Terrestrial Verification Methodology: A Down-to-Earth Approach to Developing a UVM VIP

Bryan Morris, P.Eng.

Ciena

Ottawa, ON, Canada

www.ciena.com

ABSTRACT

The Universal Verification Methodology (UVM) is a comprehensive class library and framework to effectively develop structured reusable verification environments. However, when a simple (or quick-n-dirty) environment is required, the UVM structure enforces an arguably large overhead and complexity that presents a barrier to developing such an environment.

This paper describes a simple class structure using the Façade software pattern that enables you to create a simple UVM-based environment, while still allowing the future potential of expanding to more typical UVM structure. In this pattern, the UVM 'plumbing' code is handled under-the-hood, and the user only need create the driving, monitoring and scoreboarding functions specific to their environment.

Table of Contents

1. Introduction 2
2. Façade Pattern 4
   But UVM is not *that* Complicated 5
3. Terrestrial Verification Methodology (TVM) Agent 6
   TVM Use Model 6
   TVM Implementation 7
   TVM Call Flow 10
   Driving Call Flow (`tvm_driver`) 10
   Monitoring Call Flow (`tvm_monitor`) 12
   Comparing Call Flow (`tvm_scoreboard`) 13
   All Together Now 14
   Ummm… where's the Façade-Thing in all that? 15
4. Conclusions 15

Table of Figures

Figure 1 - Facade Pattern
Figure 2 - TVM UML Class Diagram (Facade)
Figure 3- TVM UML Class Diagram
Figure 4 – Example tvm_agent based environment
Figure 5 - TVM UML Sequence Diagram (Driving)
Figure 6 - TVM UML Sequence Diagram (Monitoring)
Figure 7 - TVM UML Sequence Diagram (Comparing)

1. Introduction

The Universal Verification Methodology (UVM) is a comprehensive class library and framework providing all the 'plumbing' and utilities required to effectively develop a reusable verification environment for ASICs and FPGAs. However, it is argued that there is some overhead with using the UVM class library for all verification environments[1]. The phrase that always comes to mind is the Law of the Instrument: "If all you have is a hammer, everything looks like a nail."[2]. While the author doesn't think that's entirely fair for many of the environments we create, there is some merit to the statement when creating a simpler verification environment.

To illustrate this point, to create a simple Verification Intellectual Property (VIP) using the recommended UVM structure requires a minimum of nine classes: (derived from) `uvm_env`, `uvm_agent`, `uvm_sequencer`, `uvm_sequence`, `uvm_sequence_item`, `uvm_driver`, `uvm_monitor`, an interface class and a `uvm_scoreboard`.

This paper proposes an approach allowing the rapid development of these simpler VIPs using UVM as the underlying methodology.

This approach uses the Façade pattern [2] to create a simpler API on top of the existing UVM verification components. Briefly, the Façade takes a complicated use-case model and wraps it into a simpler API. In this paper, I create a Façade class where the engineer implements the functionality of the UVM monitor, driver and scoreboard. Compared to the "Universal" methodology, my proposed "Terrestrial" Verification Methodology (TVM) should seem less intimidating. However, I'll give a fair warning that this paper does assume a solid understanding of the UVM class library and the typical best practices for UVM.

In addition, the TVM provides the common 'plumbing' aspects of a verification environment e.g.. building and connecting the verification components -- eliminating the boiler-plate coding required to hook up the various components within a typical UVM environment. Potentially reducing the number of classes needed to create a simple VIP from nine down to 1. This approach contrasts to the UVM code-generators that spew out all the (as discussed above, nine) classes and connecting logic, leaving you to fill in the blanks with your application-specific logic. The TVM provides the simplicity by creating an API where you are expected to add your "fill in the blanks" application-specific logic.

At this point, let me clarify the main purpose of this paper: I want to demonstrate the value of the Façade pattern using a concrete example. I believe the Façade pattern is a valuable addition to a verification engineer's 'toolkit'. But rather than put up pretty diagrams with abstract or 'toy problem' classes, I'm demonstrating a concrete implementation using the TVM classes. While the TVM is an interesting idea, and may be useful for some simpler environments, the tradeoff in using TVM is a simpler environment versus a more flexible one that you get by using the normal UVM environment structure. To summarize: the TVM classes are a demonstration of the Façade pattern, and are not (at this time) an optimal solution for all UVM VIP developments.

The remainder of this paper provides the following information:

- Articulate the difference between the TVM approach and the existing set of code-generators. The code generators make it easy to create a new environment, by creating all the boiler-plate code for you.
- Then provide a general description of the Façade pattern and what problems it can be useful for. I also identify the issues with using the Façade pattern e.g., by aggregating all the functionality into one class you reduce the good design principle of "separation of

[1] "You're Either With Me Or You're With: The UVM Register ... - AgileSoC." 9 Mar. 2014, http://www.agilesoc.com/2014/03/09/youre-either-with-me-or-youre-with-the-uvm-register-package/. Accessed 15 Jan. 2017.

[2] "Law of the instrument - Wikipedia." https://en.wikipedia.org/wiki/Law_of_the_instrument. Accessed 15 Jan. 2017.

concerns". Meaning that it may be difficult to extend a TVM-based environment beyond a simple structure.

- Then provide the specific details into the design, implementation and use-model for the Terrestrial Verification Methodology class.
- Lastly explore the set of situations where this is and is NOT appropriate and/or known limitations of this approach.
- Describe other possibilities for this

2. Façade Pattern

This section provides a brief introduction to the Façade pattern. See the Wikipedia[3] entry for more detailed information.

The justification for creating a Façade class is to reduce complexity in a system that needs multiple collaborating classes to complete some piece of functionality. In some systems, the complexity has evolved naturally over time. It's generally a *good thing* to use the `Single Responsibility Principle`[4] to ensure that your classes do just one thing, and then use a set of classes for some complex functionality.

However, at some point when a recurring functionality requires a set of steps to be done in a particular order, then creating a Façade class can help reduce the complexity by performing these steps.

In the following diagram, a call to the Façade class' `doSomething` function replaces the set of calls: `Class1::doStuff()`, `Class3::setX()` and `Class3::getY()`.

[3] "Facade - Wikipedia." https://en.wikipedia.org/wiki/Façade. Accessed 7 Jan. 2017.

[4] "solid - Wikipedia." https://en.wikipedia.org/wiki/SOLID_(object-oriented_design). Accessed 7 Jan. 2017.

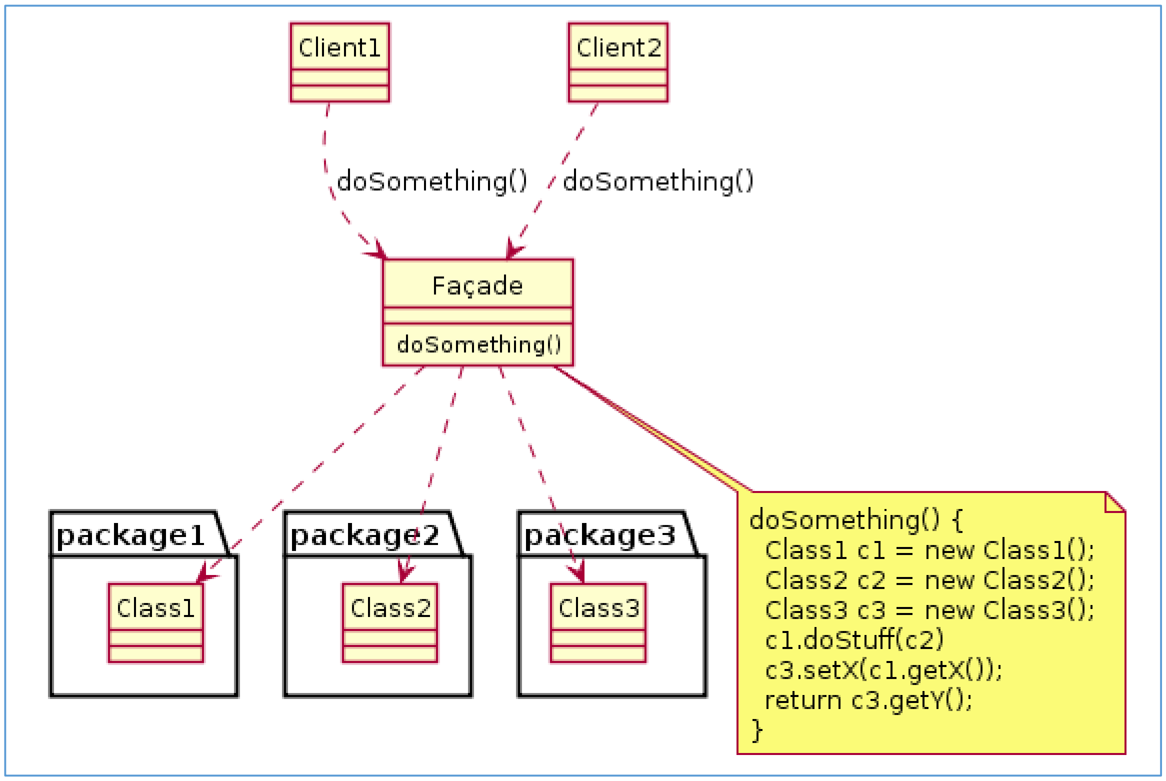


Figure 1 - Facade Pattern

You may not be aware but if you've used any of the callback mechanisms in your environments, these are examples of the Façade pattern[5], where the Façade is the callback class and this is instantiated in another class to provide additional functionality.

But UVM is not *that* Complicated

For the remainder of the paper I provide a concrete implementation of the Façade pattern by creating an API to drive, monitor and compare sequence items in a simple VIP. Generally, these functions are handled in the UVM by many collaborating classes. In our example, I'll aggregate them into one Façade class that provides the API, along with collaborating classes that the engineer can safely ignore.

Why do we need the Façade pattern for this functionality since the UVM driver, monitor and scoreboard are not *that* complicated? Using the UVM scoreboarding "compare/miscompare" functionality as a concrete example, the following sequence of events needs to happen (let's ignore for brevity's sake that all of these objects need to be built and connected):

- A `uvm_sequence` generates a `uvm_sequence_item` and passes it to a `uvm_sequencer`.
- The `uvm_sequencer` passes the `uvm_sequence_item` to a `uvm_driver` who drives it onto an interface as a series of pin wiggles.
- A `uvm_monitor` is listening to the interface and captures the signals and correctly

[5] "Verification Martial Arts » Blog Archive » How to use VMM callbacks." 19 Nov. 2008, https://www.vmmcentral.com/vmartialarts/2008/11/how-to-use-vmm-callbacks/index.html. Accessed 7 Jan. 2017.

transforms the pin wiggles back into a `uvm_sequence_item`, which is in turn, delivered to a `uvm_scoreboard`.

- The `uvm_scoreboard` might transform the `uvm_sequence_item` into an *expected* item, or accept it as the *actual* item and compare it against a matching *expected* item.

Whew! A total of six classes all working together to deliver a compare/miscompare at the scoreboard. *Unnecessarily* complicated? Perhaps not, but certainly there are a lot of "moving parts" involved in delivering this functionality.

The goal of the TVM Façade and supporting TVM classes is to implicitly handle all the following steps from above:

- creating and connecting all the underlying sequencer, driver, monitor and scoreboard
- delivering the collected sequence items to the scoreboard
- creating a simple in-order, head-of-queue comparison scoreboard algorithm.

While aggregating the VIP-specific functionality into one Façade class to:

- drive and monitor the pin-wiggles on the interface
- transform the `uvm_sequence_item` into the *expected* item
- if required, provide a more sophisticated scoreboarding comparison algorithm.

3. Terrestrial Verification Methodology (TVM) Agent

This section outlines the classes comprising the TVM, and an overview of their functionality. Using the Façade pattern, we create a `tvm_agent` class (the Façade) providing a hook to the user (through polymorphism) to implement the functionality of the driver, monitor and agent.

TVM Use Model

The use model of this `tvm_agent` is:

- Derive your VIP-specific agent from the `tvm_agent` class (providing the required set of parameterized types for the sequence items and sequencers to the `tvm_agent`).
- Implement a `drive()` method in your derived agent that accepts a sequence item and translates the sequence item to the pin-wiggles on your VIP's interface.
- Implement a `collect()` method in your derived agent that monitors your VIP's interface pin-wiggles, and collects the information to populate the supplied sequence item.
- If your VIP can use a simple in-order, head-of-queue scoreboarding methodology, then you're done. Otherwise, implement any or all of the following methods:
    - `convert2expected()`: converting the supplied sequence item to the *expected* sequence item.
    - `is_criteria_met()`: to indicate to the scoreboard that you have all the information you need to do a comparison e.g., at least one sequence item on both the *expected* and *actual* queues in the scoreboard
    - `compare()`: compares the supplied *expected* and *actual* sequence items and a returns a boolean value to indicate whether they compare or mis-compare.

The classes comprising the TVM take care of the following:

- *Creating* all the underlying verification components: monitor, driver, scoreboard and their associated 'plumbing', e.g., analysis ports and `write*` implementations.
- Connecting all the verification components as required, i.e., sequencer to driver, monitor to scoreboard.

- Automatically starting all the verification components.
- The in-order, head-of-queue comparison as described above. NOTE: this relies on the `uvm_sequence_item::compare()` functionality of the *expected* and *actual* sequence items' base class.

The TVM approach of pulling all the VIP-specific functionality into one class is in contrast to the typical (and useful) "make UVM easier" frameworks that generate all the derived classes and boiler-plate code to build and connect the verification components. These frameworks then expect the engineer to add the VIP-specific functionality into the individual classes.

TVM Implementation

Similar to a typical `uvm_agent`, the `tvm_agent` class holds the following set of parameterized classes:

- `tvm_config`
- `tvm_monitor`
- `tvm_driver`
- `tvm_scoreboard`.
- `tvm_agent` and `tvm_agent_api`

The `tvm_monitor`, `tvm_driver` and `tvm_scoreboard` derive from the UVM classes that you'd expect, and perform the functions you've come to know and love. As indicated in the UML diagram (see Figure 2 - TVM UML Class Diagram (Facade) below), the `tvm_agent_api` derives from the `uvm_agent`, but this class provides the API discussed above to drive, collect and compare functions. In turn, the `tvm_agent` derives from the `tvm_agent_api` to inherit this API. The `tvm_agent_api` is the Façade class providing `virtual` functions to collect and simplify the functionality of a typical VIP into one class.

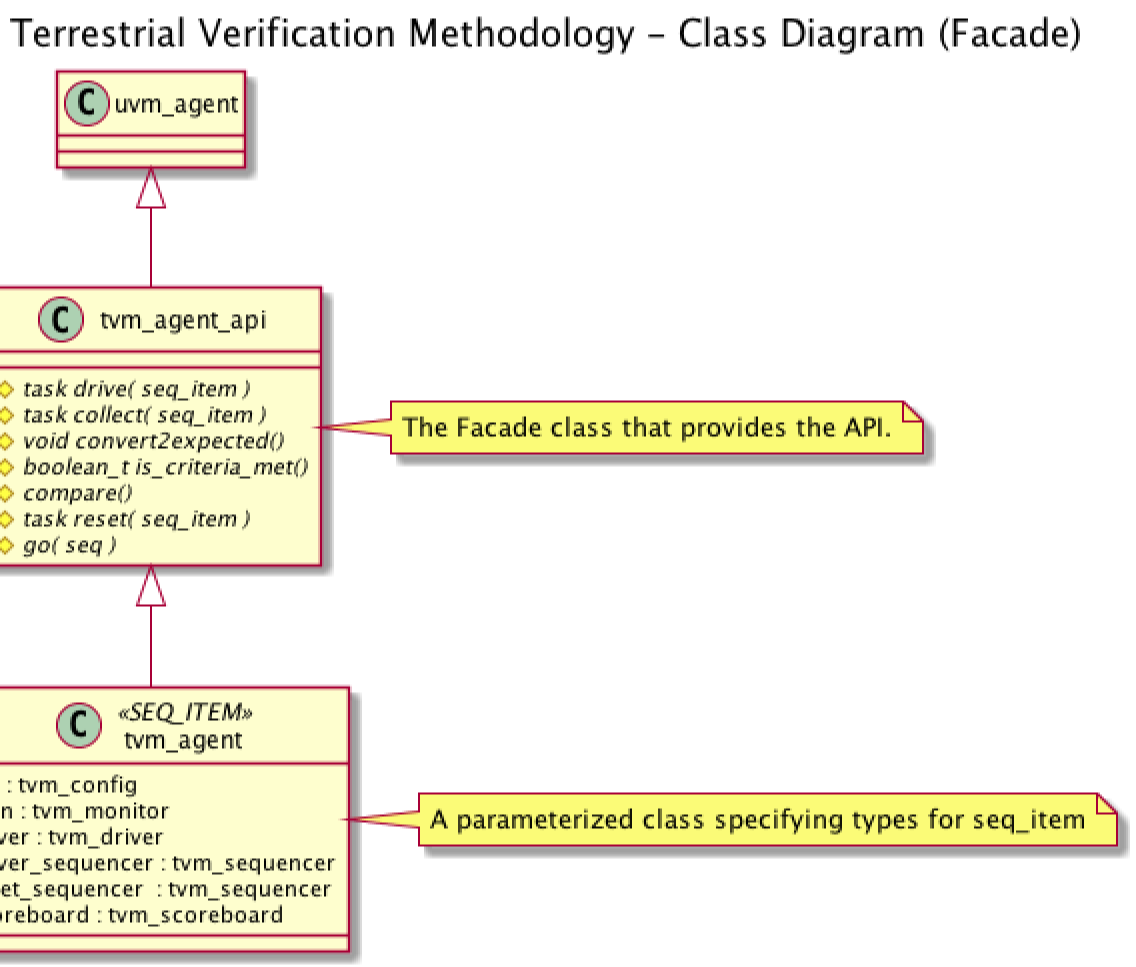

Terrestrial Verification Methodology – Class Diagram (Facade)
uvm_agent
tvm_agent_api
task drive(seq_item)
task collect(seq_item)
void convert2expected()
boolean_t is_criteria_met()
compare()
task reset(seq_item)
go(seq)
The Facade class that provides the API.
«SEQ_ITEM»
tvm_agent
cfg : tvm_config
mon : tvm_monitor
driver : tvm_driver
driver_sequencer : tvm_sequencer
reset_sequencer : tvm_sequencer
scoreboard : tvm_scoreboard
A parameterized class specifying types for seq_item

Figure 2 - TVM UML Class Diagram (Facade)

Figure 3- TVM UML Class Diagram is the UML Class diagram providing a high-level overview of all these classes.

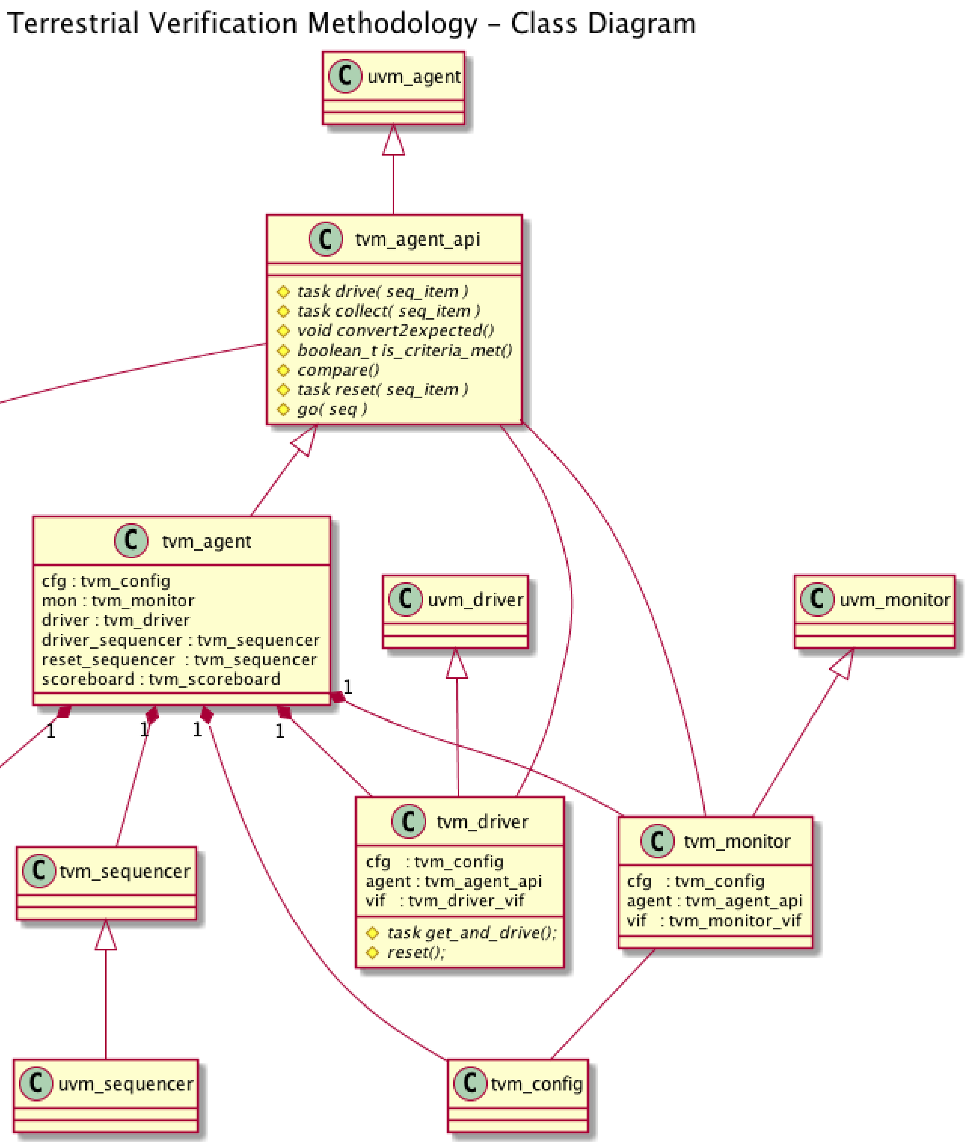


Figure 3- TVM UML Class Diagram

These are `virtual` tasks defined in the `tvm_agent_api` and should be defined in the user's derived class, providing the functionality as described above. Although not indicated in the above UML diagram, the `tvm_agent` is a parameterized class where the user should specify which `uvm_sequence_item` is used by the drive, collect and compare functions, respectively.

A typical UVM environment containing a `ubus_agent` (derived from the `tvm_agent`) is shown in the following block diagram:

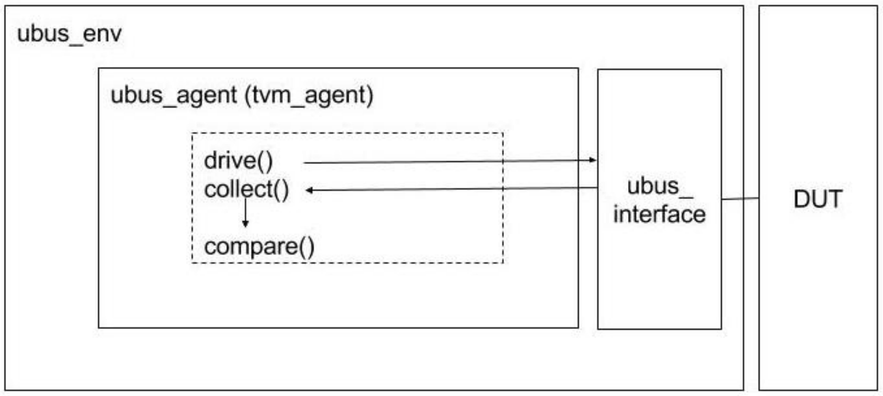


Figure 4 – Example tvm_agent based environment

The `tvm_config` class is an empty base class allowing you to derive a class to hold environment-specific configuration information e.g., random variables and associated constraints. While available to any of the TVM classes it is most likely useful to the logic found in the `tvm_agent_api`'s `drive()`, `collect()` and `compare()` tasks.

TVM Call Flow

At this point it's easier to demonstrate the utility of the TVM classes by describing the call flow for the driving and then monitoring process of a TVM-based VIP. For clarity, I separate the TVM into three concurrent processes[6]:

- *driving*: pushing sequence items onto the DUT interface;
- *monitoring*: receiving sequence items from the DUT interface;
- *comparing*: comparing the *expected* and *actual* sequence items.

Each of these are discussed in more detail below. In the following I use a `ubus_agent` as an example of a derived `tvm_agent`.

Driving Call Flow (`tvm_driver`)

In the driving process we separate the sequence item going into the DUT and converted to the predicted sequence item as the *expected* sequence item; and the sequence item coming from the DUT as the *actual* sequence item.

As well, we create a `ubus_agent` derived from a `tvm_agent`. It provides the `drive()`, `collect()` tasks and `convert2expected()` function.

1. The test environment creates an `ubus_agent`, the base `tvm_agent` creates the associated `tvm_sequencer`, `tvm_driver`, `tvm_monitor` and `tvm_scoreboard`[7]. The `ubus_agent` also creates a `ubus_config` class (derived from the `tvm_config`) class to hold the configuration information for this VIP.
   The `tvm_config` derived classes hold random variables and associated constraints that the logic found the driver, monitor and scoreboard can use in their functionality. For example, this information can include environmental parameters on the number of

[6] NOT to be confused with `uvm_phases`

[7] Which are all parameterized classes providing the type of the various UVM classes they work with.

frames/packets to send.
An important point to remember is that the tvm_driver, tvm_monitor and tvm_scoreboard hold the handle to their parent the tvm_agent_api object so that they can call the derived ubus_agent's virtual drive(), collect() and compare() tasks.
The test environment should create the appropriate interface classes for the UBUS DUT, and place the handle to these interface objects into the uvm_config_db. The ubus_agent can then get access to that interface for its drive(), and collect() tasks.

2. The tvm_agent class connects the sequencer to the tvm_driver, and the tvm_monitor to the tvm_scoreboard (let's assume it's in UVM_ACTIVE mode) via the scoreboard's analysis ports.
3. The tvm_agent grab their associated interfaces for the driver (request path) and the monitor (response path) from the uvm_config_db. The tvm_driver and tvm_monitor start their processes of waiting for the sequence item (the driver), or waiting for interface pin wiggles (monitor).
4. The test creates a uvm_sequence (where the body() of the sequence generates a set of random sequence items) and starts the request sequence item by calling the tvm_agent's go() task with the handle to the uvm_sequence object.
5. The tvm_agent starts this uvm_sequence on its attached tvm_sequencer. The tvm_sequencer uses the default arbitration algorithm, and passes the set of sequence items to the tvm_agent's tvm_driver object.
6. Typically, a uvm_driver uses the flow:

    Loop:
    - Waits for the next driving sequence item.
    - Convert the abstract sequence item to the pin wiggles on the attached interface as required by the protocol.
    - Indicate to the sequence that the driving is done.
    - Back to the top of the loop.

    Instead, the tvm_driver delegates the driving of this flow to the tvm_agent:

    Loop:
    - tvm_driver waits for the next request sequence item.
    - Calls its parent tvm_agent_api::drive() pure virtual task with the sequence item.
      Control passes to the ubus_agent::drive() -- the concrete implementation of the tvm_agent_api::drive() virtual task -- converts the incoming sequence item to the pin wiggles on the attached interface as required by the protocol.
    - Control returns back to the tvm_driver which indicates to the sequencer that the driving is done.
    - Back to the top of the loop.

In summary, the tvm_driver handles receiving the sequence item stream from the sequencer and any responses back to the sequence, but delegates the driving of the sequence item it receives to the derived ubus_agent::drive() task. This is shown in Figure 5 - TVM UML Sequence Diagram (Driving) below

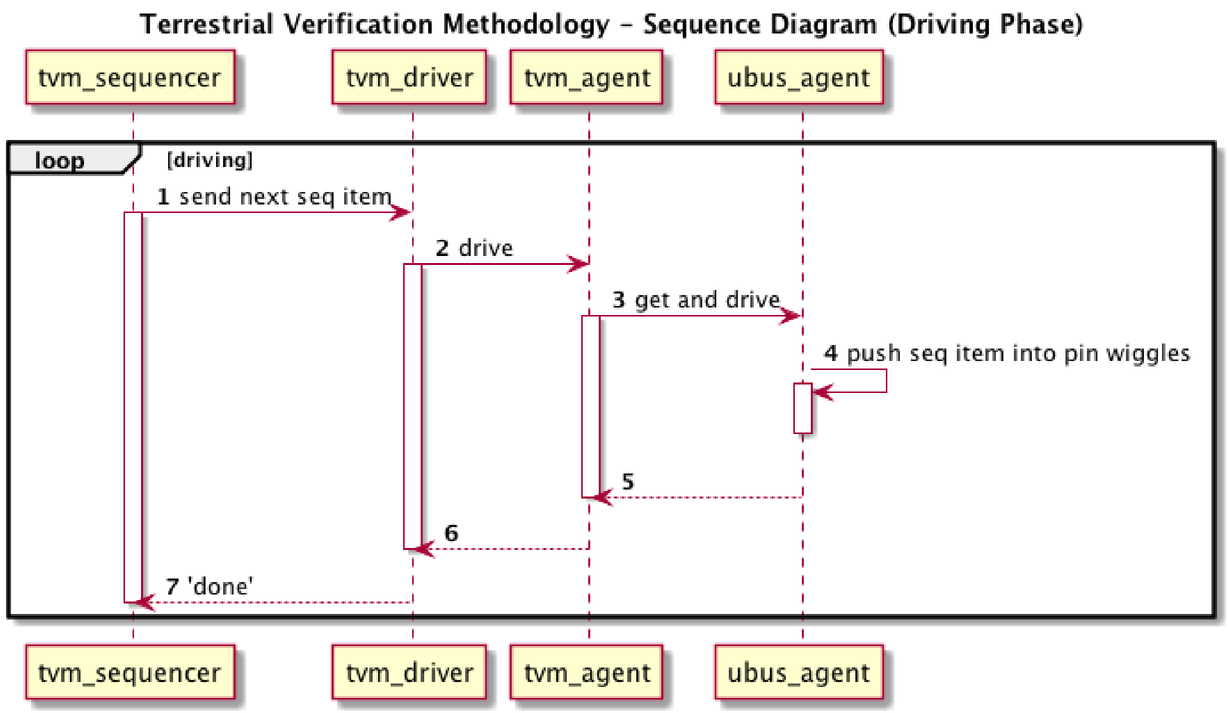


Figure 5 - TVM UML Sequence Diagram (Driving)

Monitoring Call Flow (`tvm_monitor`)

The usual flow for a `uvm_monitor` is:

- Create a new sequence item.
- Wait for the indication from the attached interface of a start of transaction.
- As per the interface protocol, collect the pin wiggles and convert to a sequence item.
- Send the sequence item to monitor's analysis port (usually destined to a scoreboard).
- Back to the top of the loop.

Similar to the `tvm_driver`, the `tvm_monitor` delegates the monitoring responsibility (or specifically, the collecting of the interface signals) to the `tvm_agent_api::collect()` virtual task, as follows:

- Create a new sequence item.
- Call the `tvm_agent_api()::collect()` pure virtual task's derived `ubus_agent::collect()` with the handle to the new sequence item. The `collect()` task:
  - Waits for the indication from the attached interface of a start of transaction.
  - As per the interface protocol grab the pin wiggles and convert to the sequence item.
  - Send sequence item to `tvm_monitor`'s analysis port (usually destined for a scoreboard).
- Back to the top of the loop.

This TVM Monitoring process is shown in the UML sequence diagram below:

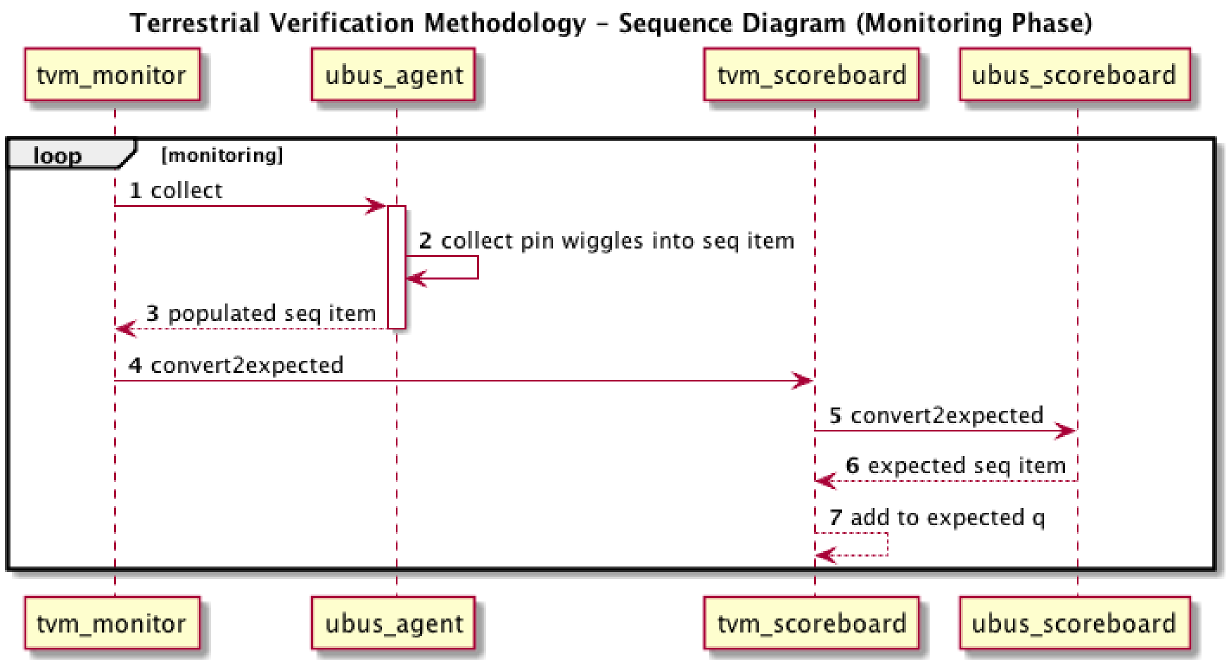


Figure 6 - TVM UML Sequence Diagram (Monitoring)

Comparing Call Flow (`tvm_scoreboard`)

Lastly, describing the call sequence for the comparing process that takes care of scoreboarding the *expected* and *actual* sequence items:

The typical flow for the scoreboard is:

Expected Seq Item Loop:

- wait for the driving sequence item from the input monitor (via a call to the analysis import `write()` task).
- convert the received sequence item to the *expected* sequence item (as per the DUT's specified behaviour for the provided stimulus and configuration.
- place the expected sequence item on an "expected" queue.
- back to top of loop.

Actual Seq Item Loop:

- wait for the received sequence item (in this case, the *actual* sequence item) from the output monitor (via a call to the analysis import `write()` task).
- attempt to find a match to this response sequence item in the *expected* queue.
- if a match is found, remove the sequence item from the *expected* queue
- if no match is found (and a match is expected), issue an error.
- back to top of loop.

The flow in the `tvm_scoreboard` is similar, except it delegates some functions to the `tvm_agent`'s `convert2expected()`, `is_criteria_met()` and `compare()` virtual tasks:

Request Loop:

- wait for the request sequence item from the input monitor (via a call to the analysis import `write()` task).

- call the `tvm_agent_api::convert2epxected()` pure virtual with the received sequence item
- the `tvm_agent_api::convert2expected()` delegates to the derived `ubus_agent::convert2expected()` functionality as above: converting the received sequence item to the *expected* sequence item (as per the DUT's specified behaviour for the provided stimulus and configuration).
- place the *expected* sequence item returned from `convert2expected()` onto an "expected" queue..
- back to top of loop.

Actual Loop:

- wait for the *actual* sequence item from the output monitor (via a call to the analysis import `write()` task).
- attempt to find a match to this response sequence item in the expected queue.
- if a match is found, remove the sequence item from the expected queue
- if no match is found, issue an error.
- back to top of loop.

In summary, `tvm_monitor`, `tvm_driver` and `tvm_scoreboard` tasks handle the functionality that was generic in nature and dealt with moving sequence items between components; while all the tasks unique to the VIP, such as converting abstract sequence items to pin wiggles, etc., are now delegated to tasks inside the `tvm_agent`'s derived class.

The TVM Comparing process -- while not inspiring -- is included for completeness with the following UML Sequence Diagram.

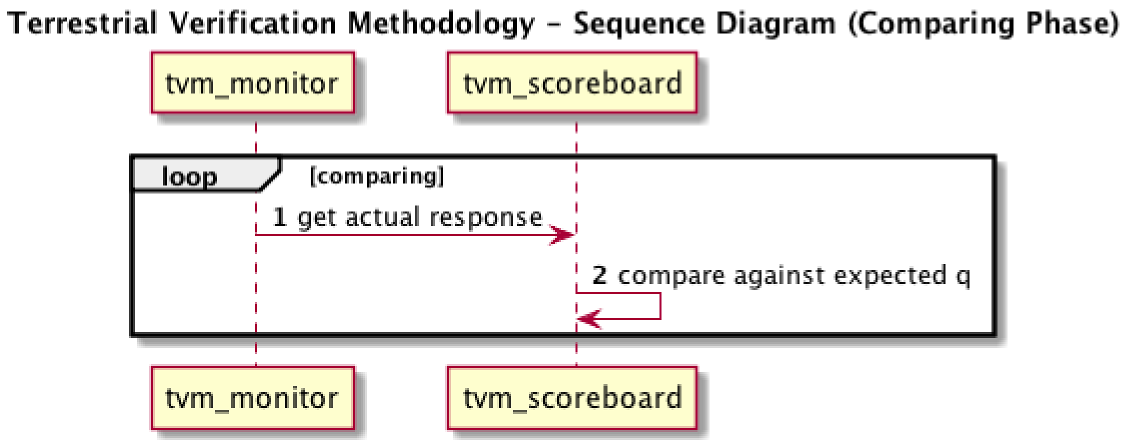
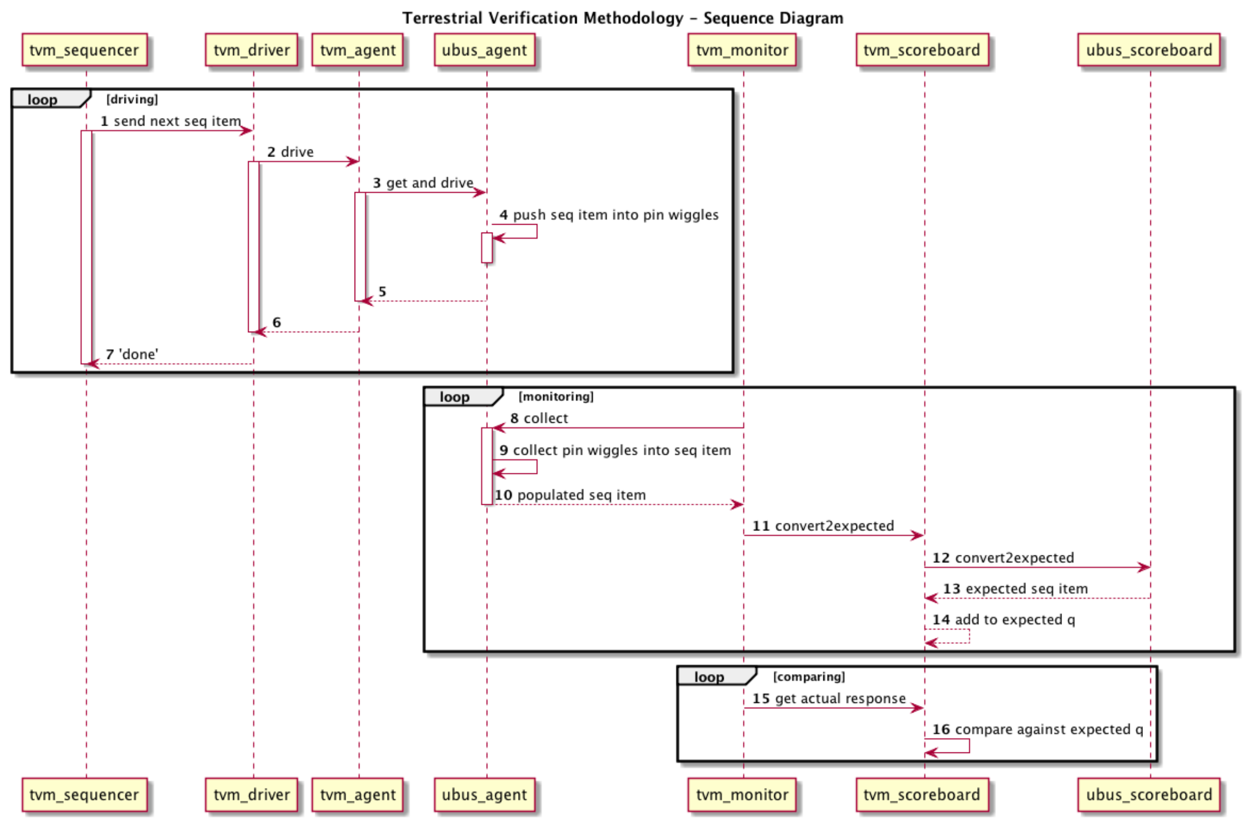

Terrestrial Verification Methodology - Sequence Diagram (Comparing Phase)
tvm_monitor
tvm_scoreboard
loop
[comparing]
1 get actual response
2 compare against expected q
tvm_monitor
tvm_scoreboard

Figure 7 - TVM UML Sequence Diagram (Comparing)

All Together Now...

The following UML sequence diagram pulls these three driving, monitoring and comparing processes together. It illustrates how the objects in the `tvm_agent` and derived classes interact.



Ummm… where's the Façade-Thing in all that?

Getting back to the original intent of this paper… the Façade pattern. Recall that the Façade pattern provides a simplified API to introduce common functionality. For TVM, we want to simplify the building and connecting of a VIP's verification components, *and* driving, monitoring and comparing the sequence items that test a DUT's functionality.

For TVM, the abstract class `tvm_agent_api` provides a set of `pure virtual` tasks defining the API to:

- `drive()` the sequence item onto the interface as a series of pin-wiggles;
- `collect()` the pin-wiggles from the interface and create a sequence item to deliver to TVM's `compare` function;
- `compare()` the *expected* and *actual* sequence items.

While the `tvm_agent` provides the base functionality to build and connect the verification components. Both classes then implement the Façade pattern with the `tvm_agent_api` providing the API and the `tvm_agent` providing the 'glue' logic for.

4. Conclusions

In this paper we discussed the Façade pattern as a useful construct where you can simplify functionality requiring multiple collaborating classes into a single Façade class. The value of this pattern is that it allows you to refactor existing functionality into something simpler, with no changes to the existing functionality. As well, you can add new functionality in the Façade building on already working code.

We then took the (arguably) complex but typical functionality of an UVM agent and collapsed the main functionality into one Façade class: the `tvm_agent` -- providing the functionality to drive, collect and compare sequence items.

While the value of the TVM is likely not suitable for any significantly complex agent functionality, it could be used in a simple, or potentially a temporary 'throw-away', agent (i.e., one that could in the future be replaced with a full UVM agent architecture). The main purpose of the TVM classes is to demonstrate the value of the Façade pattern -- which should be a valuable pattern in every verification engineer's toolkit.