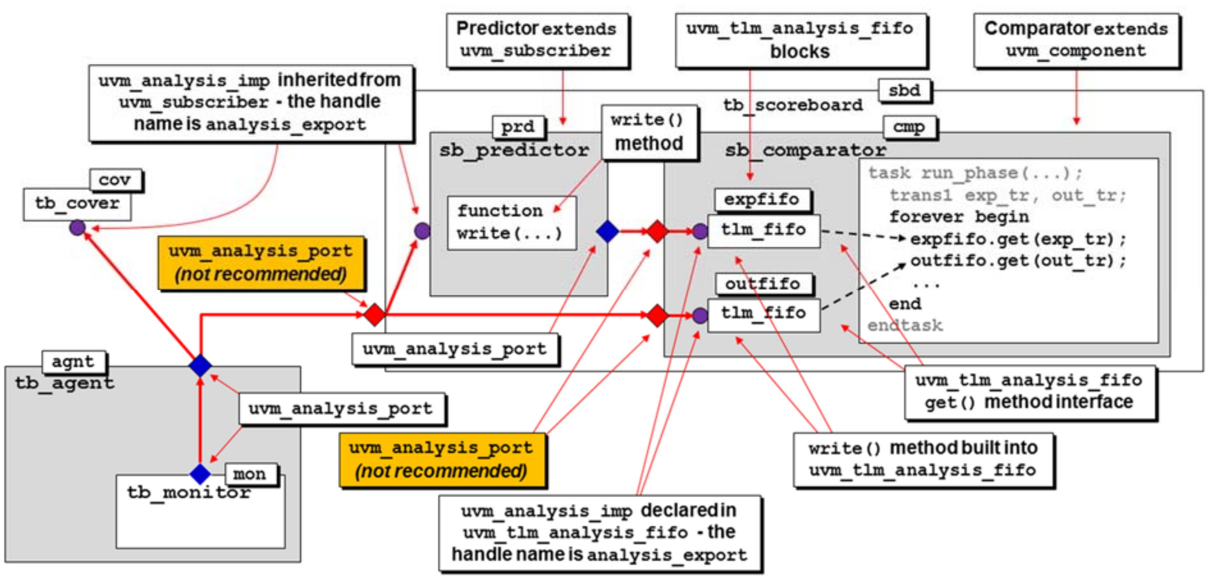


Figure 6 - NOT Recommended – analysis ports & no analysis exports (but it works!)

Similarly, replacing all of the non-broadcasting **`uvm_analysis_port`**(s) (as shown in Figure 3) with **`uvm_analysis_export`**(s) (as shown in Figure 7) does not change the behavior of the UVM testbench. Replacing the non-broadcasting ***ports*** with ***exports*** just changes the type of transfer port. The simulation continues to run the same as it did in Figure 3.

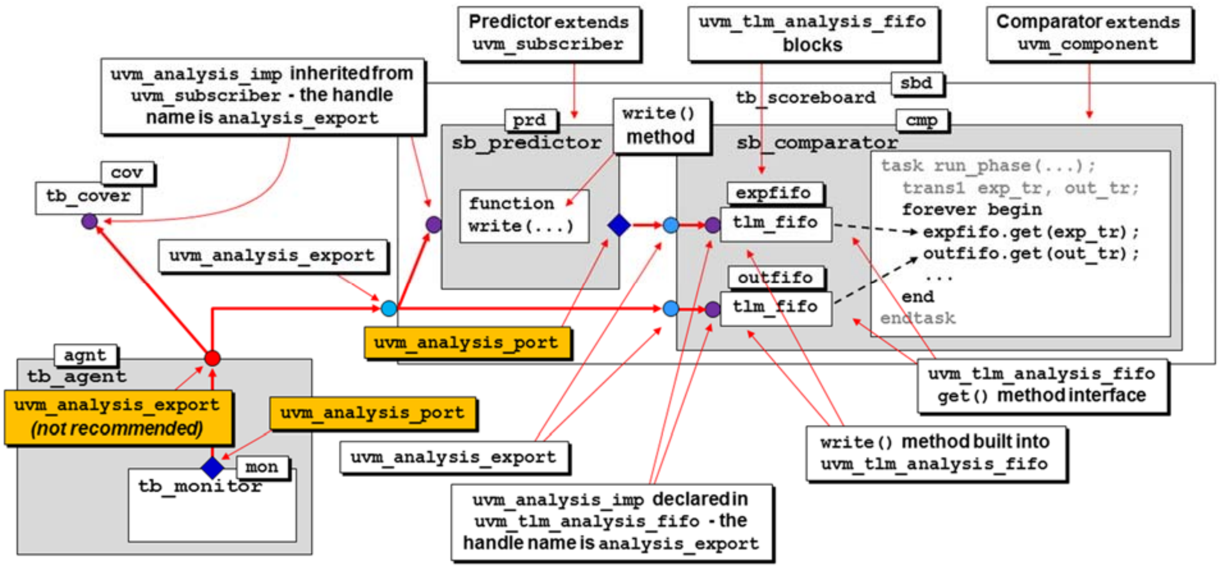


Figure 7 - NOT Recommended – analysis exports & 1 analysis port per analysis source (but it works!)

3.1.7 UVM Port & Export usage guidelines

We do not recommend randomly replacing ***exports*** with ***ports*** and ***ports*** with ***exports*** since the practice is confusing to most UVM verification engineers.

Guideline: use `uvm_analysis_port`(s) for component outputs that are forwarding a transaction to other ***ports*** on an anlysis path, as shown in Figure 8

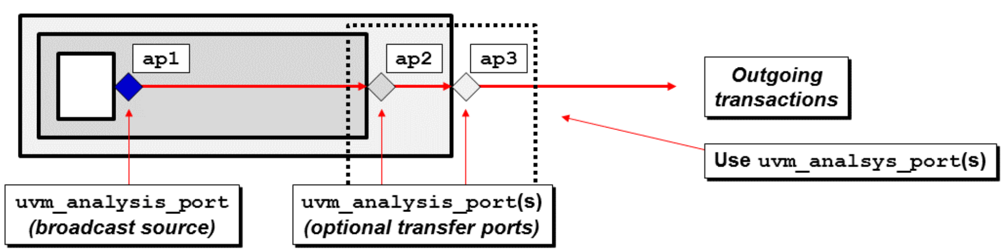


Figure 8 - UVM analysis ports – recommended usage block diagram

Guideline: use `uvm_analysis_export`(s) or `uvm_analysis_imp` for component inputs that are receiving a transaction from other ***ports*** on an anlysis path, as shown in Figure 9.

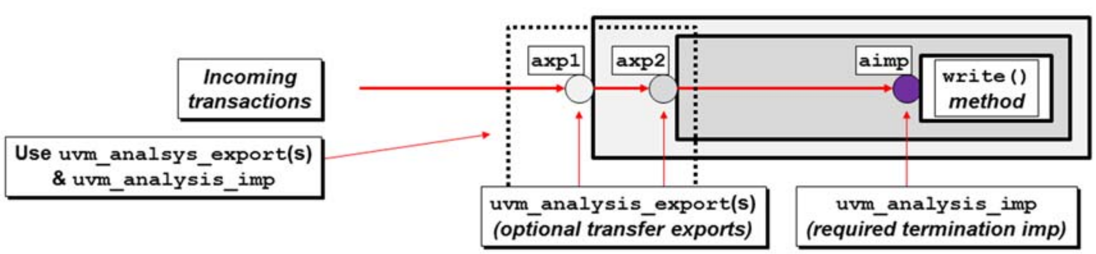


Figure 9 - UVM analysis exports & imps – recommended usage block diagram

3.2 uvm_port_base connect() method

The **`virtual`** **`uvm_port_base`** class includes multiple methods, **`virtual`** and non-**`virtual`**, that are used to define the TLM1 port types. Among the methods in the **`uvm_port_base`** is the **`virtual connect()`** method. The **`virtual connect()`** method includes 46 lines of code that performs certain important inspections to ensure that the TLM connections are legal.

The **`connect()`** method includes functionality that examines the UVM testbench code to make sure that a UVM ***export*** is not connected to a UVM port. The port must connect to an ***export***. Similarly, the **`uvm_port_base connect()`** method checks to make sure a UVM ***imp*** is not connected to any other port. The port or ***export*** must connect to a final ***imp***.

3.3 Scope of port discussion in this paper

A full understanding of all of the different UVM port types, how they can be connected and the methods that are available to pass transactions between components using TLM1 and TLM2 is beyond the scope of this paper.

This paper focuses on the use of the analysis port-chains, some of the underlying implementation basics and the proper use of methods within analysis-chains. This paper also describes common mistakes that are made with analysis chains and how those mistakes manifest themselves in a UVM verification environment. The proper use of the transaction **`copy()`** method is used to avoid many analysis-chain problems, and is discussed in this paper.

4. UVM analysis ports, exports and imps

The UVM analysis path originates with a **uvm_analysis_port** that broadcasts a transaction, which can pass through one or more **uvm_analysis_port**(s) and/or **uvm_analysis_export**(s), and has one or more **uvm_analysis_imp** termination points. If a **uvm_analysis_port** connects to any **uvm_analysis_export**(s), then there must be a **uvm_analysis_imp** at the end of each analysis path chain. It is legal for a **uvm_analysis_port** to connect to other **uvm_analysis_port**(s) without connecting to any **uvm_analysis_export**(s) or **uvm_analysis_imp**. This is analogous to a satellite broadcast where nobody is watching or recording the program. The satellite does not query to find out if anybody is watching the broadcast program and the **uvm_analysis_port** does not query to find out if there are any **uvm_analysis_imp** termination points on the UVM analysis path.

Details about these different port types are described below.

4.1 uvm_analysis_port - broadcast port

As already noted, the **uvm_analysis_port** is a broadcast port that broadcasts a transaction from the port until it reaches zero or more **uvm_analysis_imp** ports where the transaction is either used immediately (in **0**-time), or a copy of the transaction is made so that the copy can be manipulated over time without modifying the original broadcast transaction. This is why the transaction **copy()** command I so important (see Section 4.1.1).

A **uvm_analysis_port** can be connected to other **uvm_analysis_ports**, **uvm_analysis_exports** and **uvm_analysis_imps**, but there is only one **uvm_analysis_imp** per analysis path.

4.1.1 Why is the transaction copy() method so important?

Any component that needs to use the transaction over multiple cycles must take a copy of the transaction because the broadcast transaction can be changed at any time and there is no way for a component to communicate to the **uvm_analysis_port** to hold the original transaction.

Any component that needs to modify any of the fields of the transaction must also take a copy of the transaction because there may be many components that are accessing the broadcast transaction; therefore, modifying the fields of the broadcast transaction will cause problems for other subscribers that needed to access the original transaction contents.

Even if there are no other subscribers, a component should take a copy of the transaction before modifying any fields, since another component might later be added to the analysis path and it would rely on an original unmodified transaction.

4.2 uvm_analysis_export – transfer port

The **uvm_analysis_export** is little more than a transfer-point connection between the broadcasting **uvm_analysis_port** source and each **uvm_analysis_imp** termination point. **uvm_analysis_export**(s) can be viewed as a transfer port.

4.3 uvm_analysis_imp – termination port

A **uvm_analysis_imp** provides the required **write()** method implementation to terminate a UVM analysis path. The verification engineer is required to override the **write()** method with an implementation for the **uvm_analysis_imp**.

4.4 port, export, imp confusion

When it comes to the behavior of ***ports***, ***exports*** (and ***imps***), there is a great deal of confusion surrounding these names. ***Ports*** initiate activity by executing commands while ***exports*** (and ***imps***) are the targets of the commands and actually provide the implementation of the commands. For many engineers new to Transaction Level Modeling this seems backwards. Many believe that the initiator is the export and the target should be the port. What is the reasoning behind these names? This can be best described with an example.

4.4.1 Driver – Car / Port - Export

Every car has a steering wheel. When the driver turns the steering wheel in the clockwise direction, the car turns to the right. When the wheel is moved in the counter clockwise direction the car turns to the left. The driver does not necessarily know if the steering is accomplished through rack and pinion steering, power steering, steer-by-wire or some other mechanism. The driver just knows how to turn the steering wheel to make the car turn to the right or the left.

Every car has an accelerator pedal. When the driver pushes on the accelerator pedal, the car will accelerate. When the driver lets up on the pedal, the car will coast and slow down. The driver does not know if the acceleration is accomplished through a carburetor, fuel injection or some type of electric motor. The driver just knows that pushing the accelerator pedal will increase the car speed while letting up on the pedal will cause the car to coast and slow down.

Every car has a brake pedal. When the driver pushes on the brake pedal, the car will slow down and stop. When the driver lets up on the brake pedal the car will start to move and can now be accelerated. The driver does not know if the braking system uses disc brakes, drum brakes or some type of electronic recovery and battery charging system. The driver just knows that depressing the brake pedal slows the car or brings it to a stop, and that releasing the brake pedal allows the car to go forward again.

In each of these scenarios, the driver issues the commands but does not have the ability to execute any of the described actions. It is the car that must export to the driver the capabilities that the driver will control, but it is the car that has the actual implementation of each of the required functions. Similarly, the car cannot execute any of the commands autonomously but must wait until a driver initiates the appropriate commands. The driver is the initiator-port, while the car is the target-export.

Note, the driver cannot successfully issue any command that is not exported by the car. The driver might try to place a hands-free call over the automobile Bluetooth connection to the driver's mobile phone, but if the car does not export the Bluetooth-Phone control capability (because it is not a feature of that particular car), such a command by the driver will fail. The driver can only issue commands that are exported to the driver by the car.

This is what happens with TLM connections. The port can only execute commands that are exported by the connected ***export*** or ***imp***.

4.4.2 Export, imp confusion

The UVM documentation describes all connections as ***port-export*** connections, but in reality, it is an ***imp*** (implementation ***export***) that provides the actual exported functionality. Two examples found in common UVM testbenches include:

Driver **`seq_item_port`** connects to the sequencer **`seq_item_export`**.

1. **`seq_item_port`** is just the handle name of the **`uvm_seq_item_pull_port`** (***port***)
2. **`seq_item_export`** is just the handle name of the **`uvm_seq_item_pull_imp`** (***imp***-lementation)

The UVM subscriber component has a built-in **`analysis_export`**, which is really just the handle name for a **`uvm_analysis_imp`** (***imp***-lementation).

The UVM documentation seemingly tries to hide the existence of ***imps*** by giving them "export" handle names instead of "imp" handle names. We personally believe it is a mistake to declare ***imps*** with "export" handle names but that is how the UVM base classes are defined so an engineer just has to understand this naming inconsistency.

5. Declaring and constructing ports & TLM FIFOs

TLM FIFO is an important component that is used for testbench synchronization, especially in a UVM scoreboard. TLM FIFOs are built from SystemVerilog mailboxes. Mailboxes are described in Section 7. TLM FIFOs are described in Section 8.

Ports and TLM FIFOs are UVM base classes that are NOT registered with the factory and are used directly in a UVM testbench. Because they are not registered with the factory and because they are used directly, they are **`new()`**-constructed and not factory **`::type_id::create`**-ed.

5.1 How are analysis ports and TLM FIFOs declared?

The declaration of the different analysis port types has a subtle difference. An example **`uvm_analysis_port`** is declared with the transaction type parameter displayed near the top of the **`tb_monitor`** class shown in Example 12. The declaration requires the transaction-type parameter. In this example, the **`tb_monitor`** broadcasts a transaction using the **`ap.write()`** command in the **`run_phase()`**.

```
class tb_monitor extends uvm_monitor;
  ...
  uvm_analysis_port #(trans1) ap;
  ...
  function void build_phase(uvm_phase phase);
    ...
    ap = new("ap", this); // build the analysis port
    ...

  task run_phase(uvm_phase phase);
    ...
    ap.write(tr);
```

Example 12 - tb_monitor with uvm_analysis_port declaration and ap.write() command

An example **uvm_analysis_export** is declared with the transaction type parameter displayed near the top of the **tb_scoreboard** class shown in Example 13. The declaration requires the transaction-type parameter. The **uvm_analysis_export** is typically connected to other **uvm_analysis_port**(s), **uvm_analysis_export**(s) and possibly one **uvm_analysis_imp**. There are now methods that need to be excuted to propagate transactions on the **uvm_analysis_export**.

```
class tb_scoreboard extends uvm_scoreboard;
  ...
  uvm_analysis_export #(trans1) axp;
  sb_predictor                  prd;
  sb_comparator                 cmp;
  ...
  function void build_phase(uvm_phase phase);
    axp = new("axp", this);
    prd =  sb_predictor::type_id::create("prd", this);
    cmp = sb_comparator::type_id::create("cmp", this);
  endfunction

  function void connect_phase( uvm_phase phase );
    axp.connect           (prd.analysis_export);
    axp.connect           (cmp.axp_out);
  endfunction
  ...
```

Example 13 - tb_scoreboard with uvm_analysis_export declaration and connection to other analysis-type ports

The **uvm_analysis_imp**, unlike the **uvm_analysis_port** and **uvm_analysis_export** declarations, is required to declare TWO parameters, the transaction type and the class name. An example **uvm_analysis_imp** is displayed near the top of the **sb_predictor** class shown in Example 14. The declared **uvm_analysis_imp** must be built and must override the **write()** method to provide the required implementation.

```
class sb_predictor extends uvm_component;
  ...
  uvm_analysis_imp  #(trans1, sb_predictor) analysis_export;
  uvm_analysis_port #(trans1)               results_ap;
  ...
  function void build_phase(uvm_phase phase);
    ...
    analysis_export = new("analysis_export", this);
    results_ap      = new("results_ap",      this);
  endfunction

  function void write(trans1 t);
    ...
    results_ap.write(exp_tr);
    ...
```

Example 14 - sb_predictor with uvm_analysis_imp declaration and write() method

5.2 Where are TLM ports and TLM FIFOs constructed?

The construction of TLM ports and TLM FIFOs can either be done in the component **new()** constructor, or in the **build_phase()** method.

The UVM Class Refence[4] has examples of constructing ports and TLM FIFOs in the component **new()**-constructor (e.g. Section 14. TLM1 Interfaces, Ports, Exports and Transport Interfaces, and Section 16. Analysis Ports) and in the **build_phase()** (e.g. Section 14. TLM1 Interfaces, Ports, Exports and Transport Interfaces (same example)).

Aside from the ports, all components are built in the **build_phase()** method to allow runtime, build-this, factory lookup and creation of components registered with the factory. Since ports and TLM FIFOs are not registered with the factory, they do not have to be created in the **build_phase()** method.

That being said, we prefer to **new()**-construct ports and FIFOs in the **build_phase()** of a component. We believe it makes more sense to put all building of factory-selected components, as well as ports and TLM FIFO components in a single place for easy examination.

Removing **new()**-construction of ports and TLM FIFOs from the constructor also means that 90%+ of all component constructors can use the same identical, boring, three lines of **new()**-constuctor code:

```
function new (string name, component parent);
  super.new(name, parent);
endfunction
```

In short, it does not matter whether ports and FIFOs are placed in the component **new()**-constructor or in the component **build_phase()**, but we prefer to put them in the **build_phase()**.

5.3 uvm_analysis_imp port usage options

When terminating an analysis path, there are three options.

Option #1: is to explicitly declare **uvm_analysis_imp** ports inside of components. This option has three important requirements:

(1) Unlike the **uvm_analysis_port** and **uvm_analysis_export** port declarations, the **uvm_analysis_imp** declaration requires two parameters, the transaction type and the name of the class where the **uvm_analysis_imp** is declared.
(2) The **uvm_analysis_imp** must be built in either the **build_phase()** (our preference) or in the component **new()**-constructor.
(3) The class with the **uvm_analysis_imp** declaration must override the **write()** method and provide an implementation for that same method.

Option #2: is to declare a component that is an extension of the **uvm_subscriber** base class. This alternative includes a pre-declared and constructed **uvm_analysis_imp**. The user is only required to override the **write()** method and provide an implementation for that same method. The **uvm_subscriber** is described in Section 6.

Option #3: is to declare and build a **uvm_tlm_analysis_fifo** and then connect the **uvm_tlm_analysis_fifo** to a **uvm_analysis_export** on the component. The **uvm_tlm_analysis_fifo** has already pre-declared a **uvm_analysis_imp** port with corresponding **write()** method to store the transaction into the **uvm_tlm_analysis_fifo**.

6. uvm_subscriber

The **uvm_subscriber** base class is appropriately named. A subscriber is connected either directly or indirectly to a **uvm_analysis_port** and provides the **write()** method required by a **uvm_analysis_imp**.

6.1 pure virtual write function

The **pure** keyword is only legal in a **virtual** class. A **pure virtual** method is a method that is only a prototype in the **virtual** class (only the method header) and requires that the extended class actually provide the method implementation. The **uvm_subscriber virtual** class includes a **pure virtual write()** method. Any class that extends the **uvm_subscriber** must override the **write()** method with an actual implementation.

6.2 multiple uvm_analysis_imp ports on the same component

Each **uvm_analysis_imp** requires a function called **write()**. When there are multiple **uvm_analysis_imp**(s) on the same component, each must have its own **write()** method, but of course, each component class scope is only allowed to have one method named **write()**. So what can be done if more than one **uvm_analysis_imp** is required on the same component? A solution to this problem is shown in Section 9.

7. SystemVerilog mailbox

The SystemVerilog language added the **mailbox** keyword and a **mailbox** is a special type of SystemVerilog queue or FIFO that is very useful in verification environments.

A circular queue is a sequentially accessed memory with write and read pointers that wrap back to zero when the queue depth is reached, thus allowing the reuse of each memory location as the pointers "wrap" back to location zero.

FIFOs are circular queues designed with a fixed number of addressable words or entries. FIFOs also have full and empty flags to indicate when all of the available locations have either been filled or all of the available locations are empty.

A SystemVerilog queue can have a bounded size, like a FIFO, or be unbounded in size, which is the default. A SystemVerilog queue can be manipulated using a queue-specific algebra in SystemVerilog (see IEEE Std 1800-2012[2] section 7.10.1), or they can be manipulated using queue-specific built-in methods (see section 7.10.2).

7.1 Mailbox –vs- queue

SystemVerilog added both the queue and **mailbox** dynamic types, which did not previously exist in Verilog. As will be described later, the **uvm_tlm_fifo** and **uvm_tlm_analysis_fifo** both use the **mailbox** and not the queue because the **mailbox** offers an important blocking feature that is used in UVM scoreboards.

Both the queue and **mailbox** can be declared to be unbounded, which is to say that both can hold an unlimited number of entries, or both can be declared to be bounded, which is to say the declaration can indicate the maximum number of entries allowed in each.

Both have **put()** and **try_put()** methods where the **put()** method is a built-in **task** and the **try_put()** is a built-in **function**. Since a queue or **mailbox** can be bounded, they might be full forcing the **put()** command to block for a period of time until the queue or **mailbox** has space to allow a new entry to be added (when a corresponding **get()** or **try_get()** has removed an item from the queue or **mailbox**). The **try_put()** command is a **function** that completes in 0-time by either successfully placing a new item on the queue or **mailbox** and returning status that the **try_put()** command succeeded, or the **try_put()** fails because a bounded queue or **mailbox** is already full, in which case the **try_put()** command returns a "fail" status of **0**. If the queue or **mailbox** is unbounded, both the **put()** and **try_put()** commands will always succeed with the only difference being that the **try_put()** command will return status to indicate that the **try_put()** operation succeeded (returns a positive value of **1**).

The **try_put()** method can be called by either a task or function since it completes in 0-time, while a **put()** method can only be called by a task since the **put()** command might block and consume simulation time.

8. TLM FIFOs

There are two types of built-in TLM FIFOs in UVM, (1) **uvm_tlm_fifo**, described in Section 8.2 , and (2) **uvm_tlm_analysis_fifo**, described in Section 0. Both of these TLM FIFOs are derivatives of the **uvm_tlm_fifo_base** class, descrbed in Section 8.1 TLM FIFOs are valuable synchronization structures that are commonly used in UVM, especially in scoreboards. We find the **uvm_tlm_analysis_fifo** to typically be more valuable in a UVM scoreboard.

8.1 uvm_tlm_fifo_base

The **uvm_tlm_fifo_base** virtual class in the UVM Base Class Library (BCL) defines four ports, a **build_phase()** method and 17 additional virtual methods. The purpose of the **uvm_tlm_fifo_base** class is to reserve method prototypes to be used by the **uvm_tlm_fifo**, extended from the **uvm_tlm_fifo_base**, and the **uvm_tlm_analysis_fifo**, extended from the **uvm_tlm_fifo**. Although legal, verification engineers typically do not extend the **uvm_tlm_fifo_base** class.

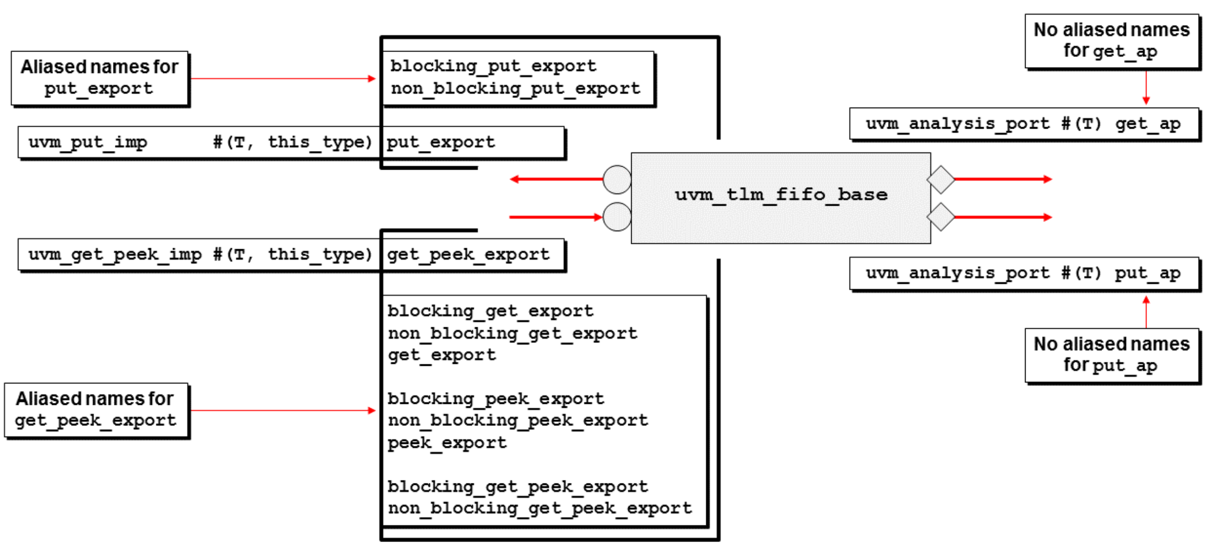


Figure 10 - uvm_tlm_fifo_base ports

As can be seen in Figure 10, the **uvm_tlm_fifo_base** class has two input ports and two output analysis ports. The input ports have many aliased names while the output **uvm_analysis_port**(s) only have the names **get_ap** and **put_ap**.
The port types and aliased names for those ports are shown in Table 1. The reserved method names and notes about method implementation are shown in Table 2.

Table 1 - uvm_tlm_fifo_base port names and port name aliases

Port type	Port name or alias	Port commonly used in **uvm_tlm_fifo** or **uvm_tlm_analysis_fifo**?
uvm_put_imp	**put_export** **blocking_put_export** **non_blocking_put_export**	No
uvm_get_peek_imp	**get_peek_export** **blocking_get_export** **non_blocking_get_export** **get_export** **blocking_peek_export** **non_blocking_peek_export** **peek_export** **blocking_get_peek_export** **non_blocking_get_peek_export**	No
uvm_analysis_port	**put_ap**	No
uvm_analysis_port	**get_ap**	No

Table 2 - uvm_tlm_fifo_base methods and usage notes

Virtual empty method	Note	Native mailbox method?	Implemented by `uvm_tlm_fifo`?	Implemented by `uvm_tlm_analysis_fifo`?
`put()`	blocks	Yes	Yes	Yes
`get()`	blocks	Yes	Yes	Yes
`peek()`	blocks	Yes	Yes	Yes
`try_put()`	returns bit	Yes	Yes	Yes
`try_get()`	returns bit	Yes	Yes	Yes
`try_peek()`	returns bit	Yes	Yes	Yes
`can_put()`	returns bit	No	Yes	Yes
`can_get()`	returns bit	No	Yes	Yes
`can_peek()`	returns bit	No	Yes	Yes
`build_phase()`	UVM phase	UVM phase		
`flush()`	clears fifo	No	Yes	Yes
`ok_to_put()`	returns uvm_tlm_event	Not implemented	Not implemented	Not implemented
`ok_to_get()`	returns uvm_tlm_event	Not implemented	Not implemented	Not implemented
`ok_to_peek()`	returns uvm_tlm_event	Not implemented	Not implemented	Not implemented
`is_empty()`	returns bit	No	Yes	Yes
`is_full()`	returns bit	No	Yes	Yes
`size()`	returns int	No	Yes	Yes
`used()`	returns int	Yes	Yes	Yes

8.2 uvm_tlm_fifo

The `uvm_tlm_fifo` can be used in UVM scoreboard design. The `uvm_tlm_fifo` is internally built using a SystemVerilog `mailbox` and is defined with a default depth of just 1 transaction, which is nearly useless, but this can be changed and is alsmost always changed when the `uvm_tlm_fifo` is `new()`-constructed.

8.2.1 uvm_tlm_fifo new()-constructor & size

The `new()` constructor for the `uvm_tlm_fifo` takes three arguments, `name`, `parent` and `size`. The `size` argument has the default depth of `1`. Setting the `size` argument to `0` causes the `uvm_tlm_fifo` to have unbounded depth, and setting the `size` to `0` is generally recommended for scoreboard designs. Note, the `uvm_tlm_analysis_fifo`, described in Section 0has a default `size` already set to `0`, which is generally ideal for verification purposes.

The tb_scoreboard in Example 15 declares two **uvm_tlm_fifo**(s) to hold expected and actual transactions. Both are **new()**-constructed with an unbounded size value of **0**, and both are written to using **void**-casted **try_put()** methods. The **write_prd()** (write-predictor) method also takes a copy of the broadcast transaction before predicting and storing the correct output values into the expected transaction (**etr**).

```
class tb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(tb_scoreboard)
  ...
  uvm_tlm_fifo #(trans1) expfifo;
  uvm_tlm_fifo #(trans1) outfifo;
  ...

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ...
    expfifo = new("expfifo", this, 0); // Unbounded tlm_fifo
    outfifo = new("outfifo", this, 0); // Unbounded tlm_fifo
  endfunction

  function void write_prd(trans1 tr);
    ...
    etr.copy(tr); // create copy of tr object
    ...
    void'(expfifo.try_put(etr));
  endfunction

  function void write_out(trans1 tr);
    void'(outfifo.try_put(tr));
  endfunction

  task run_phase(uvm_phase phase);
    forever begin
      expfifo.get(exp_tr);
      outfifo.get(out_tr);
      ...
```

Example 15 - tb_scoreboard that uses two uvm_tlm_fifo(s)

8.2.2 uvm_tlm_fifo put() & try_put() methods

Since the **uvm_tlm_fifo** is extended from the **uvm_tlm_fifo_base**, the **uvm_tlm_fifo** has two input ports and two **uvm_analysis_port**(s). Although it is possible to connect to any of these ports, they are frequently left unconnected in a scoreboard design.

The **put()** and **try_put()** methods store transactions into the **uvm_tlm_fifo** mailbox. These commands also broadcast the same transaction out of the **put_ap** as shown in Figure 11. The **put_ap uvm_analysis_port** is rarely used.

Setting the **uvm_tlm_fifo size** to **0** means that the **put()** and **try_put()** methods both automatically succeed, with the latter returning a status value, which will always indicate that the **try_put()** action succeeded.

Generally when calling the **try_put()** method on an unbounded **uvm_tlm_fifo** a **void**-cast is used to throw away the return value (as shown in Example 15).

Since the **write()** method defined for a **uvm_analysis_imp** is a function, only the **try_put()** method can be used to write to the mailbox even though both would succeed for an unbounded **uvm_tlm_fifo**.[1]

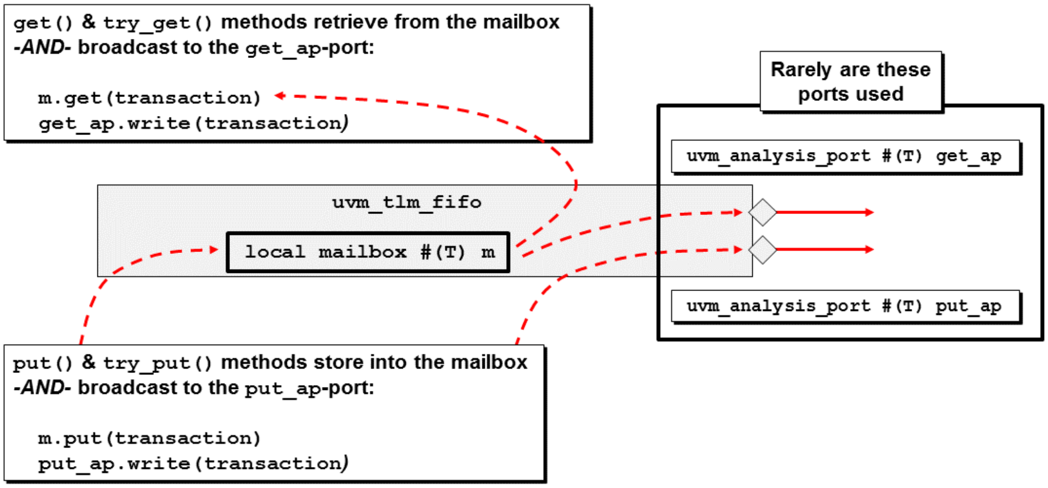


Figure 11 - uvm_tlm_fifo put() and get() method behavior

8.2.3 uvm_tlm_fifo get() & try_get() methods

The **get()** and **try_get()** methods retrieve transactions from the **uvm_tlm_fifo** mailbox. These commands also broadcast the same transaction out of the **get_ap** as shown in Figure 11. The **get_ap uvm_analysis_port** is rarely used.

get() is a blocking task that waits until there is a transaction in the mailbox of the **uvm_tlm_fifo** to be retrieved. The blocking **get()** command is ideal for use in a scoreboard comparator since it waits (blocks) until a transaction is available to retrieve. There is no equivalent blocking **get()** method defined for the SystemVerilog queue, which is why the **uvm_tlm_fifo** is a better choice instead of using a queue in a scoreboard comparator.

Engineers who try to use a queue in a scoreboard comparator typically have to either do sampling or use event triggers to wait until there is a queued transaction that can be properly extracted.

8.2.4 uvm_tlm_fifo disadvantage

Placing a **uvm_tlm_fifo** into an analysis path has the disadvantage that the scorebard must somewhere implement a **uvm_analysis_imp write()** method to receive the broadcast transaction and then does the **void'(try_put(etr))** to store the transaction into the **uvm_tlm_fifo**. The **uvm_tlm_analysis_fifo** described in the next section removes the need to implement the **write()** method to execute the **try_put()** command.

---

[1] Note: one of the EDA vendors used to allow (and may still allow) the **put()** method to be called from the **uvm_analysis_imp write()** method when the **uvm_tlm_fifo** was declared to be unbounded, because the **put()** method would execute in 0-time and succeed. The other EDA vendors properly disallowed the **put()** method since it should not be legal to call a **put()** task from a **write()** function. We recommend that engineers not use a **put()** method with an unbounded **uvm_tlm_fifo** even if your chosen vendor allows the operation.

8.3 uvm_tlm_analysis_fifo

The **uvm_tlm_analysis_fifo** extends the **uvm_tlm_fifo** (described in Section 8.2) and therefore inherits all of the ports and capabilities of the **uvm_tlm_fifo**.

The **uvm_tlm_analysis_fifo** has five ports; the same four ports defined in the **uvm_tlm_fifo** as described in Section 8.2 plus one additional **uvm_analysis_imp** port with handle name **analysis_export** as shown in Figure 12. The four **uvm_tlm_fifo** inherited ports are rarely used in a UVM scoreboard.

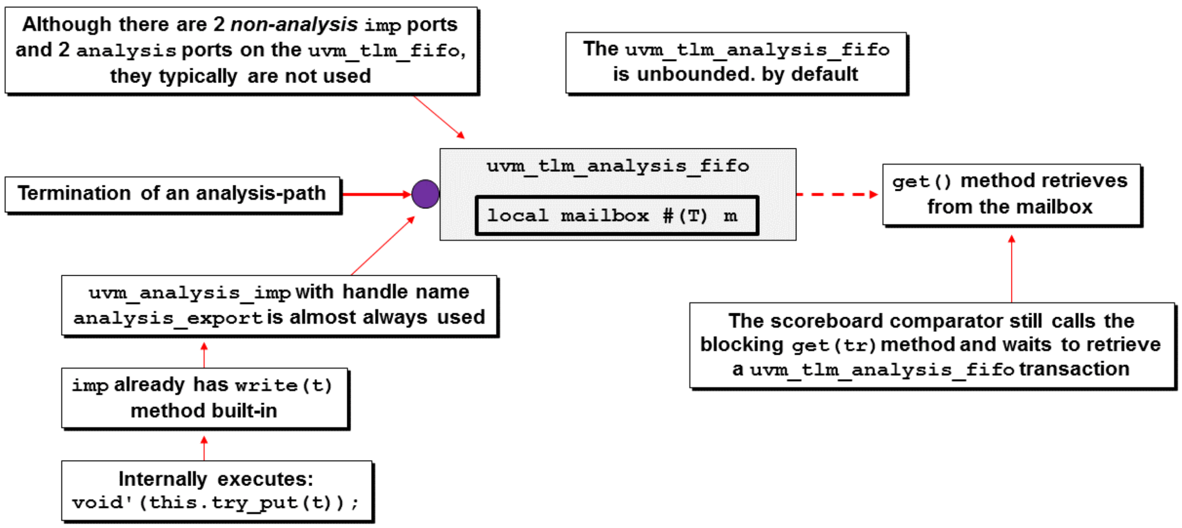


Figure 12 - uvm_tlm_analysis_fifo – most common usage

The **uvm_tlm_analysis_fifo** is ideal to store transactions that were broadcast from a **uvm_analysis_port**. Within an analysis path, the **uvm_tlm_analysis_fifo** has two distinct advantages over the **uvm_tlm_fifo**: (1) by default, the **uvm_tlm_analysis_fifo** has unbounded size, which is perfect for UVM scoreboard development, and (2) the **uvm_tlm_analysis_fifo** has a built-in **uvm_analysis_imp** port with corresponding **write()** method to store the broadcast transaction.

Unlike the **uvm_tlm_fifo**, the **uvm_tlm_analysis_fifo** has an extra **uvm_analysis_imp** port that must be connected inside of the scoreboard. Section 8.4 describes and graphically shows the differences between using the **uvm_tlm_fifo** versus the **uvm_tlm_analysis_fifo** in a scoreboard design.

8.4 Quick-summary of uvm_tlm_analysis_fifo –vs- uvm_tlm_fifo

Most UVM scoreboards are most efficiently implemented using unbounded TLM FIFOs. The **uvm_tlm_analysis_fifo** is unbounded by default, which is perfect for scoreboard development. The **uvm_tlm_fifo** must be **new()**-constructed with a **size** = **0** to become unbounded. This is a minor advantage to using the **uvm_tlm_analysis_fifo**.

The biggest advantage of the **uvm_tlm_analysis_fifo** over the **uvm_tlm_fifo** is that the **uvm_tlm_analysis_fifo** has a built-in **uvm_analysis_imp** port with corresponding built-in **write()** method to capture transactions that were broadcast from an analysis port. This is an extremely useful advantage over the **uvm_tlm_fifo** and saves a great deal of work in developing a UVM scoreboard.

These differences are shown graphically in Figure 13.

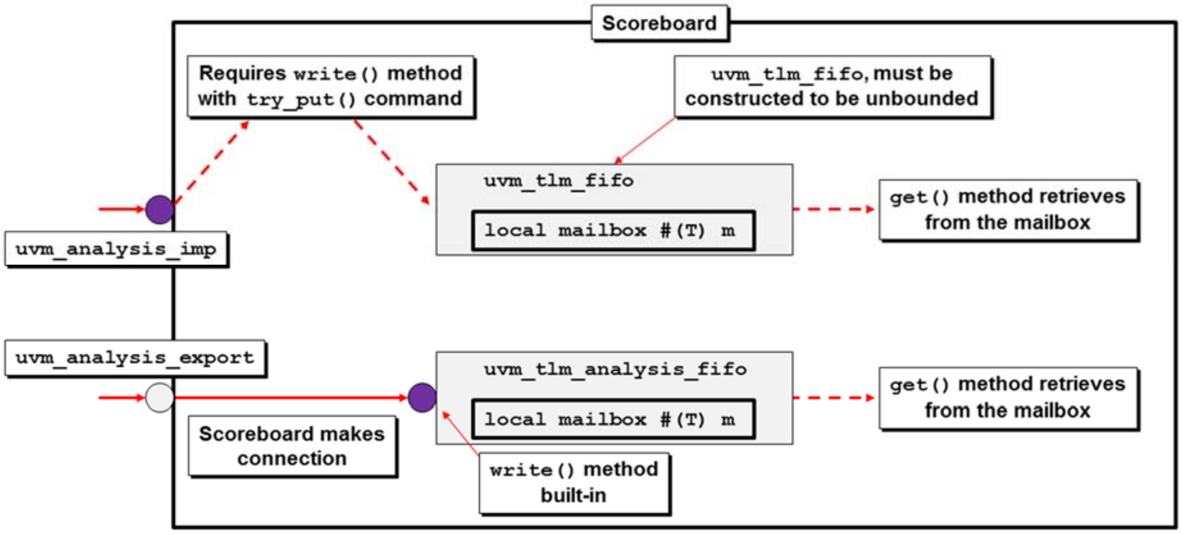


Figure 13 - uvm_tlm_fifo –vs- uvm_tlm_analysis_fifo usage

Comparing the usage requirements of the **uvm_tlm_fifo** to the **uvm_tlm_analysis_fifo** in a scoreboard, as shown in Figure 13, is described below:

(1) The **uvm_tlm_fifo** typically requires the declaration and construction of a separate **uvm_analysis_imp** (or **uvm_subscriber** with built-in **uvm_analysis_imp**) and then requires the implementation of a corresponding **write()** method, which uses a **void'(try_put())** call to store the transaction into the **uvm_tlm_fifo**. Of course the separate **uvm_tlm_fifo** must also be declared and constructed.

(2) The **uvm_tlm_analysis_fifo** typically requires the declaration and construction of separate **uvm_analysis_export** and the **uvm_tlm_analysis_fifo**. The **uvm_analysis_export** port is then connected to the **uvm_tlm_analysis_fifo**. The **uvm_tlm_analysis_fifo** already has the required **uvm_analysis_imp** and corresponding **write()** method.

9. `uvm_analysis_imp_decl(SFX) macro

If there are multiple **uvm_analysis_imp**(s) in a component, the user must define multiple uniquely named **uvm_analysis_imp_*SFX*** ports with corresponding **write_*SFX* ()** methods.

UVM has a macro to define new **uvm_analysis_imp** ports with unique port-suffix names and unique write-method-suffix names. The macro is **`uvm_analysis_imp_decl(*SFX*)**. This macro is typically used for each **uvm_analysis_imp** port on a multi-**uvm_analysis_imp** component.

The first two lines of Example 16 use the **`uvm_analysis_imp_decl(*SFX*)** macros. The ***SFX*** values can be numbers or characters and can include the "_" as shown in this example or omit the "_". This example uses suffix values of " **_prd** " for "predictor" and " **_out** " for "output."

Using the **`uvm_analysis_imp_decl(*SFX*)** macros will define two new **uvm_analysis_imp** port types that must include the suffix values declared in the macros. In this example, the port types are **uvm_analysis_imp_prd** and **uvm_analysis_imp_out**. The macros also create two new **write()** methods that must include the suffix values declared in the macros. In this example, the write method names are **write_prd()** and **write_out()** and these are the corresponding write methods for the **uvm_analysis_imp_prd** and **uvm_analysis_imp_out** ports respectively.

Note that the **uvm_analysis_imp_*SFX*** handle names are not required to use the ***SFX*** values, but we believe it is a good idea to use the suffix names as part of the handle names to reduce coding confusion.

```
`uvm_analysis_imp_decl(_prd)
`uvm_analysis_imp_decl(_out)

class tb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(tb_scoreboard)

  ...

  uvm_analysis_imp_prd #(trans1, tb_scoreboard) ap_prd;
  uvm_analysis_imp_out #(trans1, tb_scoreboard) ap_out;

  ...

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ap_prd  = new("ap_prd",   this);
    ap_out  = new("ap_out",   this);
    ...
  endfunction

  function void write_prd(trans1 tr);
    ...

  function void write_out(trans1 tr);
    ...
```

Example 16 - tb_scoreboard with two `uvm_analysis_imp_decl(SFX) macros, ports and write() methods

9.1 How many ports are allowed on a scoreboard?

We have talked to a surprisingly large number of engineers who were under the impression that a scoreboard could have only one or two ports of any type and that they had to use the same transaction type on the 2-port variety. This is not true.

UVM scoreboards can have any number of ***ports***, ***exports*** and ***imps*** and the ports can be parameterized to any number of transaction types.

It is true that most block-level scoreboards only have one or two input-port types to sample a common transcation and to perform simple calculations of expected values to compare against actual sampled output values, but for larger UVM enviironments, it is not uncommon to have multiple ports that are parameterized to multiple transaction types and then allow the scoreboard to perform a transfer function on one of the transactions before comparing specific fields between multiple transaction types.

UVM even has a **uvm_algorithmic_comparator** documented in the UVM Class Reference[4] (Section 18.2) that is designed for this purpose. The **uvm_algorithmic_comparator** is described as a comparator that:

> *"compares two streams of transactions; however, the transaction streams might be of different type objects. This device will use a user written transformation function to convert one type to another before performing a comparison."*

10. Example with typical analysis/copy() problems

Consider a UVM testbench example that appeared to be working but that had hidden problems for months. Once we discovered that there was a problem, it still took hours to identify the cause and fix the problem. We want to help the reader to avoid the same time consuming mistakes. You're welcome!

We had a training lab that had a scoreboard with two **`uvm_analysis_imp`** ports to demonstrate the multi-***imp*** port solution described in Section 9. The **`write_prd()`** method took the transaction and calculated the expected output values and wrote them back to the transaction and put them into the expected **`uvm_tlm_fifo`**. The **`write_out()`** method simply put the transaction into the actual output **`uvm_tlm_fifo`**. The scoreboard comparison logic then compared the expected outputs to the actual outputs to determine if each simulation vector passed or failed. Once the testbench was working we thought we were done.

Some months later we decided to add a feature to the lab. We asked engineers to break the DUT (Design Under Test) and observe the reported errors to see if they made sense relative to the bug. Much to our surprise, the testbench continued to report that the tests passed. We broke the DUT more, and the testbench still continued to pass. We traced the signals on the DUT and noticed that the expected wrong outputs were being generated but the testbench output continued to show and report valid outputs.

We finally figured out that the **`write_prd()`** calculate-expected function was using the transaction inputs to modify the outputs of the broadcast transaction before putting them into the expected **`uvm_tlm_fifo`**. This meant that the original transaction now had corrected outputs that had overwritten the erroneous DUT outputs so the transaction that was put in the actual output **`uvm_tlm_fifo`** had been corrected and the testbench reported no failures while showing the updated output values.

This is why any subscriber that intends to modify a transaction should first take a copy of the transaction and use the inputs of the copied transaction to calculate the outputs to be placed back into the copied transaction.

Guideline: once the UVM testbench is working, break the DUT to see if the UVM testbench can catch the bug. This will show you if you have the same problem that we described in this section.

11. Summary & Conclusions

The **`uvm_analysis_port`** is a port that broadcasts transactions to zero or more destinations, typically called subscribers. The end of each analysis path subscriber chain is a **`uvm_analysis_imp`** that must provide an implementation by overwriting the ***imp***'s **`write()`** method.

Each component that subscribes to a common transaction analysis path has a handle to a common transaction. Any component that needs to modify any of the transaction fields should first take a copy of the transaction and only modify the fields of the copied transaction. This is the primary reason to have a **`copy()`** method defined within the transaction. Failure to make a local copy of the transaction can have adverse effects on other components that reference the common transaction. Modifying the fields of an **`analysis_port`** broadcast transaction should never be done.

We demonstrated in Section 3.1.6 that most **uvm_analysis_port**(s) and **uvm_analysis_export**(s) are interchangeable. Even though they are interchangeable, use **uvm_analysis_port**(s) for component outputs that are forwarding a transaction to other ports on an anlysis path and use **uvm_analysis_export**(s) or **uvm_analysis_imp** for component inputs that are receiving a transaction from other ports on an anlysis path.

For scoreboard development, setting the FIFO **size** to **0** is generally recommended and the **uvm_tlm_analysis_fifo**, described in Section 8.3 already has a default **size** set to **0**,

If there are multiple **uvm_analysis_imp**(s) in a component, the user must define multiple uniquely named **uvm_analysis_imp_*SFX*** ports with corresponding **write_*SFX*()** methods. This is most easily accomplished by using the **`uvm_analysis_imp_decl(_*SFX*)** macros to create new **uvm_analysis_imp** port types with corresponding **write()** methods.

We mentioned that the **uvm_analysis_imp_*SFX*** <u>*handle*</u> names do not require the use of the ***SFX*** values, but we believe it is a good idea to use the same suffix names as part of the handle names to reduce coding confusion.

Finally, prove that your scoreboard analysis paths are working correctly. Once the UVM testbench appears to be working, break the DUT to see if the UVM testbench can catch the bug. This is a good correctness indicator for your UVM testbench.

One of our paper reviewers, Jeff Vance, mentioned another option that he has seen in practice that is worth considering. We described the use of the **uvm_subscriber** in Section 6. Another option is to declare multiple **uvm_subscriber** instances inside a component. I.e., the scoreboard declares an array of subscribers, and passes a handle that points to itself (the scoreboard) to each subscriber. Then each subscriber can have an extended **write()** method that pushes transactions to the scoreboard queues. The advantage being, all these port connection details are encapsulated and it is easy to add or remove connections by just adding/removing subscribers. But it requires defining an extended subscriber class. We thank Jeff for sharing this interesting technique. We did not have time to try this ourselves but it certainly appears to be a worthy technique.

12. Acknowledgements

We acknowledge our colleagues Jeff Vance, Kelly Larson, Don Mills, David Lee and Dan Chaplin for their reviews and suggested improvements to this paper and the presentation slides. Their contributions helped identify areas that needed additional explanation and modifications to the presentation slides and their ordering to make this difficult topic easier to understand. We are grateful for their insightful and generous contributions.

13. References

[1] Clifford E. Cummings, "OVM/UVM Scoreboards - Fundamental Architectures," SNUG (Synopsys Users Group) 2013 (Silicon Valley, CA). Also available at www.sunburst-design.com/papers

[2] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2012

[3] "Observer pattern," Wikipedia article. https://en.wikipedia.org/wiki/Observer_pattern

[4] "Universal Verification Methodology (UVM) 1.2 Class Reference," Accellera Systems Initiative Inc., 1370 Trancas Street #163, Napa, CA 94558, USA., June 2014

14. Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 36 years of ASIC, FPGA and system design experience and 26 years of SystemVerilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past 15 years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee from 1994-2012, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.

Email address: cliffc@sunburst-design.com

Heath Chambers is President of HMC Design Verification, Inc., a company that specializes in design and verification consulting and high tech training. Mr. Chambers is a consultant with 22 years of Verilog Experience 15 years of SystemVerilog experience, 18 years of consulting and verification lead experience for multiple projects and has been an instructor for Sunburst Design since the year 2000. Heath has 18 years of SystemVerilog, Verilog, synthesis and UVM Verification methodology training experience for Sunburst Design, Inc., and was previously a contract Specman Basic Training instructor for Verisity. Heath has ASIC and system verification, firmware, and self-test design experience and is capable of answering the very technical questions asked by experienced verification engineers.

Mr. Chambers, was a member of the IEEE 1364 Verilog and IEEE 1800 SystemVerilog Standards Groups from 2000 to 2012, and has helped to develop and improve Sunburst Design Verilog, SystemVerilog, UVM and synthesis training courses.

Mr. Chambers specializes in verification of ASICs and systems using top-down design methodologies and is proficient in SystemVerilog, Verilog, UVM, 'e', C, and Perl. Mr. Chambers specializes in the Questa, Cadence, Synopsys simulation tools.

Before becoming an independent Consultant, Mr. Chambers worked for Hewlett-Packard doing verification of multi-million gate ASICs and systems containing multiple chips. Mr. Chambers was the lead verification engineer for the last two projects he worked on before leaving the company.

Mr. Chambers holds a BSCS from New Mexico Institute of Mining and Technology.

Email address: hmcdvi@msn.com

Last Updated: October 2018