Synchronicity: Bringing Order to SystemVerilog/UVM Synchronizing Chaos

Bryan Morris, P. Eng.
Ciena Corporation
Ottawa, ON, Canada

***Abstract*-SystemVerilog and UVM provide several mechanisms to synchronize modules and verification components. Most of the native SystemVerilog synchronization constructs (fork-join, events, semaphores, mailboxes) are well understood, but sometimes used incorrectly or inefficiently. UVM introduces similar and more powerful classes for synchronization. However, being more complex, they are not fully understood or only a fraction of their capabilities are used.**

This paper provides a detailed and clear explanation of all the synchronization mechanisms available to verification engineers for both native SystemVerilog and then the corresponding UVM classes. In addition, we provide concrete examples of how to use the synchronization mechanism effectively along with guidelines that we've developed for how to choose the best mechanism for your requirements.

This paper explores the following SystemVerilog (LRM 1800-2017) constructs:

- **fork-join and its variants**
- **named events**
- **semaphores**
- **mailboxes (and parameterized mailboxes).**

In addition, we explore the following UVM (1.2) classes:

- **uvm_event and uvm_event_callback**
- **uvm_barrier**
- **uvm_objection**
- **uvm_subscriber**
- **uvm_heartbeat**
- **uvm_callback**
- **TLM1 FIFO and analysis ports.**

I. Introduction

This paper explores the SystemVerilog and UVM constructs that permit two or more dependent processes to synchronize. The various mechanisms discussed here are intended for use in your verification environment. Which structures you use and what behaviors you enable depend on what you are trying to model.

In the first section this paper explores the synchronization constructs that are part of the SystemVerilog language. The second section describes the synchronization constructs in the UVM 1.2 Class library. Where necessary we provide concrete examples to clarify where the constructs could be used, the use-model and some code examples of how to use each construct.

Synchronization Constructs

In this section, we describe the synchronization constructs supported by SystemVerilog (as per the 1800-2017 LRM [1]), and those provided in the UVM 1.2 class library [2].

For each construct we provide three sub-sections:

- **Overview**: provides a brief explanation of the synchronizing mechanism.

- **Use Model:** Describes how to use the synchronizing mechanism in the context of developing a verification environment using either a SystemVerilog or UVM-based environment. Providing a concrete example of using the construct where necessary.

  It also cites any previously published papers related to the construct to provide opportunities for the reader to explore the topic further, or to highlight novel uses of the construct.

- **Guideline(s)**: Provides a set of recommendations of how to effectively use the construct.

Native SystemVerilog Synchronizing Constructs

This section *briefly*[1] outlines the basic synchronization mechanisms using constructs that are available as part of the SystemVerilog language. Currently, SystemVerilog provides the following constructs:

Synchronization Construct	Description
`fork-join`	Ability to launch N parallel processes, and optionally wait for one or all to process to complete.
`named events`	A broadcast synchronizing mechanism where 0:N consumers are waiting for an *interesting* condition to occur.
`semaphore`	Useful for modelling a shared resource that has a finite capacity (e.g., memory, shared communications port, processor operation queues, etc.). A semaphore helps you track the resource's capacity.
`mailboxes`	Like a real-world mailbox, it allows one process to send data to another process *and* synchronize with it. This models a Producer-Consumer relationship -- where the Consumer must wait for something from the Producer before it does its processing.

---

fork - join: Process Control

*Adder's **fork** and blind-worm's sting,*
Lizard's leg and owlet's wing,
- *Macbeth (Act I, Sc 1)*

Overview

The `fork-join`, `fork-join_any` and `fork-join_none` [LRM 9.3.2 [1]] allow a process to launch separate processes and then wait for every process (`fork-join`), at least one (`fork-join_any`), or none (`fork-join_none`) of the launched processes to complete before continuing.

Use Model

- `fork-join`: Launches N separate processes. Any statement after the *join* is blocked from continuing until ALL N processes are complete.

  As shown in Figure 1- fork-join below, the Master process launches three Slave processes. Only when the

[1] The assumption is that these constructs are familiar to most readers.

last Slave process (`Slave2`) completes, does the Master process continue.

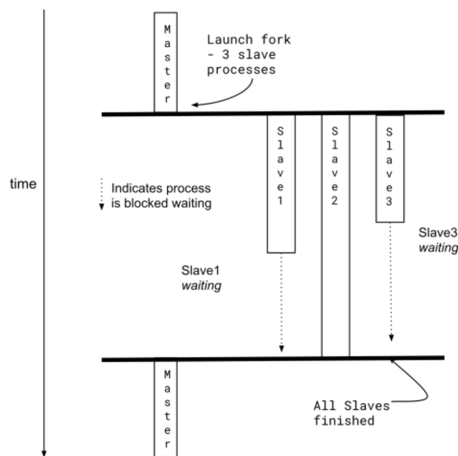

Figure 1- fork-join

- `fork-join_any`: Launches N separate processes. Any statement after the `join_any` is blocked from continuing until AT LEAST ONE of the N processes completes.

  As shown in Figure 2 - fork-join_any below the Master process launches three Slave Processes and the Master continues once the first Slave process completes (`Slave3`). As shown in the diagram, `Slave1` and `Slave2` processes continue. If you do not want them to the continue at the join, use `disable fork` (discussed later).

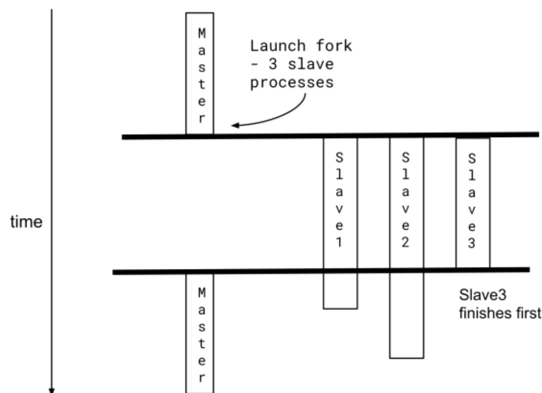

Figure 2 - fork-join_any

- `fork-join_none`: Launches N separate processes and then immediately continues with the next statement after the `join_none` -- it does not wait for launched any process to complete.

  As shown in Figure 3 - fork-join_none below, the Master process launches three independent Slave processes, and the Master continues processing. The three Slave processes complete their processing (but are usually launched with a `forever` loop).

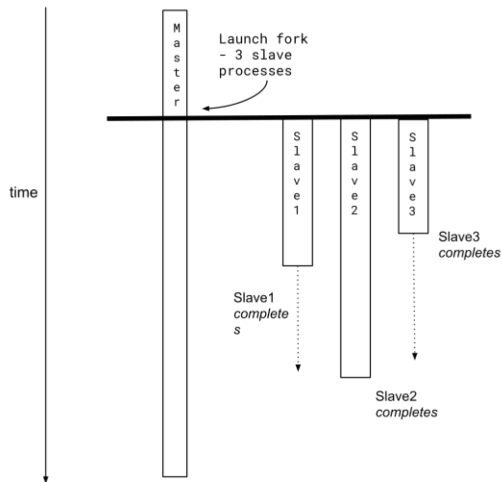

Figure 3 - fork-join_none

Guideline

The `fork-join` and its variants are required when you need to launch a new task/function in a separate process, and then synchronize when they must complete.

The use of `fork-join` is useful throughout a typical verification environment; regardless whether it's SystemVerilog or not. The main concern around processes is that they are sometime difficult to debug. The recommendations here are principally for debug assist.

- always provide a meaningful, unique name to the `fork` and to any `begin-end` block that encapsulates a process in a `fork-join` e.g., in Code Snippet 2 below, the launching `fork` is called `PKT_GEN_START_SEQS_FORK`. This allows the `fork` to be uniquely identified in your hierarchy.

  By naming a `begin-end` block[2] you create a new hierarchical scope – which is useful when you are debugging your processes either interactively in a debugger, or on the waveform as the scope is shown.
  - To ensure uniqueness, it is recommended that the `fork-join` follow the format:

    ```
    <enclosing class/task name> + <meaningful name> + _FORK
    ```

    e.g., `XBUS_MONITOR_WAIT_FOR_PKT_FORK` found in the class `xbus_monitor`.

    All letters are capitalized. Follow a similar pattern for each process launched, except remove the `_FORK` suffix e.g., `XBUS_MONITOR_WAIT_FOR_PKT`

---

[2] If your `begin-end` block is long or there are many nested levels, you can supply the same name on the `end` statement as you did with the `begin` statement. This helps reading the code to understand which scope you're in. However, if your `begin-end` is so long you have to use this aid then you should consider refactoring the code.

- Where appropriate, add a watchdog process for every `fork-join` and `fork-join_any`. Use the same name as the `fork` but use `_WATCHDOG` suffix to indicate which `fork` it is protecting e.g., `XBUS_MONITOR_WAIT_FOR_PKT_WATCHDOG` is protecting the `XBUS_MONITOR_WAIT_FOR_PKT_FORK`.

- Create a `task` for each process that you are starting in the `fork-join` block. And ensure that it is declared as an `automatic`. By default, a `task` is a `static` element, by declaring it `automatic` allows creation of one call stack per call.

```
1    task automatic do_something();
2       … something interesting here
3    endtask : do_something
4
5    task launch();
6       fork : LAUNCH_IT_FORK
7          begin : LAUNCH_IT_DO_SOMETHING
8             do_something();
9          end
10         begin : LAUNCH_IT_WATCHDOG
11            … do watchdog
12         end
13      join
14    endtask : launch
```

Code Snippet 2 - fork-join : Automatic tasks

Stopping Processes launched in a fork-join_any

When using `fork-join_any`, it is often necessary to stop all the other processes still running. For example, assume you're launching three processes using a `fork-join_any` i.e., a stimulus generating process, a protocol checking process, and a watchdog process. When any one of those three complete, you can stop the remaining two using the `disable fork` command [LRM 9.6.3, [1]] see line 14 in the Code Snippet 3 below). There are many other sources where more information can be found on this topic: [3] includes a great idea to use macros to simplify/clarify this issue), [4] provides a great of slides that provide excellent details on this entire `fork-join` topic.

If you place the `disable fork` immediately after the `join_any` statement, then any process started in the `fork-join_any` will be stopped. As recommended above, name the `fork` and each block in your `fork-join`. When you need to create nested `fork`s you can use this name in the `disable fork` command to *explicitly* identify which fork you want to `disable`.

It is also recommended that you wrap the `fork-join_any` inside another `fork-join`. (see lines 2 and 15 in Code Snippet 3). The outside `fork-join` ensures the `disable fork` only applies to the `fork-join_any` scope.

```
1    task launch();
2       fork
3          fork : LAUNCH_MONITORS
4             begin : LAUNCH_MONITORS_FIRST
5                monitor1();
6             end
7             begin : LAUNCH_MONITORS_SECOND
8                monitor2();
9             end
10            begin : LAUNCH_MONITORS_WATCHDOG
11                … do watchdog
12            end
13         join_any
14         disable fork;
15      join
16    endtask : launch
```

Code Snippet 3- fork-join_any with disable

named event: Broadcast Synchronization

> *Mark his condition and the event; then tell me*
> *If this might be a brother.*
> - *Tempest (Act I, Sc. 2)*

Overview

The *named event* is the simplest synchronization mechanism available using SystemVerilog [LRM 15.5, [1]].

The named event creates an M:N Producer-Consumer relationship (or a Master-Slave). The Producer(s) are monitoring the system waiting for a set of conditions to occur and then triggering the event; the Consumers are waiting for the event. The Producers and Consumers are independent of each other and run in separate processes. The Consumer can choose to block their current process waiting for the event to be triggered, or it can poll for the event level (i.e., ON or OFF): waking up at regular intervals to see if the event was triggered sometime in the last polling interval. The Producer's process can be blocked waiting or use polling for the set of conditions to be met. When it detects that all conditions are met, it triggers the event at which time all Consumers that are blocked waiting are unblocked and continue their process at the *same* simulation time. Where Consumer is polling for the event, its next polling sees the interrupt *level* is ON.

As shown in Figure 4 - named event below, three Slave processes all come to a point in their processing where they must wait for the `event` – and they block waiting. The Master process triggers the event[3], and all three Slave processes continue from that simulation time.

[3] But it can be any process that has a handle to the `event`.

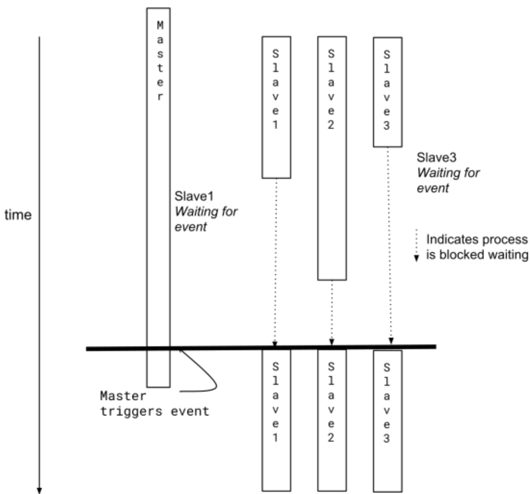

Figure 4 - named event

There can be from 1 to M Producers that can trigger the event, and zero to N Consumers (or 1:N Master-Slaves) waiting for the event to be triggered. That is, if an event is triggered, it is possible there may be no Consumers currently blocked waiting for it.

Guidelines

- Do you need to broadcast a significant phase in your simulation to multiple Consumers ("listeners")?
  - For example: when your simulation has completed reset; has been fully configured; is ready to send traffic; is injecting a specific error, etc.
- Use a named `event` only when you are creating a SystemVerilog-only verification environment.
- Use a `uvm_event` instead of a named `event` in a UVM environment. See the section on `uvm_event` described later in this paper.

semaphores: Modelling a Finite Shared Resource

> *This* ***token*** *serveth for a flag of truce,*
> - *Henry IV, Part 1 (Act III, Sc 1)*

Overview

A `semaphore` is bucket that holds a set of tokens [LRM 15.3, [1]]. When a `semaphore` is created, the bucket is empty. Any process can add tokens to or remove tokens from the bucket. When the bucket is empty of tokens and a process attempts to get a token, it is blocked until a token is returned to the bucket.

The `semaphore` is used to model a finite shared resource. Each token in the `semaphore`'s bucket represents a single element that is shared. Typical examples of a shared resource are number of time slots in an arbiter, the number of items in a shared FIFO.

The `semaphore` also provides a non-blocking mechanism to its processing. In this model, the task *tries* to get their required number of tokens and regardless of whether the tokens are not available the task returns.

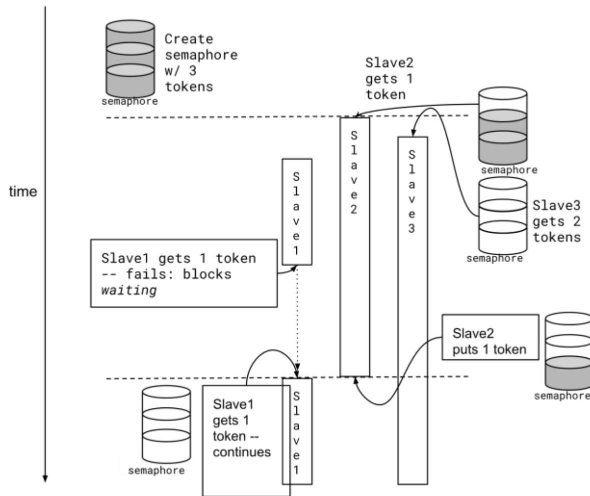

Figure 5 - semaphore

Guideline

- Before using a `semaphore` in a UVM environment, consider using a UVM `tlm_fifo` instead. It provides the same behaviour along with some additional functionality.
- If you are creating a non-UVM environment it is recommended that you create a wrapper class around the `semaphore` that can provide the normal semaphore capabilities along with the following:
  - Keep track and be able to query the number of available tokens;
  - Consider adding statistics for duration (max, min, average, variance) a token is used – useful for checking performance characteristics of the shared resource you are modelling.
  - trace logging identifying who and when a token is taken or returned
- Create a `semaphore` in your environment add it into the `uvm_config_db`. Give the semaphore a unique meaningful name in the `uvm_config_db`. It is recommended that you suffix the name with `_sem` to further clarify its purpose.
- Always initialize the `semaphore` with a number of tokens. It is recommended that the number of tokens be configurable via constraints and/or the command line.
- Add the `_bl` to the task name that attempts to get the tokens to indicate it is a blocking task.

mailbox & parameterized mailboxes: Producer - Consumer Relationship

> *No enigma, no riddle, no l'envoy; no salve in the* ***mail****, sir: O, sir, plantain, a plain plantain! no l'envoy, no l'envoy; no salve, sir, but a plantain!*
>
> - *Love's Labour's Lost (Act III, Sc 1)*

Overview

SystemVerilog `mailbox` construct provides a higher-level abstraction compared to the `semaphore` and the `event`. Similar to a real mailbox, one or messages can be pushed into the `mailbox` and the receiving process can pull from the `mailbox` using the information contained in the message for its processing. One limitation of the `mailbox` is that it provides a FIFO ordering -- where the receiving process always pulls the messages from the `mailbox` in the order they were pushed into the `mailbox`. The maximum number of messages the `mailbox` can hold can be specified, but by default is only limited by memory.

The `mailbox` is the easiest way to establish a Producer-Consumer relationship between two processes e.g., a packet generator (the Producer) sending packets to a driver (the Consumer) for pushing out of the DUT. One behavioural difference between the `mailbox` and the `semaphore` is that the Producer can only push *one*

message into the mailbox at a time, and the Consumer can only pull one message out (compared to the semaphore where either side can request and add/remove one or more tokens). Another difference is while the semaphore provides a simple counter, the mailbox provides a message that holds some meaningful data e.g., a packet, statistics, etc.

A mailbox provides either a blocking, or non-blocking synchronization to either side. That is, you can choose whether the Producer, the Consumer, or both are synchronized via a blocking or polling fashion.

A mailbox's behaviour is such that:

- *when a mailbox is full*: the Producer can choose to block its processing until space is made in the mailbox, or it can continue processing and come back later to see if there's room (polling).
- *when a mailbox is empty*: the Consumer chooses to block until a message arrives or polls the mailbox later to see if a message has arrived in the interim. It resumes its processing when a message is retrieved from the mailbox.

It is also important to note that the mailbox can be configured into a M:N relationship (where M,N > 1). That is, you can have:

- multiple Producers feeding one Consumer,
- a single Producer feeding multiple Consumers, or
- multiple Producers feeding multiple Consumers.

Guideline:

1. Only use a mailbox for non-UVM verification environments.
2. For UVM environments, always use TLM constructs e.g., analysis ports, or TLM FIFOs.
3. The UVM environment is responsible for creating the mailbox. These are stored in the uvm_config_db if using UVM, and it is recommended for non-UVM environments create a similar "resource db" class that holds a pool of all the mailboxes (and semaphores and events) in an associative array, keyed by the mailbox's unique name. Create this "resource db" as a global, singleton object. See Code Snippet 7 - Non-UVM resource db below.

```
1  class global_resource_db;
2    static local bit is_singleton = 0;
3    local event     event_pool[string];
4    local mailbox   mailbox_pool[string];
5    local semaphore semaphore_pool[string];
6    // Add appropriate get and set methods to access the pools
7    function event get_event( input string ev_name );
8    function void set_event( input string ev_name, input event ev );
9  endclass : global_resource_db
```

Code Snippet 7 - Non-UVM resource db

4. The main processing loop for the Consumer should fork off a new task that waits for the incoming messages and then dispatches the messages to a separate processing task or function.
    - When creating a blocking-style synchronization, the Consumer's waiting task should be guarded by a watchdog that ensures that Consumer gets a message within a reasonable duration.
    - It is also advisable to have an enable flag guarding the processing each message received.
5. Name an instance of a mailbox to indicate the expected processing for that mailbox i.e.,
    - the mailbox should provide a meaningful name indicating what type of messages it holds e.g., packets, frames, error injection types, etc.
    - add _bl suffix to indicate it is a blocking mailbox e.g., pkt_bl_mbox holding pkts
    - add _poll suffix to indicate it is a polling mailbox e.g., pkt_poll_mbox
    - add _mbox suffix to indicate it is a mailbox

6. If you're creating a bounded `mailbox`, ensure that the size of the queue is configurable i.e., through constraints, command-line or parameter.

UVM Synchronizing Constructs

In this section we explore the synchronization mechanisms provided by the UVM 1.2 library. The UVM library provides classes with (at least) similar behaviour to their equivalent native SystemVerilog. If you're building a UVM-based verification environment, you will benefit from using these UVM synchronization classes instead of their native SystemVerilog equivalent. They provide additional functionality that make them the superior choice.

The constructs we'll discuss are:

Synchronization Construct	Description	…could replace SV…[4]
uvm_callback	Allows you to add a set of functions that are run when certain conditions apply. For example, the `uvm_event_callback` derivation allows functionality to run before and after the event is triggered.	`event` and/or `mailbox`
uvm_event	Similar to the SystemVerilog `event`... but better.	`event`
uvm_objection	Allows multiple processes to contribute to a decision about what to do next.	`semaphore` or `mailbox`
uvm_subscriber	One process is *listening* to another process. When activated it executes its processing.	`event` and/or `mailbox`
uvm_barrier	Synchronizes multiple processes to wait until all processes are at the same stage of processing.	`semaphore` or `mailbox`
uvm_heartbeat	Creates a Parent-Child relationship where the Parent monitors its Child processes for activity i.e., is it *still* alive?	`event` or `mailbox`
tlm_fifo	Transaction Level Modelling (TLM) constructs for a FIFO queuing mechanism	`mailbox`
Analysis port	Transaction Level Modelling (TLM) construct (providing) and analysis ports (providing a dynamic message broadcast mechanism).	`mailbox`

uvm_callback: Dynamic Processing

> *What says the fellow there? **Call** the clotpoll **back.***
> - *King Lear (Act I, Sc 4)*

Overview

The intent of a `uvm_callback` is to dynamically add new processing to an object. That is, once you've setup your class to accommodate callbacks, your environment can choose what functionality is done when the callbacks are started. You can also choose in what order the processing is done. This type of processing creates a Chain of Responsibility pattern: where a series of functions are called in the order defined.

While there is no direct synchronization aspect with a callback they are used indirectly in the other UVM synchronization classes (i.e., with `uvm_event` and `uvm_objection`). There are many uses for callbacks, with some typical verification examples of adding errors into a sequence item before delivering it to the driver, pre-filtering an event to decide whether it should be generated, or for collecting coverage.

---

[4] This column illustrates that the UVM class could replace functionality provided by the native SystemVerilog constructs. When multiple constructs are added, the use of the SV constructs is dependent on *what* you're modelling i.e., for the `uvm_subscriber` functionality you would need *at least* an `event`, but you may also need a `mailbox` to pass information to the "listener" depending on what your model requires.

A process can have one or more callbacks at any one time.

As illustrated in Figure 6 - uvm_callback Processing below, a process has N `uvm_callback` objects attached to it. When process calls the `` `uvm_do_callbacks `` macro (specifying which method in the `uvm_callback` to call), each attached `uvm_callback` has its method called. The `uvm_callbacks` are executed in the same order that they were added. When all N `uvm_callbacks` have completed their execution, the process continues from that point.

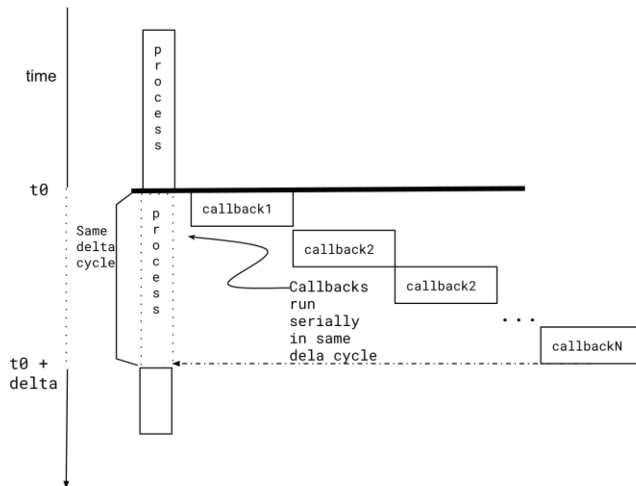

Figure 6 - uvm_callback Processing

Use Model

The use-model of the `uvm_callback` is straightforward – we'll use the example of collecting coverage with callbacks to illustrate. A user derives a new abstract base class for all your callbacks from the `uvm_callback` class (e.g., create a callback to sample coverage data) and then adds at least one *meaningful* `virtual` function that performs the processing. In our example, we create a `sample()` function that accepts a coverage group.

```
class agent_cov_cb extends uvm_callback;
   `uvm_object_utils(agent_cov_cb)
   covergroup agent_cfg_cg with function sample( agent_config agent_cfg );
      … add coverpoints
   endgroup : agent_cfg_cg
endclass : project_cov_cb
```

For every unique processing required, derive your concrete callback from the abstract base class, and add the functionality you need e.g., create the `sample` method to capture the coverage data.

```
class rx_agent_cov_cb extends project_cov_cb;
  `uvm_object_utils(rx_frame_cov_cb)
   covergroup rx_agent_cg with function sample(rx_agent_cg);
      … add coverpoints
   endgroup : rx_agent_cg
endclass : rx_frame_coverage_cb
```

If there are multiple unique processing steps, derive a new callback for each step and add its functionality into the callback. Then register the callback into the class that will call the set of callback(s) at the appropriate time.

Guideline

- Use the suffix "`_cb`" to indicate the class is a callback.
- Use callbacks to extend functionality dynamically. Using the "Single Responsibility" design principle, ensure that each callback does one thing.

uvm_event + meta-data, uvm_event_callback: Broadcast Synchronization

*I'll after him, and see the **event** of this.*
- *Taming of the Shrew (Act III, Sc. 2)*

Overview

The `uvm_event` class provides the same functionality and intent as the SystemVerilog `event`. Its purpose is to allow one or more processes to trigger an event when specific conditions are met. Other processes can be waiting for the event to be triggered. In addition, the `uvm_event` can add meta-data that travels along with the event. This is useful for when you need to broadcast information about the event to the receiving processes e.g., notifying all components that the DUT is out of reset.

The `uvm_event` can also be used in a level-sensitive way instead of a transient trigger `event` i.e., the `uvm_event`'s state is either ON or OFF. The waiting process can either poll the current level with the `is_on()` or `is_off()` functions; or can block waiting for the `uvm_event` to be `ON` or `OFF` using the `wait_on()` and `wait_off()`, respectively. As well, the `uvm_event`'s level can be reset (`OFF`) by anyone with a handle to the `uvm_event`, and the waiting process can indicate it's no longer waiting for the event with the `cancel()` function.

Use-Model

The use-model for the `uvm_event` is similar to the SystemVerilog event. As shown in Figure 7- uvm_event Before trigger called, the three Consumers call the `wait_trigger()` (or better yet the `wait_ptrigger()`) task to block until the event is triggered.

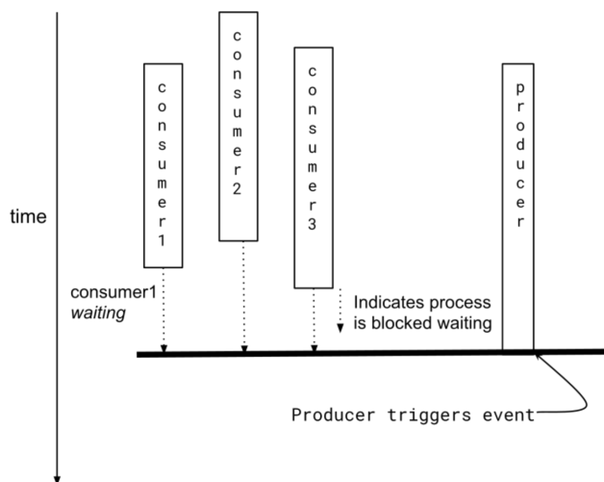

Figure 7- uvm_event Before trigger called

The Producer process uses the `trigger()` task when the required conditions are met. In Figure 8 - uvm_event After Trigger all three Consumer processes unblock and continue.

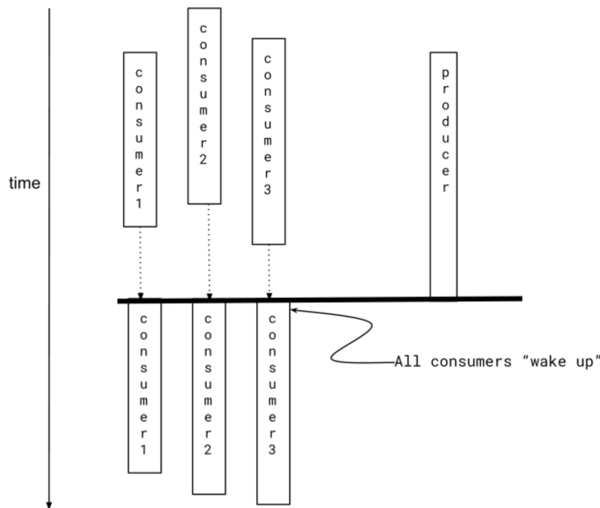

Figure 8 - uvm_event After Trigger

The triggering mechanism is not complicated: it's binary -- either it's triggered or not. The event triggering is done by calling the `trigger()` task. However, the *waiting* process can either use `wait_trigger()` or `wait_ptrigger()` tasks. Both tasks unblock the waiting process during the same simulation time step as when the `trigger()` task was called. However, the `wait_ptrigger()` provides an additional defensive feature. It is possible that the wait* tasks could be called in the same simulation time step as when the `trigger()` is executed – resulting in a race condition where the waiting process *missed* the event trigger. Using the `wait_ptrigger()` ensures that the waiting process unblocks regardless of scheduling order of the wait* or trigger() tasks. If your modelling ensures that this cannot happen then `wait_trigger()` is safe to use. However, for the sake of a minimal amount of extra checking it is recommended to always use `wait_ptrigger()`.

The `uvm_event` is a parameterized class that accepts any valid type. A variable of this type is created with the event and *travels* along with the event as meta-data. This meta-data or "trigger" data is useful to hold information about the event that can be updated and a handle to the meta-data is passed as a parameter to the `trigger()` task. You can get a handle to the trigger data by calling the `uvm_event`'s `get_trigger_data()` function.

The related `uvm_event_callback` class adds some useful functionality to the `uvm_event`. It executes functionality before *and* after the `uvm_event` triggers via a set of callback classes (as discussed in previous section). Each callback class provides the ability to pre-filter or post-filter the event. These callback classes are attached to the `uvm_event` with the `add_callback()`, and removed with the `delete_callback()`. When the `uvm_event` is triggered there are two functions[5] called in the callback class: `pre_trigger()` and `post_trigger()`. As their name implies, the `pre_trigger()` is called *before* the `uvm_event` triggers, and the `post_trigger()` is called *after* the `uvm_event` is triggered.

The `pre_trigger()` is a pre-filter for the event, it returns a `bit` value:

- When `pre_trigger()` returns a 0, the event is triggered and `post_trigger()` is called.
- When the `pre_trigger()` returns a `1,` the `uvm_event` is NOT triggered. This could be useful when a Consumer wants to dynamically filter an event e.g., if the event signals the injection of a fault this can be disabled dynamically if required.

---

[5] These are `functions` and so cannot consume simulation time.

Similarly, the `post_trigger` callbacks are useful for post-filtering of the event generally using the trigger data attached to the event. Or it can be used to capture performance statistics related to the stream of events e.g., time between each event. There is no requirement to implement either the `pre_trigger()` or `post_trigger()` functions, these are `virtual` functions that are empty in the base `uvm_event_callback` and so do nothing when called.

The callback objects are called in the same order that they were registered, and both the `pre_trigger()` and `post_trigger()` functions are called in the same simulation delta-cycle as the waiting processes i.e., the simulation clock does NOT advance.

Figure 9 - uvm_event_callback below, illustrates the processing of a `uvm_event`:

@1. `Producer` calls the `trigger()` method

@2. `uvm_event` calls the `pre_trigger()` function for all registered callbacks defined (as shown only `Consumer1` and `Consumer3` have the callbacks with `pre_trigger()` defined). The callbacks are called in the order they were added.
→ Simulation time does NOT advance
If ***any*** of the `pre_trigger` functions return a 1 the underlying event is NOT triggered i.e., filtered. All blocked processes continue to wait.

@3. If the event is not filtered, the underlying SystemVerilog `event` is triggered.
→ Simulation time does NOT advance.

@4. The `post_trigger` functions of all registered callbacks are called in the same simulation delta-cycle. These are called in the same order as the callbacks were registered.
→ Simulation time does NOT advance

@5. Consumer processes are unblocked, perform any functionality in the current simulation delta-cycle and then continue
→ Simulation time advances to the next scheduled event.

Figure 9 - uvm_event_callback

Guideline

- Always use the `wait_ptrigger` task instead of the `wait_trigger`. The main difference between the two is that `wait_ptrigger` checks to see if the event has already been triggered during the current simulation delta-cycle and returns immediately if it does. Using `wait_ptrigger` avoids race conditions where the event is triggered by one process and waited on by another during the same simulation delta-cycle. This can be sometimes a difficult debug task since the scheduled order of the triggering process and waiting process is non-deterministic -- in one simulation the wait might happen before the trigger and everything operates as expected; and in another simulation the reverse happens, and the waiting process misses the event.
- When adding callbacks you can control the order they are called by using the `append` bit you supply to the `add_callback()` function. By default, every callback is added to the end of the callback list; setting `append==0` adds the callback to the front of the list.

  However, the callbacks are stored in a `protected` SystemVerilog queue, so if you do need control over the order the callbacks are stored in the queue (which defines the order they are called), then you'll need add this capability into a derived `uvm_event` class.

- Suffix a `uvm_event` with `_ev` and a `uvm_event_callback` with `_ev_cb` to indicate their purpose.
- By default, a *copy* of meta-data (passed to the `uvm_event` in the `trigger()` task) is sent to the `pre_trigger()` and `post_trigger()` functions. Usually getting a copy of the meta-data is sufficient.

uvm_barrier: Multiple Processes in Lock-Step

...and turn the dregs of it upon this varlet here,—this, who, like a block, hath denied my access to thee.

- *Corialanus (Act V, Sc. 2)*

Overview

The `uvm_barrier` ensures a set of processes run in lock-step in a Peer-to-Peer relationship. This relationship establishes a threshold for the number of processes using the barrier. Each process in the set must do its processing and then blocks waiting for ALL the other processes in the lock-step set to also wait on the barrier.

Similar to a semaphore, a barrier is created with a threshold defining the number of processes that are expected to use the barrier. Any process (with a handle to the `uvm_barrier` object) can wait on the barrier. When the last process in the set completes its processing and indicates it is waiting (making the number of waiting processes equal to defined threshold), then ALL the other processes in the lock-step set are unblocked and continue processing.

As shown in Figure 10 - uvm_barrier Waiting for Threshold below, we have N processes waiting for the barrier. `Process2` is the last process to complete, when it calls the `uvm_barrier`'s `wait_for()` the threshold (==N) has been achieved.

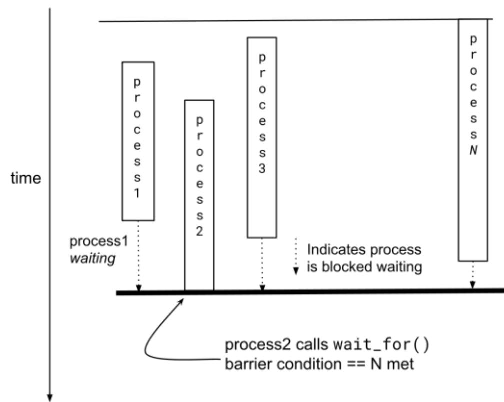

Figure 10 - uvm_barrier Waiting for Threshold

When the threshold is hit, all waiting processes (including `Process2`) continue from the same simulation time as shown in Figure 11 - uvm_barrier After Threshold below.

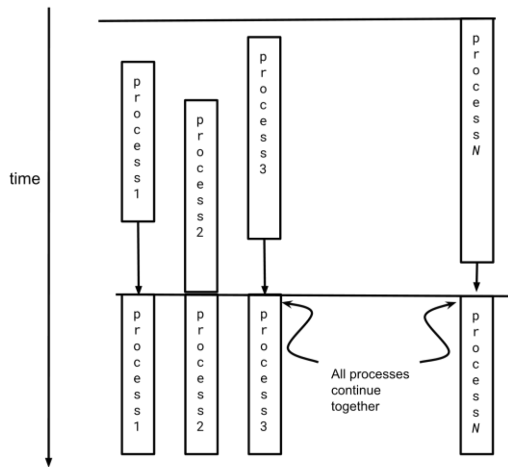

Figure 11 - uvm_barrier After Threshold

Use Model

When creating an instance of the `uvm_barrier`, you can specify the number of processes involved in the lock-step set as an initial threshold setting to the barrier. Each time a process calls the barrier's `wait_for()` function the number of waiting processes increments by one. Once the number of waiting process equals the current threshold, all processes in the set are unblocked. You can change the threshold with the `set_threshold()` function, and the corresponding `get_threshold()`.

Processes can take themselves off the waiting list by calling the barrier's `cancel()` function – this decrements the current count of waiting processes. Consider carefully how you want to model this as it could cause a dead-lock situation where you cancelled one process without decrementing the threshold and all the waiting processes could be left waiting indefinitely. Unless, or course, that's exactly the behavior you're trying to model!

An interesting behavior of the `uvm_barrier` is the ability to automatically reset the threshold once all processes are unblocked. This allows the processes that are part of the lock-step set to coordinate their processing in continuous lock-step flow i.e., all processes wait for the others; they are all released at the same time; the barrier is reset; all processes complete their processing at different rates and wait for the all other processes to get to the same point. Rinse and repeat. The auto-reset is on by default. To change this behavior use the `set_auto_reset()` function.

An example of this use of the barrier is if you were modelling a multi-stage arbiter where each stage can take a different amount of time to select. Each arbiter stage (Producer) processes its data and contributes its selection to then next stage arbiter (Consumer) -- who takes input from multiple arbiters. The Consumer arbiter must wait for all Producer arbiters before making the final selection; and each Producer arbiter cannot take in its next data until the Consumer arbiter has pulled from it. The auto-reset feature would then automatically setup for the next arbitration stage.

A great paper on the `uvm_barrier` class is Mark Glasser's "Shutdown Agreements in a UVM Testbench" [2] demonstrating the use `uvm_barrier` to coordinate the verification components that contribute to the decision to stop the test.

Guideline

- Always provide a positive, non-zero threshold when creating an instance of a `uvm_barrier` object. Alternatively, have each component that uses the barrier be responsible for incrementing the `uvm_barrier`'s threshold using the function `set_threshold()` == `get_threshold() + 1`.

  NOTE: If you leave the threshold as the default (0), the first call to `wait_for()` task returns immediately.

- Consider deriving from the `uvm_barrier` in your company/project-layer class e.g., `project_barrier` class which provides an `increment_threshold()` and `decrement_threshold()` (which could be an alias for the `cancel()` function).

  In addition, it has been found useful to create a barrier to provide a simpler phasing mechanism. When the barrier indicates a phase or a step in the processing e.g., frame pattern found across multiple channels, error injection start/stop, adding an `uvm_event` to the barrier can be triggered when that phase has been reached.

  As well, like the `+UVM_OBJECTION_TRACE` that logs when an objection is raised and lowered, a company/project-layer derived class can add an `+UVM_BARRIER_TRACE` to provide debug logging.

- Add the `uvm_barrier` object into the `uvm_config_db` so that any process can access it. Provide it with a unique name.
- Use the suffix `_barrier` for any `uvm_barrier` instance to indicate its purpose.
- Use a watchdog pattern when calling the `wait_for()`. This allows you to `cancel()` the wait if another condition applies e.g.,

```
1  fork
2     fork : INTERESTING_BARRIER_FORK
3        begin : INTERESTING_BARRIER_WAIT
4           my_barrier.wait_for();
5        end
6        begin : INTERESTING_BARRIER_WATCHDOG
7           my_barrier_watchdog_ev.wait_ptrigger();  // event triggered from other
8           my_barrier.cancel();  // indicate we're out of the synchronizing set
9        end
10    join_any
11    disable fork;
12 join
```

Code Snippet 10 - uvm_barrier with watchdog

uvm_objection and *uvm_objection_callback: Multiple Components Contribute to Go-Forward Decision*

To bear with their perverse ***objection****s;*
Much less to take occasion from their mouths
To raise a mutiny betwixt yourselves:
Let me persuade you take a better course.
- *Henry IV, Part 1 (Act IV, Sc 1)*

Overview

Most readers are familiar with the use-model of the `uvm_objection` class as the `uvm_phase` class uses objections to ensure a sequential order of the UVM phases by *raising* and *dropping* objections. In a similar way you can use objections to synchronize when one or more processes all need to start and/or complete at the same time.

The conditions for synchronization when using a `uvm_barrier` is opposite to a `uvm_objection`. Where the `uvm_barrier` resumes all waiting processes when the number of processes increments to a threshold; the `uvm_objection` resumes the waiting processes when their number decrements to 0 i.e., all previously raised objections have been dropped. It is also different from the `uvm_barrier` in that it establishes a Master-Slave relationship.

As shown in Figure 12 - uvm_objection below, the Master process creates the objection (and should add it into the `uvm_config_db` for all Slaves to retrieve). Each Slave process can raise and drop an objection at any time; the Master process is blocked from proceeding until there are no raised objections that remain.

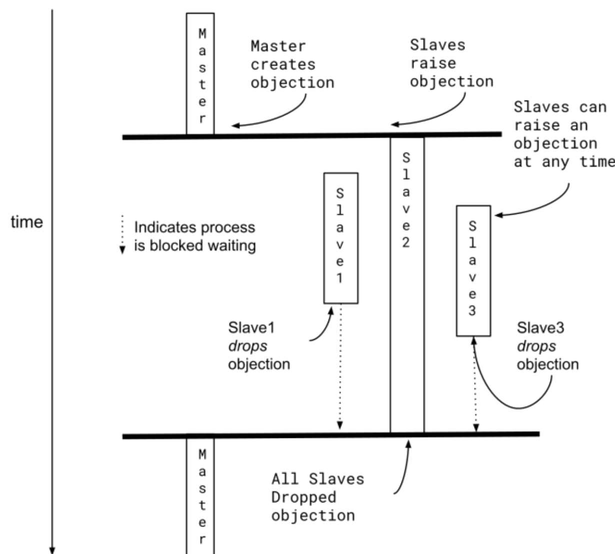

Figure 12 - uvm_objection

Use Model

The use-model of the `uvm_objection` is very simple. It creates a Master-Slave relationship where a Master component is responsible for controlling the synchronization point where all the Slaves at are at the same point in their processing. All the Slave processes communicate with the Master indicating they've started their processing by calling the `raise_objection()` method. When they are done, they call `drop_objection()`. The Master waits for all Slaves to drop their objections before it proceeds.

Objections also have built-in callback capabilities. This allows you to add processing when an objection is raised (by adding a `uvm_objection_callback` to the `uvm_objection` object and defining a `raised()` function), or

when an objection is dropped (implement the `uvm_objection_callback::dropped()` function). In addition, when ALL the objections have been dropped, the callbacks can use the `all_dropped()` function). NOTE: You can create ONE callback with any or all of the `raised()`, `dropped()` and `all_dropped()` methods defined. A useful technique for these callbacks is to record to the waveform when the objections are raised and dropped.

There are other useful utilities the `uvm_objection` provides:

- *Events attached to the objection:* You can wait for events that indicate when the individual, specific objections are raised and dropped using `wait_for()` blocking task. In addition, you can wait for an event indicating that ALL the objections are dropped. This is particularly useful for other components e.g., scoreboards, that need to know when a phase has been completed. You indicate which event you're waiting by supplying one of `UVM_RAISED`, `UVM_DROPPED`, or `UVM_ALL_DROPPED` to the `wait_for()` task.
- *Drain Time*: Sometimes it's useful to allow the components a little extra time after they have dropped their objection before the Master continues (e.g., to flush out data, all status bits to clear, etc.), every objection can `set_drain_time()` to specify the time to wait. As illustrated in Figure 13 -uvm_objection with drain time below. It is similar to the usual objection synchronization mechanism, except it delays the Master's processing after all objections are dropped by the specified drain time.

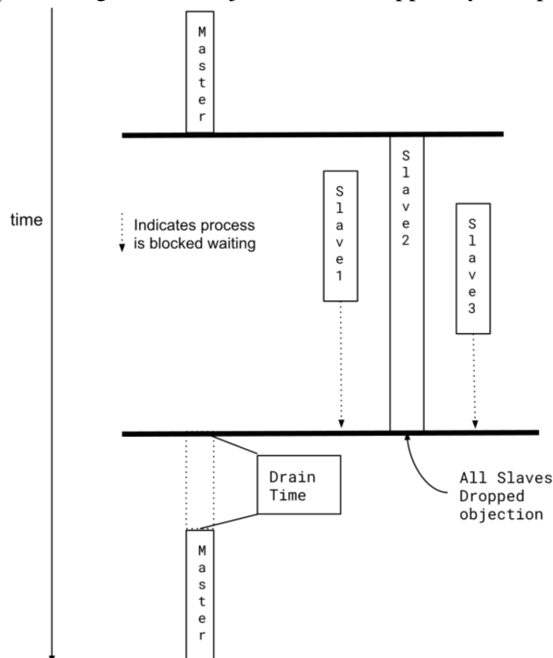

Figure 13 -uvm_objection with drain time

It is important to note that the `all_dropped()` callback, and the associated `ALL_DROPPED` event are called AFTER the drain time completes.

- *Reset objections*: If you need to cancel the objection, you can use the `clear()` method. Any process waiting for the objection is immediately unblocked. All status and counters are reset. This can be useful when the conditions for an objection are no longer relevant e.g., a reset or reconfiguration of the environment has occurred. If you're using the `wait_for()` task only the `UVM_ALL_DROPPED` event is triggered
- *Propagate mode*: Propagates the objection to hierarchy i.e. up to the parent of the component holding the objection. Use the `set_propagate_mode()` to enable and disable this capability. Note that there is a performance penalty in propagating objections depending on the depth of your hierarchy. This mechanism

is ON by default. If your model does not require this capability, it is recommend that you disable it upon creation

Guideline

- *Debugging*: Use the `+UVM_OBJECTION_TRACE` on the command line to log when *every* objection is raised and lowered. You can also query the current number of objections using the `get_objection_total()`, or dump current objections using the `display_objections()` or `print()` methods[6].
- If you extend the `uvm_component` class into a company/project layer, it is suggested that you automatically add the raise/drop objection handling for the UVM phases into this base class.
- Use the suffix `_objt` to any `uvm_objection` instance to indicate its purpose.
- The `drop_objection()` and `raise_objection()` methods have a `description` parameter that is an empty string by default. It is logged when the `+UVM_OBJECTION_TRACE` mode is enabled. *Always* provide a meaningful message in this `description` parameter to assist in debug.
- Disable the default objection propagation mechanism, if your model does not need it.

uvm_heartbeat: Parent-Child Relationship

*Ay, with all my **heart**, and thou art worthy of it.*
- *All's Well That Ends Well (Act II, Sc. 3)*

Overview

The intent of the `uvm_heartbeat` is to ensure that one process monitors a set of other processes to see if they are still alive. Typically, this creates a Parent-Child (or Master/Slave) relationship i.e., the parent wants to know if all its child processes are still running.

The `uvm_heartbeat` is an abstraction that works with a combination of the `uvm_event` and `uvm_objection`. As illustrated in Figure 14 - uvm_heartbeat below, the Parent process establishes a polling frequency when it checks to see if all its children are alive. Each Child process indicates it is alive by triggering the heartbeat event. If all the Child processes have NOT indicated they are alive during the polling interval, the Parent process can take appropriate action (e.g., log errors, start shutting down, etc.).

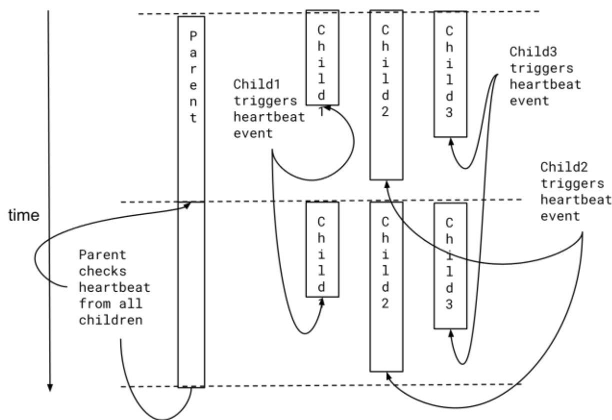

Figure 14 - uvm_heartbeat

[6] All debug tools provide additional support to make this debug even easier -- search the manual of your favorite debugger tool for UVM-centric debug capabilities.

Use Model

When creating a new `uvm_heartbeat` you can optionally add a `uvm_objection` object. Then you associate a set of Child `uvm_components` to monitor, and a `uvm_event` for triggering the heartbeat using the `set_heartbeat()` function. You can add an remove more `uvm_components` using the `add()` and `remove()` functions, respectively. You can change the `uvm_event` being used by using `stop()` to stop the monitoring and `start()` to restart monitoring but supplying it with a new (or the same) `uvm_event`.

You provide a set of `uvm_components` that need to be watched for activity within a specific window of time, and you must decide whether you want to stop under any of the following conditions:

- when exactly ONE of those components indicates they are alive, or
- when more than one, but *not* all the components report activity, or
- when ALL the components report activity within the time window.

Each of the monitored Children `uvm_components` should report activity by regularly triggering the `uvm_event` associated with the `uvm_heartbeat` (that is, the one you setup using the `set_heartbeat()` function). It is recommended that the triggering is done in a background forked process as shown in Code Snippet 11 - Child's Heartbeat Triggering below.

```
1   uvm_event child_hb_ev;
2   parent_heartbeat.set_heartbeat( child_hb_ev, … q of Child uvm_components )
3   fork : HEARTBEAT_FORK
4      begin : HEARTBEAT_EV_TRIGGER
5         forever begin
6            #(HEARTBEAT_TIMER_IN_NS * 1ns) child_hb_ev.trigger();
7         end
8      end
9    join_none
```

Code Snippet 11 - Child's Heartbeat Triggering

A good example of using the `uvm_heartbeat` is explained in [3]. This article uses the `uvm_heartbeat` to ensure that no components have locked-up during the simulation. If any fail to indicate that they are still alive, the simulation stops -- thus avoiding wasting simulation cycles on a simulation that will likely timeout eventually. This example uses the `uvm_heartbeat` to create a good defensive coding style.

Guideline

- All child processes involved in a `uvm_heartbeat` are derived from `uvm_component`. If you have a company or project layer component class e.g., `project_base_component` that derives from `uvm_component` add a `virtual start_heartbeat()` task to provide the heartbeat `uvm_event` to the parent.
- Use the suffix `_hb` for any `uvm_heartbeat` instance to indicate its purpose. Should also suffix the associated `uvm_event` with `_hb_ev` to further refine it.

uvm_subscriber: Listener

> ***Listen**, but speak not to't*
> *- Macbeth (Act IV, Sc. I)*

Overview

The uvm_subscriber creates a *push message* channel[7] from a Producer to zero or more Consumers. However, each Consumer is **not** another process, instead it is *functional* interface by deriving from a uvm_subscriber class and creating a write() function -- that executes in zero-time in the same simulation delta-cycle as the Producer.

This subscriber communication channel behaves *similarly* to a parameterized SV mailbox described above. The uvm_subscriber is a parameterized class, where a type of that class defines what message is sent from the Producer to the Consumers.

The uvm_subscriber uses the Transaction Level Modelling (TLM) analysis_ports (described later in this document for its messaging channel). However, for purposes of this discussion it is sufficient to describe the analysis_port as a "pipe" that accepts only one type of message from anywhere and results in calling a function to process that message in zero time. When the Producer sends the message to the Consumer, the Consumer receives the message *immediately* i.e., in the *same* simulation-time/delta-cycle as the Producer with no queuing. Once the Consumer processes the message -- without consuming time -- it returns processing to the Producer. If more than one Consumer is *subscribing*, then all Consumers receive a copy of the same message at the same time (i.e., simulation time does not advance) and must process it and return.

The uvm_subscriber shares many functional similarities to the uvm_callback and can (for most cases) be used interchangeably. For example, there are many papers describing using uvm_subscriber [4] or uvm_callback to sample coverage.

Use Model

Under the hood, the uvm_subscriber extends from a uvm_component and then adds a TLM analysis_port (i.e., the write() function) to provide the communication channel. Thus it provides all the usual UVM phasing and access to the uvm_config_db – the same as any uvm_component. Each Child process can extend from a uvm_subscriber, and then add itself to the uvm_config_db so that the Parent process can communicate with it.

Decide on the type of the message you need to transfer from the Producer to the Consumer. This information can be any SystemVerilog type i.e., bit, int, string, struct, class etc. In Code Snippet 12 - uvm_subscriber extension below, our subscriber's write() function accepts a frame object.

```
1   class frame_cov_sub extends uvm_subscriber#(frame_t);
2      … create the coverage group for the frame_t
3      virtual function void write(frame_t frame);
4          … sample the frame values
5       endfunction : write
6    endclass : frame_cov_sub
```

Code Snippet 12 - uvm_subscriber extension

Extend a class from uvm_subscriber that accepts the message type as its parameter. Then create your subscriber in the build_phase and add it into the uvm_config_db. During the connect phase, any Producer can connect to the built-in analysis_export (the TLM analysis_port contained in uvm_subscriber).

A concise example of using a uvm_subscriber for sampling functional coverage can be found in [4].

Guideline

- Use the suffix "_sub" for any class name to indicate it is a subscriber.

[7] Actually, under-the-hood it's a uvm_analysis_port—which is described below.

- The Consumer's `write()` method must process the message immediately and return.

TLM1 Interfaces

Overview

Most users of the UVM base class library are familiar with the abstraction of FIFOs, analysis ports and exports. This section provides a brief overview of their capabilities. Like the mailbox, they create a M:N Producer-Consumer relationship.

`tlm_fifo`: A Mailbox Wrapped Inside an Enigma

The simplest structure is the `tlm_fifo` providing similar synchronization behaviors to the SystemVerilog parameterized `mailbox`[8]. It is typically configured for an M:1 Producer-Consumer relationship, where multiple Producers are sending to a single Consumer. That is, messages flow from one process to a FIFO queue that holds the messages. Similar to the `mailbox`, the receiving process blocks waiting when it attempts to get a message from an empty FIFO. In addition, the sending process can block when the FIFO reaches its programmable maximum capacity.

The processing flow of the `tlm_fifo` is illustrated in Figure 15- uvm_tlm_fifo Processing below.

@0. The Consumer creates the `tlm_fifo` and it initializes empty.
@1. The Consumer attempts a `get` on the FIFO. Since the FIFO is empty, the Consumer blocks waiting.
@2. Sometime later, the Producer `puts` a single message onto the `tlm_fifo`.
@3. In the same simulation delta-cycle the Consumer is unblocked. The first message in the queue is returned to the Consumer and it continues its processing from that point.

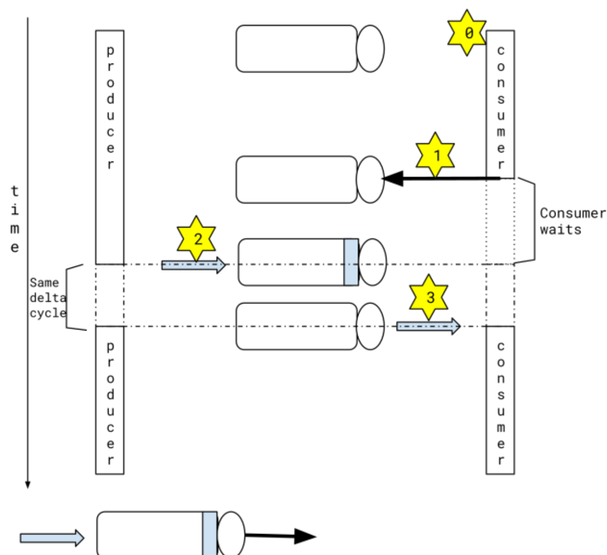

Figure 15- uvm_tlm_fifo Processing

`uvm_analysis_ports`: Any Port in a Storm of Messages

---

[8] Probably because it *uses* a SystemVerilog `mailbox`!

The analysis port is typically configured for an M:N Producer-Consumer relationship. Where the M and N can be any value from [0, infinity], and is configured dynamically when the simulation starts. The Consumer provides a `write` function that is *exported* to any component connected to the analysis port. When a Producer sends a message onto the analysis port, the Consumer's `write` task is called (immediately) with a copy of the message.

As shown in Figure 16 - TLM analysis_port below, the call flow for the analysis ports is similar to the flow for `uvm_callbacks`.

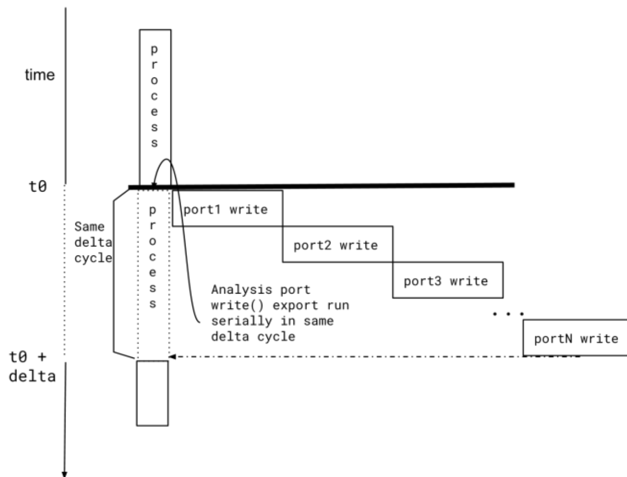

Figure 16 - TLM analysis_port

This abstraction provides a good separation of concerns, where:

- the Consumer always receives messages and processes them accordingly – without considering who they are received from.
- the Producer always sends its messages not concerning itself with who is receiving them.
- another coordinating component takes care of connecting Producer(s) to Consumer(s).

Use Model

Analysis ports are parameterized classes that can be defined for any valid SystemVerilog type, but once created can only accept that type.

As described above, the analysis port is a *functional* interface. The Consumer must create the receiving analysis *export* function specifying the type of message it accepts and must implement a `virtual write` function that accepts that message type (and does not consume time). The Producer must create a sending analysis *import* for the same message type that allows it to send to the Consumer's `write` function.

The next step is for some high-level component to decide which Producers are connected to which consumers. For illustration purposes, let's assume that multiple Producers are connected to one consumer (M:1 relationship), these are connected during the UVM `connect_phase` depending on how the environment is configured. One example of this kind of relationship is when multiple `uvm_monitor`s are connected to a single `uvm_scoreboard`. Once each `uvm_monitor` receives its full message, it sends the message out on its analysis port. The message is processed by the scoreboard and returns immediately.

However, it is possible that one or more of the monitors that send to the scoreboards are NOT created e.g., you can attach 8 monitors to a DUT's 8 output channels in one configuration, or attach 4 monitors for a different configuration, or even 0 monitors when all the channels are disabled. In all these cases, the `uvm_scoreboard` is created along with its analysis *export*; while only the required number of `uvm_monitor`s connect to these analysis

ports. The key design aspect for your scoreboard is that it should process whatever messages it receives, and the `write()` export function should not set any expectations on receiving messages. Some higher-level aspect of your scoreboard can make the decision whether receiving NO messages is acceptable.

The use of analysis ports enforces a good design separation of message reception from message checking.

Similarly, a N:1 Producer-Consumer relationship can also be created with analysis ports. Using the same mechanism as before, the Consumer create its analysis *export*, each Producer creates their analysis *import*, and some higher layer component connects the imports to the export. Provided the Consumer's `write()` function does not consume any time while processing each message, each Producer can send their message in the same simulation delta-cycle. Again, the Consumer should separate the message receipt from the message checking, so the message checking can be scheduled after all the Producers complete sending their messages.

An important consideration in using analysis ports is whether you need to modify the object passed to the `write()` function. By default, the object is passed by copy.

Guidelines

- Always use the either a `tlm_fifo` or a `uvm_analysis_port`[9] instead of a SystemVerilog `mailbox`. These provides additional capabilities and adhere to the *expected* UVM use-model.
- Use the suffix "`_fifo`" for any `tlm_fifo`.
- Use the suffix "`_ap`" for any `uvm_analysis_port` .

[9] Depending on the correct abstraction that you are modelling.

Summary

This paper provided a concise[10] exploration of the programming constructs available in native SystemVerilog that launch new processes and then allow us to synchronize processing between the individual processes.

From that base it described the similar and/or complimentary UVM 1.2 base classes that provide additional capabilities for synchronizing including built-in debug capabilities. In modelling our verification components the UVM 1.2 synchronizing classes provide superior functionality over their corresponding native SystemVerilog equivalents.

For each construct we provide a set of guidelines that attempt to define any best practices for using the construct.

This paper only scratches the surface on some of the 1.2 UVM classes. The reader is encouraged to explore each of the classes for additional functionality.

References

[1] IEEE, IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language, IEEE, 2017.

[2] Accellera Systems Initiative, Universal Verification Methodology (UVM) 1.2 Class Reference, Accellera, 2014.

[3] J. Bromley, "StackOverflow," 23 January 2013. [Online]. Available: https://stackoverflow.com/questions/14287446/proper-use-of-disable-fork-in-systemverilog.

[4] D. Smith, "Stick a Fork In It: Applications for SystemVerilog Dynamic Processes," Doulos, 2010. [Online]. Available: https://www.doulos.com/knowhow/sysverilog/SNUG10_fork_paper/SNUG10_fork_slides.pdf.

[5] Glasser, Mark, "Shutdown Agreements in a UVM Testbench," in *SNUG*, San Jose, 2017.

[6] Larson, David, "Master UVM - What the user's guide and examples haven't shown you," Synapse Deisgn.

[7] ChipVerify, "ChipVerify Tutorial - Subscriber [uvm_subscriber]," [Online]. Available: https://www.chipverify.com/uvm/uvm-subscriber.

[8] S. Birman, "How to Protect FIFOs Against Overflow - Part 1," Amiq Consulting, 25 07 2018. [Online]. Available: https://www.amiq.com/consulting/2018/07/25/how-to-protect-fifos-against-overflow-part-1/.

---

[10] In fact, some sections of this document are *too* concise. Some of the UVM classes provide more functionality than is described here due to word-count limitations for conference papers.