UVM Framework Users Guide

Version 2021.3

Contents

Chapter 1

Introduction to the UVM Framework (UVMF)

1.1 Motivation for Using UVMF

1.1.1 Schedule Reduction

The steep learning curve of UVM often prevents product teams from realizing the productivity and quality benefits of using advanced verification methodology. The UVM Framework (UVMF) provides a jump-start for learning UVM and building UVM verification environments. It defines an architecture and reuse methodology upon UVM, enabling teams new to UVM to be productive from the beginning while ascending the UVM learning curve. The UVM code generator provided by UVMF automates the creation of the files, infrastructure and interconnect for interface packages, environment packages and project benches. Interface packages, environment packages, and project benches are characterized using text based YAML. The UVMF generator uses these characterizations to create UVM source. Once generated, developers can promptly focus on adding functionality specific to the design and interfaces used.

1.1.2 Reuse Methodology

The UVM Framework is a reuse methodology that verification teams can leverage. It supports component level verification reuse across projects and environment reuse from block through chip to system level simulation. The UVM Framework is an established UVM use model that is in use at many companies in multiple industries across North America and Europe.

1.1.3 Single Architecture for Simulation and Emulation

The UVM Framework provides an architecture that supports pure simulation and accelerated simulation using emulation. This enables the creation of a single unified environment that supports block, subsystem, chip and system level tests, and with the choice of running on a pure simulation platform (e.g. Questa) or a hardware-assisted acceleration platform using emulation (e.g. Veloce and Strato).

1.1.4 Consistency

One of the key requirements in order to realize verification reuse is consistency. This canâĂŹt be emphasized enough. Increased consistency in how UVM is used decreases integration effort when reusing verification components. Teams in different locations and contractors brought into a project will have different opinions on how things should be done. The UVMF code generators help ensure content created across sites can be integrated with and into other components using the UVMF code generators. It is tempting for contractors or teams to modify the infrastructure of the generated code to suit their particular preferences. This always creates schedule delays during integration.

1.1.5 Resistance to change

Verification teams typically include engineers with a range of UVM experience. Some engineers have a strong opinion of what the âĂIJrightâĂİ way of using UVM is. UVM provides building supplies. How those building supplies are used can vary widely. UVMF defines a UVM use model by providing a base class package and code generators. This definition ensures horizontal reuse from project to project, vertical reuse from block to top, and platform reuse from simulation to emulation. Some differences in UVM use are just personal preferences. Other differences create issues that are not apparent until later in the project or when reuse is attempted. There are common mistakes made at every level of experience. Engineers new to a methodology

tend to make language or other mistakes related to OOP. Engineers with experience in a methodology tend to make mistakes that limit reuse on future projects. Some engineers with UVM experience will resist change from their home-grown solution to another solution. A home-grown methodology can not get the same level of âĂŸhardeningâĂŹ or validation the UVMF has received through over ten years of use across multiple industries in multiple continents.
Driving a common methodology across a large number of engineers and locations requires a lot of resources. Because of its wide use over a long period of time, there are a lot of resources available to teams adopting and using UVMF. There are local Mentor AEâĂŹs who have worked with UVMF. A training video series on UVMF is available on Verification Academy. Courses on UVMF are available through Mentor Training Services. Documentation, examples, and tutorials are available in the UVMF installation. Mentor Corporate support AEâĂŹs have worked with teams using UVMF. Consultants with UVMF experience are available through Mentor Consulting. When using UVMF, this results in support resources that are already familiar with your UVM test bench infrastructure and flow. In short, there is a lot of implementation and training support available through Mentor.

1.1.6 Free and Open Source

The UVMF base class package and source generator are licensed under the Apache License, Version 2.0. It is an outcome of and expression of Mentors partnership with customers and our commitment to their productivity.

1.2 Coverage within UVMF

UVMF provides a mechanism for rapid creation of reusable simulation infrastructure. Coverage collection components can be defined and connected in the environment using the UVMF code generators. The list below identifies components where functional coverage can be collected and how to add coverage to these components:

1. Coverage components: Create the class definition for this component using the UVMF generator and connect it to other components in the environment using the generator. The coverage component will likely

only have analysis exports for receiving transactions and no analysis ports.

2. Predictors: Manually add required cover groups to the predictor that was generated.
3. Scoreboard: Extend UVMF scoreboards on a per-environment basis to define cover groups and sample coverage based on DUT output transactions.

It is important to collect coverage on data that has been validated. In order to avoid agent coverage being confused with coverage of scoreboard validated data the default value of the `has_coverage` bit in the `uvmf_parameterzed_agent` is zero. This will prevent the coverage component within the agent from being constructed. Users will have to manually change this bit to enable agents to record transaction coverage at the interface agent.

The UVMF scoreboards contain features that help avoid the falsely optimistic level of coverage:

1. `end_of_test_activity_check`: This flag is set by default. It generates a uvm error if no transactions were received. This will help avoid mistakes that result in the scoreboard not being attached to prediction or prediction not sending expected transactions. This flag can be set for specific scoreboards in the build_phase of the environment.
2. `end_of_test_empty_check`: This flag is set by default. It generates a uvm error if transactions remain in the scoreboard in the check_phase. This will help avoid mistakes that result in no transactions being received for comparison against expected transactions. This flag can be set for specific scoreboards in the build_phase of the environment.
3. `wait_for_scoreboard_empty`: This flag is clear by default. It postpones the termination of run_phase until there are no transactions in the scoreboard. The UVM based timeout, `UVM_DEFAULT_TIMEOUT`, is used to prevent simulations from hanging.
4. The end-of-test results summary of UVMF scoreboards contain the scoreboard hierarchy. A cursory view of the message summary at the end of the transcript will identify the absence of a scoreboard.

1.3 Major Divisions of Functionality within UVMF

1.3.1 UVMF Base Package

The UVMF base package, `uvmf_base_pkg`, is a library of base classes that implement core functionality of components found in all simulation benches. This includes base classes for transactions, sequences, drivers, monitors, predictors, scoreboards, environments and tests. All classes in the UVMF base package are derived from UVM classes. User extensions then provide variables, tasks and functions specific to the design under test. The UVMF base package and the package structure used within UVMF define a UVM reuse methodology. This methodology supports horizontal reuse, i.e. reuse of components across projects, as well as vertical reuse, i.e. environment reuse from block to chip to system.

1.3.2 Interface Packages

UVMF interface packages and their associated BFMs provide all of the functionality required to monitor and optionally drive a design interface. Interface packages and BFMs are reusable across projects. An interface package is composed of three pieces: a signal bundle interface, BFM interfaces and the package declaration. The signal bundle contains all signals used in the protocol. The BFMs implement the protocol signaling to drive and monitor transactions at the pin level. The package declaration includes all class definitions and type definitions used by the interface agent.

1.3.3 Environment Packages

The environment package is a key aspect that enables vertical reuse of environments within the UVMF. The environment package contains the environment class definition, its configuration class definition and any environment level sequences that could be used in higher level simulations. Block level environments contain agents, predictors, scoreboards, coverage collectors and other components. All other levels of environment include other environments. Environments are structured hierarchically similar to the way RTL is composed hierarchically.

1.3.4 Verification IP Directory

The `verification_ip` folder contains all packages that are reused across projects and from block to top. This folder contains environment packages, interface packages, utility packages, etc. Multiple `verification_ip` directories are supported. Each are referenced using separate environment variables.

1.3.5 Project Benches Directory

The `project_benches` directory contains bench level code that is not reuable. The simulation bench is composed of top level elements that are not generally intended to be reusable horizontally nor vertically. It defines test level parameters, the top level modules, top level sequence and top level UVM test. It also includes derived sequences and tests used to implement additional test scenarios.

1.3.6 Example Groups

UVMF examples are divided into groups. The groups include `base_examples` and `vip_examples`. The `base_examples` are the core examples and run in simulation and emulation. The `vip_examples` contain Questa VIP and emulatable VIP example for AXI4. A Questa VIP license and software installation is required in addition to a Questa license to run the QVIP example. A Veloce software license and software installation is required in addition to a Questa license to run the VIP example. Each example group contains a `verification_ip` and `project_benches` folder. The `verification_ip` folder contains all reusable packages used and shared among benches in the `project_benches` folder. The benches can also use packages found in the `verification_ip` folder in the `base_examples` group.

1.4 UVMF Base Class Package Overview

1.4.1 Overview

The `uvmf_base_pkg`, located under the `$UVMF_HOME` directory, provides a library of classes that implements the methodology used by the UVMF. In order to support emulation UVMF is divided into two packages: `uvmf_base_pkg` and `uvmf_base_pkg_hdl`. The latter only contains the synthesizable typedefs and parameters required by the emulated portion of a test bench. The former includes all class definitions and other non-synthesizable typedefs. The `uvmf_base_pkg_hdl` package is imported by `uvmf_base_pkg`. Each class within `uvmf_base_pkg` is described below.

1.4.2 Stimulus Classes

1.4.2.1 `uvmf_transaction_base.svh`

This is the base class for all sequence items, i.e. transactions, within UVMF. It provides a unique transaction ID variable and associated functions useful for debug. It also provides variables used for transaction viewing and a unique key for storing the transaction in associative arrays.

1.4.2.2 `uvmf_sequence_base.svh`

This is the base class for all sequences within UVMF. It extends `uvm_sequence` but provides no additional functionality. It provides a location where functionality common to all UVMF sequences can be added.

1.4.3 Agent Classes

1.4.3.1 Built-in Analysis Port

Each UVMF agent contains an analysis_export named `monitored_ap`. Information observed during the protocol transfer by the monitor BFM are sent to the monitor class. The monitor class places the information within a UVM sequence item. The sequence item is then broadcasted from the agent's analysis_port named `monitored_ap`.

1.4.3.2 `uvmf_monitor_base.svh`

This is the base class for all UVMF monitors. Only monitors extended from `uvmf_monitor_base` should be used as specialization of the `MONITOR_T` parameter of the `uvmf_parameterized_agent` base class. When extending the uvmf_monitor_base, only the notify_transaction task must be defined. This task receives a struct from the monitor BFM that contains information about the bus transfer. It is recommended that signal level bus monitoring be done in the monitor BFM for optimal run-time performance, especially with emulation. The notify_transaction task is how the monitor BFM pushes observed data to the monitor class for broadcasting by the agent.

1.4.3.3 uvmf_driver_base.svh

This is the base class for all UVMF drivers. Only drivers extended from uvmf_driver_base should be used as specialization of the DRIVER_T parameter of the uvmf_parameterized_agent base class. When extending uvmf_driver_base, only the access() task must be defined. The access() task either drives bus activity directly or calls a task in the driver BFM which drives activity. It is recommended that signal level bus driving be done in the driver BFM for optimal run-time performance, especially with emulation.

1.4.3.4 uvmf_parameterized_agent_configuration_base.svh

This is the base class for all interface agent configurations. Only configurations extended from uvmf_parameterized_agent_configuration_base should be used as specialization of the CONFIG_T parameter of the uvmf_parameterized_agent base class. Variables common to all agents and specific to the uvmf_parameterized_agent are in the uvmf_parameterized_agent_configuration_base. Add protocol specific configuration variables to an extension of uvmf_parameterized_agent_configuration class.

1.4.3.5 uvmf_parameterized_agent.svh

This class implements an interface agent. This agent can be used with any protocol. The protocol specific features reside in the configuration, driver, monitor, coverage and transaction class types used as parameters to this class. If the agent is active then it automatically places its sequencer within the uvm_config_db and within the agent configuration object for retrieval and use by the top level sequence. Prior to constructing a monitor, the parameterized agent checks the uvm_config_db for a monitor of the required type. If one is returned then it is used instead of constructing a local monitor. This is to support construction of a shared monitor in an upper level environment. The shared monitor is created in an upper level environment where

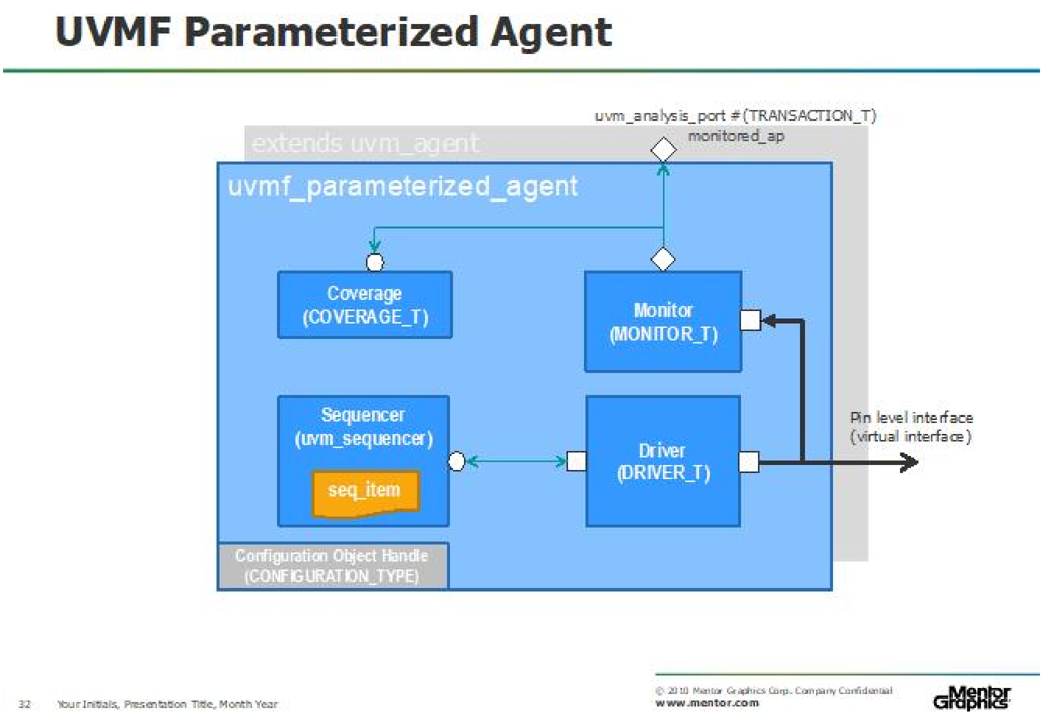

Figure 1.1: UVMF Parameterized Agent Structure

lower level environments monitor common interfaces. Figure 1.1 illustrates the structure of the parameterized agent.

1.4.4 Predictor Classes

The use of UVMF predictor classes, `uvmf_predictor_base` and `uvmf_sorting_predictor_base` has been replaced by application specific predictors and other analysis components that can be generated using the UVMF code generators. The `util_components` section in the YAML based UVMF code generators can be used to characterize a predictor with any combination of analysis ports and analysis exports. Once generated, the user only need to implement the behavioral model within the generated predictor class.

1.4.5 Analysis Component Classes

The UVMF environment generator provides a way for users to specify analysis components with any combination of analysis exports and analysis ports. This allows the user to automatically generate any analysis component required. One type of analysis component that can be generated are predictors. Predictor specification for the UVMF code generator includes the number and type of analysis exports required as well as the number and type of analysis ports required. Another type of analysis component that can be generated are coverage components. Coverage component specification for the UVMF code generator includes the number and type of analysis exports required. Coverage components typically do not have analysis ports so the list of analysis port type and names would be empty. Another type of analysis component that can be generated are scoreboards. Scoreboard specification for the UVMF code generator includes the number and type of analysis_-exports required as well as the number and type of analysis ports required. The scoreboard specification for an analysis component is for defining and instantiating custom scoreboards.

1.4.6 Scoreboard Classes

The scoreboards provided within UVMF perform comparison between predicted and actual DUT output transactions. UVMF scoreboards perform no prediction operations. Within UVMF all prediction is performed using predictors. This allows reuse of scorebaords. The UVMF provides a set of scoreboards for use with various data flow characteristics. These scoreboards include the `uvmf_in_order_scoreboard`, `uvmf_in_order_scoreboard_array`, `uvmf_in_order_race_scoreboard`, and `uvmf_out_of_order_scoreboard`. The `uvmf_scoreboard_base` provides base functionality and can be extended by the user to create custom scoreboards without using the UVMF generator.

1.4.7 Scoreboard Features

All UVMF scoreboards provide the features described in the table below.

Function	Description

`enable_scoreboard()` (Default Setting)	Enable the scoreboard to receive and compare transactions.
`disable_scoreboard()`	Prevent the scoreboard from accepting transactions. Transactions received by the expected_analysis_export and the actual_analysis_export will be discarded.
`enable_entry_comparison()` (Default Setting)	Enable comparison of expected and actual transactions.
`disable_entry_comparison()`	Disable comparison of expected and actual transactions. Transactions are received, stored, and retrieved from storage for comparison. However, the comparison between expected and actual transactions is not performed.
`enable_end_of_test_empty_check()` (Default Setting)	Generate an error if transactions remain in the scoreboard at the end of the simulation.
`disable_end_of_test_empty_check()`	Do not check for transactions remaining in the scoreboard at the end of the simulation.
`enable_end_of_test_activity_check()` (Default Setting)	Generate an error if no transactions have been received by this scoreboard during this simulation.
`disable_end_of_test_activity_check()`	Do not check whether this scoreboard received transactions during the simulation.
`enable_wait_for_scoreboard_empty()`	Enable the scoreboard to delay termination of `run_phase` until scoreboard contains no remaining transactions.
`disable_wait_for_scoreboard_empty()` (Default Setting)	Scoreboard will not delay termination of `run_phase` to allow remaining transactions to drain from the scoreboard.
`enable_sprint_use_to_display_compare_results()`	Use `sprint()` method to generate string messages used in `compare_message()` function.
`disable_sprint_use_to_display_compare_results()` (Default Setting)	Use `convert2string()` method to generate string messages used in `compare_message()` function.

`set_max_remaining_-transaction_print()` (Default Value:10)	Controls the number of transactions that will be printed at the end of the simulation if transactions remain in the scoreboard at the end of the `run_phase`.
`flush_scoreboard()`	Remove all transactions currently stored in the scoreboard.
`compare_message()`	Virtual function that allows users to create custom messages to report transaction comparison results. Message is generated for each transaction comparison performed.
`report_message()`	Virtual function that allows users to create custom end-of-test summary report.

The use of UVMF scoreboards require the transaction class being compared to contain a `compare()` function. The use of UVMF out-of-order and in-order-array scoreboards also require the transaction class being compared to contain a `get_-key()` function.

1.4.7.1 Built-in Analysis Ports

Each UVMF scoreboard contains two analysis_exports. The `expected_analysis_-export` receives sequence items from a predictor, golden model. The expected sequence item defines what the predictor expects the DUT to produce in response to input stimulus and configuration settings. The `actual_analysis_export` is the sequence item broadcasted by the agent connected to the DUT output.

1.4.7.2 `uvmf_scoreboard_base.svh`

This is the base class for all scoreboards within the UVM Framework. It provides the two analysis exports for receiving transactions, `expected_analysis_export` and `actual_analysis_export`. It also provides basic end of test use checks and reporting.

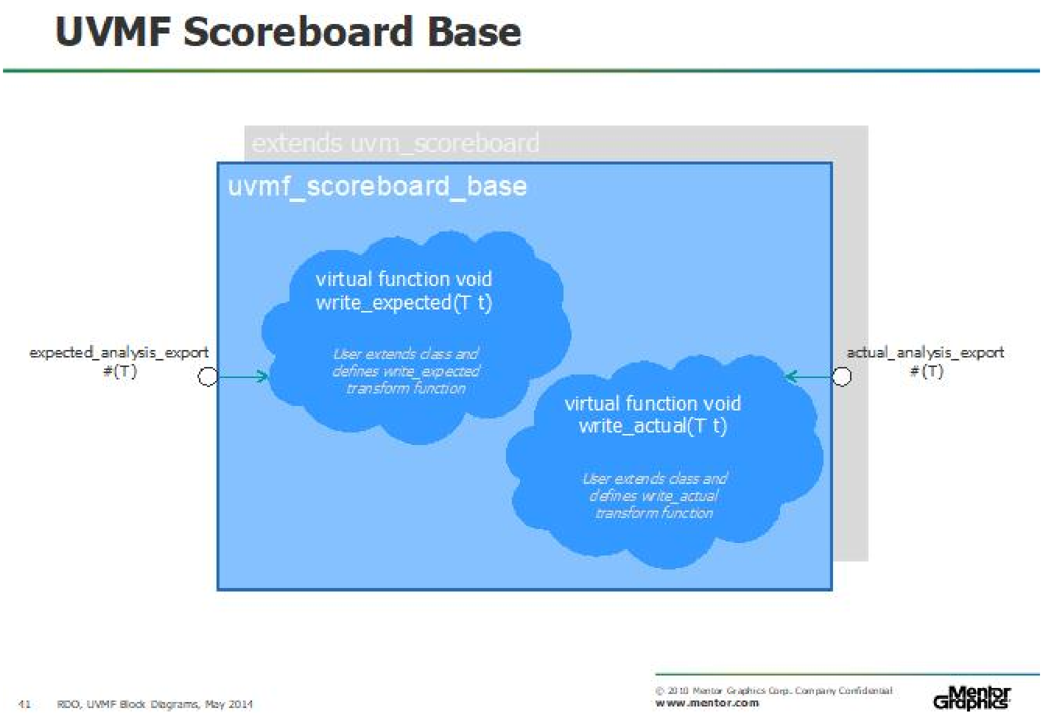

Figure 1.2: UVMF Scoreboard Base Class Structure

1.4.7.3 `uvmf_in_order_scoreboard.svh`

This scoreboard is used in circumstances where the data order through the DUT is preserved or predictable. The in order scoreboard extends the scoreboard base. It adds a queue for storing expected results. Transactions received through the `expected_analysis_export` are placed into the queue. Transactions observed on the DUT output are sent to the actual export for comparison. The arrival of a transaction on the `actual_analysis_export` causes the next transaction to be removed from the queue and compared to the actual transaction. An error is generated if the queue is empty.

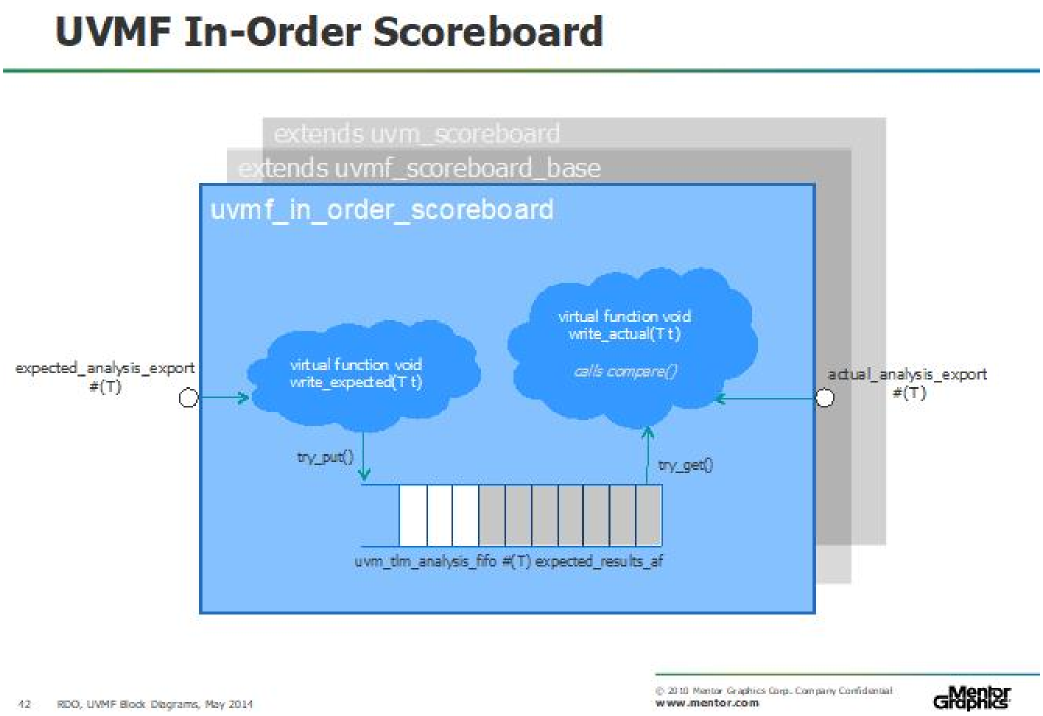

Figure 1.3: UVMF In-Order Scoreboard Class Structure

1.4.7.4 `uvmf_in_order_scoreboard_array.svh`

This scoreboard is used in circumstances where data through a physical channel is divided into multiple logical channels and data order within a logical channel is in order or in a predictable order. The in order scoreboard array uses a queue for each logical channel. The behavior for each channel is identical to the in order scoreboard. The logical channel for each incoming transaction is identified using the `get_key` function of the transaction.

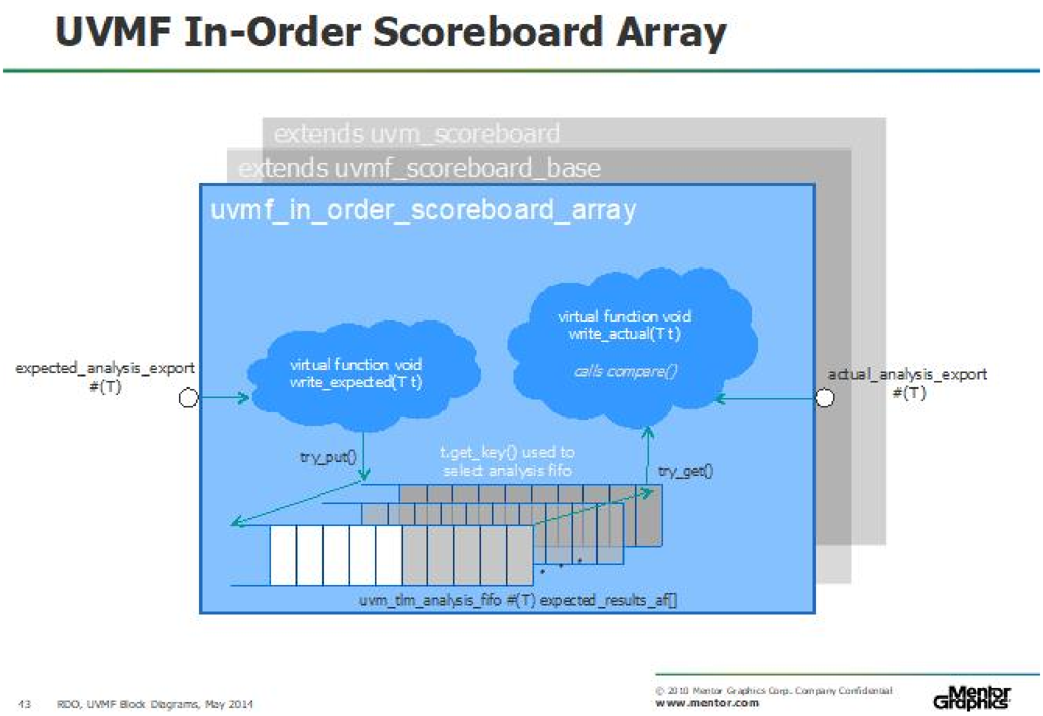

Figure 1.4: UVMF In-Order Scoreboard Array Class Structure

1.4.7.5 `uvmf_out_of_order_scoreboard.svh`

The out of order scoreboard is used in circumstances where data order through the DUT is not guaranteed or not predictable. The out of order scoreboard extends the scoreboard base. It adds a SystemVerilog associative array for storing expected results. Transactions received through the `expected_export` are placed into the associative array using the value returned from the `get_key` function as the key of the entry. Transactions observed on the DUT output are sent to the actual export for comparison. The arrival of a transaction on the actual export causes a transaction to be removed from the associative array and compared to the actual transaction. The `get_key` function of the actual transaction is used to identify a matching transaction in the associative array. An error is generated if the associative array does not have an entry that matches the key returned from the actual transactions `get_key` function.

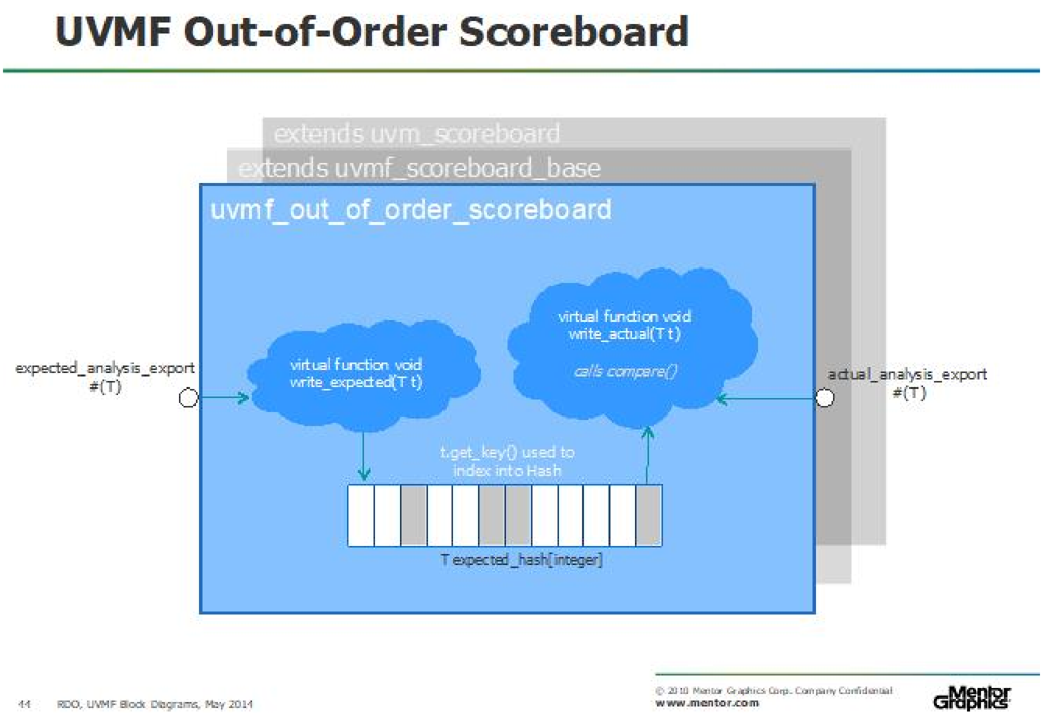

Figure 1.5: UVMF Out-of-Order Scoreboard Class Structure

1.4.7.6 `uvmf_in_order_race_scoreboard.svh`

The in order race scoreboard is used in circumstances where data order through the DUT is preserved and the DUT can send output transactions before input transactions are received. The in order race scoreboard extends the scoreboard base. It adds a queue for the `expected_export` and `actual_export`. When a transaction arrives on either port the other port is checked for an entry to compare against. If an entry exists in the queue for the other port then the entry is pulled from the queue for comparison. If an entry does not exist in the queue for the other port then the entry is queued for later comparison.

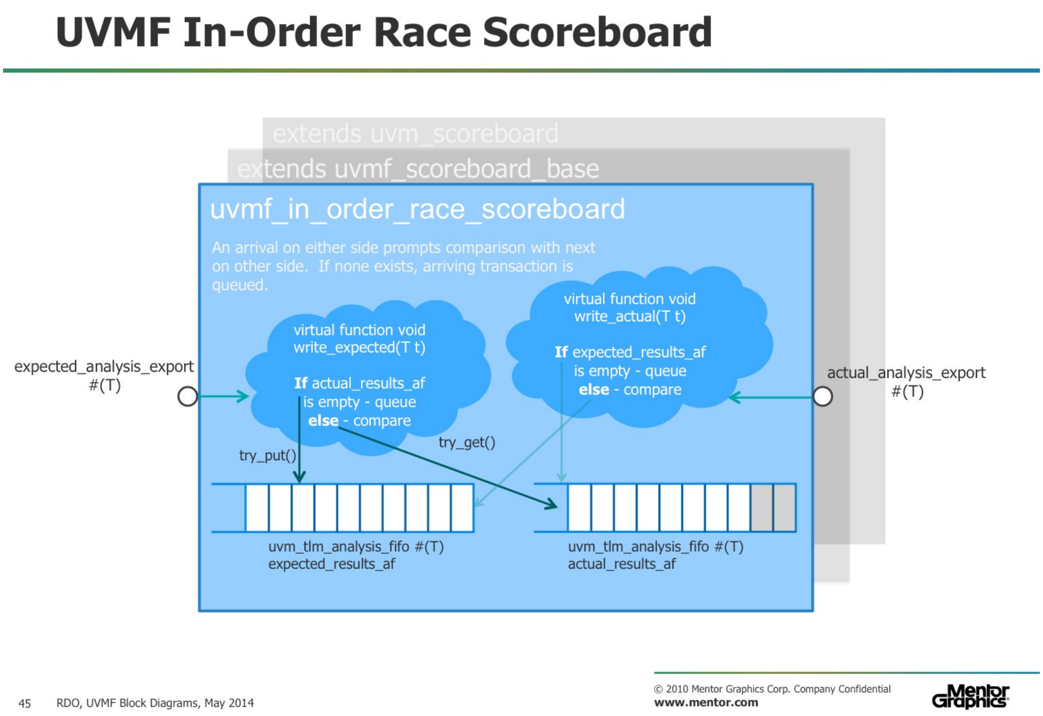

Figure 1.6: UVMF In-Order Race Scoreboard Class Structure

1.4.8 Environment Classes

1.4.8.1 `uvmf_environment_configuration_base.svh`

The `uvmf_environment_configuration_base` is the base environment configuration class for all UVMF environments. It provides flags used for register model integration. It also provides debug features for the initialize call and its handling of agent interface name and agent activity arrays.

1.4.8.2 `uvmf_environment_base.svh`

The `uvmf_environment_base` is the base class for all UVMF based environments. It provides a handle to the environments configuration class.

1.4.8.3 Parameterized Environments

The parameterized environments contained in the `uvmf_base_pkg` are no longer recommended. It is recommended that users generate design specific environments using the UVMF environment generator. The parameterized environments include `uvmf_parameterized_*_environment` classes.

1.4.9 Test Classes

1.4.9.1 `uvmf_test_base.svh`

This is the base class for the base test for all simulation benches. The `uvmf_test_base` instantiates the top level configuration, top level environment and top level sequence. The test class directly extended from `uvmf_test_base` is named `test_top` and must define the parameters for the top level configuration, environment and sequence and calls the `initialize` function of the configuration class. Test top is then extended to create additional test cases by specifying factory overrides.

1.5 UVMF Protocol Interfaces

1.5.1 Protocol Components and Flow Basics

Generally, a protocol is a defined mechanism for providing interaction between two or more entities. Specifically, it is a defined signaling scheme for communicating data between two or more components. Within a protocol, one end initiates data communication and the other end responds to data communication operations. The initiator end initiates a data transfer, waits for a response, then captures the response. The responder end waits for a transfer to be initiated, captures data from the initiation, determines how to respond, then responds to the transfer.

1.5.2 UVMF Protocol Agent and Interface Overview

The components used in UVMF to implement a protocol agent and its associated interfaces can be divided into two categories, dynamic and static. The dynamic components are class based and are defined within the protocol package. They include the agent, agent configuration, driver, monitor, sequencer, coverage, transaction and sequence classes. These components handle transaction object and transaction variable operations. The static components are interface based and include the driver BFM, monitor BFM, and signal bundle. These components handle transaction variable and signal level operations. A video named, âĂIJAgents, Architecture and OperationâĂİ, describes these components and their operation. It is available on verificationacademy.com at the following link: `https://verificationacademy.com/courses/UVM-Framework-One-Bite-at-a-Time`

1.6 UVMF Environment Overview

Figure 1.7 shows a block level environment. It is from the `wb2spi` example. Block level environments include agents, predictors, scoreboards, coverage collectors and other components that are connected based on design data flow.

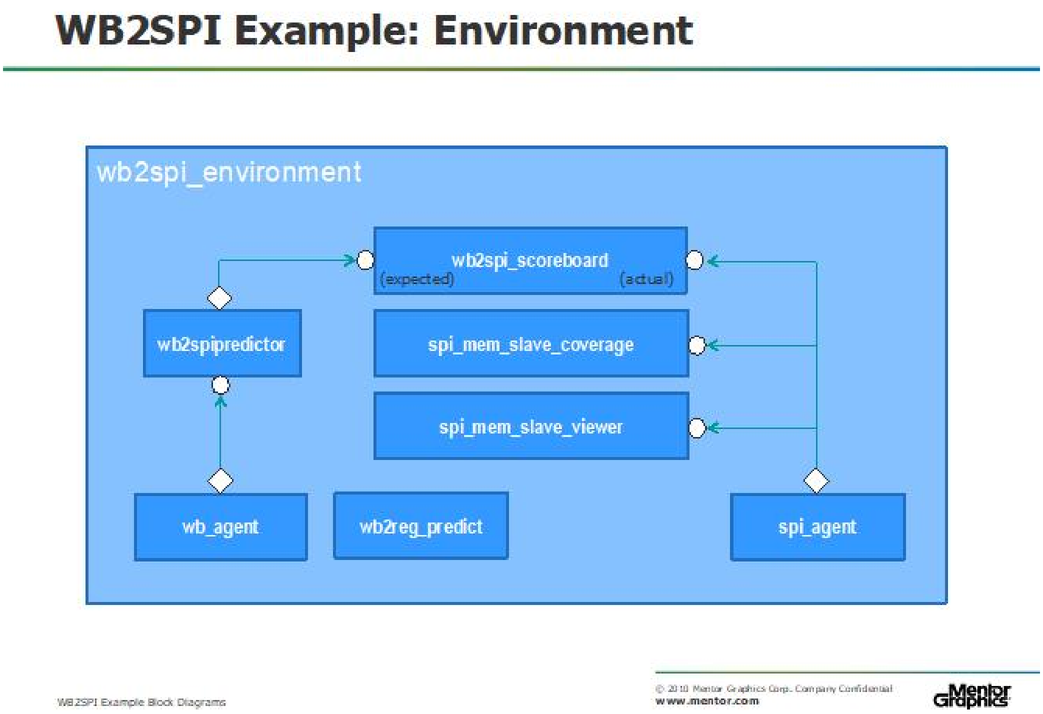

Figure 1.7: WB2SPI Example Environment Structure

Figure 1.8 shows a chip level environment. It is from the `ahb2spi` example. Chip level environments include other environments. This is true for all environments other than block level environments. Environments are structured in a hierarchical manner similar to how RTL blocks are composed hierarchically. This allows for environment reuse as RTL components are reused.

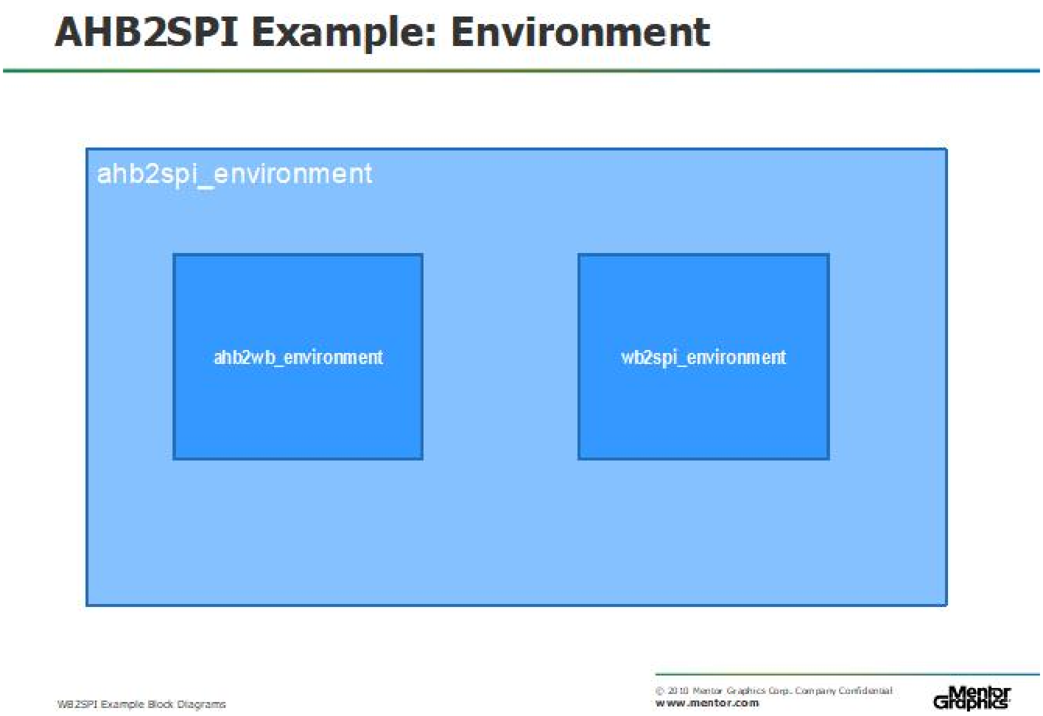

Figure 1.8: AHB2SPI Example Environment Structure

1.7 UVMF Simulation Bench Overview

The structure of the simulation bench is shown in figure 1.9. It is from the `ahb2wb` simulation bench. The three top level elements in every UVMF simulation bench are `hdl_top`, `hvl_top` and `test_top`. Synthesizable content that may be run in emulation is placed in `hdl_top`. This includes the DUT, driver BFMs, monitor BFMs, signal bundle interfaces and any other logic required by the DUT. The non-synthesizable elements that must be in a module are placed in `hvl_top`. This includes the test package import and the call to the UVM `run_test` task to start the UVM phases. The top level configuration, top level environment and top level sequence are placed in `test_top`. This defines the base test from which all other tests are derived.

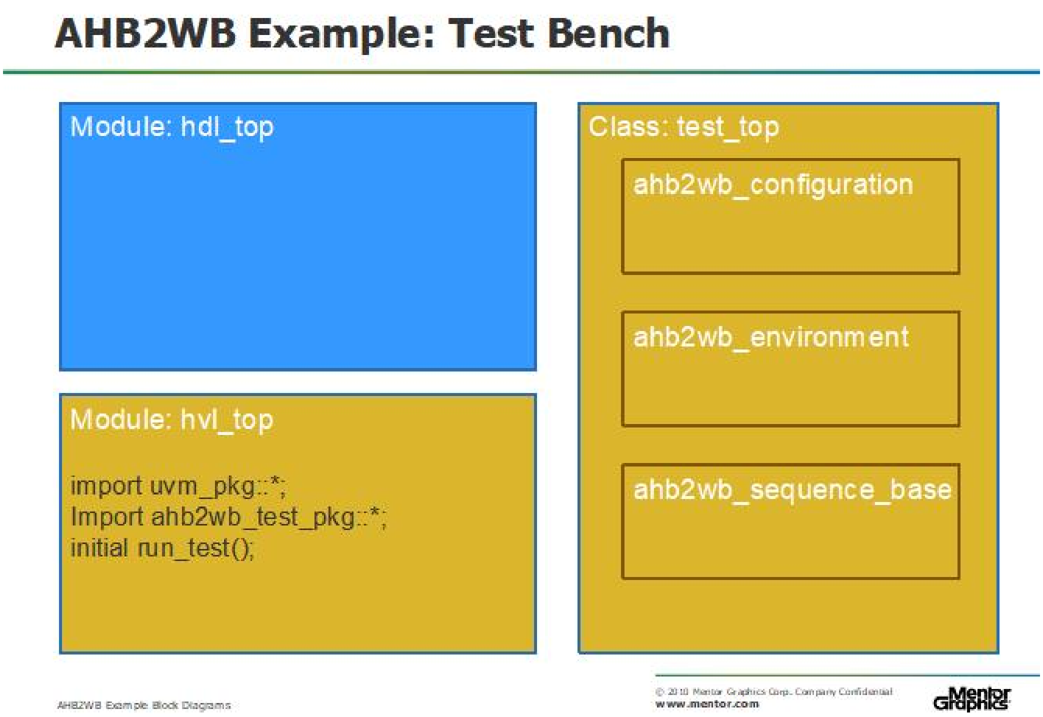

Figure 1.9: AHB2WB Example Bench Structure

Chapter 2

UVM Framework Examples

2.1 Example Benches

2.1.1 AHB to Wishbone Example

The AHB2WB example demonstrates a block level environment. It is located in the `base_examples` group. This block level environment is reused in the AHB2SPI chip level environment example. This example also demonstrates the use of a parameterized environment. The use of a parameterized environment alleviates the need to write an environment class. This example demonstrates the following:

1. Block level environment that will be reused at the chip level
2. Use of a parameterized environment
3. Test plan import
4. Merging of test results
5. Generation of a custom coverage report

A specification for the `ahb2wb` DUT can be found in the `docs` folder of the example

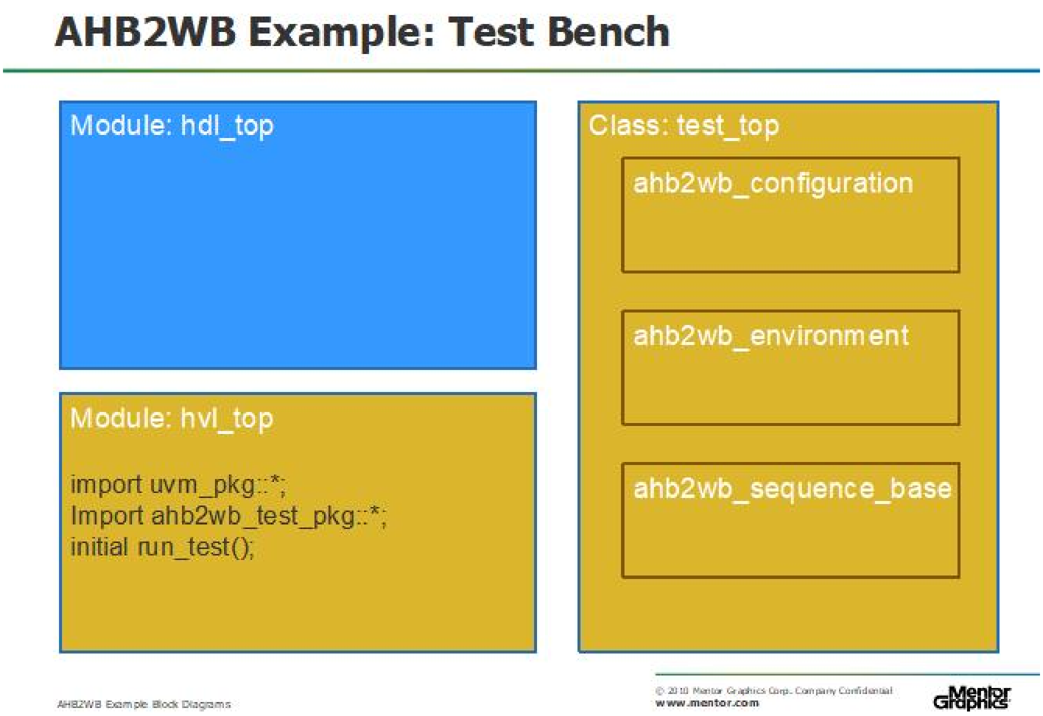

Figure 2.1: AHB2WB Bench Example

As with all UVMF test benches, the AHB2WB test bench is composed of three top levels: `hdl_top`, `hvl_top` and `test_top`. The module named `hdl_top` contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in `hdl_top` is synthesizable to support emulation. The module named `hvl_top` contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling `run_test` to start the UVM phases. The class named `test_top` is the top level UVM test class. It is selected using the `+UVM_TESTNAME` argument on the command line and constructed by the UVM factory.

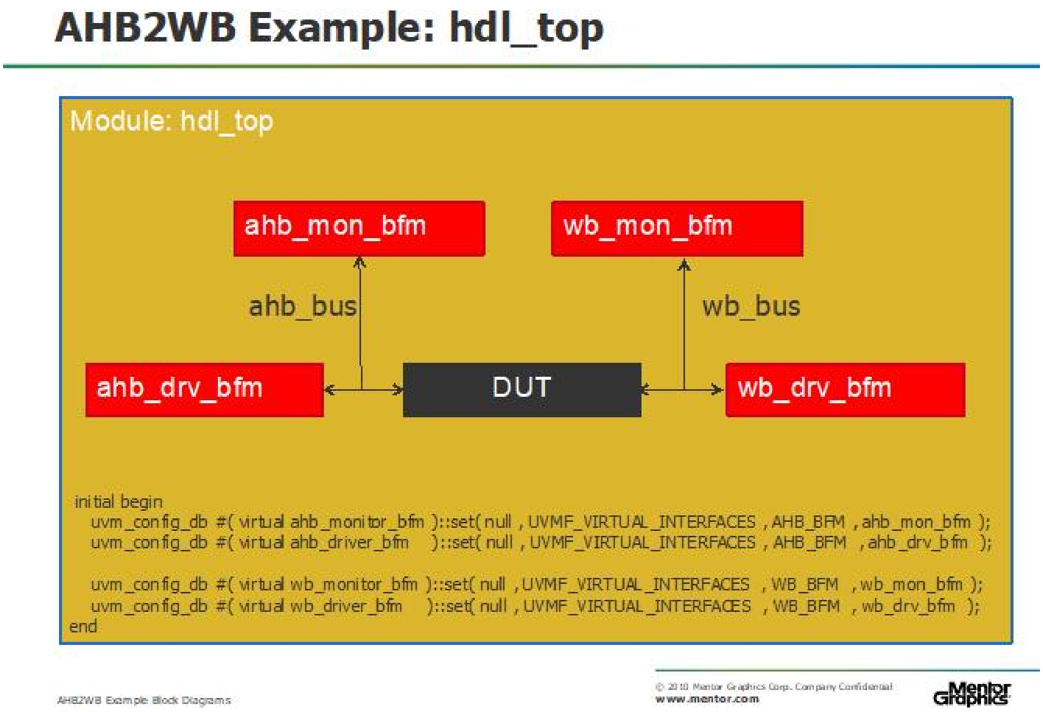

Figure 2.2: AHB2WB hdl_top

The `ahb2wb_configuration` class contains a configuration for each agent in the environment. A function named initialize provides the agent configurations with the active/passive state of the agent, the path to its agent in the environment and the string name of the interface to be retrieved from the uvm_config_db. The ahb2wb configuration also contains DUT configuration specific variables that can be randomized as needed. The ahb2wb configuration class is constructed, randomized and initialized before the environment build phase is executed. This ensures that the environment can be built according to the configuration for the simulation.

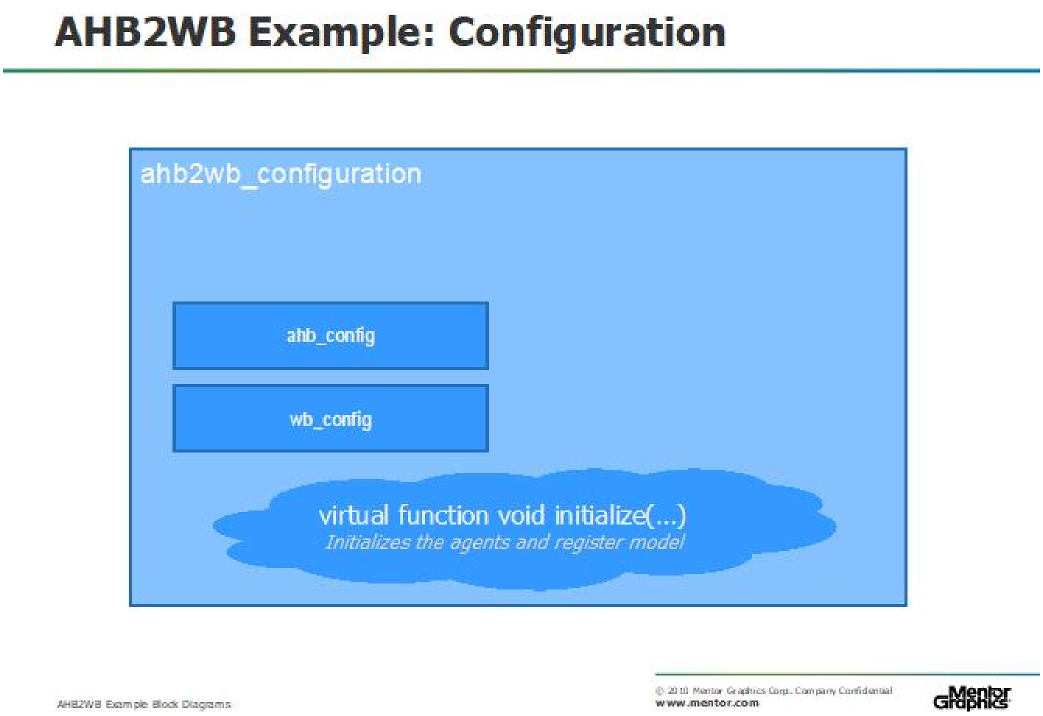

Figure 2.3: AHB2WB Configuration

The ahb2wb environment utilizes the `uvmf_parameterized_2agent_environment`. This alleviates the need to write an environment class. The ahb2wb environment is a type specialization of the parameterized environment. Creating a type specialization of the parameterized agents only requires the creation of a typedef. The typedefs used in the ahb2wb example can be found in `./verification_-ip/environment_packages/ahb2wb_env_pkg/src/ahb2wb_environment.svh`

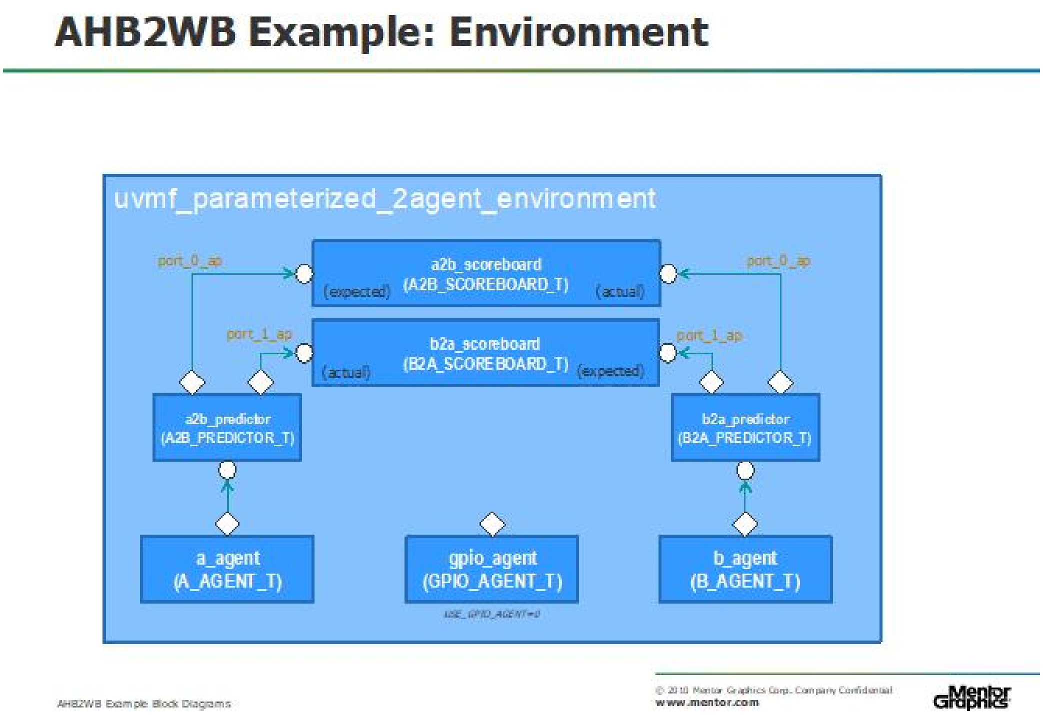

Figure 2.4: AHB2WB Environment

The top level sequence, named **`ahb2wb_sequence_base`**, orchestrates and controls all stimuli within the simulation. The stimulus flow is shown in figure 2.5. The first sequence to be started is the ahb reset sequence. This causes the **`ahb_drv_bfm`** to assert and then release reset. Once the reset sequence has completed the wishbone memory slave sequence is started. This sequence is forked off because it will remain active throughout the simulation. This is because a slave device is always active and ready to respond to activity initiated by the master. Once the slave sequence is forked a series of writes and reads are performed on the ahb2wb DUT.

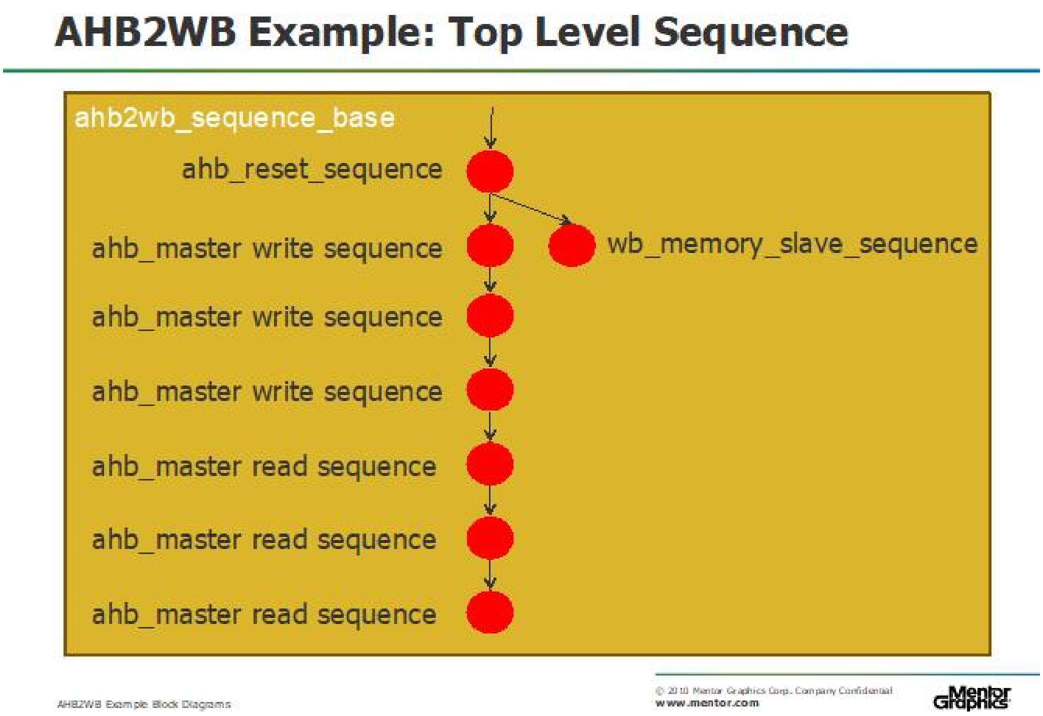

Figure 2.5: AHB2WB Top Level Sequence

2.1.2 WB to SPI Example

The WB2SPI example demonstrates a block level environment. It is located in the `base_examples` group. This environment includes a register model based on the UVM register package. This block level environment is reused in the AHB2SPI chip level environment example. A specification for the wb2spi DUT can be found in the doc folder of the example. This example demonstrates the following:

1. Block level environment that will be reused at the chip level
2. Block level UVM register model that will be reused at the chip level

As with all UVMF test benches, the WB2SPI test bench is composed of three top levels: `hdl_top`, `hvl_top` and `test_top`. The module named `hdl_top` contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in `hdl_top` is synthesizable to support emulation. The module named `hvl_top` contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling `run_test` to start the UVM phases. The class named `test_top` is the top level UVM test class. It is selected using the `+UVM_TESTNAME` argument on the command line and constructed by the UVM factory.

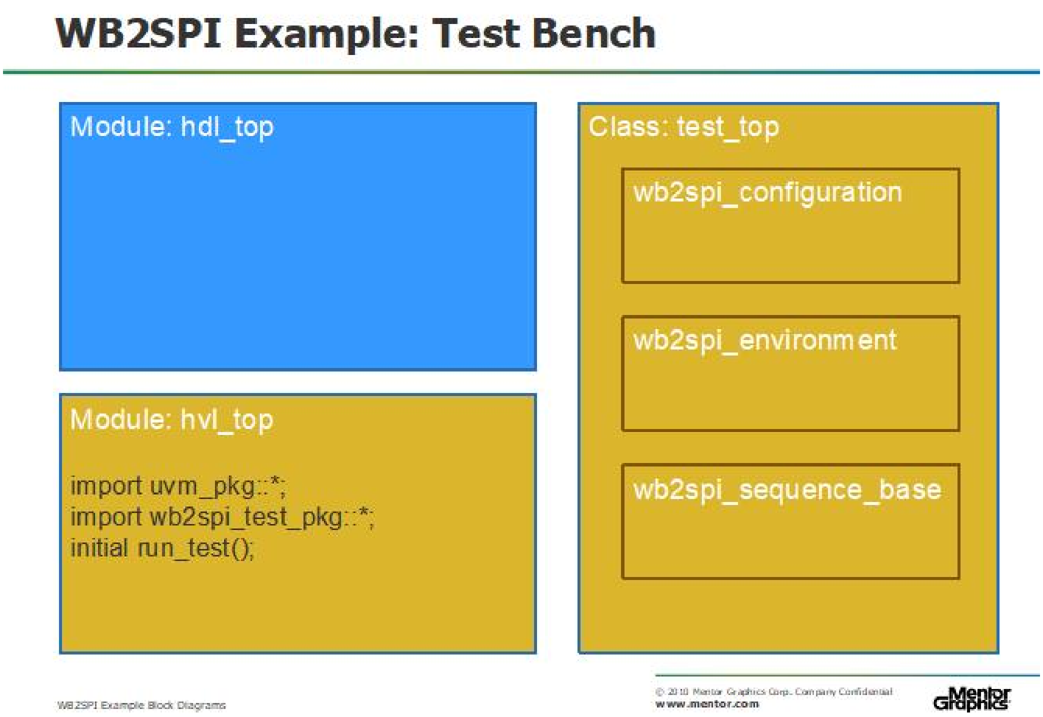

Figure 2.6: WB2SPI Bench

The `hdl_top` module contains the DUT and BFMs used to drive and monitor bus activity. The `_drv_bfm` interfaces provide signal stimulus. The `_mon_bfm` interfaces observe signal activity and capture signal information for broadcasting to the environment for prediction, scoreboarding and coverage collection. An interface containing all of the signals for the bus ties the monitor BFM, driver BFM and DUT signal ports together. Protocol signals are separated into an interface to enable block to top reuse of environments and monitor BFMs. All BFMs are placed into the uvm_config_db by `hdl_top` for retrieval by the appropriate agent configuration. This mechanism is described in the section on resource sharing and initialization within UVM Framework.

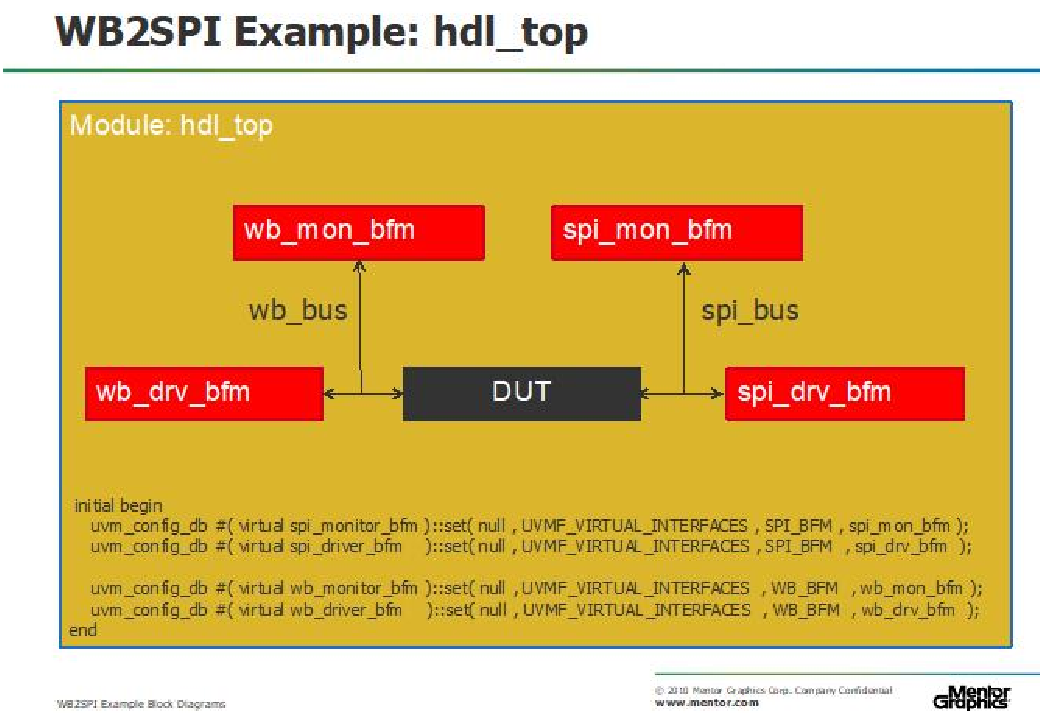

Figure 2.7: WB2SPI HDL Top

The `wb2spi_configuration` class contains a configuration for each agent in the environment and the register model. The register model is a UVM register block for the wb2spi DUT. This register block will be a sub block of the ahb2spi register block used in the chip level simulation. A function named `initialize` provides the agent configurations with the active/passive state of the agent, the path to its agent in the environment and the string name of the interface to be retrieved from the uvm_config_db. The wb2spi configuration also contains DUT configuration specific variables that can be randomized as needed. The wb2spi configuration class is constructed, randomized and initialized before the environment build phase is executed. This ensures that the environment can be built according to the configuration for the simulation.

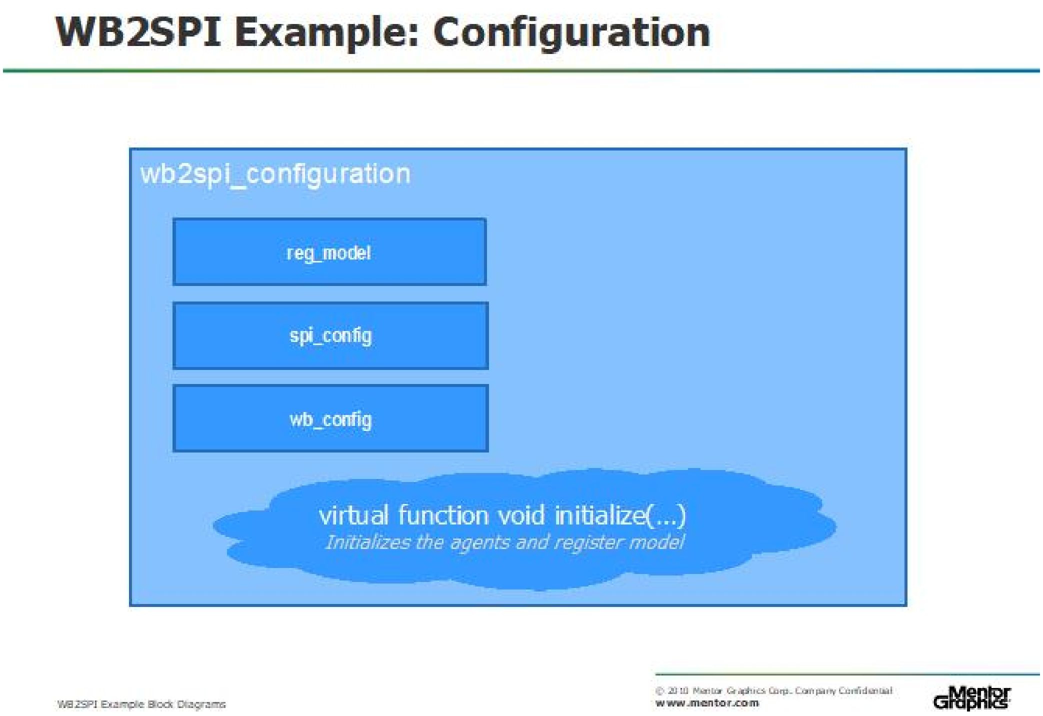

Figure 2.8: WB2SPI Configuration

The wb2spi environment contains the agents, predictor, scoreboard, coverage and transaction viewing components shown in the above diagram. The wishbone agent interacts with the `wb_drv_bfm` and `wb_mon_bfm`. It receives stimulus information from sequences in the top level sequence. Bus operations are observed and broadcasted to the wb2spi predictor. The predictor creates an expected SPI transaction based on DUT configuration and input from the wishbone bus. Output from the wb2spi predictor are sent to the scoreboard to be queued until DUT activity is received for comparison. The SPI agent interacts with the `spi_drv_bfm` and `spi_mon_bfm`. It receives stimulus information from sequences in the top level sequence. Bus operations are observed and broadcasted to the wb2spi scoreboard, coverage component and transaction viewing component. The coverage component records functional coverage of SPI operations. The transaction viewing component provides a transaction viewing stream of SPI memory slave transactions. The monitor within the SPI agent provides transaction viewing of the base SPI transfer. This allows for viewing of raw SPI transfers as well as the functional meaning of each bit within the raw SPI transfer.

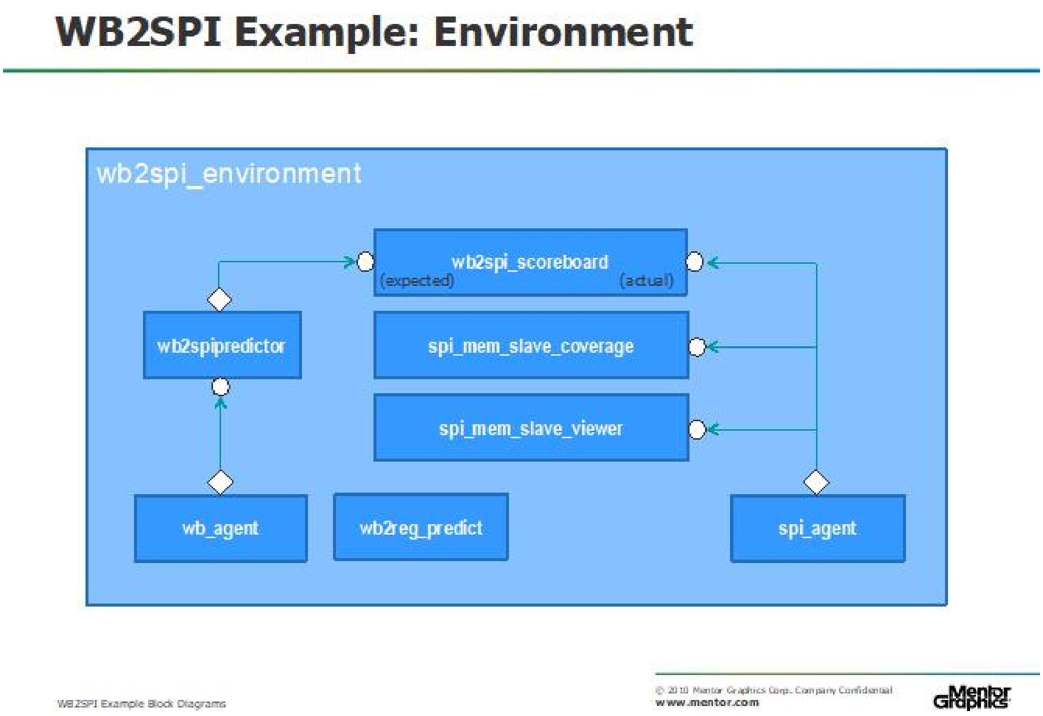

Figure 2.9: WB2SPI Environment

The top level sequence, named `wb2spi_sequence_base`, orchestrates and controls all stimuli within the simulation. The stimulus flow is shown in Figure 2.10.

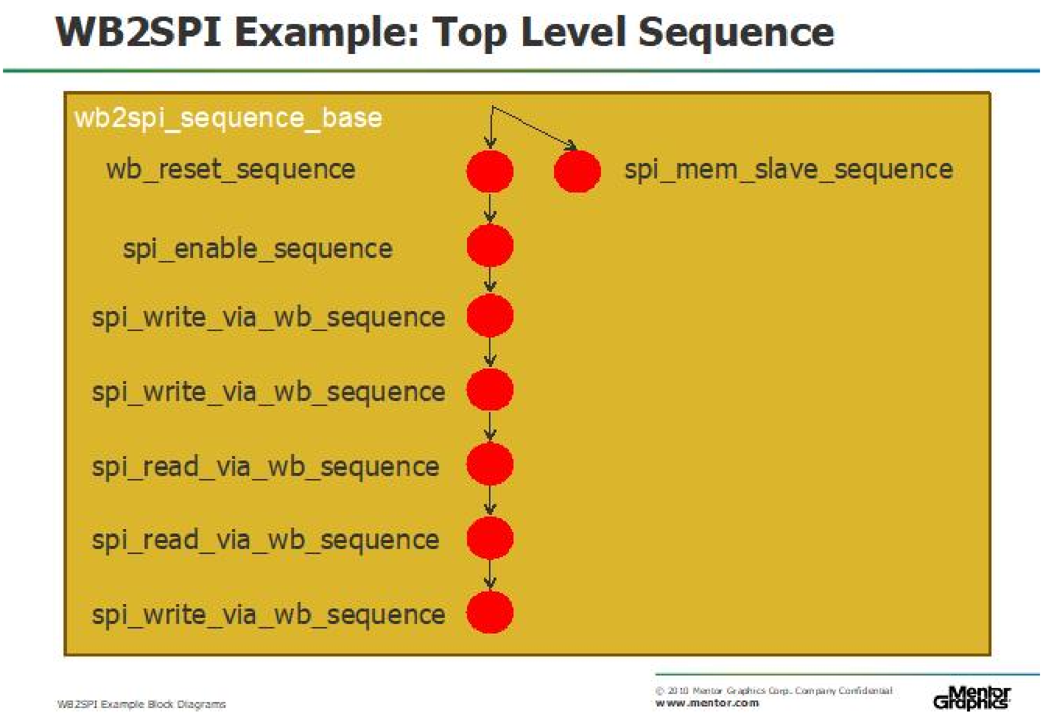

Figure 2.10: WB2SPI Top Level Sequence

The first sequence to be started is the SPI memory slave sequence. This sequence is forked off at the beginning because it will remain active throughout the simulation. This is because a slave device is always active and ready to respond to activity initiated by the master. Once the slave sequence is forked the wishbone reset sequence is started. This causes the `wb_drv_bfm` to assert and then release reset. Once the reset sequence has completed a series of writes and reads are performed on the wb2spi DUT. The format of the SPI transfer as a memory slave is shown in the table below.

SPI Slave Bus Protocol			
Signal	Data[7]	Data[6:4]	Data[3:0]
MOSI	RW 1:Write, 0:Read	Address[2:0]	Data[3:0]
MISO	STATUS (prev command) 1:Success, 0:Error	Address[2:0]	Data[3:0]

2.1.3 AHB to SPI Example

The AHB2SPI example demonstrates a chip level environment. It is located in the `base_examples` group. This chip level environment reuses the AHB2WB and WB2SPI block level environments. This environment includes a register model based on the UVM register package. This chip level register model contains a register block for the WB2SPI block level environment. This example demonstrates the following:

1. Chip level environment that reuses block level environments
2. Chip level UVM register model that reuses a block level UVM register model

As with all UVMF test benches, the AHB2SPI test bench is composed of three top levels: `hdl_top`, `hvl_top` and `test_top`. The module named `hdl_top` contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in `hdl_top` is synthesizable to support emulation. The module named `hvl_top` contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling `run_test` to start the UVM phases. The class named `test_top` is the top level UVM test class. It is selected using the `+UVM_TESTNAME` argument on the command line and constructed by the UVM factory.

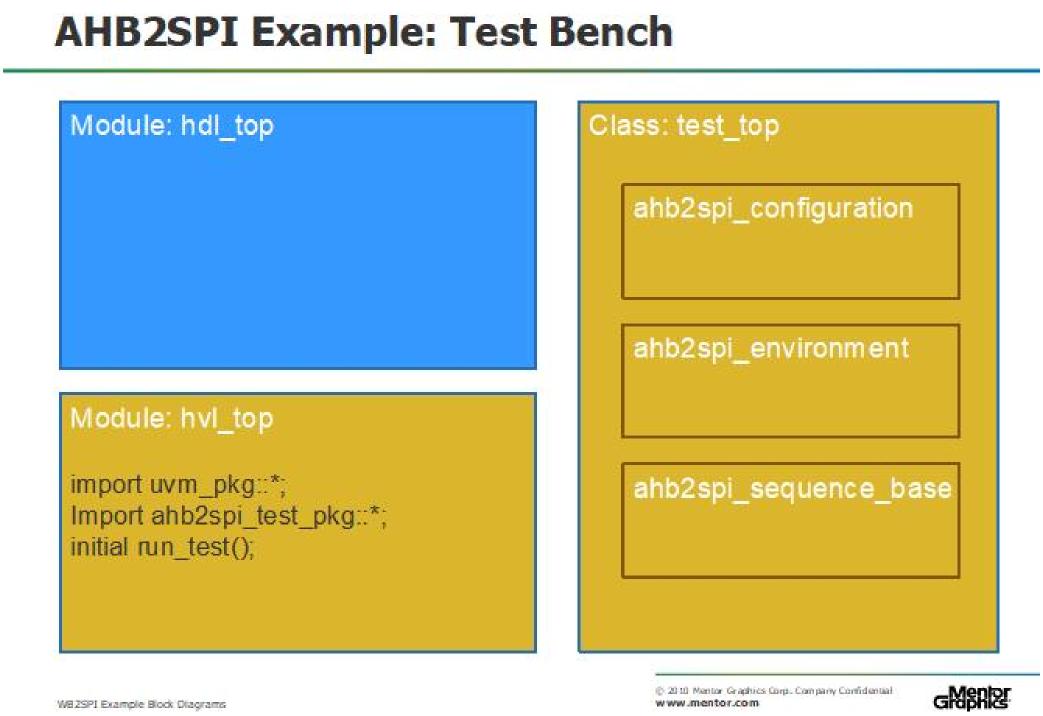

Figure 2.11: AHB2SPI Test Bench

The `hdl_top` module contains the DUT and BFMs used to drive and monitor bus activity. The `_drv_bfm` interfaces provide signal stimulus. The `_mon_bfm` interfaces observe signal activity and capture signal information for broadcasting to the environment for prediction, scoreboarding and coverage collection. An interface containing all of the signals for the bus ties the monitor BFM, driver BFM and DUT signal ports together. Protocol signals are separated into an interface to enable block to top reuse of environments and monitor BFMs. All BFMs are placed into the uvm_config_db by `hdl_top` for retrieval by the appropriate agent configuration. This mechanism is described in the section on resource sharing and initialization within UVM Framework. In this example the wishbone bus is internal to the DUT and driven by RTL within the DUT. A wishbone signal bundle interface, `wb_bus`, is connected to the wishbone bus in the DUT in order to observe bus activity. This can be done using either the SystemVerilog bind construct or hierarchically connecting the signal bundle into the DUT. This wishbone signal bundle is connected to two wishbone monitor BFM. This is to provide the wishbone agent within each of the block level environments a wishbone monitor BFM virtual interface handle. This allows independent prediction, scoreboarding and coverage for each block level environment.

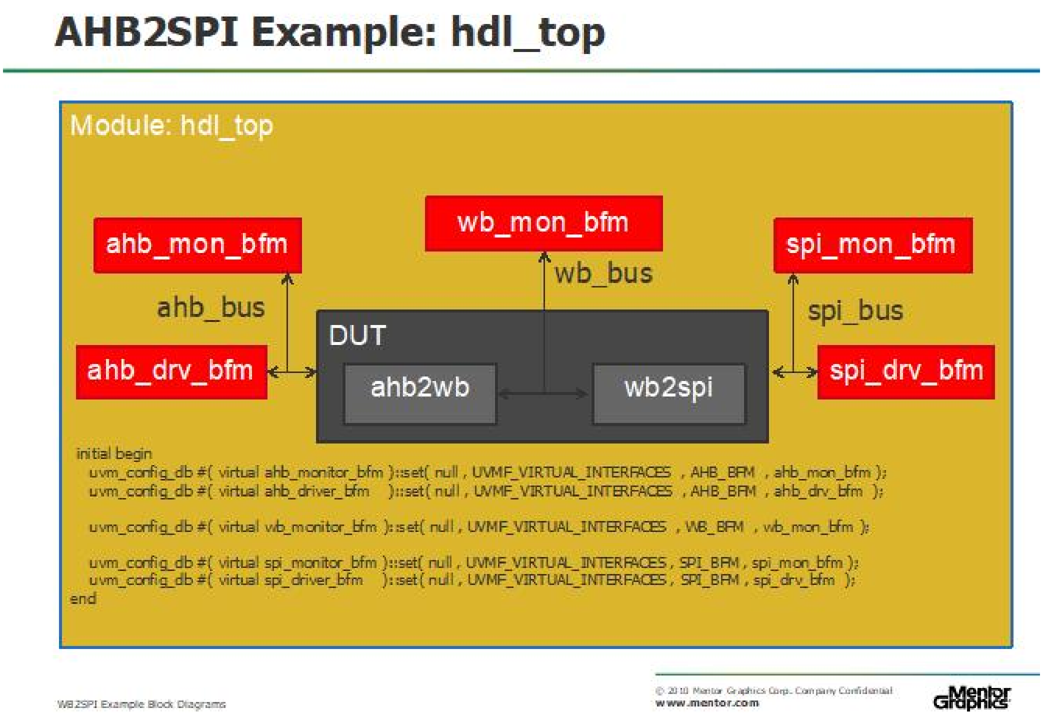

Figure 2.12: AHB2SPI hdl_top

The `ahb2spi_configuration` class contains a configuration for each block level environment within this chip level environment and a register model. The register model is a UVM register block for the ahb2spi DUT. This register block contains a wb2spi register block for use by the wb2spi block level environment. A function named `initialize` provides the environment configurations with the simulation level, `BLOCK/CHIP`, the hierarchical path down to the chip level environment and an array of string names of the interface to be retrieved from the uvm_config_db. The ahb2spi configuration also contains DUT configuration specific variables that can be randomized as needed. The ahb2spi configuration class is constructed, randomized and initialized before the environment build phase is executed. This ensures that the environment can be built according to the configuration for the simulation. Randomization occurs down through the configuration classes. The `post_randomize` of `ahb2spi_configuration` randomizes `ahb2wb_configuration` and `wb2spi_configuration` applying implication constraints to enforce lower level value options based on upper level values randomly selected.

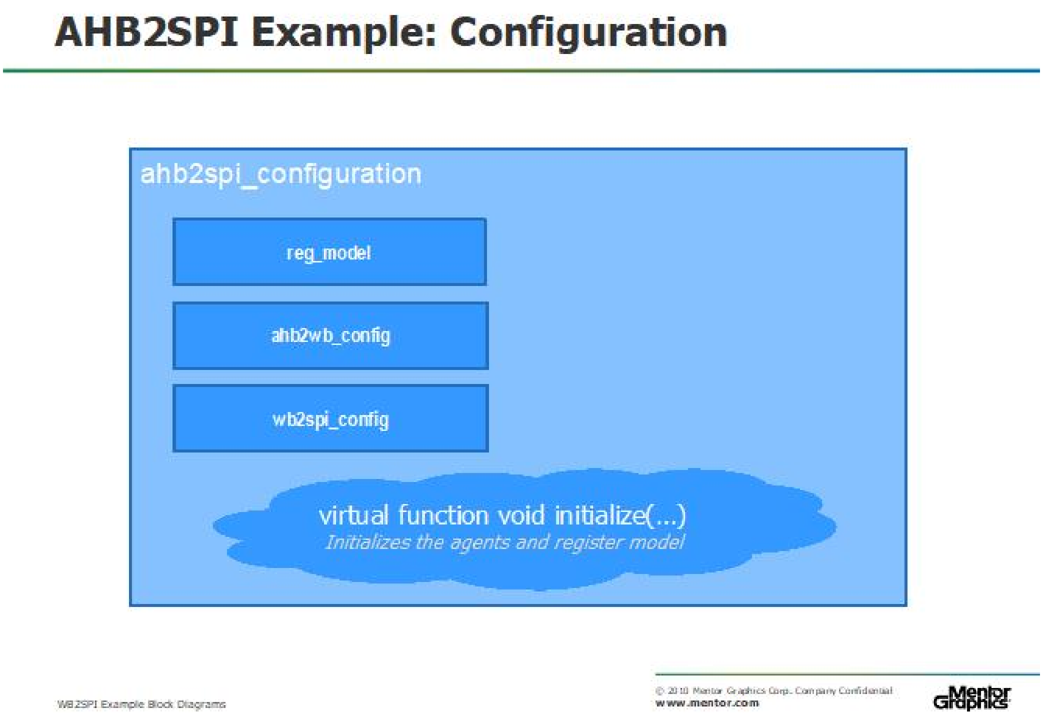

Figure 2.13: AHB2SPI Configuration

The ahb2spi environment contains the ahb2wb environment and the wb2spi environment. These two block level environments perform the same prediction, scoreboarding and coverage provided when run in the block level bench. Stimulus is driven into the design via the ahb interface. The wishbone interface is now buried in the DUT. It is observed by both environments for prediction, scoreboarding and coverage. Data is sent out through the SPI interface of the DUT. The `ahb2wb_environment` is configured by the `ahb2spi_configuration`. The `ahb2wb_environment` is configured by the `ahb2wb_configuration`. The `wb2spi_environment` is configured by the `wb2spi_configuration`. The ahb2spi environment creates a `wb_monitor` to be shared between the two environments that need to observe the wishbone bus. This wishbone monitor is constructed by the ahb2spi environment and placed into the uvm_config_db for retrieval by the wb agent within each block level environment. The shared `wb_monitor` is connected to the single `wb_monitor_bfm`. WB transactions observed by the `wb_monitor_bfm` are sent to the shared `wb_monitor` and broadcasted within each environment.

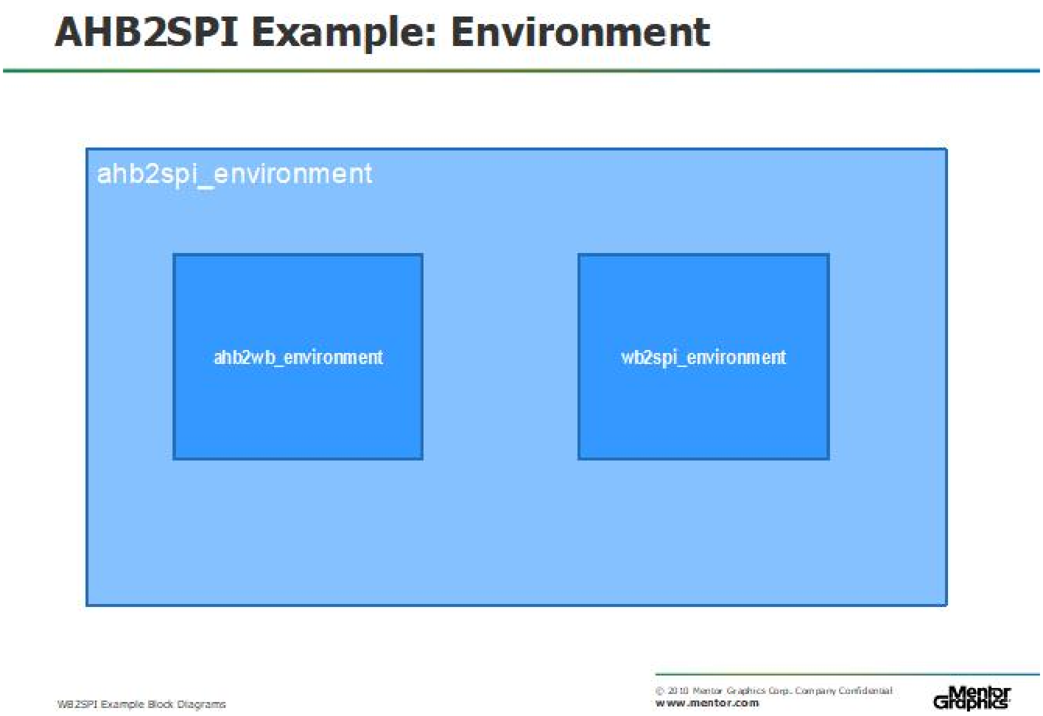

Figure 2.14: AHB2SPI Environment

The top level sequence, named `ahb2spi_sequence_base`, orchestrates and controls all stimuli within the simulation. The stimulus flow is shown in the diagram above. The first sequence to be started is the ahb reset sequence. This causes the `ahb_drv_bfm` to assert and then release reset. Once the reset sequence has completed then the SPI memory slave sequence is started. This sequence is forked off because it will remain active throughout the simulation. This is because a slave device is always active and ready to respond to activity initiated by the master. Once the slave sequence is forked a series of writes and reads are performed on the DUT through the ahb port. These operations write and read the SPI memory slave attached to the SPI port of the DUT.

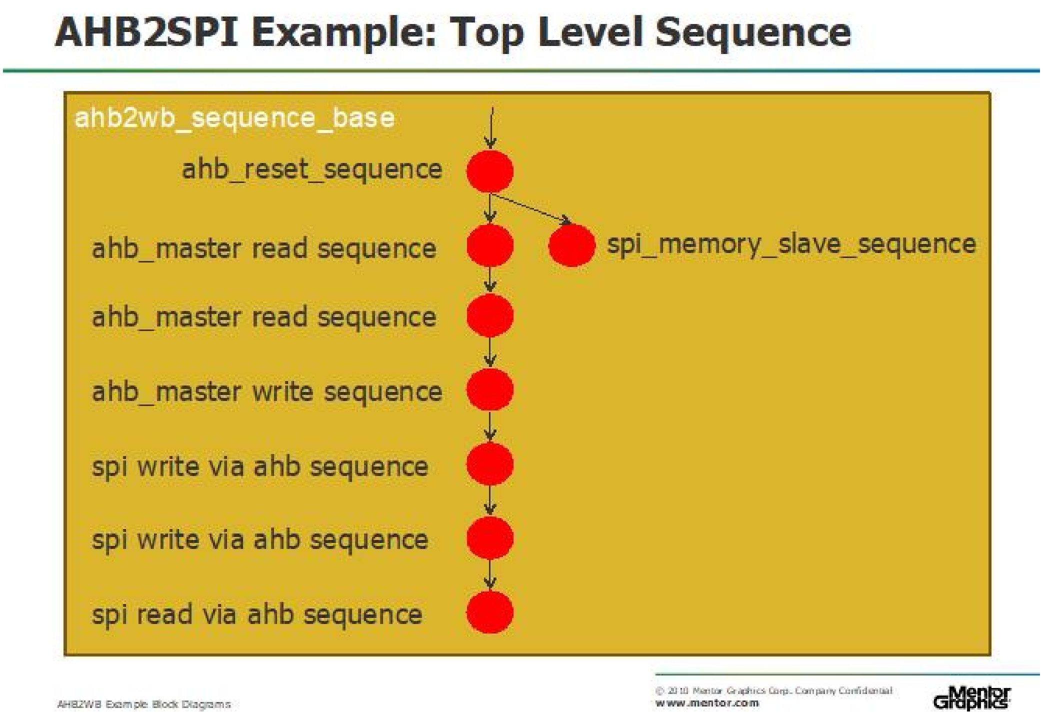

Figure 2.15: AHB2SPI Top Level Sequence

2.1.4 GPIO Example

The GPIO example demonstrates the use of a parameterized interface. The `WRITE_PORT_WIDTH` and `READ_PORT_WIDTH` parameters are used to instantiate the BFM interfaces, agent, configuration, and sequence classes. The value for these parameters are defined in the `gpio_example_parameters_pkg`. This example demonstrates the following:

1. The creation and instantiation of a parameterized interface

As with all UVMF test benches, the GPIO test bench is composed of three top levels: `hdl_top`, `hvl_top` and `test_top`. The module named `hdl_top` contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in `hdl_top` is synthesizable to support emulation. The module named `hvl_top` contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling run_test to start the UVM phases. The class named `test_top` is the top level UVM test class. It is selected using the `+UVM_TESTNAME` argument on the command line and constructed by the

UVM factory.

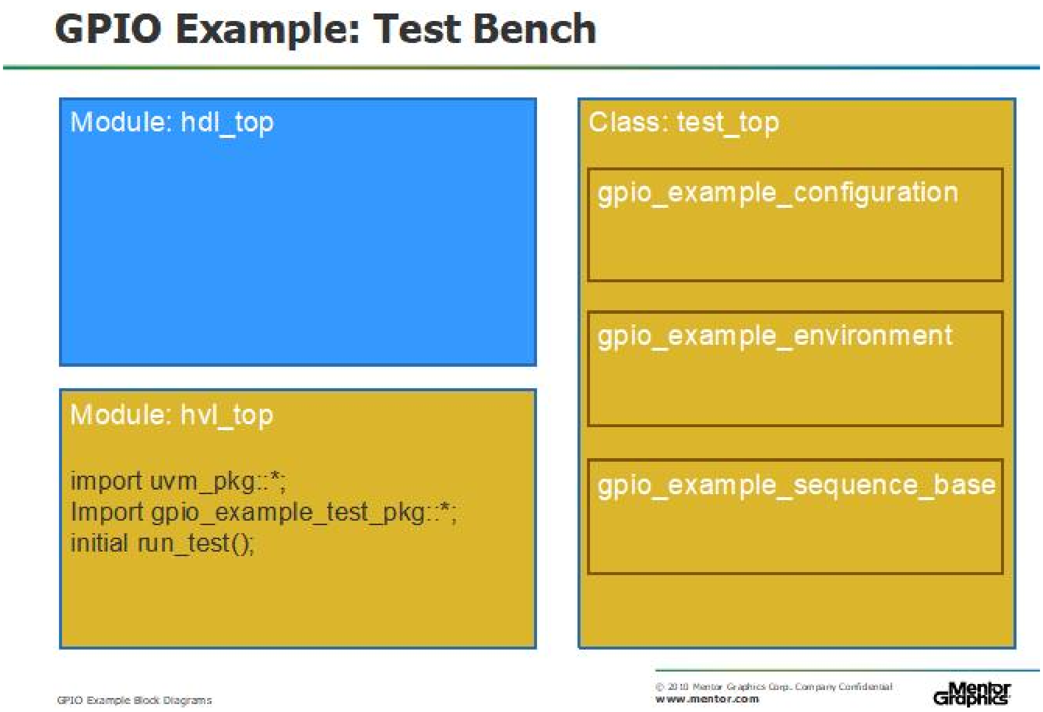

Figure 2.16: GPIO Bench

The `hdl_top` module contains the DUT and BFMs used to drive and monitor bus activity. The `_drv_bfm` interfaces provide signal stimulus. The `_mon_bfm` interfaces observe signal activity and capture signal information for broadcasting to the environment for prediction, scoreboarding and coverage collection. An interface containing all of the signals for the bus ties the monitor BFM, driver BFM and DUT signal ports together. Protocol signals are separated into an interface to enable block to top reuse of environments and monitor BFMs. All BFMs are placed into the uvm_config_db by `hdl_top` for retrieval by the appropriate agent configuration. This mechanism is described in the section on resource sharing and initialization within UVM Framework. In this example the DUT is a simple register named `read_port_`. The input to the register is the `write_port` of the `gpio_bus`. On each clock edge the value on `write_port` is output on `read_port_`. The `read_port_` value is then assigned to the `read_port` of the `gpio_bus`. This inserts a one clock delay between the `write_port` output and `read_port` input. This loopback delay is only for demonstration purposes.

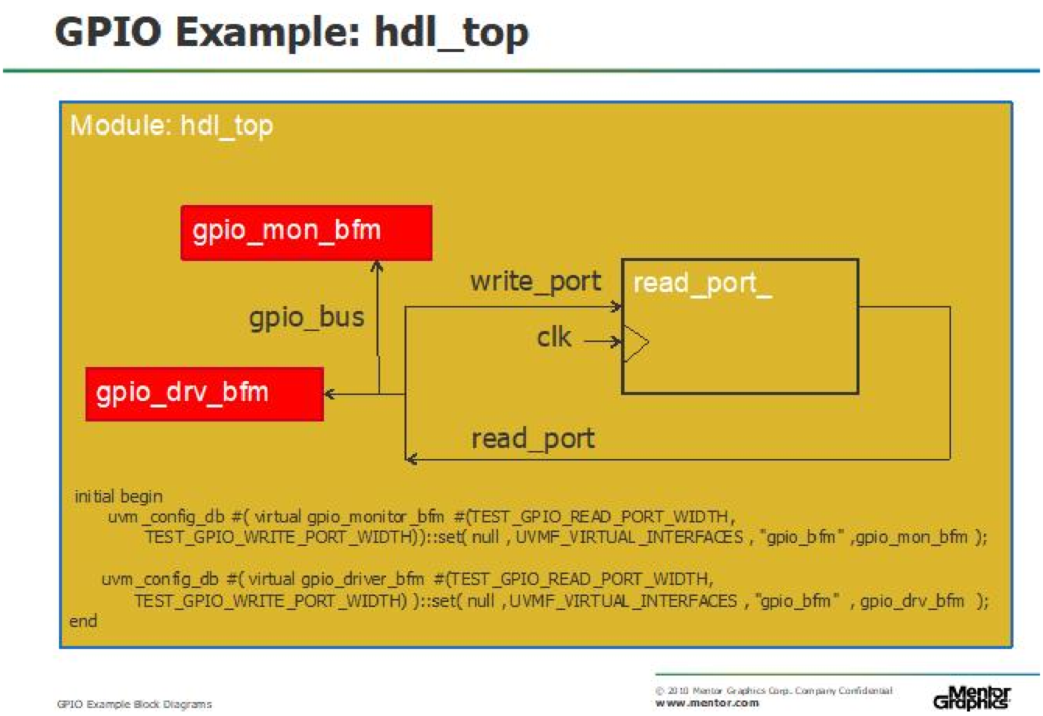

Figure 2.17: GPIO hdl_top

The `gpio_example_configuration` class contains a configuration object for the GPIO agent in the environment. A function named `initialize` provides the agent configuration with the active/passive state of the agent, the path to its agent in the environment and the string name of the interface to be retrieved from the uvm_config_db.

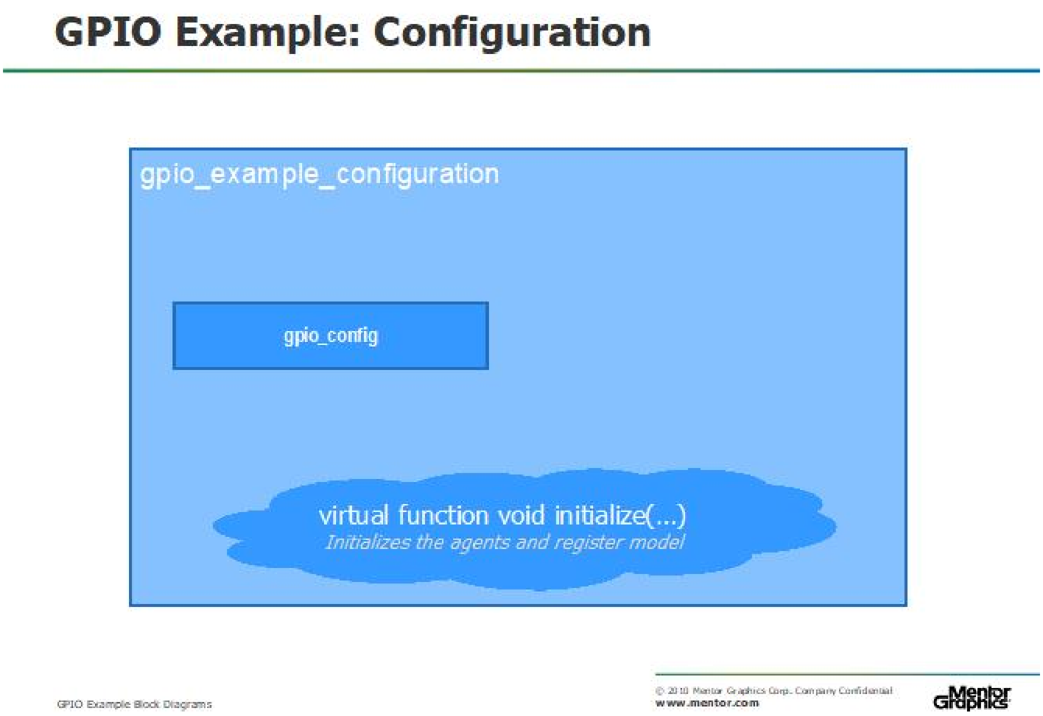

Figure 2.18: GPIO Configuration

The `gpio_example_environment` only contains the `gpio_agent`. This is because the purpose of this example is to demonstrate the use of a parameterized interface, agent, configuration and sequence.

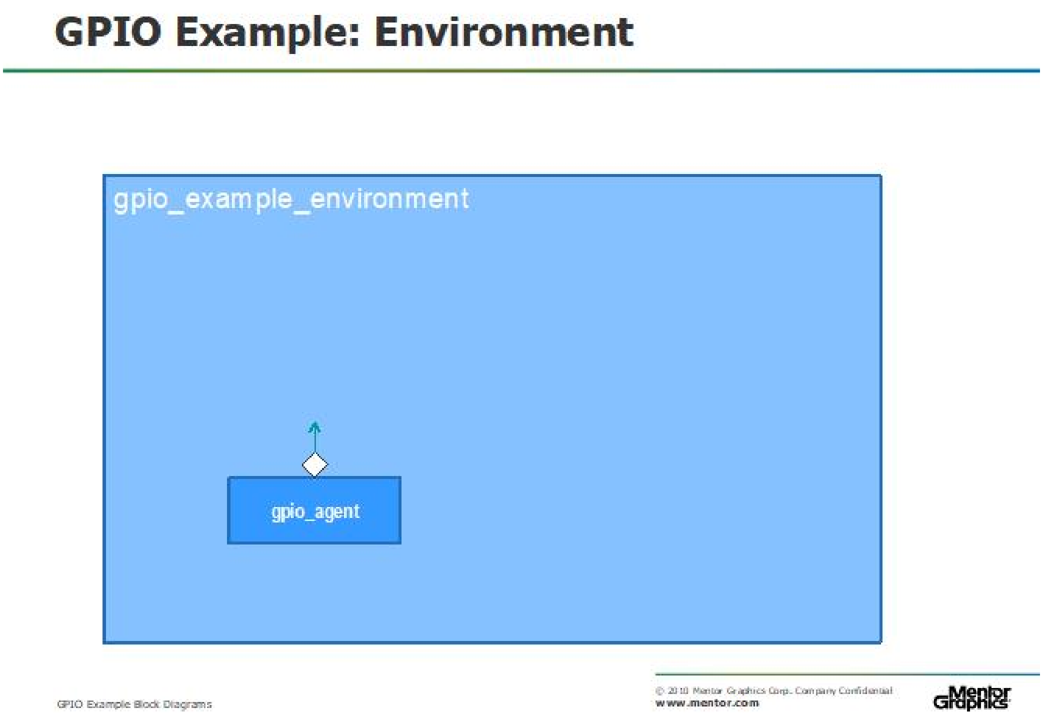

Figure 2.19: GPIO Environment

The sequence used in this example is different from most sequences in that it is started at the beginning of the simulation and remains throughout the simulation. Writing values to the GPIO `write_port` and reading values from the GPIO `read_port` are done through tasks in this sequence. The sequence, named `gpio_seq`, is an extension to the `gpio_sequence` located in the `gpio_pkg`. This extension defines bit assignments to the `write_port` and `read_port`. In this case `bus_a` and `bus_b` are assigned to the `write_port`, `bus_c` and `bus_d` are assigned from the `read_port`.

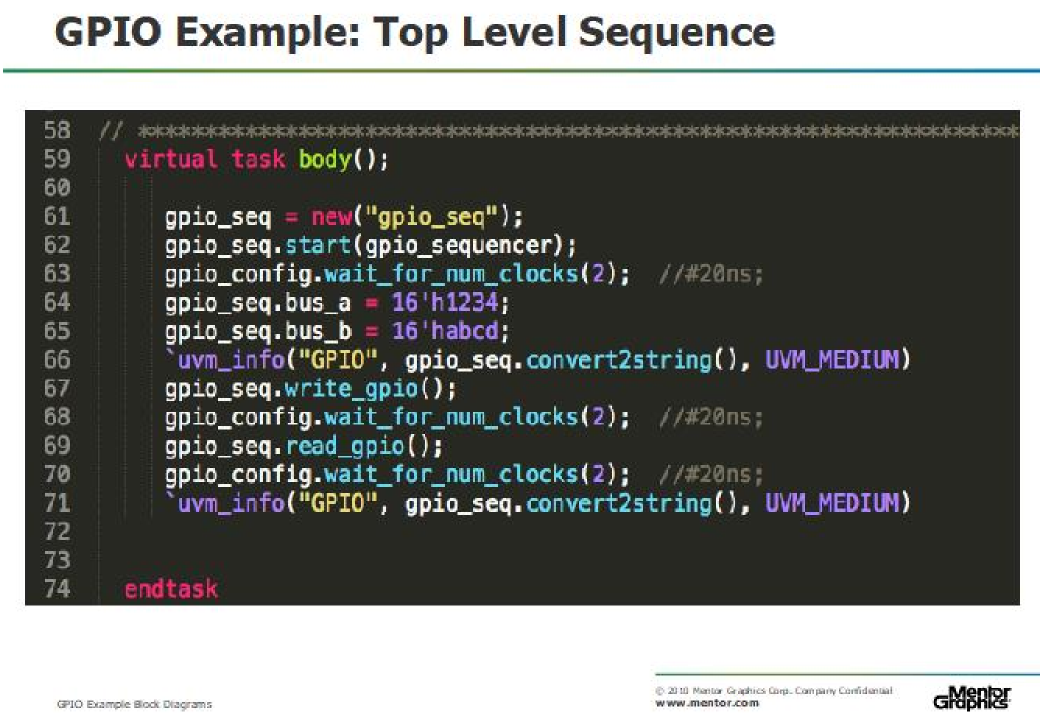

Figure 2.20: GPIO Top Level Sequence

The flow of the top level sequence is outlined below: Initialization:

- Line 61: Construct the `gpio_seq` sequence.
- Line 62: Start the `gpio_seq` sequence. This sequence remains resident throughout the simulation.
- Line 63: Wait for two clocks using the `wait_for_num_clocks` task within the `gpio_agents` configuration class.

Write operation:

- Line 64 and 65: Set the values of `bus_a` and `bus_b` variables.
- Line 66: Display the variable values in the sequence item within `gpio_seq`.
- Line 67: Write the new values of `bus_a` and `bus_b` to the GPIO `write_port`
- Line 68: Wait for two clocks using the `wait_for_num_clocks` task within the `gpio_agents` configuration class.

Read operation:

- Line 69: Read the values currently on the GPIO `write_port` and `read_port`.
- Line 70: Wait for two clocks using the `wait_for_num_clocks` task within the `gpio_agents` configuration class.
- Line 71: Display the variable values in the sequence item within `gpio_seq`.

2.1.5 Questa VIP Examples

The Questa VIP examples provide UVM Framework environments with instantiations of Questa VIP. They reside in the `vip_examples` group. These examples can be used to understand where constituent pieces of Questa VIP reside in the environment. They can also be used as a starting point for designs that have standard protocols.

2.1.5.0.1 AXI4 Example

This example demonstrates the various features of the QVIP AXI4 as listed below. This example can be used as a production environment by substituting a design for either `axi4_master`, `axi4_slave` or both. This example demonstrates the following:

1. Block level environment with a QVIP AXI4 master connected to a QVIP AXI4 slave with a QVIP AXI4 monitor observing bus activity.

The following table lists the interfaces and classes used from the QVIP Library and where they are located in the environment.

Component Description	**Component Used**	**Location in UVMF**

SystemVerilog Interface	`axi4_master`
	`axi4_slave`
	`axi4_monitor`
		`hdl_top.sv`

Configuration	`axi4_vip_config`	`vip_axi4_configuration.svh`
Agent	`axi4_agent`	`vip_axi4_environment.svh`
Sequence	`axi4_out_of_-` `order_sequence`	`qvip_axi4_bench_sequence_base.svh`

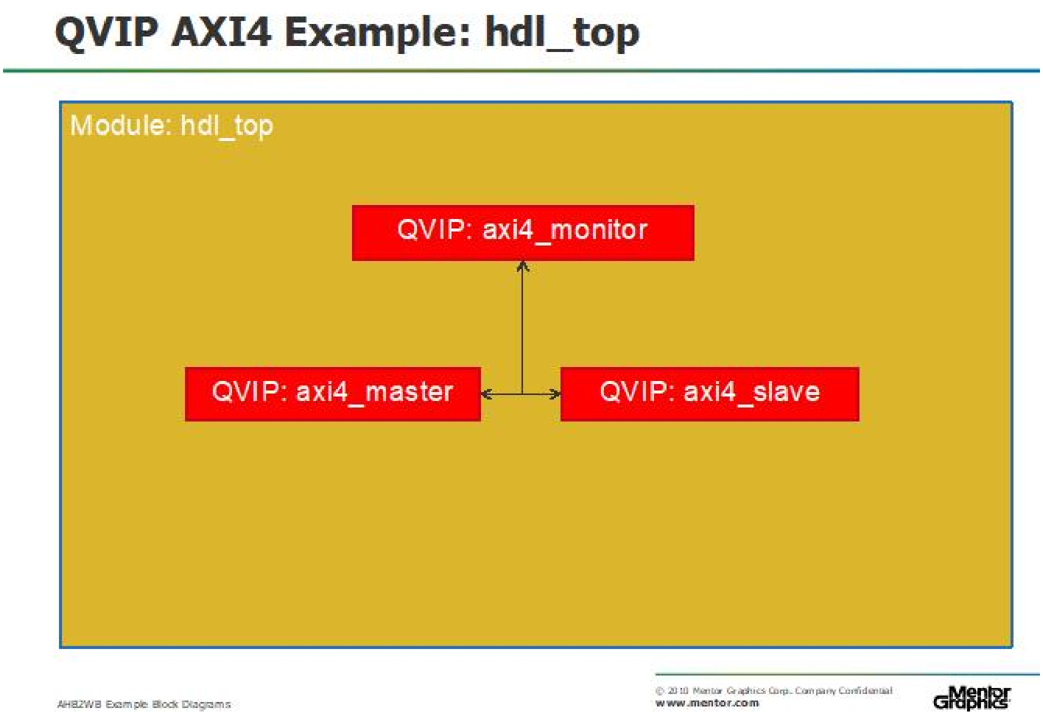

Figure 2.21: QVIP AXI4 hdl_top

2.1.5.0.2 Scatter_gather_dma Example

The scatter_gather_dma example demonstrates mixing both QVIP and custom protocols within a single bench. The DUT has an AXI4-Lite Slave interface for the control registers used to configure the DMA (Direct Memory Access) operation to be performed. This interface is connected to a QVIP AXI4-Lite Master which is used to initiate the register writes. Once programmed, the DUT initiates DMA operations via its AXI4 Master interface which is connected to a QVIP AXI4 Slave responder. A custom interface, `ccs_if`, is used to determine when a DMA operation has completed and is instantiated as `dma_done_rsc`. This example demonstrates the following:

1. Using the QVIP Configurator to instantiate and configure the two AXI4 agents. The Configurator is also used to instantiate the appropriate AXI4 interfaces, via the provided connectivity modules, for the AXI4 master and slave instances.

2. A QVIP Configurator generated Block level environment containing the

QVIP AXI4-Lite master and the QVIP AXI4 slave agents. This environment is instantiated as a sub-env within the top-level environment.

3. A top-level environment containing the sub-env and an instance of the custom ccs_agent which is configured as a responder.

The following table lists the interfaces and classes used from the QVIP Library and the custom agent plus where they are located within the example directory.

Component Description	**Component Used**	**Location within example directory**

SystemVerilog Interface	`axi4_master`
	`axi4_slave`
	`ccs_if`
		`hdl_scatter_gather_dma_qvip.sv`
		
		`hdl_top.sv`

Configuration	`axi4_vip_config`
	
	`ccs_configuration`
		`scatter_gather_dma_qvip_env_-configuration.svh`
		`scatter_gather_dma_env_-configuration.svh`

Agent	`axi4_agent`
	
	`ccs_agent`
		`scatter_gather_dma_qvip_-environment.svh`
		`scatter_gather_dma_environment.svh`

Sequence	`axi4_m0_dma_cmd_-seq`
	`ccs_responder_-sequence`
		`scatter_gather_dma_sg_seq.svh`
		
		`scatter_gather_dma_bench_sequence_-base.svh`

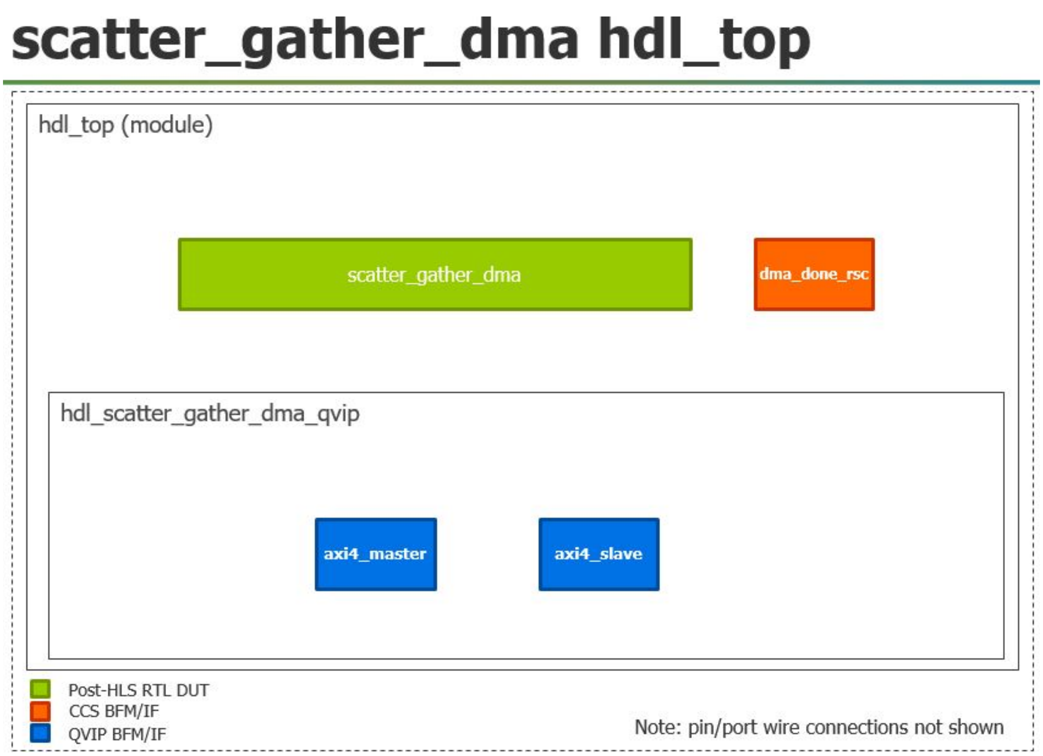

Figure 2.22: scatter_gather_dma hdl_top

2.2 Running UVMF Example Benches

The UVMF examples will run on Windows and Linux. UVMF examples are run from the sim directory located under the <example_group>/project_benches/<bench_name> directory. The examples can be run in either command line or GUI mode. The sections below describe how to run the examples in Linux and Windows. Running the base_examples from within the Questa installation is not recommended. Copy the base_examples folder from the UVMF installation into a scratch area. Run the examples within the scratch area.

2.2.1 Running the Examples in Linux

Makefiles are provided under the sim directory of each example bench for running on Linux. Example benches are under the <example_group>/project_benches directory. Under each example bench is the sim directory where simulations are run. When in the sim directory do the following to run a simulation:

1. Set environment variables listed in the Makefiles section of this document.

2. Run one of the two following commands:
   - `make cli`
   - `make debug`

The `cli` make target runs the sim in command line mode. The `debug` make target runs the sim in GUI mode. You can view signals and transactions in debug mode. Refer to section 3 for details on how to run single simulations via the Makefile structure.

2.2.2 Running the Examples in Windows

The examples as well as template generated code can be run using the makefile on Windows using a windows make utility. There are also `compile.do` and `run.do` scripts provided for running the examples as well as the template generated code in Windows. To use the Tcl scripts set the environment variables listed in the Makefiles section of this document.

2.2.3 Running the Examples on Veloce

By default, the makefiles are configured to run in simulation mode. The following make variables are used to run in emulation. Add these variables to the `make` command in the format shown as indicated.
`make [cli|debug] [USE_VELOCE=1]` (default is 0, i.e. pure simulation mode)

2.2.4 Running the Examples using Questa Verification Run Manager

Questa Verification Run Manager (VRM) is a utility built into Questa that facilitates the execution of regressions in a highly automated way. In addition to being able to build and run simulations, it also easily integrates with grid management software like LSF, automates the process of determining PASS/FAIL for individual runs and automatically manages the collection and merging of coverage data as well as many other capabilities. The UVMF examples all use the same VRM configuration file (RMDB). That common `default.rmdb` file resides in `$UVMF_HOME/scripts`. A test list control file resides in each bench's `./sim` directory to specify which tests are associated with the given bench. To invoke a VRM regression simply invoke the "`make vrun`" Makefile target or issue the "`vrun`" command yourself in a bench's sim directory, as shown:

```
vrun -rmdb $UVMF_HOME/scripts/default.rmdb
```

Refer to section 4 for details on how to invoke UVMF regression test runs with VRM.

Chapter 3

Makefiles

The UVMF uses a two level makefile structure. Individual packages have a makefile that contains make targets for compiling the package and associated modules. The simulation bench makefile includes the makefiles for all packages used by the bench as well as the `uvmf_base_pkg` makefile. This gives the bench makefile access to the make targets needed to compile required packages located under `verification_-ip`. The bench makefile also contains make targets for all packages located under the projectâĂŹs `tb` directory.

3.1 Package Makefile

Each package located under the `verification_ip` directory has a makefile that contains the make targets required to compile the package. These makefiles are included in the simulation bench makefile depending on which packages under `verification_ip` are used by the bench. Including each package makefile in the bench makefile allows project bench access to make targets for all packages under `verification_ip` that are required by the bench.

3.2 Common Makefile

The `uvmf_base_pkg` makefile is located in the `scripts` directory. This makefile contains common makefile variables and conditionals used to create commands for compiling and running example UVMF code. The common makefile is included by each project bench makefile.

3.3 Simulation Bench Makefile

Each project bench contains a makefile located in the `sim` directory under `project_benches/<benchName>`. This makefile includes the common makefile for compiling required code located under `verification_ip`. The bench makefile also contains make targets for all packages and modules located under the `tb` directory.

3.4 User Makefile Variables

3.4.1 Environment variables used for directory structure control

The makefile in `verification_ip/scripts` contains the following variables that can be set using environment variables. These variables are used to indicate the location of code to be used by UVMF.

1. `UVMF_HOME`: This variable points to the root directory of the UVMF core code. This represents released, non-user modified code. This directory should contain the `uvmf_base_pkg`,`common`,`scripts`,`base_examples` directories. Example:
   `${QUESTA_HOME}/examples/UVM_Framework/UVMF_2019.1`
2. `UVMF_VIP_LIBRARY_HOME`:This variable points to the directory where reusable UVMF IP is located. It should point to and include the `verification_ip` directory. Example:
   `/repository/sim/reuse/verification_ip`
3. `UVMF_PROJECT_DIR`:This variable points to the directory where project benches are located. It should point to and include the `project_benches` directory. Example:
   `/projects/sim/project_benches/<project_name>`

3.4.2 Command Line Makefile Variables

The makefile in the `project_benches/bench/sim` directory contains the following make variables that can be set from the command line. These variables have default values that can be seen in the makefile. To change the values of a makefile variable use the following format:

```
TEST_NAME=my_test
```

Variable Name	Description	Default
TEST_NAME	Specify which UVM test to execute	test_top
TEST_SEED	Specify a random seed to use for the simulation	random
USE_INFACT	Use inFact as part of the simulation run	0
USE_VELOCE	Use Veloce emulation as part of the simulation run	0
USE_VELOCE_LINT	Adds TBX linting capabilities to the flow to detect code constructs that might break Veloce compatibility. When USE_VELOCE is set only use the 'build' makefile target.	0
USE_VIS	Use the Visualizer debug environment	0
USE_VIS_UVM	Use the Visualizer debug environment, including advanced UVM and class debug capabilities	0
CODE_COVERAGE_ENABLE	Enable code coverage collection for the given run	0
CODE_COVERAGE_TYPES	Specify which types of code coverage to collect	bsf
CODE_COVERAGE_TARGET	Specify the target for code coverage collection	/hdl_top/DUT
EXTRA_VLOG_ARGS	Specify additional command-line switches for all vlog commands	Empty
EXTRA_VCOM_ARGS	Specify additional command-line switches for all vlog commands	Empty
EXTRA_VOPT_ARGS	Specify additional command-line switches for all vopt commands	Empty
EXTRA_VSIM_ARGS	Specify additional command-line switches for all vsim commands	Empty
EXTRA_VELHVL_ARGS	Specify additional command-line switches for velhvl commands used during a Veloce compile	Empty
EXTRA_VELANALYZE_ARGS	Specify additional command-line switches for the velanalyze commands used during a Veloce compile	Empty

`VRUN_ARGS`	Default switches to use on the vrun command-line when invoking VRM using 'make vrun'	Empty
`RMDB_PATH`	Specify the location of the RMDB file to use when invoking VRM using 'make vrun'	`$UVMF_HOME/scripts/default.rmdb`
`VRUN_DISABLE_TIMEOUTS`	If set, disable all runtime and queue timeouts for VRM invocation	0
`VRUN_MINTIMEOUT`	If defined, will explicitly specify global minimum timeou for VRM. This will override `VRM_DISABLE_TIMEOUTS`	Empty
`UVM_DISABLE_FILE_LINE`	If set to '1', will put `+define+UVM_REPORT_DISABLE_FILE_LINE` on all vlog commands, suppressing file and line number information and producing more concise log output. If set to '0', full file path and line number information will be included in all UVM log output.	1

3.5 Makefile Targets

Simulations are run using either the `cli` or `debug` make targets or using VRM. The following additional make targets can also be used. One should note that the dependencies for these targets are not complete. The result of this is that dependent code that has been modified may not be compiled automatically. The safest way to ensure all modified code is compiled and optimized is to use either the `cli` or `debug` make targets, which will both compile and optimize everything from scratch. The following targets are available for use by those who understand the dependencies associated with modified code. Command line makefile variables should be used as desired and appropriate.

3.5.1 Make Targets for Compiling Individual Packages

3.5.1.1 Packages Under `verification_ip`

Each package under `verification_ip` has a makefile for compiling that package. Within the makefile for each package is a target for compiling the package. The name of said target is the name of the package with a "comp_" prefix. For example, within the makefile for the `wb_pkg` is a target called `comp_wb_pkg`. Compilation using `comp_<packageName>` also includes compilation of C source if the package contains DPI-C calls.

3.5.1.2 Packages Under `project_benches`

The bench level parameters package, sequence package, and test package within a project bench are compiled using a make target within the Makefile located in the bench's `sim` directory. The name of the target is the name of the package with a "comp_" prefix. For example, the name of the test package for the bench named `block_a` is `block_a_tests_pkg`. The make target for compiling the `block_a_tests_pkg` is `comp_block_a_tests_pkg`.

3.5.2 Make Targets for Compiling Related Packages and Source

The `build` target can be used to compile all source necessary to run a simulation. It triggers the execution of the following targets:

1. `comp_<bench_name>_dut` - Compiles the design under test
2. `comp_uvmf_core` - Compiles the UVMF base package
3. `comp_hvl` - Compiles all packages associated with a bench, environment, or interface
4. `comp_test_bench` - Compiles static testbench modules

3.5.3 Make Targets for Optimization

The `optimize` target can be used to optimize compiled source. Performing optimization only works after a compile and must be done prior to invoking simulation.

3.5.4 Make Targets for Running a Simulation

The `gui_run` target invokes an interactive simulation. The `cli_run` target runs a simulation in batch mode.

3.5.5 Example Use of Makefile Targets

In the following example a user modified code within the following packages since the last simulation: `mem_pkg`, `block_a_sequences_pkg`, and `block_a_tests_pkg`. The user can employ any of the following makefile target sets to compile and run a simulation:

1. `make debug`
2. `make cli`
3. `make comp_wb_pkg comp_block_a_sequences_pkg comp_block_a_tests_pkg optimize gui_run`
4. `make comp_wb_pkg comp_block_a_sequences_pkg comp_block_a_tests_pkg optimize cli_run`

Option 1 compiles all source and runs the simulation in GUI mode.
Option 2 compiles all source and runs the simulation in command line mode.
Option 3 compiles only the modified source and runs the simulation in GUI mode.
Option 4 compiles only the modified source and runs the simulation in command line mode.

3.5.6 Make Targets with Veloce

Veloce compilation and runtime is only supported on Linux systems.

In general, adding the `USE_VELOCE=1` option to any make command will modify the flow to incorporate Veloce compile and run commands where appropriate. Further, the `hvl_build` target can be used to limit a recompile of only the HVL side code. Anything that is contained in a shared package (compiled by both Questa and Veloce) or purely on the HDL side (BFMs, pin interfaces, design source) will require a full Questa and Veloce compile using `'make build USE_VELOCE=1'` or one of the other more specific build targets mentioned earlier.

`make cli USE_VELOCE=1` is not recommended for Veloce compilation; since it will attempt to also run Questa and Veloce simulation.

The intended flow is as follows:

1. `make build USE_VELOCE=1`
   – Build testbench and design code for Questa & Veloce.
2. `make hvl_build USE_VELOCE=1`
   – Compile only the HVL sources.
   – Link and optimize the Questa testbench with the Veloce model.
3. `make cli_run USE_VELOCE=1`
   – Run Questa and Veloce simulation.

3.5.6.1 Veloce Servers and Processes

When using Veloce, the compilation and run-time processes occur across different sets of workstation servers. It is recommended to use Job Management to control the Veloce compilation and runtime.

Refer to the Supported Platforms and Operating Systems section in the Veloce Software Installation Guide,

- Runtime Host Hardware Requirements
- Comodel Host Hardware Requirements
- Compile Server Hardware Requirements

Reference: `$VELOCE_HOME/docs/pdfdocs/veloce_install_user.pdf`

Refer to Chapters 3 and 6 in the Veloce User Guide.

Chapter 3 – Compiling Your Design

- Analyzing RTL Source Files
- Compiling for Emulation

Chapter 6 – Job Management

Veloce supports the following job distribution systems:

- Simple Machine List
- Load Sharing Facility (LSF)
- Grid Engine
- Custom Job Distribution

Reference: `$VELOCE_HOME/docs/pdfdocs/veloce_ug.pdf`

Chapter 4

Running Regressions with VRM

4.1 Overview

Questa VRM, or Verification Run Manager, is a key component of the Questa Verification Management suite. With it, a relatively small amount of configuration can describe a highly complex and efficient regression flow that can enable a number of useful capabilities. Configuration of VRM is accomplished through an RMDB file, or "Regression Management DataBase file. This document focuses mostly on what the UVMF RMDB file has been configured to accomplish and will only touch on basic VRM capabilities when necessary. For detailed information on VRM and RMDB syntax refer to the Verification Run Manager User Guide in the Questa installation. The RMDB file for UVMF is centrally located at `$UVMF_HOME/scripts/default.rmdb`.

Alongside this is the `default_rmdb.tcl` file which defines a number of Tcl routines that the RMDB file uses to carry out its functions. All UVMF example testbenches as well as any generated UVMF testbenches should point to this RMDB file when invoking VRM, and all VRM invocations should be made from a bench's `./sim` directory. In other words, there should be no need to create special RMDB files for individual benches.

4.2 Invocation

The simplest way to invoke a VRM regression for a given bench is to use the makefile target "make vrun". Information about how to compile the given bench as well as which tests to invoke is pulled from a test list file. The default location and name for the test list file is `./sim/testlist`. Information on the content and format of the test list file can be found in a later subsection. By default, the following RMDB behavior is invoked by UVMF:

- Advanced test list file
  - Per-test extra arguments
  - Nested test list files
  - Repeat of tests
  - Control of random seeds on a per-test basis
- Parallel build of separate testbenches
- Parallel run of simulations on a per-bench basis
- Automatic parallel merge of UCDB output
- Automatic generation of HTML coverage report
- Random test seed generation and control
- Automatic integration of Questa test plan file, if specified
- Automatic email notification
- Automatic invocation of Questa CoverCheck
- Use of grid management systems (LSF/SunGrid, etc.)

4.2.1 RMDB Controls and Parameters

The following features and capabilities can be controlled and modified by the user in multiple ways. Some capabilities are controlled via RMDB parameters, others through VRM command-line switches. For those controlled via parameter, override with the VRM "-G" switch syntax, e.x. `"vrun -GMASTER_SEED=0"` or alternatively, via the use of a UVMF VRM Initialization Tcl file. See later section for details on the format of that file. In the following table, the "INI Variable Exists" column indicates whether that parameter can be specified via the initialization file. If "Yes", the variable name is the same as the parameter name but not case-sensitive (i.e. the INI variable associated with "NO_RERUN" can be specified as either "NO_RERUN" or "no_rerun").

Feature	Parameter Name	INI Variable Exists	Default	Notes
Parallelism control	N/A	No	Infinite	Use vrun "-j" switch to reduce the number of allowed parallel processes.
Automatic re-run	`NO_RERUN`	Yes	Disabled	Enable automatic rerun of failing tests
Automatic UCDB merge	N/A	No	Enabled	Disable with vrun switch "-noautomerge"
Test list file name	`TESTLIST_NAME`	No	`testlist`	Overrides name of test list file but location continues as `.sim`
Test list location	`TOP_TESTLIST_FILE`	No	`./sim/testlist`	Specify absolute or relative path to test list file
Enable code coverage collection	`CODE_COVERAGE_ENABLE`	Yes	0	Collect code coverage against DUT
Specify code coverage types	`CODE_COVERAGE_TYPES`	Yes	bsf	Branch, statement and FSM code coverage collected by default
Specify code coverage target	`CODE_COVERAGE_TARGET`	Yes	`/hdl_top/DUT.`	Recursively collect coverage against the default location for the DUT
HTML Coverage Report Options	`HTML_REPORT_ARGS`	Yes	-details -source -testdetails -htmldir (%VRUNDIR%)/covhtmlreport	Specify how the HTML coverage report will be generated. Location defaults to the ./sim directory and will include source annotation and coverage details
Generate waves	`DUMP_WAVES`	Yes	0	Set to 1 to produce Visualizer waveforms for all simulations
Generate waves on rerun	`DUMP_WAVES_ON_RERUN`	Yes	0	Set to 1 to produce Visualizer waveforms for simulations that are re-run on failure.
Seed control	`MASTER_SEED`	No	random	Seeds the top-level random number generator used to seed individual simulations. May be the string "random" or any 32-bit unsigned integer value
Apply Coverage Exclusions	`exclusionfile`	Yes	Undefined	If defined, should refer to a valid Tcl file that is intended to be applied to the final merged UCDB prior to coverage report generation. Intended to use "coverage exclude" commands.
Dofile Control	`PRE_RUN_DOFILE`	Yes	Empty	If non-empty and pointing to a valid file, will be executed before starting any simulation. The path must be relative to the VRM invocation directory.
Dofile Control	`USE_TEST_DOFILE`	Yes	0	If variable is set and a Tcl file is found matching the UVM test name currently being run, the file will be executed before starting that simulation. File paths are relative to the VRM invocation directory.
Grid System Control	`GRIDTYPE`	Yes	LSF	Specify which grid system to use. Can be one of the built-in systems or a custom string
Grid System Control	`USE_JOB_MGMT_BUILD`	Yes	0	If set, will attempt to invoke build commands into the specified grid system
Grid System Control	`USE_JOB_MGMT_RUN`	Yes	0	If set, will attempt to invoke run commands into the specified grid system
Grid System Control	`GRIDCOMMAND_BUILD`	Yes	Empty	Specify the command (bsub, qsub, etc.) with arguments needed to invoke a build into the grid. Should be encapsulated in curly brackets and look similar to this: `bsub -q my_queue (%WRAPPER%)`

Grid System Control	GRIDCOMMAND_RUN	Yes	Empty	Specify the command (bsub, qsub, etc.) with arguments needed to invoke a simulation into the grid. Should be encapsulated in curly brackets and look similar to this: `bsub -q my_queue (%WRAPPER%)`
Execution Control	USESTDERR	Yes	1	By default, if any process produces output on STDERR the process will be tagged as having failed. Setting this to 0 will cause STDERR to be ignored in all processes.
Timeouts	TIMEOUT	Yes	3600 (seconds)	Defines default timeout for all execution VRM timeouts
Timeouts	QUEUE_TIMEOUT	Yes	60 (seconds)	Defines default timeout for all queue VRM timeouts
Timeouts	BUILD_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for build operations. If unspecified, will use "TIMEOUT" value
TImeouts	BUILD_QUEUE_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for queuing build operations. If unspecified, will use "QUEUE_TIMEOUT" value.
Timeouts	RUN_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for individual simulation runs. If unspecified, will use "TIMEOUT" value
TImeouts	RUN_QUEUE_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for queuing individual simulation runs. If unspecified, will use "QUEUE_TIMEOUT" value.
Timeouts	EXCLUSION_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for applying coverage exclusions. If unspecified, will use "TIMEOUT" value
TImeouts	EXCLUSION_QUEUE_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for queuing the application of coverage exclusions. If unspecified, will use "QUEUE_TIMEOUT" value.
Timeouts	COVERCHECK_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for producing automatic coverage exclusions. If unspecified, will use "TIMEOUT" value
TImeouts	COVERCHECK_QUEUE_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for queuing automatic coverage exclusions. If unspecified, will use "QUEUE_TIMEOUT" value.
Timeouts	REPORT_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for generating coverage reports. If unspecified, will use "TIMEOUT" value
TImeouts	REPORT_QUEUE_TIMEOUT	Yes	Empty	Specify timeout value (in seconds) for queuing the generation of coverage reports. If unspecified, will use "QUEUE_TIMEOUT" value.
Email	EMAIL_MESSAGE	Yes	Empty	Specify message content for automatic email notifications. If left empty, rely on default VRM message content
Email	EMAIL_ORIGINATOR	Yes	Empty	Specify originator email address for automatic email notifications. If left empty, rely on the default VRM originator value

Email	EMAIL_RECIPIENTS	Yes	Empty	Specify recipient list for automatic email notifications. This must be specified to enable this feature
Email	EMAIL_SERVERS	Yes	Empty	Specify list of SMTP servers for use when sending email notifications. This must be specified to enable this feature.
Email	EMAIL_SECTIONS	Yes	Empty	Specify which built-in sections of information are included in the email. Default is "all", which is to include all sections
Email	EMAIL_SUBJECT	Yes	Empty	Specify the subject header for email notifications. If left empty, use the VRM default header
Auto-Triage	triagefile	Yes	Empty	If specified, enable auto-triage capability. Points to a triage DB file. Path should be relative to the VRM invocation directory
Auto-Triage	triageoptions	Yes	Empty	Specify any arguments to the command used to produce the triage DB file
Auto-Trending	trendfile	Yes	Empty	If specified, enable auto-trend analysis. Points to a trending UCDB file, path relative to the VRM invocation directory

4.2.2 UVMF VRM Initialization File

A Tcl-based initialization file can be used to specify many of the parameters described in the table above in a more permanent way. This can be used to override defaults on a user-by-user basis or specify preferences for an entire group.

Use the initialization file by setting the environment variable `$UVMF_VRM_INI` to the full path to the desired file. An example file can be found in `$UVMF_HOME/scripts/uvmf_vrm_ini.tcl` but the file can and should be located elsewhere.

This file, if pointed to, will be sourced prior to running VRM and can be used to both initialize parameters as well as specify Tcl procedures for overriding default VRM behavior or specifying how to invoke non-standard grid management systems.

Parameter overrides take place by specifying a Tcl proc called `vrmSetup`. From within this proc use calls to the pre-defined routine `setIniVar` to specify the desired value for a given parameter. For example, one can more widely enable support for Visualizer with the following Tcl content:

```
proc vrmSetup {} {
  setIniVar use_vis 1
}
```

Attempts to use `setIniVar` to initialize parameters not in the list above will result in a fatal error. It is legal to have the initialization file source other Tcl files and this mechanism can be used to specify project-level preferences along with user-level preferences.

4.2.3 Test List Format

Test lists provide the RMDB with information about how a given bench is to be built, which bench is associated with a given simulation, and how simulations are to be invoked. Comment lines start with a pound (#). Other non-blank lines must start with a keyword followed by information specific to that keyword.

4.2.3.1 `TB_INFO` Keyword

The `TB_INFO` keyword specifies how a given testbench should be built. The format is as follows:

```
TB_INFO <bench_name> { <build arguments> } { <runtime arguments> }
```

The build arguments will be applied to the 'make' command used to both compile and optimize the testbench. Any runtime arguments will be applied to all invocations of vsim for simulations against the given testbench.

4.2.3.2 `TB` Keyword

The `TB` keyword specifies which test bench should be used for all subsequent tests. The format is as follows:

```
TB <bench_name>
```

After a given TB keyword is found, all subsequent tests will target this test bench. If another TB keyword is used with a different bench name, all subsequent tests after that will target the new bench.

4.2.3.3 `TB_MAP` Keyword

The `TB_MAP` keyword enables automatic hierarchy mapping of code coverage data from the currently targeted bench into another higher-level bench where the current DUT is instantiated as a sub-component. This facilitiates comprehensive code coverage collection when running both block-level and top-level simulations. The format is as follows:

```
TB_MAP <bench_name> <source hierarchy> <destination hierarchy>
```

With this line specified, the code coverage data will be quickly edited after each simulation completes in order to modify the hierarchy for the DUT in the specified bench.

4.2.3.4 `TEST` Keyword

The `TEST` keyword specifies a test to be executed against the bench that should have been specified earlier with the TB keyword. The format is as follows:

```
TEST <test_name> <repeat_count> <seed0> <seed1> ... <seedN> { <per-test arguments> }
```

Only `<test_name>` is required, all other arguments are optional. If `<repeat_count>` is omitted, a count of 1 is used. A seed for each repeat count may be provided but if any or all are omitted, a random seed is generated. The final argument, if provided, will be passed in as additional arguments to vsim for each iteration of the test.

4.2.3.5 `INCLUDE` Keyword

The `INCLUDE` keyword allows one to include one test list within another, creating nested structures. The format is as follows:

```
INCLUDE <file_name>
```

Relative paths are allowed. Any relative path is based on where VRM was invoked (the ./sim directory for a bench, by default).

4.3 Operation and Results

When VRM is invoked, the test list file is parsed for information on which benches to build and which tests to execute. Once this data model is built, the following occurs:

1. Each test bench is built in parallel
2. After a given test bench has successfully been built, all associated simulation runs are executed in parallel on that bench.
3. UCDB merging takes place as individual simulations complete.
4. Once all simulations have finished an HTML coverage report is produced.

All operations and output take place under the ./sim/VRMDATA directory. The VRMDATA directory is created when VRM is invoked. Individual bench builds take place in subdirectories, as do individual simulation runs. This allows for a high degree of parallel operation.

- Individual bench compiles take place in
  VRMDATA/top/each_top˽<bench_name>/build_group/build_task
- Individual simulations run in
  VRMDATA/top/each_top˽<bench_name>/build_group/run_fork
  /run˽<bench_name>-<test_name>-<iteration>-<seed>

If a build or simulation fails, look in the execScript.log underneath the given directory for information on what went wrong.

4.4 Timeouts

All meaningful actions that the UVMF VRM process can invoke have the potential to take too much time to complete and, in some cases, could be misconfigured in such a way as to hang the entire VRM process. In order to avoid this situation, timeouts are in place that will terminate any processes that have taken too long.

There are two timeouts associated with each activity: A "runtime" timeout that tracks the amount of time a process has taken while running and a "queue" timeout that is associated with how long a given process has been waiting to begin. The "queue" timeout is only meaningful when using a grid management system like LSF or SunGrid.

All values are given in seconds. By default, all processes' runtime timeouts default to 3600s (1 hour) and queue timeouts default to 60s. Timeouts can be disabled via a VRM command line argument (controllable via the "make vrun" makefile target) and individual timeouts can be increased or decreased via VRM INI variables. These can all be found in the table of INI variables in a previous section of this document.

For more detail on how VRM handles timeouts please refer to the VRM User Guide.

4.5 Email

Email notifications can be sent by VRM when a regression has completed. Several variables are defined within by UVMF VRM to enable and configure this feature.

For more detail on how to properly set these variables please refer to the VRM User Guide.

Chapter 5

Scripts

5.1 UVMF Code Generator

5.1.1 Overview

YAML based and Python API based code generators are provided by UVMF for rapid code development. Three code generation templates are provided: interface, environment and bench. The interface template generates the files, infrastructure and interconnect required for an interface package. The environment template generates the files, infrastructure and interconnect required for an environment package. The bench template generates the files, infrastructure and interconnect required for a project bench. The SystemVerilog/UVM code generated by the templates can be simulated as is. This provides a starting point for adding required design and protocol specific code.

A tutorial on using the generators from specification to completed test bench is located in the `docs/generator_tutorial` directory. This tutorial starts with a design specification, walks through the creation of generator input files, shows how to generate the code, and finishes up by talking about the post-generation modifications necessary to complete the simulation environment.

The templates have been tested on the following Python releases: 2.6, 2.7, 3.6. Confirm the Python version on machine is at least 2.6 using the `python -V` command.

The input to the API-based generator is a Python configuration script that imports the `uvmf_gen` Python module as well as specifying the work to be done. Example configuration scripts for generating UVMF code using the templates is located under `templates/python/api_files`.

Use of the YAML based template generator is strongly recommended. It is a more data oriented approach that provides some linting of user written YAML. It eliminates some of the common mistakes made when using the Python API based template generator. It also supports a regeneration flow that can automatically incorporate manual changes to generated source into subsequent re-runs of the script. The YAML based template generator uses the Python APIs under the hood for generating SystemVerilog/UVM code. The Python APIs are available for customers who prefer a more alogorithmic or programmable mechanism for generating SystemVerilog/UVM.

Python API files can be converted to YAML using the `--yaml` switch when executing the Python API file. For example, executing `'./mem_if_config.py --yaml'` will generate a file named `mem_interface.yaml`. The Python API file and the resulting YAML file will generate identical code.

5.1.2 Installation and Operation

Instructions on installation and operation of the Python based UVMF code generators are located in `$UVMF_HOME/templates/python/templates.API.README`

5.1.3 `yaml2uvmf.py` Command Details

The following switches are supported by the `yaml2uvmf.py` command for YAML-based generation of code. All of these switches can be viewed from the command through the `-h` or `--help` switches of the script.

Usage:
`yaml2uvmf.py [options] [yaml_file1 [yaml_file2] ... [yaml_fileN]]`

Option	Description
`--version`	Show script version number and exit
`--help | -h`	Show help message and exit
`--quiet | -q`	Suppress output while running
`--dest_dir=DEST_DIR | -d DEST_DIR`	Override default destination directory of `./uvmf_template_output`
`--template_dir=TEMPLATE_DIR | -t TEMPLATE_DIR`	Override the template source directory, defaults to `template_files` relative to the location of the `uvmf_gen.py` file.
`--overwrite | -o`	Overwrite existing output. Default is to skip any files that already exist.
`--file=FILE | -f FILE`	Specify a file containing a list of YAML configuration files as input.

`--relfile=FILE | -F FILE`	Specify a file containing a list of YAML configuration files as input. Relative path references will be calculated relative to the path of the containing file list instead of the invocation directory.
`--generate=GEN_NAME | -f GEN_NAME`	Specify which elements to generate. Default is to generate everything passed in.
`--merge_source=DIR | -m DIR`	Enable auto-merge flow, pulling information from the specified directory. **Note:** Unless the destination directory is explicitly changed from the default with `--dest_dir`, the directory specified here cannot be `uvmf_template_output`.
`--merge_skip_missing_blocks | -s`	Continue merge if unable to locate a labeled block in new output that was defined in old source and produce a list of issues at the end. Default behavior is to raise an error and quit.

5.1.4 Developing Configuration Input for Use by the UVMF Code Generator

All details regarding the required input format for all YAML data structures can be found in the UVMF Code Generator YAML Reference guide. This section describes the available examples and how to leverage those examples to get started the first time.

5.1.4.1 Interface Package Generation

5.1.4.1.1 Format and Input Required by the Generator

Example interface YAML configuration is located under the `templates/python/examples/yaml_files` directory. The interface examples are `mem_if_cfg.yaml` and `pkt_if_cfg.yaml`. The user is expected to provide the following information: interface name, signals, transaction variables and configuration variables.

A file named `new_interface.yaml`, located in the `yaml_files` directory, can be used as a starting point for creating an interface YAML file. It has all but the required fields commented out. Uncomment the labels as needed to add ports, transaction variables, etc.

5.1.4.1.2 Steps for Using the Generator

1. Create the configuration file, using `new_interface.yaml` as a starting point
2. Execute the generation, using `yaml2uvmf.py <filename>.yaml`
3. Use the list in the next section to add protocol-specific code to the new interface package.

5.1.4.1.3 Adding Protocol Specific Code

The following list identifies areas where protocol specific code needs to be added to the new interface package. The following files listed assume the interface generated is named `abc_pkg`. Files generated are under the `./uvmf_template_output/verification_ip/interface_packages/abc_pkg` directory. The `UVMF_CHANGE_ME` string can be used to identify locations for code addition or changes within various files.

1. `src/abc_driver_bfm.sv`: Add code to implement protocol driving
   (a) To implement initiator functionality, add signaling code to the `initiate_and_get_response()` task. Comments in the generated task list available variables from the sequence item and signals in the port list of the BFM. This task is automatically executed by the driver class when it receives a sequence item from the sequence through the sequencer when the agent is configured as an INITIATOR.
   (b) To implement responder functionality, add signaling code to the `respond_and_wait_for_next_transfer()` task. Comments in the generated task list available variables from the sequence item and signals in the port list of the BFM. This task is automatically executed by the driver class when it receives a sequence item from the responder sequence through the sequencer when the agent is configured as a RESPONDER.
2. `src/abc_monitor_bfm.sv`: Implement protocol monitoring in the provided `do_monitor` task. Comments in the generated task list available variables in the sequence item and signals in the port list of the BFM. This task should assign variables from the signals according to the protocol. When this task exits, the variables are automatically received by the monitor class. The monitor class automatically places the variables in a sequence item and broadcasts the sequence item through its analysis_port to connected subscribers.

A forever loop in the monitor BFM automatically re-executes `do_monitor()` with only one clock consumed between calls.

3. `src/abc_responder_sequence.svh`: If the interface has responder functionality, complete the body of this sequence.
4. `src/abc_transaction_coverage.svh`: If functional coverage from the agent is desired, add bins, crosses, etc., to the generated covergroup.
5. `src/abc2reg_adapter.svh`: If the interface will be used with a UVM register model, fill in the `bus2reg()` and `reg2bus()` functions.
6. Add new sequence items based on protocol needs. All new sequence items should be extended from
textttabc_transaction.
7. Add new sequences based on protocol needs. All new sequences should be extended from `abc_sequence_base`.

5.1.4.2 Environment Package Generation

5.1.4.2.1 Format and Input Required by the Generator

Example environment YAML configuration is located under the `templates/python/examples/yaml_files` directory. The environment examples include `block_a_env_cfg.yaml`, `block_b_env_cfg.yaml`, `chip_env_cfg.yaml` and `block_c_env_cfg.yaml`. Analysis components used at the environment level are all defined in `predictor_components.yaml`.

The `block_a_env` example illustrates a simple, unparameterized environment.

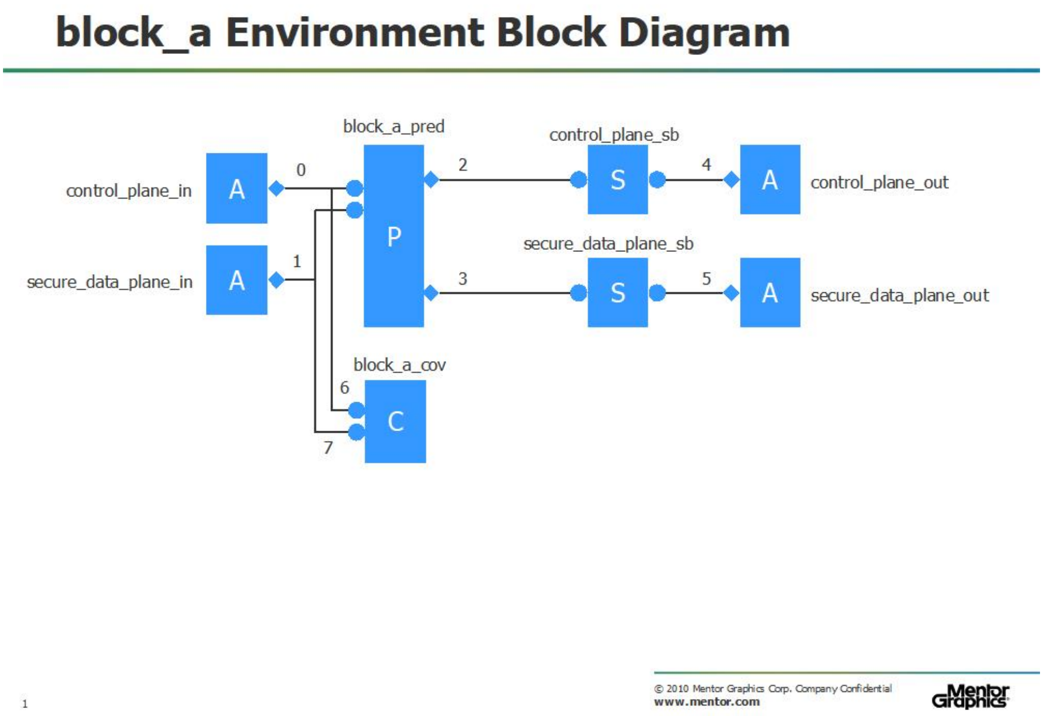

Figure 5.1: Block A Environment Example

The `block_b_env` example illustrates how a parameterized environment looks and also incorporates some DPI-C source as well as a UVM register model.

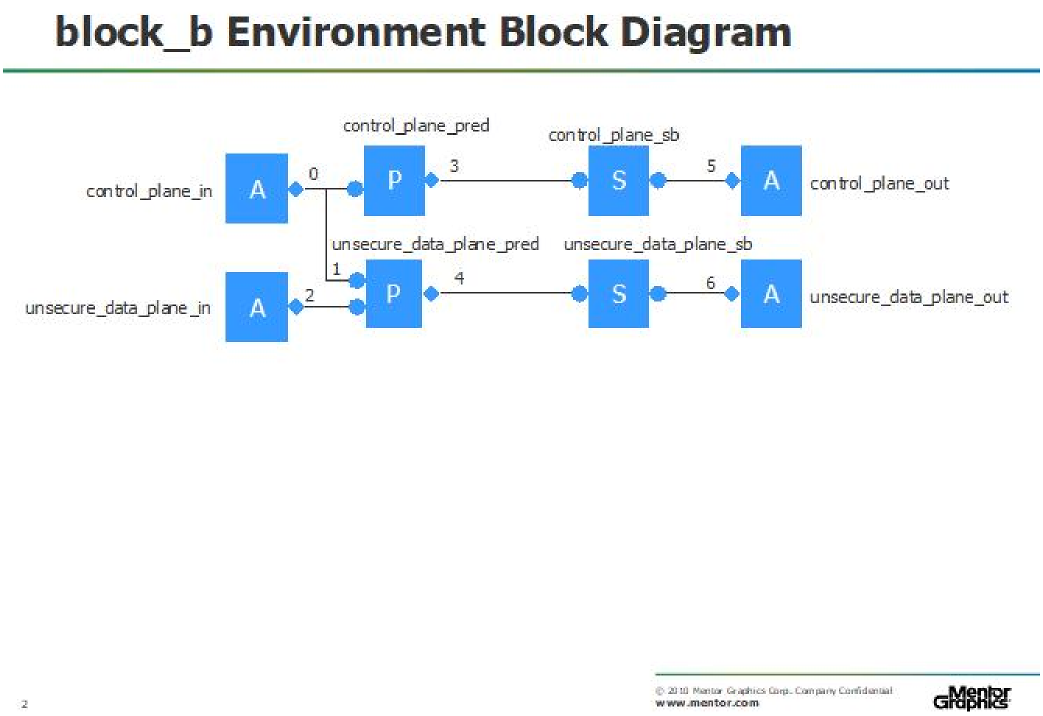

Figure 5.2: Block B Environment Example

The `chip_env` example illustrates vertical reuse by instantiating the aforementioned `block_a_env` and `block_b_env` environments.

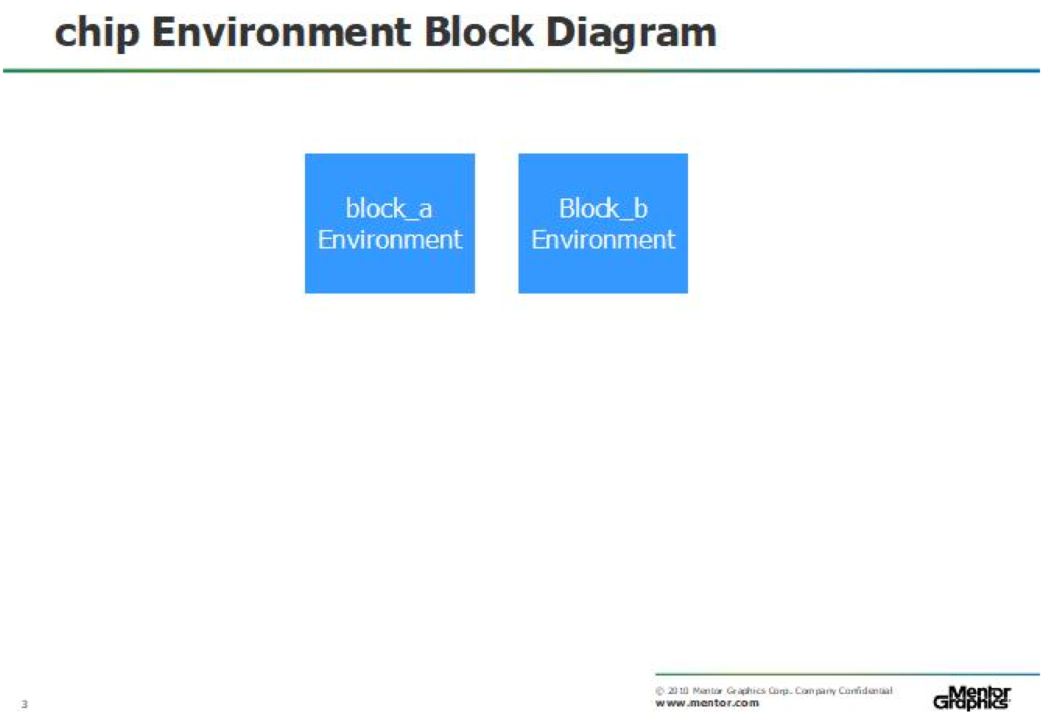

Figure 5.3: Chip Environment Example

The `block_c_env` example demonstrates the use of Questa VIP for standard protocols n an environment that also contains custom protocols, predictors, and scoreboards.

5.1.4.2.2 Steps for Using the Generator

1. Create the configuration file, using `new_environment.yaml` as a starting point
2. Execute the generation, using `yaml2uvmf.py <YAML files>`. **NOTE:** All interfaces and other subcomponents referenced in the environment YAML will need to be defined by passing in their requisite YAML as well.
3. Use the list in the next section to add protocol-specific code to the new environment package.

5.1.4.2.3 Adding DUT Specific Code

The following list identifies areas where DUT specific code needs to be added to the new environment package. The following files listed assumes the environment generated is named `abc_prj_env_pkg`. Files generated are under `uvmf_template_output/verification_ip/environment_packages/abc_prj_env_pkg` directory. The `UVMF_CHANGE_ME` string can be used to identify locations for code addition or changes within various files.

1. `src/abc_prj_env_configuration.svh`: Configure agents as required by the DUT
2. Implement the prediction model in the `write...ap` functions within the predictor class created using the `util_components` YAML label
3. Implement the coverage model in the `write...ap` functions within the coverage class created using the `util_components` YAML label
4. Implement any custom scoreboards in the `write...ap` functions within the scoreboard class created using the `util_components` YAML label
5. Add new sequences as needed in the `src` directory. All new sequences should be extended from `abc_prj_sequence_base`. Be sure to include any new sequence files to the environment package, `abc_prj_env_pkg.sv`.

5.1.4.3 Bench Generation

5.1.4.3.1 Format and Input Required by the Generator

Example bench YAML configuration is located under the `templates/python/examples/yaml_files` directory. The bench examples include `block_a_bench_cfg.yaml`, `block_b_bench_cfg.yaml`, `chip_bench_cfg.yaml` and `block_c_bench_cfg.yaml`. The YAML input only requires the name of the bench and top-level environment name as input.

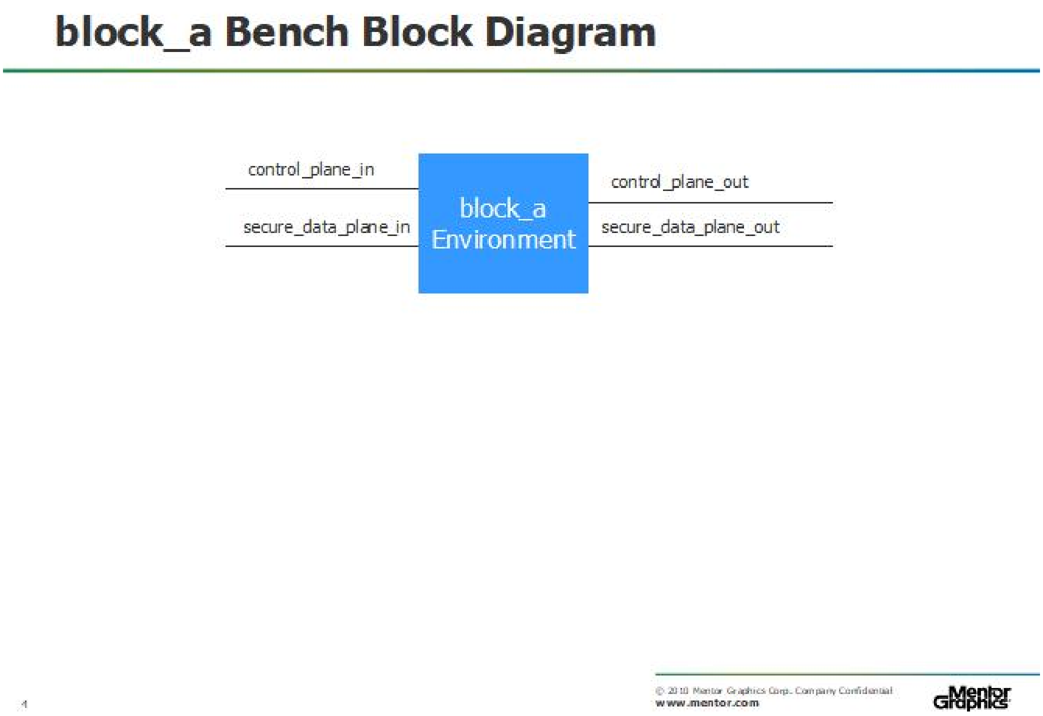

Figure 5.4: Block A Bench Block Diagram

5.1.4.3.2 Steps for Using the Generator

1. Create the bench template file as described above, using `new_bench.yaml` as a starting point.
2. Execute the generation, using `yaml2uvmf.py <YAML files>`. **NOTE:** All interfaces, environments, and other subcomponents referenced recursively by the target bench will need to be defined by passing in their requisite YAML as well.
3. Use the list in the next section to add protocol-specific code to the new bench.

5.1.4.3.3 Adding DUT Specific Code

The following list identifies areas where DUT specific code needs to be added to the new bench. The following files listed assume the bench generated is named `abc`. Files generated are under the `uvmf_template_output/project_benches/abc` directory. The `UVMF_CHANGE_ME` string can be used to identify locations for code addition or changes within various files.

1. `tb/testbench/hdl_top.sv`: Instantiate the DUT and connect ports to the signals in the interface busses.
2. `sim/Makefile`: Update the `abc_VERILOG_DUT` and/or `abc_VHDL_DUT` variables with a list of DUT files to compile.
3. `tb/sequences/src/abc_bench_sequence_base.svh`: Modify the `body()` task to reflect the typical test flow. Factory overrides can be used to change test flow as needed.
4. Add new sequences as needed in the `tb/sequences/src` directory. All new sequences should be extended from `abc_bench_sequence_base`. Be sure to include new sequences to the environment package `tb/sequences/abc_sequence_pkg.sv`.

5.1.4.3.4 Connecting to Internal DUT Ports

Interfaces that are internal only and as a result are not primary I/O to the bench need to be connected manually. In the `chip_bench` example the interfaces to be connected in this fashion are the `internal_control_plane_in` and `internal_control_plane_out` busses.

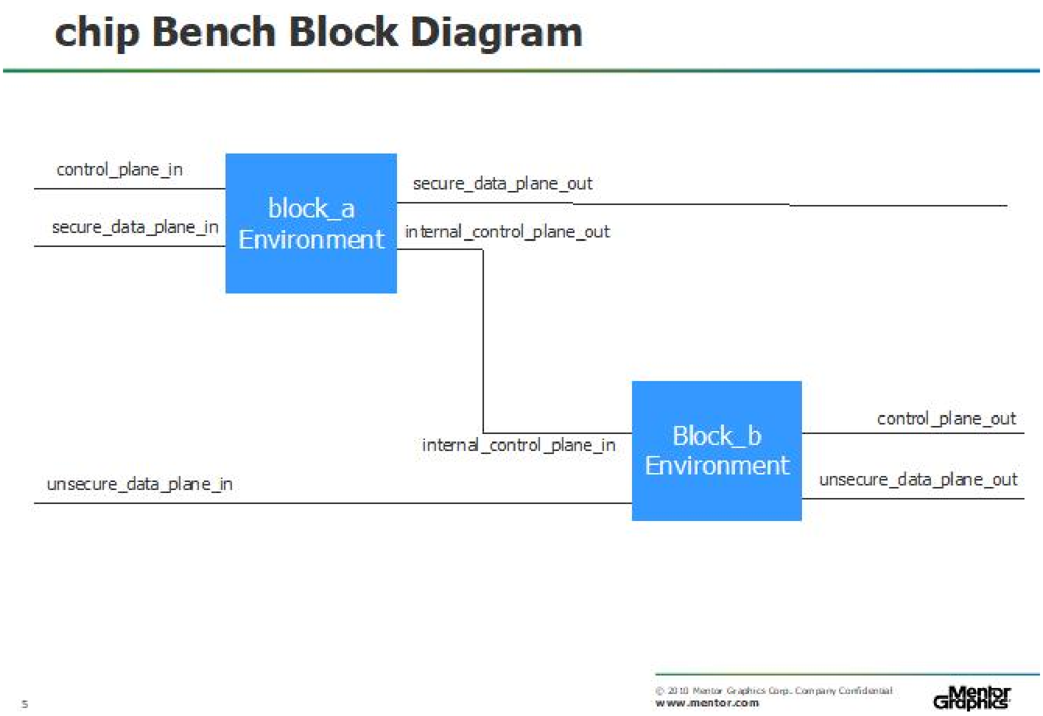

Figure 5.5: Chip Bench Block Diagram

The generated code looks like this in `hdl_top.sv`:

```
mem_if internal_control_plane_out_bus(.clock(clk),.reset(rst));
mem_if internal_control_plane_in_bus(.clock(clk),.reset(rst));
```

```
mem_monitor_bfm internal_control_plane_out_mon_bfm(
internal_control_plane_out_bus);
mem_monitor_bfm internal_control_plane_in_mon_bfm(
internal_control_plane_in_bus);
```

This code must be converted to the following to connect the internal interfaces between the two block levels. The two signal bundles of type `abc_if` are combined into one named `internal_control_plane_bus`. The two `abc_monitor_bfm`s are connected to the new signal bundle, `internal_control_plane_bus`.

```
mem_if internal_control_plane_bus(.clock(clk),.reset(rst));
```

```
mem_monitor_bfm internal_control_plane_out_mon_bfm(
internal_control_plane_bus);
mem_monitor_bfm internal_control_plane_in_mon_bfm(
internal_control_plane_bus);
```

Further optimization can be achieved by having the two environments share the same UVMF based `mem_monitor` using the mechanism shown in the `ahb2spi` environment example.

5.1.4.4 Using the Questa VIP Configurator in Tandem with UVMF Code Generation

The Questa VIP Configurator is used to generate UVMF based environments containing standard protocols. The UVMF environment generated by the configurator contains the agents and configuration policies for each standard protocol selectecd by the user. This environment can be automatically included in a UVMF environment containing custom agents, predictors, coverage components, scoreboards, etc.

Documentation on the QVIP Configurator can be found in the QVIP installation. A snapshot of the QVIP Configurator is shown below.

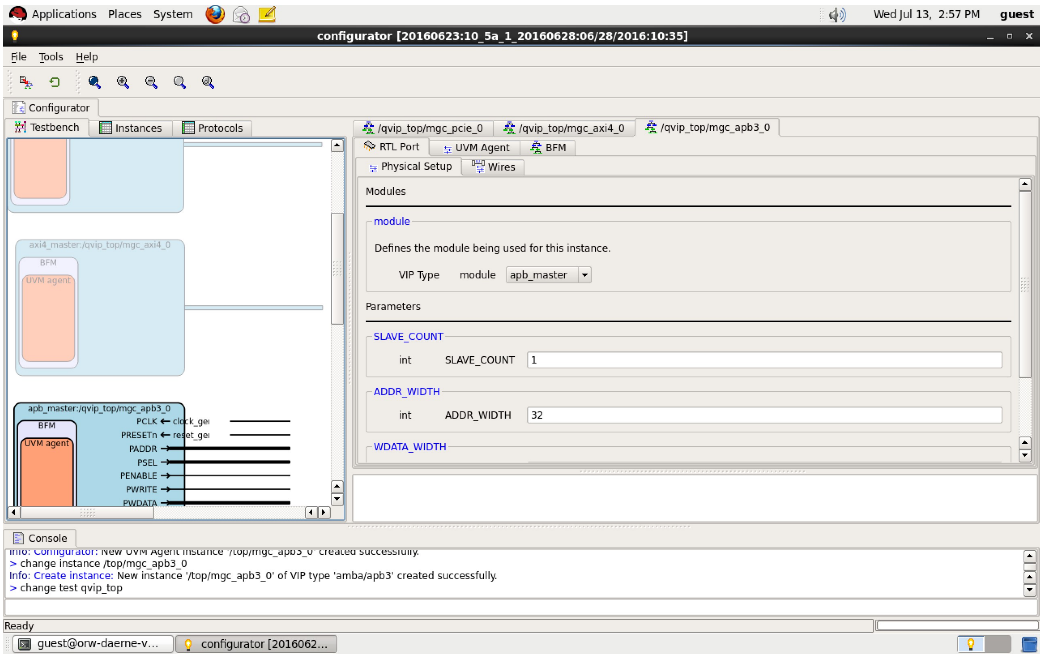

Figure 5.6: QVIP Configurator Screenshot

The following digram shows the hierarchy of the `block_c` environment which includes QVIP Configurator generated content underneath:

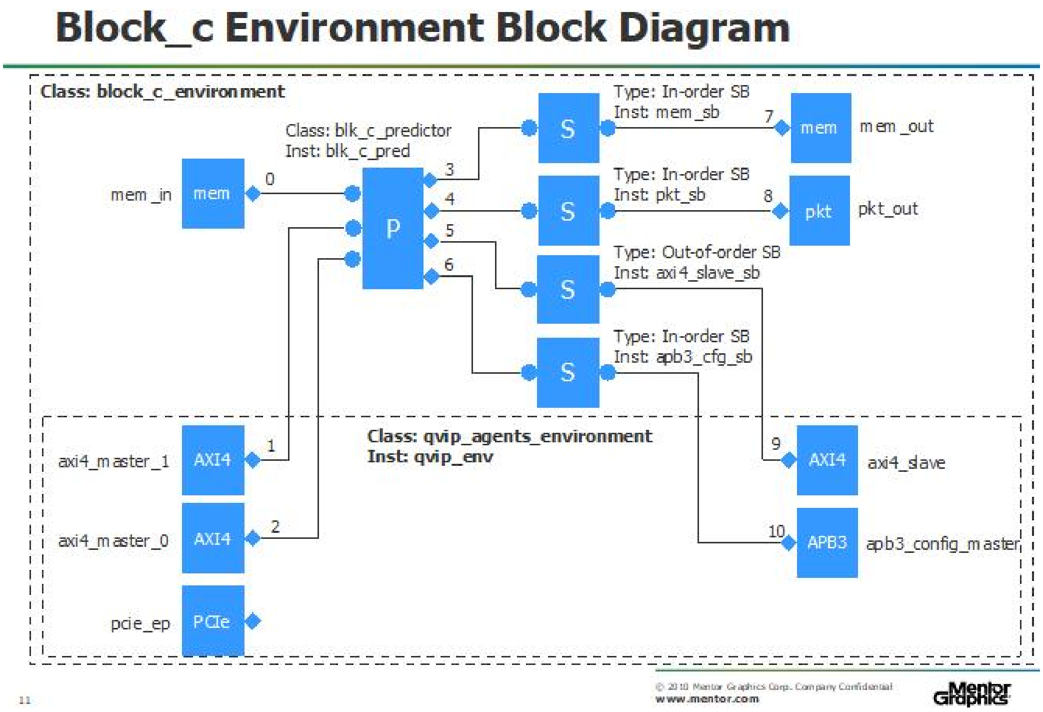

Figure 5.7: Block C Environment Block Diagram

5.1.4.4.1 Using the Configurator Tool

The configurator uses a GUI to customize environments containing custom and standard protocols. Click the "Protocols" tab to add protocols, which can then be customized on the right side of the configurator. To change the name of the testbench, right click on the whitespace under the "Testbench" tab on the left. The name of the test bench must be changed from the default name 'top' to avoid recursive module instantiation. In this example we are changing the test bench name from `top` to `qvip_agents`. When the desired settings are completed, click the "Generate Models/Testbench" button. The configurator will create a folder containing the output (In the picture below it will be called "qvip_agents_dir"). Inside this folder there will be a folder named `uvmf` which will contain the UVMF based configurator output. Set the environment variable `QVIP_AGENTS_DIR_NAME` so that it points to the `uvmf` folder. A YAML file can be found in the `uvmf` directory that describes the contents of the QVIP sub environment along with comments describing how to incorporate this into higher level YAML.

The YAML `qvip_connections` structure defines connections between QVIP analysis ports and outside components. Please see the YAML reference manual for details. These connections require the user to specify the desired QVIP agent component underneath the QVIP sub-environment. In the following YAML example, a QVIP AXI4 master component's analysis port is connected to a predictor's analysis export: `qvip_connections:`

```
  - driver:  "qvip_env.axi4_master_0"
    ap_key:  "trans_ap"
    receiver:  "blk_c_pred.axi4_master_0_ae"
```

The analysis ports within QVIP are located within an associative array of analysis ports named `ap`. This associative array is indexed using a string. In the above example the `trans_ap` argument is used as a string to index into the associative array of analysis ports within the QVIP agent to select the desired analysis port within the array. All analysis ports within `ap` broadcast transactions that are casted to the base class type named `mvc_sequence_item_base`. These analysis ports must be connected to analysis exports parameterized to receive transactions of type `mvc_sequence_item_base`. Transactions received through the analysis export should then be casted to the desired transaction type.

QVIP agents contain default analysis ports. The Users Guide for each protocol within the QVIP installation describes the analysis port name and transaction type broadcasted for each protocol. Remember that the analysis export must be parameterized to receive `mvc_sequence_item_base` and the received transaction must be casted to the type listed in the protocol specific users guide in the QVIP installation docs directory.

5.1.4.4.2 Clock Generation Within QVIP Configurator Generated UVMF Module

The QVIP configurator generates a module that contains all of the QVIP interface BFMs. The name of this module is based on the test bench name: `hdl_<benchName>.sv`. This module contains a default clock and reset generator. There is no interaction between this clock/reset generator and the clock/reset generator used in `hdl_top.sv` created by UVMF. If a single clock/reset generator is required then ports can be added to the QVIP module to bring in clock and reset from `hdl_top`.

5.1.4.4.3 Block-to-Top Reuse of QVIP Configurator Generated Environment

The Configurator generated UVMF based environment is reusable as a sub-environment at any environment level with any number of instantiations.

5.1.4.4.4 Block-to-Top Reuse of QVIP Configurator Generated Module

Block to top reuse of QVIP Configurator generated code is supported. The Configurator generated UVMF based module contains a single parameter named `UNIQUE_ID`, a parameter for each BFM that sets the ACTIVE/PASSIVE setting for each BFM, and a parameter named `EXT_CLK_RESET` that determines if the Configurator generated clock and reset driver is used.

The `UNIQUE_ID` string parameter is prepended to the BFM interface name. The result is used as the `field_name` argument to the `uvm_config_db::set` call to place each interface BFM in the UVM config database.

The parameter for each BFM that sets the ACTIVE/PASSIVE setting is named `<AGENT_NAME>_ACTIVE` where `<AGENT_NAME>` is the capitalized instantiation name of the QVIP agent. When set to 1, this parameter sets the BFM in ACTIVE mode. When set to 0, this parameter sets the BFM in PASSIVE mode.

The `EXT_CLK_RESET` parameter determines if the default clock and reset generators within the Configurator generated module are instantiated. When set to 1, then the clock and reset generators are not instantiated. This is considered EXTernal clock and reset generation. The clock and reset signals from `hdl_top.sv` must be connected to the Configurator generated module's clock and reset signals. When set to 0, then the clock and reset generators are instantiated within the Configurator generated module and connected to all QVIP BFMâĂŹs within the module.

5.1.5 Automatic Code Merging

You can point to a previous iteration of your generated code when producing a new version of the code with modified YAML input. In doing so, the script will attempt to extract any hand-edits from the old version and place them into the new output. This is accomplished through the use of labeled blocks within the source - all hand edits must be made within these blocks in order to ensure everything is reliably transferred.

The `yaml2uvmf.py --merge_source` switch enables this capability. An older generated output directory that underwent hand edits should be pointed to with this switch.

5.1.5.1 Labeled Blocks

All hand-edits should be placed within what are called "labled blocks" of generated output. These blocks are bordered with special "pragma" comments that look as follows:

```
// pragma uvmf custom <block_name> begin
// INSERT CUSTOM CODE HERE
// pragma uvmf custom <block_name> end
```

The user can search for these pragma blocks in addition to `UVMF_CHANGE_ME` in order to track down areas of code that will require hand edits.

All blocks must begin with a `begin` pragma and end with an `end` pragma using the same label name. Blocks may not be nested, and all label names must be unique within a given file.

There must be a matching labeled block between newly generated code and hand-edited code in order for data transfer to succeed. For this reason, users should not create new labeled blocks in their hand-edited code, as they will not transfer automatically. Errors will be raised if labeled blocks cannot be matched.

5.1.5.2 Code Merging Flow

When invoking this capability, the usual output directory called `uvmf_template_output` by default will not be created. Instead, any updated code will be overlaid on top of the hand-edited code pointed to by the `--merge-source` switch.

It is possible that the merged result will not compile or run as a result of incompatibilities between the old source and new YAML (new interface instantiations, changes in parameterization, etc.). The debug switch `--merge-debug` can be used to view intermediate directories if needed.

5.1.5.3 Merging Rules

- A 'matching file' is a file name that was found in both the old source and new source.
- Any labeled block found in a matching file will have its contents maintained.
- Any labeled block in a matching file that did not exist in the old output will be copied over into the old output with default content.
- Any labeled block in a matching file that was found in the old output but not in the new source will, by default, produce an error message and cauase the script to exit.
  - If the `--merge_skip_missing_blocks` switch is used, processing will continue and a list of missing blocks and their locations will be produced

at the end of the run. The user will need to consider transferring these blocks manually.

- Any file found in the new source but not in the old will be copied into the old source directory structure. Any file found in the old source that wasn't matched with something new will be left in place.

5.2 UVM Objection Tracer

A Python script called `uvm_objection_trace.py` is available under `$UVMF_HOME/scripts` and can be employed to deal with situations where a UVM simulation (UVMF or not) is not finishing properly as a result of a suspected objection not dropping.

The user must first run their simulation with the UVM standard `+UVM_OBJECTION_TRACE` plusarg on the command-line in order to produce more verbose output. This output can then be passed into the script for analysis.

Run `uvm_objection_trace.py --help` for command-line options and more detailed information.

5.3 UVMF Build/Compile/Run Script

The `uvmf_bcr.py` script is available under `$UVMF_HOME/scripts` and can be used to drive the simulation flow as an alternative to using the make files or any other customized, home-grown scripting solution. The `uvmf_bcr.py` script employs YAML-based configuration files to determine the desired compile and run commands as well as how to build complete file lists necessary to compile the desired testbench.

5.3.1 Usage

```
uvmf_bcr.py [options] [override1 [override2] ...  [overrideN]]
```

Option	Description
`--version`	Show script version number and exit
`--help | -h`	Show help message and exit
`--quiet | -q`	Suppress output while running
`--flow_file=FILE | -f FILE`	Override default flow file of `$UVMF_HOME/scripts/mentor.flows`

`--flow_file_overlay=FILE`	Provide list of flow data files to overlay on top of main flow file. If multiple files are listed, separate each with colon (":")
`--flow=FLOW | -f FLOW`	Specify the desired flow, overriding the default
`--steps=STEPS | -s STEPS`	Specify the desired steps to run, overriding the flow default. If multiple steps are required, use double-quotes to encapsulate the list separated with spaces
`--dry_run | -n`	Print resulting commands, do not run anything
`--list_flows`	List out all available flows
`--list_steps`	List out all available steps for the specified flow
`--list_variables`	List out all available variables for the specified flow and steps
`--filelists_only | -l`	Only produce file lists and stop
`--clean | -c`	Runs the `clean` step for the desired flow
`--skip_filelist_build`	Disables automatic creation of file lists
`--sim_dir`	Specify top directory for compile file structure

5.3.2 General Flow

The build/compile/run script reads a series of YAML files to determine both the command flow to employ as well as the requisite input files to use during the compile process. The script operates through the following steps:

- **Build**: Build a set of file lists
- **Compile**: Compile everything using the compile lists
- **Run**: Run the simulation

5.3.2.1 Flows, Steps & Variables

A flow file defines one or more available "flows" that can be employed. One flow is designated as default but the user can specify the desired flow using the `--flow` or `-f` switch to `uvmf_bcr.py`. A list of available flows along with descriptions can be produced with the `--list_flows` switch.

The flows are defined in a "flow file" that can be specified by the user but a default file is read in automatically and exists as `$UVMF_HOME/scripts/mentor.flows`. This particular flow file describes several commonly used flows that support both pure Questa-based simulation and Veloce-based emulation-driven verification.

Each flow consists of one or more "steps" which will be carried out in the order as defined by the flow. By default, all steps associated with the chosen flow will be executed but the user can control this with the `--steps` switch to `uvmf_bcr.py`. A list of available steps along with their descriptions can be produced with the `--list_steps` switch.

Each flow also defines a set of variables and respective default values. Variable overrides can be specified on the command-line in order to produce different behavior for individual runs. Examples of variables for a standard simulation flow would be the name of a test, a desired simulation seed, and the desired verbosity level for a test. Variables are overridden using `var_name:var_value` syntax on the `uvmf_bcr.py` command-line as shown in the following example:

```
uvmf_bcr.py test:my_test seed:1234
```

In this example, the `my_test` test will be executed using a seed value of `1234`.

A list of available variables along with their descriptions can be produced with the `--list_variables` switch. Variables can be associated with a given flow as well as a specific step. All flow-level variables are available to each underlying step.

The flow file format is YAML-based with a predetermined schema. Most users should be able to use the default flow file `$UVMF_HOME/scripts/mentor.flow`. Use the existing content of this file as a guide for any desired customization or alternatives.

5.3.2.1.1 Default Mentor Flow File

The `mentor.flows` file defines the following flows and steps. Use the `--list_variables` switch to get a comprehensive listing of all available variables for a given flow and switch. In addition, all flows define a `clean` step that will clean up files and directories from earlier runs which can be invoked manually or through the `-c` or `--clean` convenience switches.

Flow	Step	Description
`questa`		Run simulation with Questa using `qrun` executable (Default)

	`run`	Incremental build and run
`veloce`		Run a TBX simulation with HDL components running on the Veloce emulator
	`hdl_build`	Perform analysis and compile of the HDL source
	`hvl_build`	Perform analysis and compile of the HVL source
	`run`	Invocation of TBX simulation
`3step`		Run simulation in batch mode using the three-step Questa flow (`vlog/vcom`,`vopt`,`vsim`)
	`compile`	Compile code
	`optimize`	Optimize/elaborate
	`run`	Run simulation

Other less commonly used flows and steps may be available that are not listed here. Use `--list_flows`, `--list_steps`, and `--list_variables` for a comprehensive list and details for all available flows.

5.3.2.1.2 Common Variables

Most of the defined flows share a common set of variables that allow control over common items such as simulation seed, test names, and verbosity levels. The following list is not comprehensive but does outline some of the more commonly used variables.

Variable	**Description**
`test`	Specify the UVM test to run. Defaults to `test_top`
`seed`	Specify the random seed to use during the run. Defaults to 0
`live`	If set to `True` run an interactive sim. Defaults to `False`
`use_vis`	If set to `True` run using QIS and generate Visualizer waves. If combined with `live:True`, run an interactive simulation using the Visualizer GUI. Defaults to `False`
`use_vis_uvm`	Same behavior as `use_vis` but dump more information to the wave file for testbench debug. Defaults to `False`
`verbosity`	Specify UVM verbosity for run. Default unspecified (use default UVM verbosity)
`error_limit`	Specify how many errors can occur before simulation is halted. Default is 20

extra_do	Specify additional 'do' commands to invoke just prior to the 'run' simulation command
run_command	Override the usual `run -all` used to start the simulation
enable_trlog	When `True`, enable transaction stream recording. Defaults to `False`

5.3.2.1.3 Overriding Default Flow Behavior

In addition to using command-line switches and options, user-provided file input can be utilized to override default behavior defined in the default flow-file. The following environment variables are queried during initialization to determine both the start-up flow file as well as any static configuration overrides that should be applied to all script invocations for a given project, group, or user.

As stated elsewhere, the default flow file is located at `$UVMF_HOME/scripts/mentor.flows`. If a complete override of this file is required it is possible to do so via the `--flow_file` command-line switch but a more useful and consistent approach would be to utilize the `$UVMF_BCR_FLOW_FILE` environment variable to point to an alternative starting point. The script will attempt to load the flow file pointed to by this variable if it is defined in place of the default flow file.

A separate environment variable, `$UVMF_BCR_FLOW_FILE_OVERLAY`, or an associated command-line switch, `--flow_file_overlay` can be used to overlay modifications to the base flow-file (default or otherwise). This can point to one or more additional flow file snippets that can be used to override default flow behavior as well as define additional flows and variables. The format is expected to be in the form of a colon-separated list of file paths, similar to a `$PATH` search path. Each file will be parsed in order, so any conflicts will be resolved as "last wins". If the command-line switch and the environment variables are not specified but a file named `bcr_overlay.flow` exists in the invocation directory, that file will be read in as an overlay file.

These overlay files must follow the same format as the default flow file but only snippets are required. There is an example overlay file in `$UVMF_HOME/scripts/overlay_example.flow` which modifies a handful of default values. Use that as well as the default `mentor.flow` file for reference.

5.3.2.1.4 Setting Environment Variables

An overlay file can be used to set environment variables needed by the verification flow. This is accomplished by defining the `env_vars` section as shown here:

```
options:
  env_vars:
    MY_ENV_VAR_1:  <value of MY_ENV_VAR_1>
    MY_ENV_VAR_2:  <value of MY_ENV_VAR_2>
    MY_ENV_VAR_3:
      value:  <value of MY_ENV_VAR_3
      resolve_path:  True
```

The basic form is used to define the first two environment variables as simple string values. The third example will be treated as a file path which will be converted into an absolute path with respect to the current working directory (invocation directory of the script). This can be useful and sometimes necessary when using environment variable references within the `needs` section of compile files and/or within a nested file list due to how relative file paths are calculated within those contexts.

5.3.2.2 Compile Files

Once a flow has been chosen along with a series of steps and all variables have been elaborated, YAML-based compile files are parsed in order to produce one or more file lists that will drive the compile stage of the script. A compile file can contain the following information:

- **Source Files**: Source files (Verilog, SystemVerilog, VHDL, C, etc)
- **Switches**: Compile-time switches required to compile the given source
- **Include Paths**: File paths to specify as part of a compile operation (e.x. `+incdir`)
- **Include Files**: Other compile files that must be included before any local files are compiled

5.3.2.2.1 Generated Compile File Tree

The following diagram illustrates how all of the `*.compile` files are organized by default as part of a generated bench by `yaml2uvmf.py`:

- *bench*`/sim/top.compile`
  - *bench*`/sim/hvl.compile`
    - *bench*`/tb/testbench/hvl_top.compile`
      - *bench*`/tb/tests/tests_pkg.compile`
        - *bench*`/tb/parameters/parameters_pkg.compile`
        - *bench*`/tb/sequences/sequences_pkg.compile`

```
        └── top_env/env_pkg.compile
            ├── sub_env_A/env_pkg.compile
            ├── sub_env_B/env_pkg.compile
            ├── ...
            ├── sub_env_n/env_pkg.compile
            ├── intf_A/intf_pkg.compile
            ├── intf_B/intf_pkg.compile
            ├── ...
            └── intf_n/intf_pkg.compile
└── bench/sim/hdl.compile
    ├── bench/rtl/rtl.compile
    └── bench/tb/testbench/hdl_top.compile
        ├── bench/tb/parameters/parameters_pkg.compile
        ├── intf_A/intf_bfms.compile
        ├── intf_B/intf_bfms.compile
        ├── ...
        └── intf_n/intf_bfms.compile
```

User customization will typically involve modifying the `rtl.compile` file in order to compile the target DUT as part of the given testbench. Any other custom code (non-UVMF agents, modules, packages) that were not originally generated by other UVMF scripts may also need to be incorporated by editing or adding more `*.compile` files.

By default, the top-most compile file(s) are assumed to be defined in the same directory as where `uvmf_bcr.py` is invoked (a project bench's `sim` directory, for example) but this does not always need to be the case. The script can be invoked from any other location by either specifying the `--sim_dir` switch or by using a flow overlay that specifies the `sim_dir` option as follows:

```
options:
  sim_dir:  <path_to_sim_directory>
```

In either case, the provided path can be relative or absolute, and the top-most compile files for the desired flow will be read from that directory. If the command-line switch *and* the overlay are provided, the command-line switch will take precedence.

5.3.3 Compile File Schema

All compile files may contain up to four different top-level sections:

- `needs`: A list of any underlying `.compile` files that should be processed prior to processing this file.

- `src`: A list of source code (Verilog, VHDL, etc) associated with the compile file.
- `incdir`: A list of include directories required to successfully compile the listed source files. The underlying flow file determines how these include directories are processed but one example would be the application of `+incdir+` command-line options.
- `options`: A list of compile-time options to use against the provided source code. For example, `+define` calls or a `-timescale` directive. Any options specified will be applied to all compile commands unless an explicit association is made.

Specified file paths may be absolute or relative. If a relative file path is provided it should be relative to the location of the compile file itself. Environment variables can be used as well, simply use the standard `$ENV_NAME` format (no curly brackets or parenthesis).

The following is a small example of a typical compile file which utilizes all of the sections mentioned above:

```
needs:
  - $UVMF_HOME/uvmf_base_pkg/uvmf_base_pkg.compile
  - ../../interface_packages/mem_package/mem.compile
  - ../../interface_packages/pkt_package/pkt.compile

incdir:
  - $UVMF_HOME/uvmf_base_pkg

src:
  - registers/block_b_reg_pkg.sv
  - block_b_env_pkg.sv
  - dpi/my_dpi_file.c
  - extra_vhdl_stuff.vhd

options:
  - -warning error # Sent to all commands
  - ['vlog','+define+SPECIAL'] # Only send to Verilog compile
```

In this example, a UVMF environment is described that is instantiating underlying `mem` and `pkt` interfaces, hence the need to incorporate those underlying interface compile files. The `src` section lists files of various types (SystemVerilog, C and VHDL) which the script will automatically distribute to separate file lists and tools if necessary. The first option is global and will be distributed to all compile commands. The second option is associated only with the `vlog` file type, meaning it will only be applied to Verilog compile operations.

In most generated UVMF benches the only compile file that typically needs to be updated/edited by the end user will be <bench_name>/rtl/dut.compile, which defines the location and method to compile the target Device Under Test. Modifications to the generated bench typically take place within files that are included by top-level SystemVerilog package files that are already mentioned in existing compile files, meaning that very few user changes outside of the `dut.compile` file should be needed to the overall compile file structure.

Chapter 6

Verification Reuse

6.1 Overview

Reuse is an important factor in reducing verification schedule and finding bugs at each level of integration. Reuse can be divided into three main categories: horizontal, vertical, and platform. Horizontal reuse is the use of verification components from project to project. An example of this is an interface package or protocol VIP. The agent, transactions, sequences, BFMâĂŹs, etc. within the package can be used on any project that uses the protocol. Vertical reuse is the use of verification environments from block to chip to system level simulations. An example of this is an environment for an encryption engine. This environment can be used in block level simulations of the engine alone. This environment can be reused in subsystem level simulations that include a fabric and connected devices, including the engine. This environment can be reused in chip level simulations that include the processor, fabric, and all devices connected to the fabric. This environment can be reused in system level simulations that include multiple chip level environments. Platform reuse is the use of verification components across different execution engines such as simulation, emulation, or prototyping. Verification components should be reusable horizontally, vertically, and across platforms without modification. The architecture of UVMF based verification components guarantees horizontal, vertical, and platform reuse. The rest of this chapter will discuss topics related to reuse. Additional information on UVMF can be found in the âĂİUVM Framework âĂŞ One Bite at a TimeâĂİ video series on Verification academy at the following link:
UVM Framework âĂŞ One Bite at a Time

6.2 Interface Reuse

The UVMF generator creates an interface package and BFMs that are reusable in all three dimensions described above. If the `veloce_ready` section is set to `True` in the interface YAML file, the generated interface is ready for use in simulation and emulation. Environments that use UVMF based interface agents are ready for block to top platform reuse. A video describing the archtecture and operation of UVMF agents can be found at the following link:
uvmf-agents-architecture-and-operation

6.3 Environment Reuse

The UVMF generator creates an environment package that is reusable vertically and across platforms. An environment can instantiate another environment by using the `subenvs` section in the parent environments YAML file. There is no limit to the number of sub-environments an environment can instantiate. There is no limit on how many levels of environments reside between the single top-level environment and the lowest block-level environment. The `veloce_ready` YAML section is not relevant to generating environment packages since environment code only executes within a simulator. All of the environment code always executes within the simulator, even when using emulation for acceleration.
An example of generating block level environments and chip level environments that reuse block level environments are provided in the block_a, block_b, and chip examples described in this document and provided in
`$UVMF_HOME/templates/python/examples/yaml_files`.
A video describing the architecture and operation of enviornments can be found at the following link:
uvmf-environments-architecture-and-operation

6.4 Test Bench Reuse

The UVMF generator creates a test bench that is reusable across platforms if the `veloce_ready` section is set to `True` in the bench YAML file. The test bench includes the top level modules, top level environment, and top level sequence. A different DUT requires a different top level module and top level test. Test stimulus can be reused if the stimulus is specified in a sequence that is included in the environment package. DUT specific sequences that can be used in upper level simulations should be placed in the environment package. DUT specific sequences that cannot be reused in upper level simulations should be in the bench level sequence package. The "Sequence Categories" video in the "UVM Framework âĂŞ

One Bite at a Time" video series on Verification academy describes where to place sequences based on their use and reuse and can be found at the following link:
uvmf-sequence-categories
A video describing the architecture and operation of a UVMF test bench can be found at the following link:
uvmf-testbench-architecture-and-operation

6.5 Performance Considerations

As block and subsystem environments get encapsulated in chip and system level environments, simulation performance will decline. Two mechanisms have been identified that mitigate simulation performance degradation. The first is reducing monitoring activity by sharing monitors between environments connected to the same bus. The second is conditionally constructing sub-environments based on the goals of a particular test or a particular simulation run. These mechanisms can be used separately or in tandem. Both of these mechanisms require changes to generated code.

6.6 Sharing Monitors Between Environments

As environments become encapsulated in other environments, itâĂŹs likely that monitors within multiple environments will be observing the same bus. Having multiple monitor BFMs observe the same bus activity and multiple monitor classes broadcasting the same observed data consumes simulation cycles without providing additional value. It is possible for a single monitor BFM to be connected to a bus that is connected to multiple environments. This single monitor BFM would have a single monitor class that receives the observed data and broadcasts transactions to all subscribers. In this case, the subscribers will span multiple environments. This reduction in monitoring activity can provide significant reduction in simulation time depending on the number of environments that need data from the same bus and the number of busses that have multiple environments connected to them. UVMF provides a mechanism for sharing monitor classes across multiple agents. This mechanism is to instantiate and construct a monitor class in the environment and assign the monitor handle within the agent after construction of the agent. This shared monitor handle in the environment can be assigned by parent environments. This allows monitor handles to be passed down through multiple layers of environment hierarchy. It is important to note that the configuration object for agents that share the same monitor should have the same values for configuration variables that affect monitoring behavior. This can be done in test_top after the initialize call of the configuration object. It can also be done through constraints on agent configuration variables.

6.6.1 Steps for sharing monitor functionality between agents:

1. Add a monitor class handle within the environment that contains one or more agent to receive a monitor handle.
2. In the environment build phase, conditionally construct the shared monitor if the handle is null. Since the parent environments `build_phase` will be executed before the child environments `build_phase`, the parent environment can assign the monitor handle within the child environment before the child environments `build_phase` is executed.
3. Construct the agent(s) that will receive the shared monitor.
4. Assign the shared monitor handle to the monitor handle within the agent. The agent will only construct its own monitor if this handle is null when the build_phase function of the agent is executed.
5. Remove all but one of the monitor BFMâĂŹs from `hdl_top.sv` that would be observing the same bus.
6. Remove the uvm_config_db::set in hdl_top.sv for each removed BFM.
7. In the `interface_names` array within `test_top.svh`, copy the BFM name of the shared BFM and place it in the entry of each BFM that was removed.
8. Ensure that configuration variables that affect monitoring behavior have the same value in the configuration object of each agent that shares the monitor.

6.7 Configurable Environment Reuse

A chip level simulation that includes multiple subsystem and block level environments provides fine-grained prediction, coverage, and scoreboarding data. However, this comes at the cost of simulation performance. One approach to mitigating this degradation is to add chip level data checking and conditionally constructing sub-environments. This can be achieved by using predictors from lower level environment packages within the chip level environment. Chip level regressions can be run without constructing lower level environments. If a test fails, it can then be re-run with the sub-environments constructed. The failure identified at a chip level can then be identified at a subsystem or block level. This provides more granular information regarding where the failure occurred. Variables can be added to an

environments configuration that control the construction of sub-environments. The UVM Command Line Processor can be used to optionally set these flags to enable construction of sub-environments from the command line without recompilation. Additional information on the UVM Command Line Processor can be found at Verification Academy at the following link:
Command Line Processor

Chapter 7

Register Model Development

7.1 Overview

Register model development includes three major steps: definition, creation, and integration. The UVMF generator provides automation that creates a basic register model and fully integrates the register model into the generated environment. The UVMF generator also creates a register test and register test sequence within the generated bench.

7.2 Register Model Definition

The register map definition is typically found in the DUT specification. The definition includes the following: Memory blocks, registers, register fields, address offset, addresses, access mode, reset values, etc. This typically defines the memory map of the design. Just as the DUT is composed of hierarchical blocks, implementation of the memory map within the DUT is also typically composed of hierarchical blocks. Being aware of these hierarchical blocks is important because it is the basis of creating a reusable register model that can be used when moving from block level simulations to chip level simulations. Information about the register model is used in the next step of register model development, creation.

7.3 Register model creation

The UVM base class package provides classes that are used to characterize the register model of a DUT. These classes include memories, registers, register fields, register blocks, and bus maps. These classes are used to characterize the memory map defined in the DUT specification. The `uvm_mem` class is used to characterize a memory block. The `uvm_reg` class is used to characterize a register. The `uvm_reg_field` is used to characterize a field within a register. The `uvm_reg_block` is used to collect registers and memories into a hierarchical block. The `uvm_reg_block` can also contain other uvm_reg_blocks. This is how the register model hierarchy can reflect the hierarchy of the registers within the DUT. The register model is created by extending these classes to reflect the memories, registers, register fields, and register blocks of the design. Creating these class extensions can be done manually but is almost always carried out using an automated generation utility. The UVMF environment generator provides a way to specify that the environment being generated contains a UVM based register model. The YAML label `register_model` is used to enable a register model within an environment. When this label is used, the UVMF environment generator creates a skeleton register model for the environment. The name of the class extension of the uvm_reg_block is based on the environment name. The generated register model contains example registers, register fields, coverage, etc. If the environment contains sub-environments that have their own register models, an instance of each sub-environments register model is instantiated in the parent register model extension of uvm_reg_block. This forms the basis of a hierarchical register model that enables register model reuse from block to chip level simulations. Note that if an environment contains a register model, each encapsulating environment must also be defined as having a register model, even if the encapsulating environment does not add any registers or memories. The environmentâĂŹs generated register model can either be modified or it can be replaced with a register model generated using a register generator tool. If it is replaced with a register model generated using a register generator tool, using the same register block class names and register model package names will alleviate having to modify the environment class and environment configuration class to reflect the name changes.

7.4 Register model integration

Once a register model is created, it needs to be integrated into the environment. The `register_model` YAML label automatically integrates the register model in the generated environment. In the UVMF, the register model is instantiated in the environmentâĂŹs configuration class. The initialize function of the environmentâĂŹs configuration class receives a handle to a register block. If this handle is null, the environment configuration constructs its register model. If the handle is non-null, this means that the register block is a sub-block of a hierarchical register model. In this case, the handle passed in through the initialize function is used as the environments register model. The UVM register adapter and predictor are instantiated and constructed in the environment class based on settings within the `register_model` label in the YAML file used to characterize the environment. The sequencer of the interface agent identified in the `register_model` label is connected to the register models map within the environment class.

7.4.1 Register Adapter

When the interface agent identified within the `register_model` label is a UVMF based agent, the generator expects a register adapter to be present in the package that contains the agent. The UVMF interface generator automatically creates a register adapter and includes it in the package. Once generated, the user will have to complete the register adapter by completing the `bus2reg` and `reg2bus` functions. In these functions, the user will map protocol specific variables to the generic register model variables. The generated `bus2reg` and `reg2bus` functions contain example code. This register adapter will automatically be instantiated, constructed, and connected in the generated environment.
When the interface agent identified within the `register_model` label is a QVIP based agent, using the `qvip_agent` YAML label, the generator will instantiate a generic register adapter. This generic register adapter must be replaced by the desired register adapter from the QVIP installation. These adapters are contained in the protocol package from the QVIP installation. This package is automatically compiled as part of compiling QVIP configurator generated code. The protocol package will beed to be imported into the UVMF environment package for the register adapter to be available for use. Detailed instructions for inserting a protocol specific register adapter from QVIP is located in the generated environment class. Look for `QVIP_AGENT_USED_FOR_REG_MAP` within the generated environment for locations of and instructions on needed changes.

7.4.2 Register Predictor

When the interface agent identified within the `register_model` label is a UVMF based agent, the generator will instantiate a `uvm_reg_predictor` parameterized to the agents transaction type including any parameter overrides performed in the instantiation of the agent.
When the interface agent identified within the `register_model` label is a QVIP based agent, using the `qvip_agent` YAML label, the generator will instantiate a `uvm_reg_predictor` parameterized to a `uvm_sequence_item`. This parameterization must be replaced with the sequence item used by the selected register adapter. Be sure to include and any sequence item parameterizations.

7.4.3 Register Predictor Connection to Sequencer

The connection between the agent `analysis_port` and the `analysis_export` of the `uvm_reg_predictor` is commented out in the generated environment. This is due to a UVM register package bug. The bug is exposed when the environment is used as a sub-environment. This is because the UVM register package does not construct sub-maps within register sub blocks. Construction of the sub-maps in the register model must be done manually. Then the generated line in the environment can be uncommented. This situation is also described in the generated environment class definition.

7.5 Register Tests

The bench generated by UVMF includes a register test. The name of the register test is `register_test`. This test executes the generated sequence named `register_test_sequence`. If the environment hierarchy does not contain a register model, the `register_test_sequence` is largely blank so that registers can be accessed as desired by the user. If the environment hierarchy contains a register model, the `register_test_sequence` instantiates, constructs, and executes the UVM built-in `uvm_register_test_seq` sequence. The `register_test_sequence` can be modified to execute only the desired operations such as `UVM_DO_REG_BIT_BASH`, `UVM_DO_MEM_ACCESS`, etc. The register test can be executed from the command line using the following command: `make cli TEST_NAME=register_test`

7.6 Register Model Examples

The wb2spi environment example, under `base_examples`, contains a block level register model. The ahb2spi environment example, under `base_examples`, is defined as having a register model and instantiates the wb2spi environment as a sub-environment. This example shows how to define a block level register model and how UVMF reuses the block level register model in a chip level environment and register model.

Chapter 8

Data Flow Within Generated UVMF Agents

8.1 Data Flow Within a Generated Interface

The interface generated by the UVMF code generators actively drives and monitors data as generated. The driver automatically requests a transaction from the sequencer, waits a few clocks, then requests another transaction. The monitor automatically broadcasts default values every few clocks.

The `do_monitor` task within the monitor BFM must be completed in order to broadcast observed signal values. The `initiate_and_get_response()` task within the driver BFM must be completed in order to see pin activity on the bus for an agent configured as an initiator. The `respond_and_wait_for_next_transfer()` task within the driver BFM must be completed in order to see pin activity on the bus for an agent configured as a responder. The sections below outline the flow of data within the generated interface components.

8.2 Data Flow Within a Generated Monitor

The diagram below illustrates the monitor flow within an initiator or responder:

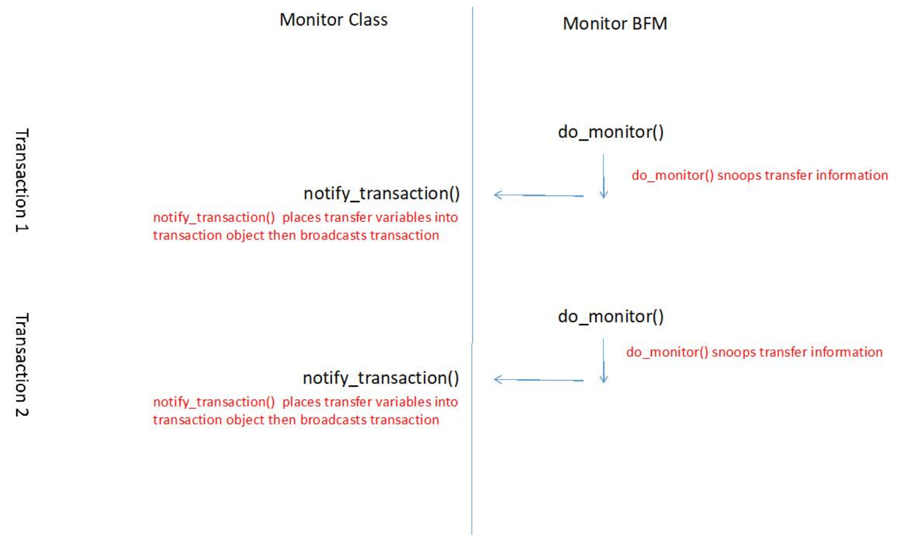

Figure 8.1: Generated UVMF Monitor Data Flow

The generated interface components associated with monitoring and broadcasting observed signal activity are operational as generated. The flow outlined below describes the setup, enabling, and operation of this flow. This flow is described for an interface protocol named `mem`. This flow is automatically implemented within the generated code.

Setup that enables monitoring:

1. `mem_monitor.svh`: `set_bfm_proxy_handle()` is called in `uvmf_monitor_base::connect_phase()` to place a handle to this UVM-based monitor within the `mem_monitor_bfm`. This handle is used by `mem_monitor_bfm` to send observed data to `mem_monitor` for broadcasting through the agent's analysis port. This is a setup activity automatically performed once in the connect phase.

2. `mem_monitor.svh`: `start_monitoring()` is called in `run_phase()` to enable signal monitoring within `mem_monitor_bfm`. This is an enabling activity automatically performed once in the run phase.

Steps performed to observe a transfer and broadcast a transaction object:

1. `mem_monitor_bfm.sv`: Once monitoring is enabled, a forever loop is entered that performs the following two steps:
   (a) `do_monitor()`: This task observes and captures signal values according to the protocol. Observed values are returned through the task argu-

ments. **This is the task that should be modified/customized by the user to be protocol specific.**

   (b) `proxy.notify_transaction()`: This function takes values from the `do_monitor()` task and sends them to `mem_monitor` for broadcasting in the UVMF environment. The proxy variable is a handle to `mem_monitor` initialized in Step 1. Since `notify_transaction()` is a function it returns in zero time allowing `do_monitor` to immediately return to observing signal activity.

2. `mem_monitor.svh`: The `notify_transaction()` function performs the following steps when called:
   (a) Construct a transaction for broadcasting
   (b) Set values in the transaction based on values received as arguments to `notify_transaction()`.
   (c) Add the transaction to the waveform transaction viewing stream if the `enable_transaction_viewing` flag is set in the agent configuration.
   (d) Broadcast the transaction out of the agent analysis port named `monitored_ap`.

8.3 Data Flow Within a Generated Driver

The generated interface components associated with driving signal activity are operational as generated. The flow outlined below describes the operation of this flow. This flow is described for an interface protocol named `mem`. The flow for an a gent acting as an initiator is different than the flow for a responder agent. Each of these flows are described below.

8.3.1 Driver Flow for an Initiator

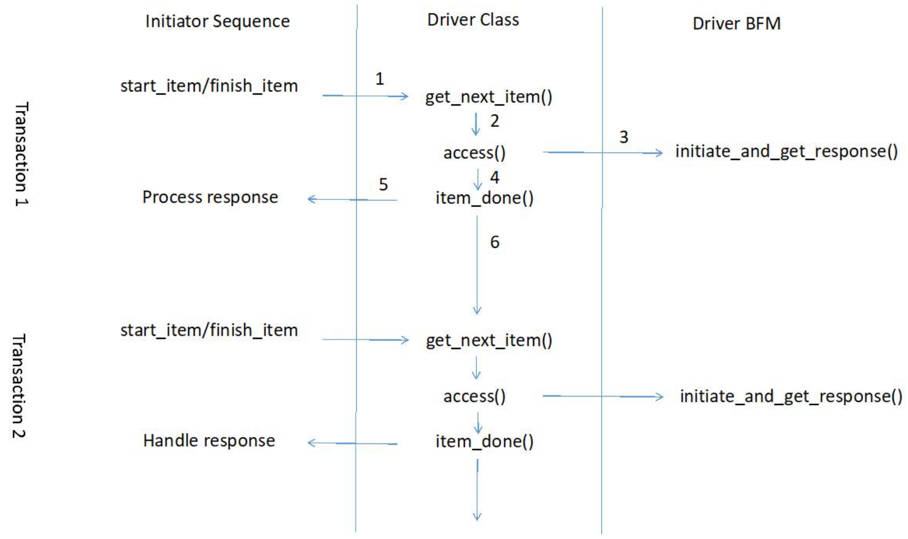

Figure 8.2: Generated UVMF Driver Data Flow (Initiator)

An initiator, or master, agent initiates the transfer of information. The information for this transfer is received from a sequence item. The agent requests this information from a sequence according to the protocol timing. The flow listed below includes steps performed at the sequence for a protocol named `mem`.

1. `mem_random_sequence.svh`: Executing the `start()` task of this sequence executes the `body()` task within this sequence. The `body()` task performs the following steps:
   (a) Construct a `mem_transaction` named `req`.
   (b) Execute `start_item(req)` to indicate to sequencer in `mem_agent` that a transaction is available for the driver. This call blocks until the driver requests a transaction from the sequencer through executing the `get_next_item()` task. When `get_next_item()` is executed, `start_item()` is unblocked and `finish_item()` executes which unblocks `get_next_item()`.
2. `mem_driver.svh`: When `get_next_item()` unblocks, the transaction handle received is passed to `access()`. The `access()` task is located in `mem_driver.svh`. The call to `get_next_item()` is located in the `uvmf_driver_base` class.

3. `mem_driver_bfm.svh`: The `access()` task calls `initiate_and_get_response()` task. This task has two arguments. The first argument is an initiator struct. The second argument is a responder struct. The elements in the initiator struct are used to initiate a transfer. The elements in the responder struct are used to pass response data back to the sequence. The `initiate_and_get_response()` task will initiate a transfer, wait for the responder to reply, and gather the responder data. **The contents of this task are meant to be customized/modified by the user in order to define protocol-specific behavior for the interface driver when configured as a bus initiator.**

4. `mem_driver.svh`: When the `initiate_and_get_response()` task returns, the `access()` task completes and `uvmf_driver_base` calls `item_done()`.

5. `mem_driver.svh`: Execution of `item_done()` unblocks `finish_item()`. The initiator sequence can then process the data received from the responder.

6. `mem_driver.svh`: After execution of `item_done()`, `uvmf_driver_base` immediately exectutes `get_next_item()` to request another sequence item.

8.3.2 Driver Flow for a Responder

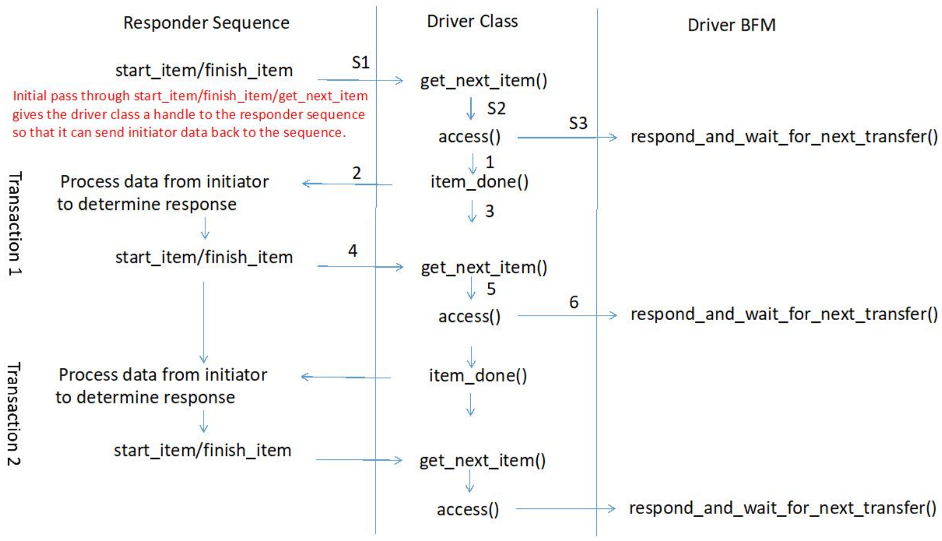

Figure 8.3: Generated UVMF Driver Data Flow (Responder)

A responder, or slave, agent responds to a transfer started by an initiator or master. Once a transfer is initiated the responder agent sends information about the transfer to the sequence. The sequence returns information needed by the responder agent to complete the transfer. The flow listed below includes steps performed at the sequence.

Setup Steps:

1. `mem_responder_sequence.svh`: Executing the `start()` task of this sequence executes the `body()` task within this sequence. The `body()` task performs the following steps:
   (a) Construct a `mem_transaction` named `req`.
   (b) Execute `start_item(req)` to indicate to sequencer in the agent that a transaction is available for the driver. This call blocks until the driver requests a transaction from the sequencer through executing the `get_next_item()` task. When `get_next_item()` is executed, `start_item()` is unblocked and `finish_item()` executes which unblocks `get_next_item()`.
2. `mem_driver.svh`: When `get_next_item()` unblocks, the transaction handle received is passed to `access()`, located in `mem_driver.svh`. This makes a call to `get_next_item()` is defined in the `uvmf_driver_base` class.
3. mem_driver_bfm.svh: The `access()` task calls `respond_and_wait_for_next_transfer()` task. This task has two arguments. The first argument is an initiator struct. The second argument is a responder struct. The elements in the responder struct are used to pass response data to the driver BFM to complete the transfer. The elements in the initiator struct are used to send initiator data from the transfer back to the responder sequence. The `respond_and_wait_for_next_transfer()` task will respond to the current transfer, wait for another transfer to be initiated, then send the initiator data to the responder sequence to determine how to respond. For this first call to `respond_and_wait_for_next_transfer()` there is no current transfer to complete. Therefore, `respond_and_wait_for_next_transfer()` will immediately start waiting for the next transfer. **The contents of this task are meant to be customized/modified by the user in order to define protocol-specific behavior for the interface driver when configured as a bus responder.**

Transaction flow steps:

1. `mem_driver.svh`: When `initiate_and_get_response()` returns, the initiator struct argument contains data from the initiator. The `access()` task then passes the initiator data to the responder sequence and calls `item_done()`

2. `mem_responder_sequence.svh`: The sequence processes the data from the initiator and determines how to respond. Response data is sent to the driver by executing `start_item()` and then `finish_item()`.

3. `mem_driver.svh`: After executing `item_done()`, `get_next_item()` is executed to retrieve the next sequence item from responder sequence.

4. `mem_driver.svh`: When the responder sequence executes `finish_item()`, the responder struct argument contains data for completing the current transfer.

5. `mem_driver.svh`: Data from the responder struct is passed to the `access()` task for transfer to the driver BFM.

6. `mem_driver.svh`: The `access()` task calls the `respond_and_wait_for_-next_transfer()` task. This task has two arguments. The first argument is an initiator struct. The second argument is a responder struct. The elements in the responder struct are used to pass response data to the driver BFM to complete the transfer. The elements in the initiator struct are used to send initiator data from the transfer back to the responder sequence. The `initiate_and_get_response()` task will respond to the current transfer, wait for another t ransfer to be initiated, then send the initiator data to the responder sequence to determine how to respond.

Chapter 9

Resource Sharing within the UVM Framework

9.1 Overview

The UVM Framework uses direct assignment to make most resources available across interfaces, environments, and bench level code. The UVM config database is used in a limited amount to pass resources. The resources available include virtual interface handles, sequencer handles, agent configuration handles, and environment configuration handles. The methods and operations listed in this section are provided in the `uvmf_base_pkg` or source generated by the UVMF code generator. The user need not write any code to implement what is described below.

9.2 Accessing UVMF Interface Resources

9.2.1 Agent Configuration Handles

The agent configuration object is located in the environment configuration object. The agent configuration contains a handle to the agent's sequencer, driver BFM virtual interface handle, and monitor BFM virtual interface handle.

The agent configuration places itself in the UVM config db using 'null' for the context and `UVMF_CONFIGURATIONS` as the `inst_name` arguments. The `interface_name` argument in the agent configuration is used for the `field_name` argument.

The top level virtual sequence base retrieves the agent configuration handles from the config db. Through the agent configurations, the top level virtual sequence base has access to all agent configurations for starting sequences on any sequencer.

9.2.2 Virtual Interface Handles

The monitor and driver BFM virtual interface handles are available through the UVM config db. The virtual interface handles are placed into the database by `hdl_top`. The context argument is null, the `inst_name` argument is `UVMF_VIRTUAL_INTERFACES`, the `field_name` argument is the `interface_name` variable in the agent configuration class. The monitor and driver BFM virtual interface handles are retrieved by the agent configuration in the `initialize()` function of the `uvmf_parameterized_agent_configuration_base` class. The agent configuration `initialize()` function receives the `interface_name` as an argument. This argument is used to retrieve the monitor and driver BFM handles from the UVM config db.

9.2.3 Sequencer Handles

The sequencer handle is made available to the top level virtual sequence through the agent configuration handle.

The sequencer handle is also available for retrieval from the config db. The `cntxt` argument is null, the `inst_name` argument is `UVMF_SEQUENCERS`, the `field_name` argument is the `interface_name` variable in the agent configuration object. Though the sequencer handle is available in the config db, it is not retrieved from the database by the UVMF or any generated code. This is to avoid simulation performance impacts caused by `uvmf_config_db::get()` calls.

9.3 Accessing UVMF Environment Resources

9.3.1 Environment Configuration Handle

The top level environment configuration object is instantiated in the `uvmf_test_base` class. The `uvmf_test_base` assigns the environment configuration handle into the environment during the build_phase.

The top level environment configuration handle is available for retrieval from the config db. The `cntxt` argument is null, the `inst_name` argument is `UVMF_CONFIGURATIONS`, the field name is `"TOP_ENV_CONFIG¨`. Though the top level environment configuration handle is available in the config db, it is not retrieved from the database by the UVMF or any generated code. This is to avoid simulation performance impact caused by `uvmf_config_db::get()` calls.

Chapter 10

Environment and Interface Initialization within the UVM Framework

10.1 Overview

The UVM Framework is architected for reuse. One key characteristic of reuse is self-containment. Reusable components automatically get needed resources, construct and configure children components and make internal resources available to other components. The two mechanisms for applying this are the `initialize()` function and `set_config()` function.

The `initialize()` function passes information down through the configuration object hierarchy. This information configures environments and agents in a simulation. It begins at the top level UVM test and ends at agent configurations. It is the mechanism by which all agents are initialized.

The `set_config()` function passes configuration object handles down through the environment hierarchy.

It is important to note that the code listed below is automatically generated when generating an interface package, environment package and project bench.

10.2 Top-down Initialization Through `initialize()`

The `initialize()` function passes information down through the configuration hierarchy. It starts at the top level UVM test and ends at the agent configurations. The configuration class for each agent in a simulation is initialized using this mechanism. At the top level UVM test and environment level the function passes the following information:

- Simulation level
- Hierarchical path down to the configurations environment
- An array of string names that uniquely identify each interface in the design.

At the agent level the `initialize()` function passes the following information:

- Active/passive state of the agent
- Hierarchical path to the configurations agent
- A unique string identifying the agents handle to the interface BFMs (driver BFM and monitor BFM).

The following is a walkthrough of typical `initialize()` function behavior as defined in a bench's base test class (`test_top` by default). Keep in mind that the `build_phase()` function of this component completes before the environment `build_phase()` is executed. This allows for the configurations to be constructed and initialized before the environment hierarchy is constructed.

```
string interface_names[] = {
    ahb_pkg_ahb_BFM /* ahb     [0] */ ,
    wb_pkg_wb_BFM /* wb     [1] */
};

uvmf_active_passive_t interface_activities[] = {
    ACTIVE /* ahb     [0] */ ,
    ACTIVE /* wb     [1] */
};

```

Figure 10.1: `test_top` Initialization Code

- **Lines 30-33:** Create an array of strings that contain the unique names of each interface BFM in the simulation. These values are typically parameters defined in the `tb/parameters` directory.

- **Lines 35-38:** Create an array of agent activity settings. The order matches the interface name order.

```
64  virtual function void build_phase(uvm_phase phase);
65
66    super.build_phase(phase);
67    configuration.initialize(
68                             BLOCK,
69                             "uvm_test_top.environment",
70                             interface_names,
71                             null,
72                             interface_activities);
73  endfunction
```

Figure 10.2: `test_top` Initialization Code

- **Line 68:** Pass the simulation level to the environment. This argument can be used to set agent activity levels. However, a much more flexible mechanism is to use the array of agent activity levels, which is what the UVMF code generator does.

- **Line 69:** Pass the path to the environment. Sub-environments will append child component hierarchical names to this path and pass the result down through the `initialize()`function.

- **Line 70:** Pass the array of agent interface names to the environment. Each environment is responsible for distributing these names to their own sub-environments and interfaces. Ultimately, each agent configuration will receive its own unique interface name. The agent configuration uses this name to retrieve interface BFM handles from the UVM configuration database. The name is also used by the agent to place its sequencer in the `uvm_config_db` for retrieval by the top-level sequence.

- **Line 71:** Pass the register model handle to the environment. This handle is the register block for this level of environment. Some environments do not require a register model so the default value is 'null'.

- **Line 72:** Pass the agent activity settings to the environment. Each environment will distribute these settings to sub-environments. Ultimately, each agent configuration will receive its own unique setting. The agent uses this value in its configuration to determine whether or not to construct its sequencer and driver. A passive agent does not have a driver BFM.

The following is a walkthrough of typical environment configuration class. It is executed by `test_top`. It is provided all of the information required to configure sub-environments and agents.

```
81   function void initialize(
82                            uvmf_sim_level_t sim_level,
83                            string environment_path,
84                            string interface_names[],
85                            uvm_reg_block register_model = null,
86                            uvmf_active_passive_t interface_activity[] = null
87                            );
88
89     super.initialize(
90                      sim_level,
91                      environment_path,
92                      interface_names,
93                      register_model,
94                      interface_activity
95                      );
96
97      ahb_config.initialize(
98                      interface_activity[0],
99                      {environment_path,".ahb"},
100                     interface_names[0]
101                     );
102
103     wb_config.initialize(
104                     interface_activity[1],
105                     {environment_path,".wb"},
106                     interface_names[1]
107                     );
108
109  endfunction
```

Figure 10.3: Environment Configuration Initialization Code

- **Line 89:** `super.initilize()` is called to perform tasks required by all environment configurations. It also contains debug code that is executed when the simulation verbosity is set to `UVM_DEBUG`.
- **Line 97:** The `initialize` function is called to pass the agent configuration information required for setup. This call includes the active/passive setting, the full path to the agent in the environment hierarchy, and the unique string name of this interface.
- **Line 98:** The activity setting determines whether the agent will be ACTIVE or PASSIVE.
- **Line 99:** Each agent has its own corresponding agent configuration. This path is used by the agent configuration to place itself in the UVM config db. The path is used to set the scope of the `uvm_config_db::set()` call so that only one agent receives this agent configuration. The agent whose hierarchical path matches this string receives this agent configuration.

- **Line 100:** The interface name uniquely identifying this agent. This variable is used by the agent configuration to retrieve the monitor BFM and driver BFM if the agent is configured as ACTIVE.
- **Lines 103-107:** These lines are used to pass initialization information to a second agent.

The following is a walkthrough of a typical agent configuration class content. It is executed by the environment configuration object. It is passed the active/passive state, the hierarchy down to the agent, and the interface name.

```
74  // ****************************************************************************
75     virtual function void initialize(uvmf_active_passive_t activity,
76                                      string agent_path,
77                                      string interface_name);
78
79      super.initialize( activity, agent_path, interface_name);
80
81      uvm_config_db #( ahb_configuration )::set( null ,agent_path,       UVMF_AGENT_CONFIG, this );
82      uvm_config_db #( ahb_configuration )::set( null ,UVMF_CONFIGURATIONS, interface_name, this );
83
84     endfunction
```

Figure 10.4: Agent Configuration Code

- **Line 79:** Call to `super.initialize()` to execute the initialize function in the base configuration class. The flow of this function is described in the next code section.
- **Line 81:** This configuration object places itself in the UVM config db for the agent identified by `agent_path` to retrieve.
- **Line 82:** This configuration object places itself in the UVM config_db using the string that identifies the interface BFMs used for this agent. This creates an automatic association between an interface and its configuration.

The following code is from `uvmf_parameterized_agent_configuration_base.svh`, whose class definition is extended by all UVMF agent configurations.

```
106 virtual function void initialize(
107                                    uvmf_active_passive_t activity,
108                                    string agent_path,
109                                    string interface_name);
110    active_passive      = activity;
111    this.interface_name = interface_name;
112
113    // Checking the config_db
114    void'(uvm_config_db #(uvm_bitstream_t)::
115      get(null,interface_name,"enable_transaction_viewing",enable_transaction_viewing));
116
117   if( !uvm_config_db #( MONITOR_BFM_BIND_T )::
118    get( null , UVMF_VIRTUAL_INTERFACES , interface_name , monitor_bfm ) )
119          `uvm_error("Config Error" ,
120           $sformatf("uvm_config_db #( MONITOR_BFM_BIND_T )::get cannot find resource %s",interface_name) )
121
122   if ( activity == ACTIVE ) begin
123      if( !uvm_config_db #( DRIVER_BFM_BIND_T )::
124       get( null , UVMF_VIRTUAL_INTERFACES , interface_name , driver_bfm ) )
125          `uvm_error("Config Error" ,
126           $sformatf("uvm_config_db #( DRIVER_BFM_BIND_T )::get cannot find resource %s",interface_name) )
127   end
128
129 endfunction
```

Figure 10.5: Base Agent Configuration Code

- **Lines 110-111:** Set the local activity level and interface string name variables from the arguments to the function call.
- **Lines 114-115:** Check the UVM config db for command-line setting of the `enable_transaction_viewing` flag for this interface.
- **Lines 117-120:** Retrieve the handle to the monitor BFM. Generate an error if the function fails.
- **Lines 122-126:** If the agent is configured as active then retrieve the handle to the driver BFM. Generate an error if the function fails.

10.3 Top-Down Passing of Environment Config Through `set_config()`

All environments should be extended from the `uvmf_environment_base`. The `set_config()` function from this class is shown below. This function is used by `test_top` and all lower environments to pass in the environment's configuration handle.

```
// FUNCTION : set_config
function void set_config( CONFIG_T cfg );
    configuration = cfg;
endfunction
```

Figure 10.6: `set_config()` Definition Code

This function is used to set the top-level environment config by the test (which occurs automatically for any test extending from `uvmf_test_base`). This function is also used to configure sub-environments by parent-environments, as shown here:

```
55  // *****************************************************************************
56    virtual function void build_phase(uvm_phase phase);
57      super.build_phase(phase);
58
59      ahb2wb_env = ahb2wb_environment::type_id::create("ahb2wb_env",this);
60      ahb2wb_env.set_config( configuration.ahb2wb_config );
61
62      wb2spi_env = wb2spi_environment::type_id::create("wb2spi_env",this);
63      wb2spi_env.set_config( configuration.wb2spi_config );
64
65    endfunction
```

Figure 10.7: `set_config()` Usage in Top-Level Environment

- **Line 59:** Construct the first sub-environment
- **Line 60:** Pass the first configuration handle into the first sub-environment
- **Line 62:** Construct the second sub-environment
- **Line 63:** Pass the second configuration handle into the second sub-environment

Chapter 11

Enabling Transaction Viewing within the UVM Framework

11.1 Overview

The code that is responsible for transaction viewing in the waveform viewer is in three locations: agent configuration, agent monitor and transaction class. The agent configuration contains a variable that turns transaction viewing on and off. The agent monitor creates the transaction viewing stream and calls the function in the transaction class that adds the transaction to the stream. The transaction class adds itself to the transaction stream.

11.2 UVM Framework Transaction Viewing Flow

11.2.1 Creating a Transaction Stream

The transaction stream is a handle to which all transactions to be viewed are added. This stream is created in the monitor extended from `uvmf_monitor_base`. The following code defined in that class creates the stream.

```
// FUNCTION: start_of_simulation_phase
virtual function void start_of_simulation_phase(uvm_phase phase);
  if (configuration.enable_transaction_viewing)
    transaction_viewing_stream = $create_transaction_stream({"..",get_full_name(),".","txn_stream"});
 endfunction
```

Figure 11.1: Transaction Stream Creation in `uvmf_monitor_base`

The stream is automatically created in the `start_of_simulation_phase` and is conditional on the `enable_transaction_viewing` flag in the agent configuration. The name of the stream is the full hierarchical path to the monitor with `.txn_stream` appended. This stream can then be found in both the Questa and Visualizer GUI.

11.2.2 Adding a Transaction to the Stream

Transactions to be viewed in the waveform viewer must be added to a transaction stream. The function that adds a transaction is located in the transaction class. The following code from the `run_phase()` of `uvmf_monitor_base` calls the `add_to_wave()` function of the transaction class. The `add_to_wave()` function adds the transaction information to the transaction stream.

```
if ( configuration.enable_transaction_viewing )
 trans.add_to_wave(transaction_viewing_stream);
```

Figure 11.2: Adding Transaction Data to a Stream

The `add_to_wave()` function definition of an example transaction class shown below.

```
56 //**********************************************************************
57   virtual function void add_to_wave(int transaction_viewing_stream_h);
58    if ( transaction_view_h == 0)
59      transaction_view_h = $begin_transaction(transaction_viewing_stream_h,op.name(),start_time);
60     if       ( op == WB_RESET ) $add_color( transaction_view_h, "red" );
61     else if ( op == WB_WRITE ) $add_color( transaction_view_h, "blue" );
62     else if ( op == WB_READ )  $add_color( transaction_view_h, "green" );
63     super.add_to_wave(transaction_view_h);
64     $add_attribute(transaction_view_h, op, "op");
65     $add_attribute(transaction_view_h, addr, "addr");
66     $add_attribute(transaction_view_h, data, "data");
67     $add_attribute(transaction_view_h, byte_select, "byte_select");
68     $end_transaction(transaction_view_h,end_time);
69     $free_transaction(transaction_view_h);
70   endfunction
```

Figure 11.3: Definition of the `add_to_wave()` Function

- **Lines 58-59:** The `transaction_view_h` is a handle to the transaction viewing object for this transaction within the transaction stream. Each transaction in the stream has a unique transaction viewing handle. If the handle is null then a new handle is generated using the `begin_transaction` system call. The `start_time` argument of the `begin_transaction` call determines the start time of the transaction in the waveform viewer.

- **Lines 60-62:** These lines set the color of the transaction within the waveform viewer based on the transaction operation type.

- **Line 63:** This line executes the `add_to_wave()` function in the base class. It adds variables in the base class to the stream entry.
- **Line 64-67:** The `add_attribute()` system function adds transaction variables to the waveform viewer. The second argument is the data variable to be added. The third argument identifies the variable value.
- **Line 68:** The `end_transaction()` system function call sets the end time of the transaction in the waveform viewer.
- **Line 69:** The `free_transaction()` system function call closes the transaction viewing handle on the stream and completes the process of adding the transaction.

11.3 Switches for Enabling Transaction Viewing

11.3.1 UVM Reporting Detail Setting

The UVM recording detail of the simulation can be set in either of the two mechanisms listed below:

- Add the following line to the UVM test case in any phase prior to and including the `run_phase`:
  `set_config_int("*","recording_detail",UVM_FULL);`
- Add the following line to the vsim command line:
  `+uvm_set_config_int=*,recording_detail,UVM_FULL`

  As described in Makefile section, extra vsim args can be passed upon invocation of Makefile via the `EXTRA_VSIM_ARGS` Variable
  `make cli EXTRA_VSIM_ARGS=+uvm_set_config_int=*,recording_detail,UVM_FULL`

Chapter 12

Top Level Modules

12.1 hdl_top

The module named `hdl_top`, located in the `tb/testbench` directory, contains the signal bundle interfaces, interface BFMs and DUT. It also includes the `uvm_config_db::set()` calls to pass virtual interface handles to UVM. The UVMF uses a two top architecture, `hdl_top` and `hvl_top`, to support emulation. The `hdl_top` module is synthesized into the emulator and `hvl_top` is run on the host simulator.

12.1.1 Instantiating Interfaces

A UVMF interface is divided into three pieces; the signal bundle, monitor BFM and driver BFM. Each instance of a protocol interface requires an instantiation of the signal bundle and monitor BFM. Each active instance of a protocol interface requires the instantiation of a driver BFM. The following code illustrates the instantiation two of these triplet SystemVerilog interfaces:

```
alu_out_if          alu_out_bus(.clk(),.rst(),.done(),.result());
alu_out_monitor_bfm alu_out_mon_bfm(alu_out_bus);
alu_out_driver_bfm  alu_out_drv_bfm(alu_out_bus);

alu_in_if           alu_in_bus(.clk(),.rst(),.valid(),.ready(),.op(),.a(),.b());
alu_in_monitor_bfm  alu_in_mon_bfm(alu_in_bus);
alu_in_driver_bfm   alu_in_drv_bfm(alu_in_bus);
```

Figure 12.1: Interface Instantiations in `hdl_top`

The first line of each group instantiates a signal bundle associated with a particular interface. The second and third lines instantiate a monitor and driver BFM, respectively. Each of these takes the signal bundle interface as its only port list item.

12.1.2 Instantiating the DUT

A Verilog DUT is instantiated using standard Verilog syntax. The port list itself can contain references to elements of the interface signal bundles as shown below, or to discrete wires that are subsequently attached to the signal bundles via `assign` statements.

```
59  alu   #(.OP_WIDTH(8), .RESULT_WIDTH(16)) DUT  (
60        .clk    (alu_in_bus.clk ) ,
61        .rst    (alu_in_bus.rst ) ,
62        .ready  (alu_in_bus.ready ) ,
63        .valid  (alu_in_bus.valid ) ,
64        .op     (alu_in_bus.op ) ,
65        .a      (alu_in_bus.a ) ,
66        .b      (alu_in_bus.b ) ,
67        .done   (alu_out_bus.done ) ,
68        .result (alu_out_bus.result ) );
```

Figure 12.2: Verilog DUT Instantiation in `hdl_top`

A VHDL DUT would be instantiated in the same way, but the compiled library name, VHDL entity and VHDL architecture name must be specified. The following line would replace line 59 above:

```
\workLib.entity(architecture) #(.OP_WIDTH(8),.RESULT_WIDTH(16)) DUT (
```

12.2 `hvl_top`

The module named `hvl_top`, located in `tb/testbench` directory, imports the test package and contains the call to `run_test()` which executes the UVM phases. The UVMF uses a two top architecture, `hdl_top` and `hvl_top`, to support emulation. The `hdl_top` module is synthesized into the emulator while `hvl_top` is run on the host simulator.

```
import uvm_pkg::*;
import alu_test_pkg::*;

module hvl_top;

  initial run_test();

endmodule
```

Figure 12.3: Contents of a UVMF `hvl_top`

Chapter 13

Creating Test Scenarios

13.1 Overview

A test scenario is a series of stimulus used to configure and stimulate the design. A test scenario can be created by writing a new test, a new sequence or both. If the desired stimulus does not exist in a sequence package then a new sequence must be created. The new sequence can be selected using either a new test or using the UVM command line processor. This section describes the creation of a new sequence, creation of a new test and how to select the sequence using either the new test or the UVM command line processor.

13.2 Creating a New Sequence

13.2.1 Creating a New Interface Sequence

If a bus operation needs to be created that is protocol specific and not design specific then a new interface sequence should be created. The new sequence should be extended from the sequence base class located in the interface package. The new sequence should be added to the interface package. The steps below describe how to create a new interface sequence and add the sequence to the package. It assumes the name of the interface package is `abc_pkg` and that the name of the new sequence is `new_sequence`.

1. In the `abc_pkg/src` folder create a file named `new_sequence.svh`
2. In `new_sequence.svh` add a class that extends `abc_sequence_base`. At a minimum this sequence should contain a factory registration macro and

constructor.

3. Add the desired behavior to this sequence.

4. Include the new sequence in `abc_pkg.sv` after the inclusion of `abc_sequence_base.svh`

13.2.2 Creating a New Environment Sequence

If a sequence needs to be created that is design specific and may be reused at block and chip level simulation then a new environment sequence should be created. The new sequence should be extended from the sequence base class located in the environment package. The new sequence should be added to the environment package. The steps below describe how to create a new environment sequence and add the sequence to the package. It assumes the name of the environment package is `abc_env_pkg` and that the name of the new sequence is `new_sequence`.

1. In the `abc_env_pkg/src` folder create a file named `new_sequence.svh`

2. In `new_sequence.svh` add a class that extends `abc_env_sequence_base`. At a minimum this sequence should contain a factory registration macro and constructor.

3. Add the desired behavior to the sequence

4. Include the new sequence in `abc_env_pkg.sv` after the inclusion of `abc_env_sequence_base.svh`

13.2.3 Creating a New Top Level Sequence

The top level sequence controls the flow and order of all lower level sequences. If a sequence needs to be created that is design specific and will not be reused at block and chip level simulation then a new top level sequence should be created. The new sequence should be extended from the sequence base class located in the sequence package of the bench. The new sequence should be added to the sequence package. The steps below describe how to create a new top level sequence and add the sequence to the package. It assumes the name of the bench sequence package is `abc2def_sequences_pkg` and that the name of the new sequence is `new_sequence`.

1. In the `tb/sequences/abc2def_sequences/src` folder create a file named `new_sequence.svh`

2. In new_sequence.svh add a class that extends abc2def_sequence_base. At a minimum this sequence should contain a factory registration macro and constructor.

3. Add the desired behavior to this sequence.

4. Include the new sequence in abc2def_sequences_pkg.sv after the inclusion of abc2def_sequence_base.svh

13.3 Creating a New Test

All UVMF generated test packages have a test named test_top. This test contains the top level configuration, top level environment and top level sequence. The test_top component constructs and connects the components. It also starts the sequence identified in the class definition of test_top. A new test case is created by extending test_top. The steps below describe how to create a new test class. It assumes the name of the test package is abc2def_test_pkg and the name of the new test is new_test. Each example in UVMF has an example derived test named example_derived_test.

1. In the tb/tests/src folder create a new file named new_test.svh

2. In new_test.svh add a class that extends test_top. At a minimum this test should contain a factory registration macro, constructor and build_phase function.

3. Add the desired factory overrides to this sequence in the build_phase. The factory overrides should be prior to super.build_phase(phase).

4. Include the new test in <benchName>_test_pkg.sv after the inclusion of test_top.svh

13.3.1 Modifying the Configuration

The configuration hierarchy can be modified in the test class prior to constructing the environment and starting stimulus. This allows for setting configuration variables to values required for specific test scenarios. The following steps within the build_phase allow for control of configuration values in the test class:

1. super.build_phase(): Constructs and randomizes configuration object

2. configuration.initiallize(): Initialize the configuration structure with bench level information

3. Modify variables in the fully constructed configuration structure as required by the test scenario. This includes but is not limited to randomizing the configuration with additional constraints, setting specific configuration variables to required values, etc.

4. Exit the test class `build_phase()` function. The environment class `build_phase()` function will be executed next. This allows for environment construction to be influenced by variable settings in the configuration object.

13.4 Selecting a New Test Scenario Using the UVM Factory

13.4.1 Using a New Test Class

The `TEST_NAME` makefile variable is used to select the test. This variable is used to set the `UVM_TESTNAME` command line variable. The default value for the `TEST_NAME` variable is `test_top`. To select another test add `TEST_NAME=new_test` to the make command.

13.4.2 Using the UVM Command Line Processor

The UVM command line processor can be used to override any class or object within the simulation. This includes overriding sequences. To select a new test scenario by performing an override using the UVM command line processor add the following to the vsim command line:

```
+uvm_set_type_override=requested_type_class_name,override_type_class_name
```

Appendix A

Adding Non-UVMF Components to an Existing UVMF Bench and Environment

A.1 Adding a Non-UVMF Based Agent

The recommended method of adding a non-UVMF based agent to a UVMF environment is to wrap the agent within a UVMF generated environment. The environment only contains the non-UVMF based agent and the glue code to configure and initialize the agent when the initialize function of the environment configuration is called. The generated UVMF makefile will need to be updated to compile the encapsulated agent, its related classes and modules. Encapsulating the non-UVMF agent within a UVMF environment enables the use of the non-UVMF agent with the UVMF environemnt generator.

A.2 Adding a Non-UVMF Based Environment

The recommended method of adding a non-UVMF based environment to a UVMF environment is to wrap the non-UVMF environment within a UVMF generated environment. The envioronment only contains the non-UVMF based environment and the glue code to configure and initialize the environment when the initialize function of the environment is called. The generated UVMF Makefile will need to be updated to compile the encapsulated environment and its related classes. Encapsulating the non-UVMF environment within a UVMF environment enables the use of the non-UVMF environment with the UVMF environment generator.

A.3 Adding a QVIP Agent

The recommended method of adding QVIP agents to a UVMF environment it to use the QVIP Configurator. The QVIP Configurator is a graphical tool for selecting and configuring standard protocols. The QVIP Configurator generates a UVMF based environment that contains all selected agents and their configuration objects. The QVIP Configurator also generates a module that contains all of the BFM's for the selected protocols. The QVIP Configurator generates a YAML file that describes the content of the generated code. This YAML file is used by the UVMF generator to intantiate the environment and module generated by the QVIP Configurator within a parent UVMF based environment that can also contain prediction, scoreboarding, coverage, and custom interfaces.

Appendix B

Making a Non-UVMF Interface VIP Compatible with UVMF

Non-UVMF VIP can be made UVMF compatible with some modifications. The requirements listed below are necessary in order to be compatible with UVMFâĂŹs UVM reuse methodology. They are also required for seamless use with UVMF environment and test bench generators. The requirements below are written assuming a non UVMF interface VIP protocol named `xyz`.

B.1 Interface Package

All classes, typedefs, and class specializations used in the interface VIP must be contained within a package with a `_pkg` suffix. For example, `xyz_pkg`. This package must contain classes such as driver, monitor, coverage, agent, transaction, sequences, etc.

This package must contain the following imports:

```
import uvmf_base_pkg_hdl::*;
import uvmf_base_pkg::*;
```

B.2 Transaction Class

The transaction class must have a `_transaction` suffix. For example, `xyz_transaction`. This can be done using a typedef. For example, `typedef my_xyz_transaction_name xyz_transaction`. The transaction class must be extended from `uvmf_transaction_base`. A class that extends `uvm_sequence_item` can be changed to extend from uvmf_transaction_base because uvmf_transaction_base extends uvm_sequence_item.

B.3 Configuration Class

The configuration class must have a _configuration suffix. For example, xyz_-configuration. This can be done using a typedef. For example, typedef my_-xyz_configuration_name xyz_configuration. The configuration class must have a convert2string() implementation. This is because a UVMF environment configuration's convert2string() will call texttt convert2string of all agent configurations within the enviornment configuration.

The configuration class must have an initialization function with the following prototype:

```
virtual function void initialize(
  uvmf_active_passive_t activity,
  string agent_path,
  string interface_name);
```

The activity argument will have a value of ACTIVE or PASSIVE based on the agent activity. If the agent is actively driving stimulus into the DUT then the agent is ACTIVE. If the agent is only passively monitoring bus traffic then the agent is PASSIVE. The agent_path argument is the full path to this configurations agent. Since the UVMF architecture dictates that the configuration classes are not within the environment hierarchy the agent_path must be used to pass the configuration handle to its agent. The interface_name argument is a unique string identifying the resources associated with this agent. The resources include its virtual interface handles(s), configuration class handle, and sequencer class handle.

B.4 Agent Class

The agent class must have a _agent_t suffix. For example, xyz_agent_t. This can be done using a typedef. For example, typedef my_xyz_agent_name xyz_-agent_t; The agent class must have an analysis port named monitored_ap which broadcasts transactions of type xyz_transaction as described in the "Transaction Class" section.

If the agent is ACTIVE then it must place its sequencer handle in the UVM config db using the interface_name argument of the configurations initialize call as the field_name argument of the uvm_config_db::set() call with the following scope: null, UVMF_SEQUENCERS.

B.5 Interface

The interfaces should include a driver BFM, monitor BFM and signal bundle. The names of these should have `_driver_bfm`, `_monitor_bfm`, and `_if` suffixes respectively. If this recommendation is not followed then the interface instantiations within the generated `hdl_top.sv` must be modified accordingly.

B.6 Makefile

A makefile named `Makefile` should be created for compiling the interface and package for the VIP. The makefile should contain a make target named `comp_xyz_pkg` where `xyz` is the protocol name. If this recommendation is not followed then the test bench makefile must be modified accordingly. The makefile from an interface package generated using the UVMF code generator can be used as a template for creating the interface makefile.

B.7 Directory Structure

The interface VIP should have the following directory structure given the protocol name of `xyz`:

- `xyz_pkg` which contains the `xyz_pkg` declaration and makefile
- `xyz_pkg/src` which contains all other source files

The `xyz_pkg` should be located in `interface_packages` directory.

If these recommendations are not followed then the test bench makefile must be modified accordingly.

Appendix C

UVM classes used within UVMF

C.1 Overview

In order to minimize risk and minimize the UVM learning curve, a minimum subset of classes from the `uvm_pkg` have been used. Most of the classes used in UVMF were contained in the predecessor of the UVM, `ovm_pkg`.

C.2 UVM Component Classes Used

- `uvm_test`
- `uvm_env`
- `uvm_agent`
- `uvm_sequencer`
- `uvm_driver`
- `uvm_monitor`
- `uvm_subscriber`
- `uvm_scoreboard`
- `uvm_analysis_port`
- `uvm_port_component_base`
- `uvm_port_list`

- `uvm_report_server`
- `uvm_reg_predictor`

C.3 UVM Data Classes Used

- `uvm_object`
- `uvm_sequence_item`
- `uvm_sequence`
- `uvm_tlm_analysis_fifo`

C.4 UVM Phases Used

- `build_phase`
- `connect_phase`
- `end_of_elaboration_phase`
- `start_of_simulation_phase`
- `run_phase`
- `extract_phase`
- `check_phase`
- `report_phase`

C.5 UVM Macros Used

- `uvm_component_utils`
- `uvm_component_param_utils`
- `uvm_object_utils`
- `uvm_object_param_utils`
- `uvm_info`

- `uvm_warning`
- `uvm_error`
- `uvm_fatal`
- `uvm_analysis_imp_decl`

C.6 Miscellaneous UVM Features Used

- `uvm_config_db`
- UVM factory
- `phase_ready_to_end`
- `raise_objection`
- `drop_objection`

Appendix D

UVMF Base Package Block Diagrams

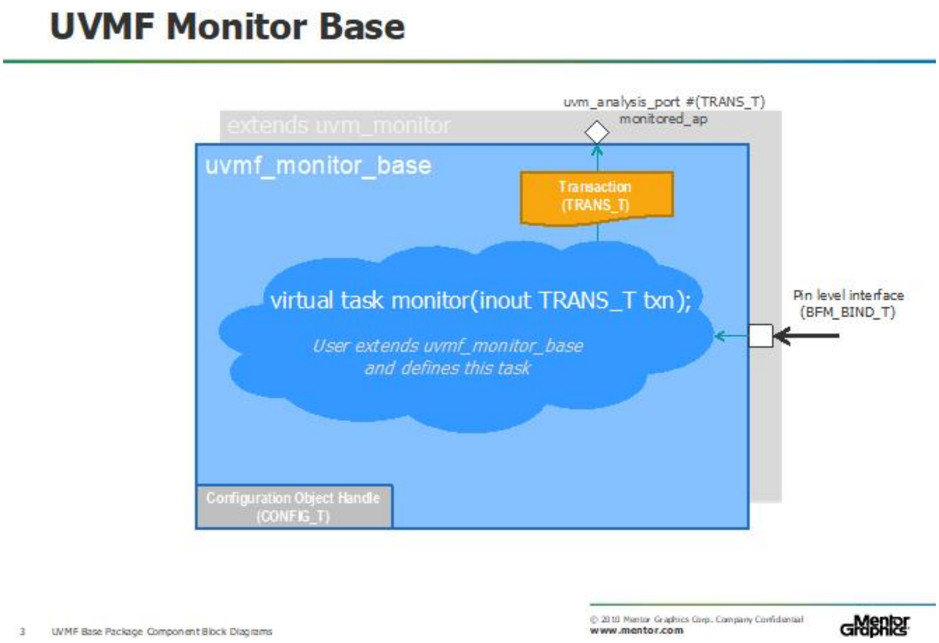

Figure D.1: `uvmf_monitor_base`

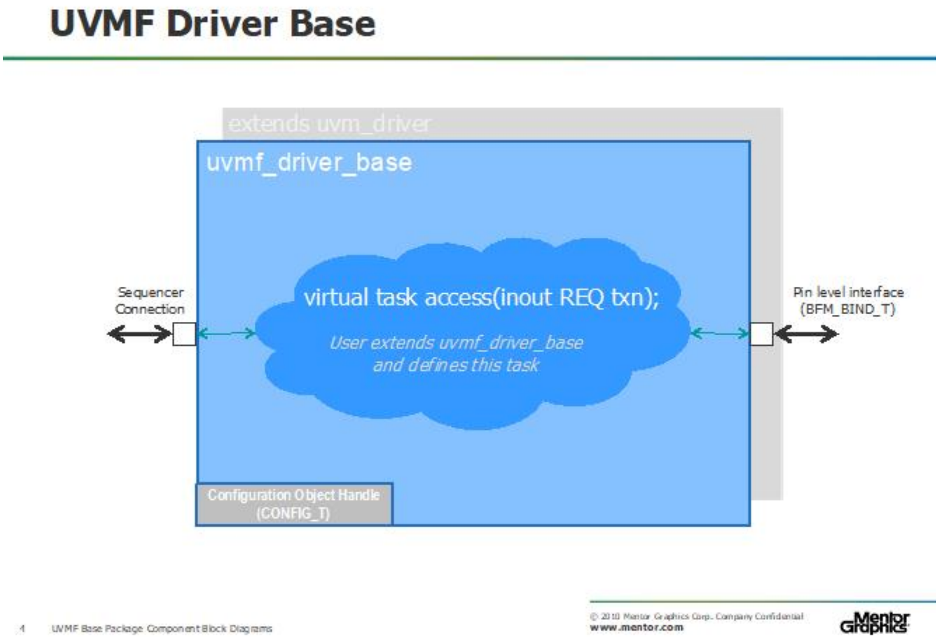

Figure D.2: `uvmf_driver_base`

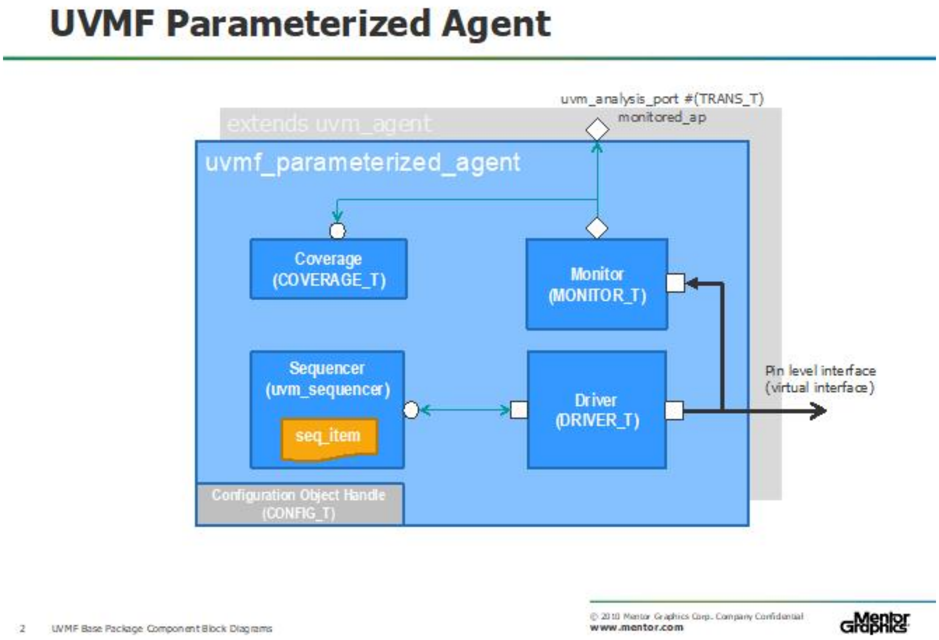

Figure D.3: `uvmf_parameterized_agent_base`

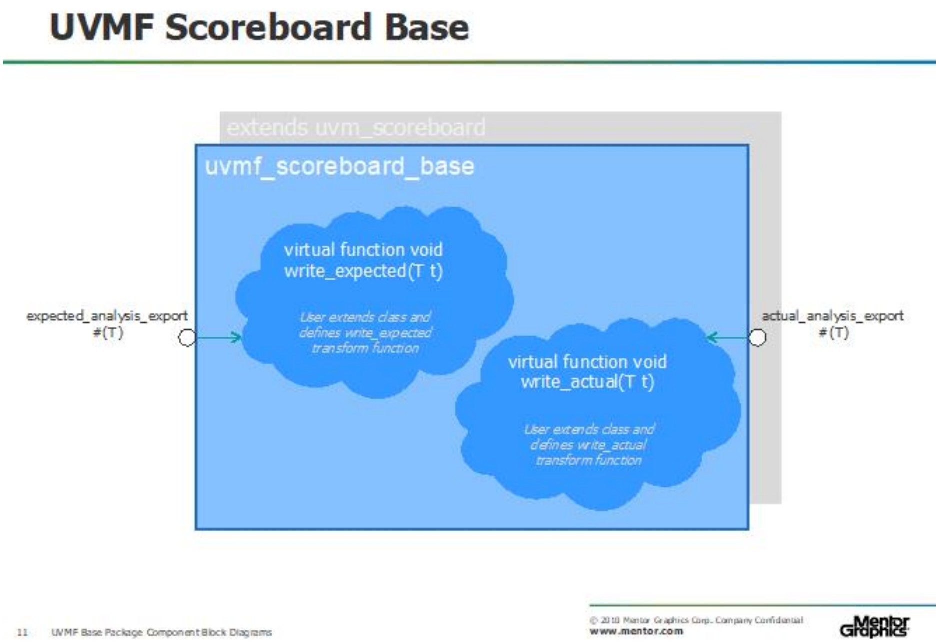

Figure D.4: `uvmf_scoreboard_base`

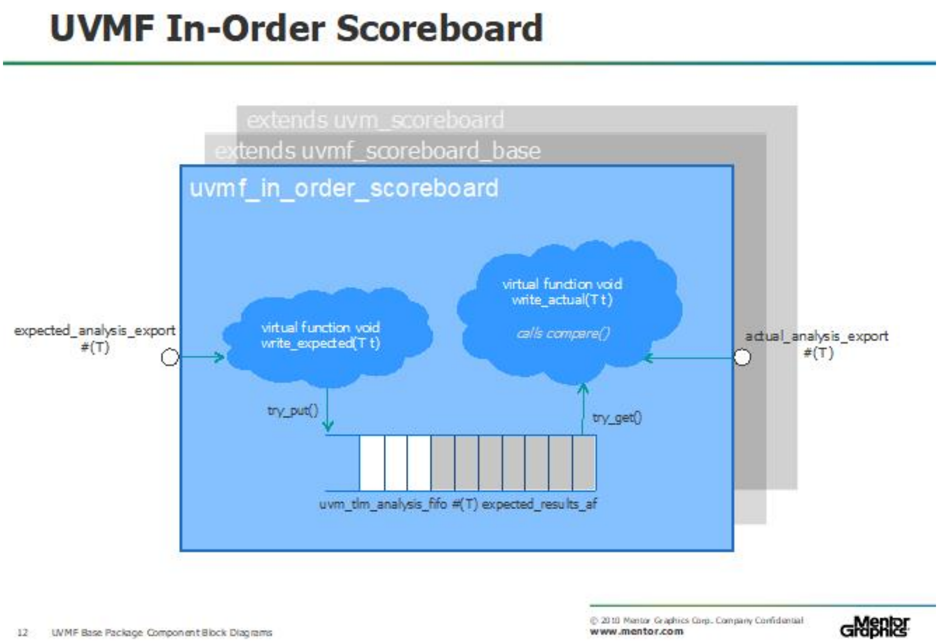

Figure D.5: `uvmf_in_order_scoreboard`

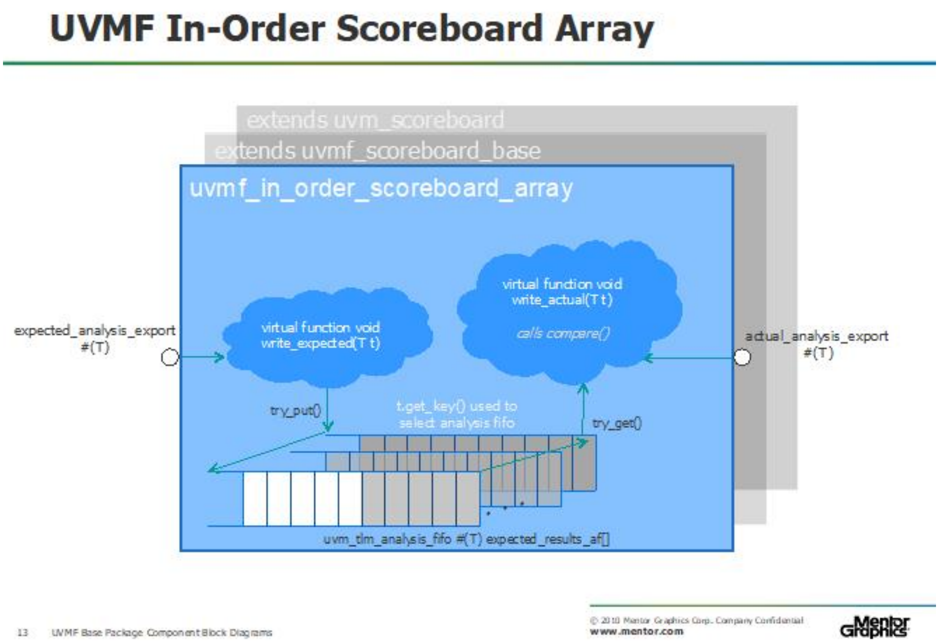

Figure D.6: `uvmf_in_order_array_scoreboard`

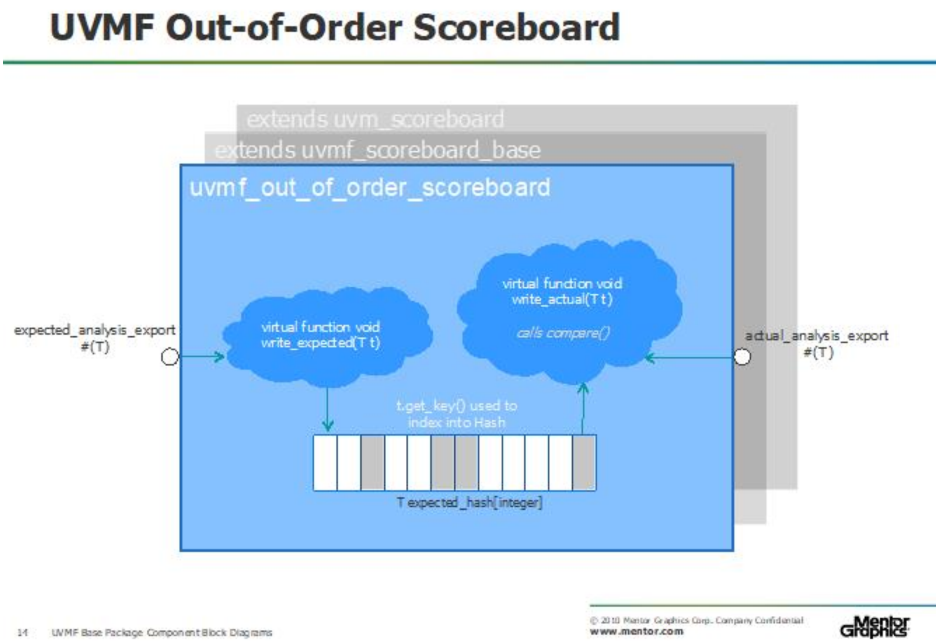

Figure D.7: `uvmf_out_of_order_scoreboard`

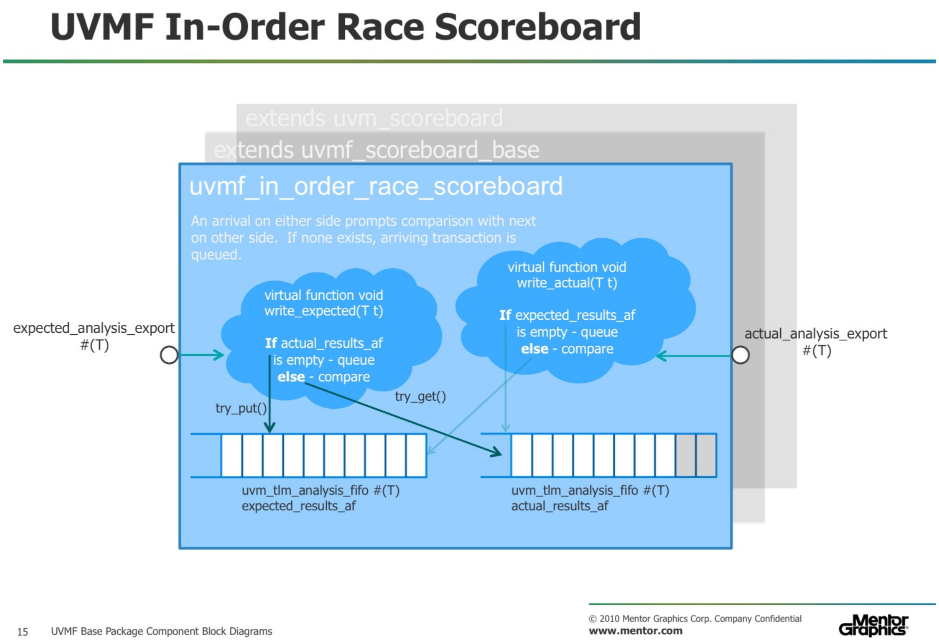

Figure D.8: `uvmf_in_order_race_scoreboard`