UVMF YAML Reference Manual

Version 2022.1

Contents

Chapter 1

Introduction to the UVM Framework (UVMF) Code Generators

1.1 Overview

The UVM Framework provides code generators for creating interfaces, environments, and test benches. A Python script, `yaml2uvmf.py` can be used to translate desired UVMF structure described as YAML-based files into the UVMF code.

Specific YAML data structures must be provided to the script in order to properly generate the desired interfaces, environments or benches. This document describes how to execute this script and also details the required control structures. There are also a number of examples available in the UVMF installation that illustrate the format. The following table describes these examples, all of which can be found under `$UVMF_HOME/templates/python/examples/yaml_files`:

Interface Examples	Description
mem_if_cfg.yaml	User input file for generating an interface package named mem_pkg. This interface is used in block_a, block_b, and chip environments and test benches.
pkt_if_cfg.yaml	User input file for generating an interface package named pkt_pkg. This interface is used in block_a, block_b, block_c, and chip environments and test benches
dma_if_cfg.yaml	User input file for generating an interface named dma_pkg. This interface is a responder interface and defines response data.
Environment Examples	**Description**
block_a_env_cfg.yaml	User input file for generating an environment that has no parametization. This environment is also used in chip_env.
block_b_env_cfg.yaml	User input file for generating an environment that has parametization. This environment is also used in chip_env.
block_c_env_cfg.yaml	User input file for generating an environment that has a QVIP configurator generated sub environment that contains standard protocols.
chip_env_cfg.yaml	User input file for generating a chip level environment that instantiates sub environments.
Test Bench Examples	**Description**
block_a_bench_cfg.py	User input file for generating a test bench to run the block_a environment.
block_b_bench_cfg.py	User input file for generating a test bench to run the block_b environment.
chip_bench_cfg.py	User input file for generating a test bench to run the chip environment.
block_c_bench_cfg.py	User input file for generating a testbench that runs the chip environment.

1.2 Generation Flow

The diagram below shows the flow utilized by the UVMF generators. The user creates one or more text files that use the provided YAML format for characterizing the interface, environment, or test bench. These files are passed into the generator script yaml2uvmf.py. Files generated can include all classes, packages, BFMs, and makefiles required for an operational test bench that simulates as generated.

In order to generate a particular level of UVMF hierarchy all YAML structures used underneath that hierarchy must be provided. For example, if an environment YAML structure is provided, the YAML describing any instantiated interfaces must also be provided.

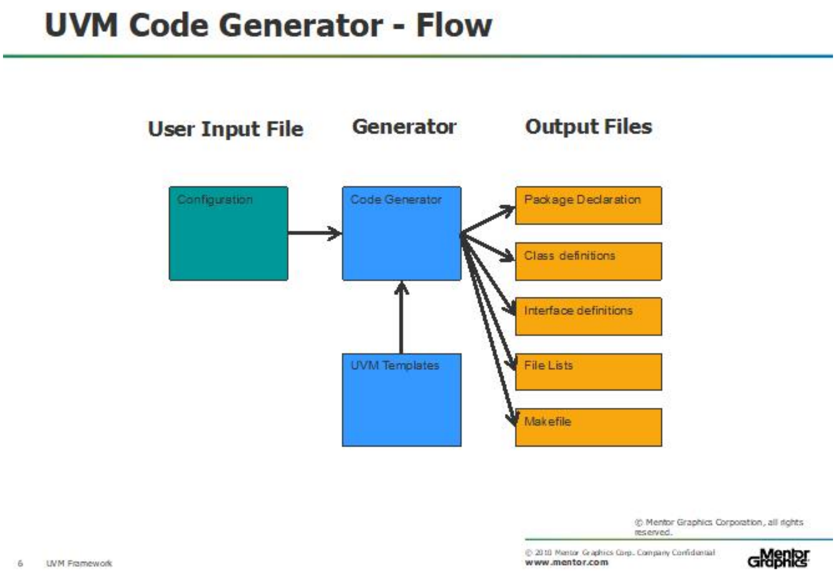

Figure 1.1: Code Generation Flow

1.3 YAML Overview

YAML is a human friendly data serialization standard that is supported by a wide array of programming languages. The name itself is a recursive acronym common in Linux development that stands for "YAML Ain't Markup Language". Its use can be considered similar to that of XML but the format is far simpler, both to read as well as to write.

For complete documentation on the YAML format, sites like www.yaml.org can be used as a starting point. For the purposes of this application, YAML is used to translate nested data structures that describe UVMF hierarchy and properties in a manner easily parsed by both users and scripts.

All UVMF YAML must be presented as part of a specific top-level format, shown here:

```
uvmf:
  interfaces:
    "<interface_nameA>"
      <properties>
    "<interface_nameB>"
  util_components:
    "<util_componentA>"
      <properties>
  environments:
    "<env_nameA>"
      <properties>
    "<env_nameB>"
      <properties>
  benches:
    "<bench_nameA>"
      <properties>
    "<bench_nameB>"
      <properties>
  global:
    <properties>
```

The allowed contents within each named subsection are described in subsequent chapters. The information can be spread across multiple files or in a single file.

Chapter 2

Running the Generator

2.1 Initial Generation

When beginning from scratch it is recommended to make the following steps when developing an initial UVMF bench with the generator script:

- Develop initial structure of desired interfaces, environments and bench in a block diagram form
- Translate diagrams into YAML structures
- Execute `yaml2uvmf.py` against YAML
- Run `make cli` against generated output before making any hand edits. Any compile or run time errors encountered at this stage should be addressed by making adjustments to the YAML configuration files and re-generating
- Proceed with hand edits, focusing on areas of code marked with `UVMF_-CHANGE_ME` comments that should reside within special "pragma" comments in the following form:
  `// pragma uvmf custom <label_name> [begin|end]`
  These labeled blocks will facilitate the ability to merge hand edits into subsequent re-runs of the `yaml2uvmf.py` script.

2.2 Merging

After some hand-edits have been made to generated output, it may be necessary to make adjustments or additions to the original YAML configuration files. The generation flow supports the ability to integrate those YAML updates with hand edited source that was generated using a previous configuration. This is facilitated through some special switches to the script that all begin with `--merge_*`. In the most basic form, use the `--merge_source` switch to point the script to a set of hand-edited source that you wish to update. When merging old and new output, the following rules are followed:

- Any manual edits that were made within UVMF pragma blocks will be transferred to the appropriate location within the updated generated output.
- Any files that were created from scratch in the old source will be left alone.
- Any files that were created by the new generation that do not match up with old source will be copied into the old source directory structure.
- Any manual edits that were made outside of UVMF pragma blocks will be lost. Furthermore, this cannot be detected, so no warnings will be issued.
- Any UVMF pragma blocks that were added by the user within old source will not be manually merged but these will be noted by the script.

In most cases the updated hand-edited source will still run but there will be situations where the modified YAML will insert changes to the source that is incompatible with the hand edits. For example, if updated YAML deletes one of the transaction variables within an interface definition, any references to the removed variable elsewhere in the generated code (driver, monitor, prediction, etc.) will prevent successful compilation until said hand edits are updated to also remove references to the variable. In addition to updating the hand-edited source it is possible to have the script produce an intermediate non-edited output based purely on the updated YAML by using the `--merge_debug` switch. By default, this will produce the usual `uvmf_-template_output` directory that, if there are no errors in the YAML, should work out-of-the-box.

Additional data regarding exactly what was impacted during a merge operation can be viewed with the `--merge_verbose` switch. When enabled, the following data will be printed out at the end of the run:

- List of all named blocks and associated files that were found in the original source
- List of all files that were not matched between original source and newly

generated source, and were copied into the original source tree

- List of all named blocks and associated files in new generated source that were not mapped to original source, so now contain their default contents in the original source

The script does not support the ability to transfer new/custom named blocks that were created by the user. Only blocks that exist in both new and old code will be merged. By default, any new/custom blocks will be detected and flagged as an error during the merge process. This behavior can be overridden with the `--merge_skip_missing_blocks` switch. When in place, the script will instead produce a list of new/custom blocks at the end of the merge that the user can employ as a list of items that will need to be transferred by hand.

2.3 Command Syntax

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

2.4 Switch Details

Switch Name	Description
--version	Show script version number and exit
-h, --help	Print help message and exit
-c, --clean	Clean up generated code instead of producing code
-q, --quiet	Suppress output while running
-d <dest_dir>, --dest_dir=<dest_dir>	Specify a destination directory for output. Default is $CWD/uvmf_template_output
-o, --overwrite	Overwrite existing output files. Default behavior is to skip files if they already exist in the destination directory
-f <file_list> --file=<file_list>	Specify a list of YAML configuration files
-g <name> --generate=<name>	Only produce the specified component. By default, all components defined by the input configuration files will be generated
-t <template_dir> --template_dir= <template_dir>	Override the location where templates for generated files are sourced. The default location is relative to the location of the uvmf_gen.py module
-m <merge_source> --merge_source= <merge_source>	Enable auto-merge flow, pulling in hand-edited source from the specified directory. Updates from YAML will be overlaid on top of the specified directory. Backup of the original source will be made by default
-s, --merge_skip_missing_blocks	Continue the merge even if a labeled block was found in old source that can't be mapped to anything in the new output. A report of all such files and labels will be produced at the end of the run. By default, missing blocks will raise an error
--merge_no_backup	Do not produce a backup of the original merge source
--merge_debug	Provide an intermediate unmerged output directory for debug purposes. The location of this directory can be specified with the --dest_dir switch
--merge_verbose	Output details on names of files and named blocks that were affected during the merging process

Chapter 3

Interface YAML Structure

3.1 Description

The interface YAML data structure contains information about an interface's name, associated transaction data, interface ports and configuration. This information is used to create the following content:

- **Classes**: Transaction, interface level sequence base, random sequence, coverage, driver, monitor, agent, agent configuration, UVM reg adapter, UVM reg predictor.
- **Package**: Protocol package including all classes listed above.
- **BFMs**: Driver and monitor.
- **Compilation flow**: File list and Makefile

3.2 YAML Format

Most of the content in an interface YAML file is optional but most of the available properties should be filled out in order to define a useful starting point for a UVMF interface. All properties are assigned a name and an expected data type. Top-level properties are listed in the section below along with references to BNF information for underlying structure. The order of underlying lists will be maintained in the generated output. All properties are optional unless noted otherwise.

3.2.1 Top-level Interface Properties

Name	Type	Description
`clock` (required)	string	Name of primary clock. Additional clocks must be added manually
`reset` (required)	string	Name of primary reset. Additional clocks must be added manually. If interface has no reset use 'dummy' and remove associated code from interface
`reset_assertion_level`	True|False	Assertion level for this protocol. If True the protocol will use an active high reset
`vip_lib_env_variable`	string	Name of environment variable that will point to the location of the source for this environment package
`veloce_ready`	True|False	Defaults to True. When True, generated code is a Veloce ready/friendly interface
`mtlb_ready`	True|False	Defaults to False. The generated interface will generate files allowing it to work with Matlab
`config_constraints`	List of constraint_schema	List defining the constraints to be applied against the constraint variables
`dpi_define`	dpi_define_schema	Structure defining DPI source associated with this interface

3.2.2 Interface Schema Definitions

3.2.2.1 import_schema

Description	Defines a single package import
Structure	{ name: "<name>" }
Example	imports : - { name: "my_pkg"} - { name: "my_other_pkg"}

3.2.2.2 parameter_def_schema

Description	Defines a single parameter. All arguments except value are required. If value is not specified the parameter will not have a default value defined
Structure	name: "<name>" type: "<type>" [value: "<value>"]
Example	parameters : - name: "ADDR_WIDTH" type: "int" value: "16"

3.2.2.3 typedef_schema

Description	Defines a typedef. All arguments are required
Structure	name: "<name>" type: "<type>"
Example	hdl_typedefs : - name: "addr_t" type: "bit [15:0]"

3.2.2.4 port_schema

Description	Defines a single port definition for use in an interface wire bundle. All arguments except for `reset_value` are required. The `width` value can be a scalar string or a list of strings. If a scalar string, that value will be used to derive the width of the port. If a list is provided, a multi-dimensional packed array will be defined using the list of widths as the width for each element of the array. Width values can be simple integers or parameter references.

Structure	`name: "<name>"`
	`width: "<width>" | [ "<width_dim0>",...]`
	`dir: "<dir>"`
	`[ reset_value: "<value>" ]`

Example	`ports :`
	`  - name: "rdata"`
	`    width: "RDATA_WIDTH"`
	`    dir: "input"`
	`    reset_value: "32'b0123_4567"`
	`  - name: "wdata"`
	`    width:"32"`
	`    dir: "output"`
	`  - name: "addr_mda"`
	`    width: ["15","3","ADDR_WIDTH"]`
	`    dir: "output"`

3.2.2.5 transaction_schema

<table>
<tr><td>Description</td><td>Defines a single transaction to be placed within an interface's sequence item definition. All arguments are required unless surrounded with square brackets. Default for <code>isrand</code> is "False" and the default for <code>iscompare</code> is "True".
If <code>isrand</code> is "True" the given transaction variable will be marked with the SystemVerilog "rand" keyword, allowing it to be modified when the transaction object's "randomize()" function is called.
If <code>iscompare</code> is "True" the given transaction variable will be taken into consideration when two transactions are compared. "Meta" data such as latency, arrival time, etc. should usually mark this value as "False" whereas more concrete data variables should be compared.
If <code>unpacked_dimension</code> is specified, this will prompt the variable to use the given value as the unpacked dimension string to the right of the variable name in the declaration.
The <code>comment</code> field is optional and if specified, will be placed on the line above the variable declaration.</td></tr>
<tr><td>Structure</td><td><pre>name: "<name>"
type: "<type>"
[isrand: "True"|"False"]
[iscompare: "True"|"False"]
[unpacked_dimension: "<dim>"]
[comment: "<text>"]</pre></td></tr>
<tr><td>Example</td><td><pre> transaction_vars :
  - name: "data"
    type: "bit [15:0]"
    isrand: "True"
    iscompare: "True"
    unpacked_dimension: "[1000]"
  - name: "latency"
    type: "int"
    isrand: "True"
    iscompare: "False"
    comment: "Data field for all operations."</pre></td></tr>
</table>

3.2.2.6 config_var_schema

<table>
<tr><td>Description</td><td>Defines a configuration variable to use in the given interface. All arguments are required unless denoted with square brackets. Default for "isrand" is "False".
If "isrand" is "True" the given configuration variable will be marked with the SystemVerilog "rand" keyword, allowing it to be modified when the object's "randomize()" function is called.
If "value" is provided, this will initialize the variable with the specified value at the beginning of simulation.
The "comment" field is optional and if specified, will be placed on the line above the variable declaration.</td></tr>
<tr><td>Structure</td><td>name: "<name>"
type: "<type>"
[isrand: "True"|"False"]
[value: "<value>"]
[comment: "<text>"]</td></tr>
<tr><td>Example</td><td>config_vars :
  - name: "block_a_cfgVar"
    type: "bit [3:0]"
    isrand: "True"
    value: "4'b1010"
    comment: "Example configuration variable"</td></tr>
</table>

3.2.2.7 constraint_schema

<table>
<tr><td>Description</td><td>Defines a constraint to be applied to the transaction variables for the given interface. The 'name' and 'value' arguments are required. The 'comment' field is optional and if specified, will be placed on the line above the constraint declaration.</td></tr>
<tr><td>Structure</td><td>name: "<name>"
value: "<value>"
[comment: "<text>"]</td></tr>
<tr><td>Example</td><td>transaction_constraints :
  - name: "address_word_align_c"
    value: "{ address[1:0]==2'b00; }"
    comment: "All addresses alligned."</td></tr>
</table>

3.2.2.8 dpi_schema

<table>
<tr><td>Description</td><td>Specifies that a set of DPI-C source should be created and compiled to be associated with this interface. User is expected to provide a shared object name, a list of desired C source files to be produced and a list of DPI import and export definitions. NOTE: DPI exports are currently unsupported.</td></tr>
<tr><td>Structure</td><td>name: "<shared_object_name>"
files: [<array_of_c_source_files>]
comp_args: "c_compile_arguments"
link_args: "c_link_arguments"
exports: [<array_of_export_function_names>]
imports: [<array_of_dpi_import_schema>]</td></tr>
<tr><td>Example</td><td>dpi_define :
  name: "pktPkgCFunctions"
  files:
    - "myFirstIFFile.c"
    - "mySecondIFFile.c"
  comp_args: "-c -DPRINT32 -O2 -fPIC"
  link_args: "-shared"
  imports:
    - name: "hello_world_from_interface"
      return_type: "void"
      c_args: "(int var1, int var2)"
      sv_args:
        - name: "var1"
          type: "int"
          dir: "input"
        - name: "var2"
          type: "int"
          dir: "input"
    - name: "good_bye_world_from_interface"
      return_type: "void"
      c_args: "(int var3, int var4)"
      sv_args:
        - name: "var3"
          type: "int"
          dir: "input"
        - name: "var4"
          type: "int"
          dir: "input"</td></tr>
</table>

3.2.2.9 dpi_import_schema

Description	Defines a DPI import function

Structure	name: "<import_function_name>"
	return_type: "<return_type_of_function>"
	c_args: "<args_string_used_in_c_function>
	sv_args:
	  name: "<name_of_sv_argument>"
	    type: "<type_of_sv_argument>"
	    dir: "input|output|inout"
	    [unpacked_dimension: "<dim>"]

Example	See dpi_schema example

Chapter 4

Utility Component YAML Structure

4.1 Description

Utility components are items within a UVMF environment that do not fall into the category of a sub-environment or interface. These types of components are defined within the `util_components` header of the overall YAML data structure. Valid types are `predictor`, `coverage` and `scoreboard`.

Utility components defined with the `predictor` type contain the base content for a predictor including construction of a transaction for broadcasting through an analysis port. The user must add the prediction algorithm to the generated write functions associated with the analysis exports. As generated, when a transaction is received through any of the predictor's analysis exports, the predictor broadcasts a transaction out of each of the predictor's analysis ports. This is to validate connections between the predictor and other components as defined using the `tlm_connections` construct. This may cause some scoreboards within some generated environments to issue an error at the end of the simulation due to transactions remaining in the scoreboard. This is common with predictors with multiple analysis exports which result in multiple transaction broadcasts to scoreboards, etc.

Utility components defined with the `coverage` type contain the base content for a coverage component including a covergroup and handle to the environment configuration object. The user must add coverpoints, bins, crosses, exclusions, etc as needed to implement the required coverage model.

Utility components defined with the `scoreboard` type contain the base content for a custom scoreboard component. This includes instantiations for desired analysis ports and exports as well as definitions for write functions for each defined analysis export. How incoming transactions are stored and compared is left up to the user.

4.2 YAML Format

Top-level properties for all utility components are listed in the table below.

4.2.1 Top-Level Properties

Name	Type	Description
`type`	`predictor` | `coverage` | `scoreboard`	Indicates what type of utility component definition this is
`analysis_exports`	List of analysis_schema	Specifies the name and type of the various analysis export components to be instantiated within the component
`analysis_ports`	List of analysis_schema	Specifies the name and type of the various analysis port components to be instantiated within the component
`qvip_analysis_exports`	List of analysis_schema	Specifies the name and type of the various analysis export components to be instantiated within the component. Unlike analysis_exports, which will instantiate an analysis "imp" of the specified sequence item type, this will trigger the creation of an "imp" of type "mvc_sequence_item_base". Incoming items will then be dynamically cast into the specified type of item as part of that port's "write" function
`parameters`	List of parameter_def_schema	List of parameter definitions for the given utility component

4.2.2 Schema Definitions

The following structures (schemas) can be used to populate information underneath the top-level properties listed in the table above.

4.2.2.1 analysis_schema

Description	Defines an analysis port/export to be instantiated within the given component. The `type` field indicates the type of sequence item that the port or export will be handling
Structure	`name: "<name>"` `type: "<type>"`

Example	`analysis_ports: :`
	`  - name: "mem_ap"`
	`    type: "mem_item"`
	`  - name: "pkt_ap"`
	`    type: "pkt_item"`

4.2.2.2 parameter_def_schema

Description	Defines a single parameter. All arguments except "value" are required. If "value" is not specified the parameter will not have a default value defined.

Structure	`name: "<name>"`
	`type: "<type>"`
	`[ value: "<value>" ]`

Example	`parameters: :`
	`  - name: "ADDR_WIDTH"`
	`    type: "int"`
	`    value: "16"`

Chapter 5

Environment YAML Structure

5.1 Description

The environment YAML data structure contains information about an environment's name, instantiated components and sub-environments, TLM connectivity and configuration. This information is used to create the following content:

- **Classes**: Environment, environment configuration, predictors, coverage collection components, environment level sequence base.
- **Package**: Environment package
- **Compilation flow**: File list and Makefile

5.2 YAML Format

Most of the content in an environment YAML file is optional but most of the available properties should be filled out in order to define a useful starting point. All properties are assigned a name and an expected data type. Top-level properties are listed in the section below along with references to BNF information for underlying structure. The order of underlying lists will be maintained in the generated output. All properties are optional unless noted otherwise.

5.2.1 Top-Level Properties

Name	Type	Description
`agents`	List of component_schema	Ordered list of underlying UVMF agents (interfaces) to instantiate within this environment. The YAML definition for agents must be provided as part of the script run. It is important to note that the built-in analysis_port on a UVMF agent is named `monitored_ap`.
`non_uvmf_components`	List of component_schema	Ordered list of components not defined using the generator script.
`qvip_memory_agents`	List of component_schema	Ordered list of agents associated with QVIP DDR memory models.
`parameters`	List of parameter_def_schema	List of parameter definitions for creating type parameters for classes.
`hvl_pkg_parameters`	List of parameter_def_schema	List of parameter definitions to be included in the interfaces_pkg package declaration
`imports`	List of import_schema	Specify packages to import for this environment's package definition
`analysis_components`	List of component_schema	Ordered list of underlying analysis components (i.e. predictors or coverage components) to instantiate within this environment. The YAML definition for each component must be provided as part of the run.

`scoreboards`	List of scoreboard_schema	List of built-in UVMF scoreboard components to instantiate within this environment. It is important to note that the built-in analysis_exports on a UVMF scoreboard are named `expected_analysis_export` and `actual_analysis_export`.
`subenvs`	List of component_schema	List of sub-environments to instantiate within this environment. YAML definitions for each sub-environment must be provided as part of the run.
`qvip_subenvs`	List of qvip_subenv_schema	List of QVIP Configurator-generated sub-environments to instantiate within this environment. YAML definitions for these environments must be provided.
`analysis_ports`	List of tlm_port_schema	List of UVM analysis port components and their connection information to be used in this environment.
`analysis_exports`	List of tlm_port_schema	List of UVM analysis export components and their connection information to be used in this environment.
`tlm_connections`	List of tlm_schema	Specify how all of the components within this environment should be connected.
`qvip_connections`	List of qvip_tlm_schema	Specify how the QVIP components within this environment should be connected.
`config_vars`	List of config_var_schema	Defines configuration variables to use in controlling this environment
`config_variable_values`	List of config_value_schema	Defines values for underlying config variable default values

`config_constraints`	List of constraint_schema	Defines constraints associated with the configuration variables for this environment
`imp_decls`	List of imp_decl_schema	Defines the names of imp_decl macros to be defined for this environment
`register_model`	register_model_schema	Specifies characteristics of the desired register model to instantiate and connect in this environment
`dpi_define`	dpi_define_schema	Structure defining DPI source associated with this environment. See interface dpi_define_schema for more details. Structure here is identical.
`typedefs`	typedef_schema	Specifies typedefs to be defined in this environment.
`mtlb_ready`	True|False	Defaults to False. The generated bench will generate files allowing it to work with Matlab

5.2.2 Schema Definitions

The following structures (schemas) can be used to populate information underneath the top-level properties listed in the table above.

5.2.2.1 component_schema

Description	Defines a component instantiation. Optional arguments are shown in square brackets. The 'extdef' value specifies if this component is defined within the YAML (default) or externally, allowing an undefined component to be instantiated.
Structure	name: "<name>" type: "<type>" extdef: "True|False" [parameters: <parameter_use_schema>]

Example #1	agents :
	  - name: "control_plane_in"
	    type: "mem"
	    parameters:
	    - name: "ADDR_WIDTH"
	      value: "CP_IN_ADDR_WIDTH"
	    - name: "DATA_WIDTH"
	      value: "CP_IN_DATA_WIDTH"

Example #2	non_uvmf_components :
	  - name: "block_pred_inst"
	    type: "block_predictor"
	    extdef: "True"
	    parameters:
	    - name: "ADDR_WIDTH"
	      value: "CP_IN_ADDR_WIDTH"
	    - name: "DATA_WIDTH"
	      value: "CP_IN_DATA_WIDTH"

Example #3	qvip_memory_agents :
	  - name: "ddr_instance_name"
	    type: "qvip_memory_agent"
	    qvip_environment: "configurator_generated_sub-environment_instance_name"
	    parameters:
	    - name: "CONFIG_T"
	      value: "ddr_vip_config"
	    - name: "TRANS_T"
	      value: "ddr_mem_xfer"

5.2.2.2 scoreboard_schema

Description	Defines a scoreboard instantiation. Optional arguments are shown in square brackets.

Structure	`name: "<name>"`
	`sb_type: "<type>"`
	`trans_type: "<type>"`
	`[ parameters: <parameter_use_schema> ]`

Example	`scoreboards :`
	`  - name: "control_plane_in_sb"`
	`    sb_type: "uvmf_in_order_scoreboard_array"`
	`    trans_type: "mem_transaction"`
	`    parameters:`
	`    - name: "ARRAY_DEPTH"`
	`      value: "NUM_CHANNELS"`

5.2.2.3 parameter_use_schema

Description	Used as part of a component schema, defines a parameter name/value pair for the component's instantiation
Structure	`name: "<name>"` `value: "<value>"`
Example	See use in the component_schema example.

5.2.2.4 parameter_def_schema

Description	Defines a single parameter. All arguments except 'value' are required. If 'value' is not specified the parameter will not have a default value defined.

Structure	`name: "<name>"`
	`type: "<type>"`
	`[ value: "<value>"]`

Example	`parameters :`
	`  - name: "ADDR_WIDTH"`
	`    type: "int"`
	`    value: "16"`

5.2.2.5 import_schema

Description	Defines a single package import
Structure	`name: "<name>"`

Example	`imports :`
	`  - name: "my_pkg"`
	`  - name: "my_other_pkg"`

5.2.2.6 qvip_env_schema

Description	Creates a QVIP sub-environment instantiation
Structure	`name: "<name>"` `type: "<type>"`

Example	`qvip_subenvs :`
	`  - name: "qvip_env"`
	`    type: "axi4_2x2_fabric_qvip"`

5.2.2.7 tlm_port_schema

Description	Defines a TLM port/export to instantiate. The "type" field defines the transaction type to which the component will be parameterized and the "connected_to" field indicates what will be driving/consuming items associated with this component.

Structure	`name: "<name>"`
	`trans_type: "<type>"`
	`connected_to: "<item>"`

Example	`analysis_ports :`
	`  - name: "control_plane_in_ap"`
	`    trans_type: "mem_transaction"`
	`    connected_to: "control_plane_in.monitored_ap"`

5.2.2.8 tlm_schema

Description	Defines a TLM connection within the environment. The "driver" field should be a reference to a port emitting items and the "reciever" should be a reference to an export/imp consuming items. The validate field is optional. It checks the provided driver and receiver against available valid drivers and receivers.

Structure	`driver: "<driving_port>"`
	`receiver: "<receiving_port>"`
	`validate: "<True|False>"`

Example	`tlm_connections :`
	`  - driver: "control_plane_in.monitored_ap"`
	`    receiver: "block_a_pred.control_plane_in_ae"`
	`    validate: "True"`
	`  - driver: "block_a_pred.predicted_item_ap"`
	`    receiver: "scoreboard.expected_analysis_export"`

5.2.2.9 qvip_tlm_schema

Description	Defines a TLM connection within the environment involving a QVIP component as the driver. The "driver" field should be a reference to a QVIP agent underneath its containing QVIP sub-environment, the "ap_key" should refer to the associative array string key within the agent's analysis port array and the "receiver" should be a reference to an export/imp consuming items. The validate field is optional. It checks the provided driver and receiver against available valid drivers and receivers. In the example below, the QVIP sub-environment name is "qvip_env" and the underlying agents are "mgc_axi4_m0" in the first entry and "mgc_axi4_m1" in the second. Refer to QVIP documentation for a list of default AP keys available for each QVIP protocol.

Structure	`driver: "<driving_agent>"`
	`ap_key: "<key>"`
	`receiver: "<receiving_port>"`
	`validate: "<True|False>"`

Example	`qvip_connections :`
	`  - driver: "qvip_env.mgc_axi4_m0"`
	`    ap_key: "trans_ap"`
	`    receiver: "block_a_pred.control_plane_in_ae"`
	`    validate: "False"`
	`  - driver: "qvip_env.mgc_axi4_m1"`
	`    ap_key: "trans_ap"`
	`    receiver: "block_a_pred.secure_data_plane_in_ae"`

5.2.2.10 config_var_schema

Description	Defines a configuration variable to use in the given environment. All arguments are required unless denoted with square brackets. Default for 'isrand' is 'False'. If 'isrand' is 'True' the given configuration variable will be marked with the SystemVerilog 'rand' keyword, allowing it to be modified when the object's 'randomize()' function is called. If 'value' is specified, the configuration variable will be provided an initial value when declared. The 'comment' field is optional and if specified, will be placed on the line above the variable declaration.
Structure	name: "<name>" type: "<type>" [isrand: "True|False"] [value: "<value>"] [comment: "<text>"]

Example	config_vars :
	  - name: "block_a_cfgVar"
	    type: "bit [3:0]"
	    isrand: "True"
	    value: "4'b0101"
	    comment: "Example variable comment."

5.2.2.11 config_value_schema

Description	Defines desired values for underlying sub-environment and agent configuration variables. Paths to configuration variables use the standard 'dot' notation, relative to the current environment's configuration object (which is a parent for all underlying sub-configuration objects). Incorrect paths or incorrect value types will result in compile or runtime errors during simulation.
Structure	name: "<name>" value: "<value>"

Example	"my_env":
	  config_variable_values:
	    - name: "my_subenv.env_cfg_value"
	      value: "32"
	    - name: "my_interface.string_val"
	      value: "foo"

5.2.2.12 constraint_schema

Description	Defines a constraint to be applied to the configuration variables for the given environment. The 'name' and 'value' arguments are required. The 'comment' field is optional and if specified, will be placed on the line above the constraint declaration.

Structure	`name: "<name>"`
	`value: "<value>"`
	`[comment: "<text>"]`

Example	`config_constraints :`
	`  - name: "address_word_align_c"`
	`    value: "{ address[1:0]==2'b00; }"`
	`    comment: "Word alligned addresses."`

5.2.2.13 imp_decl_schema

Description	Specify that an imp_decl macro be defined for this environment package
Structure	`name: "<name>"`

Example	`imp_decls :`
	`  - name: "mem_EXPECTED"`
	`  - name: "mem_ACTUAL"`

5.2.2.14 reg_model_schema

Description	Specify how a given UVM register model should be instantiated and connected in the environment. The "use_adapter" and "use_explicit_prediction" entries default to "True". *Note: At this time only one map entry is supported. More beyond the first will be ignored.* The optional `reg_model_package` can be used to specify the name of the package containing the register model. The optional `reg_block_class` can be used to specify the name of the top level register block. When using a QVIP agent for the interface, set `qvip_agent` to "True" and set the `interface` value to include the QVIP sub-environment name and agent name. For example: `interface: "qvip_env.axi4_master_1"`. When using a QVIP agent for the regmodel map entry, the correct regmodel adapter must be used from the QVIP installation. The UVM reg predictor must also be updated to reflect the QVIP sequence item used by the QVIP reg adapter. Look for `UVMF_CHANGE_ME` in the generated environment class for areas that need customization. If omitted, `qvip_agent` defaults to "False".

Structure	`use_adapter: "True|False"`
	`use_explicit_prediction: "True|False"`
	`reg_model_package: "<reg_package_name>"`
	`reg_block_class: "<top_reg_block_class_name>"`
	`maps:`
	`  - name: "<map_name>"`
	`    interface: "<interface_name>"`
	`    qvip_agent: "<True|False>"`

Example	`register_model :`
	`  use_adapter: "True"`
	`  use_explicit_prediction: "True"`
	`  reg_model_package: "reg_model_pkg"`
	`  reg_block_class: "reg_block"`
	`  maps:`
	`  - name: "bus_map"`
	`    interface: "control_plane_in"`
	`    qvip_agent: "False"`

5.2.2.15 typedef_schema

Description	Defines a typedef. All arguments are required
Structure	`name: "<name>"` `type: "<type>"`

Example	`typedefs :`
	`  - name: "addr_t"`
	`    type: "bit [15:0]"`

Chapter 6

Bench YAML Structure

6.1 Description

The test bench YAML data structure contains information about a bench's name, top-level environment and a host of optional data regarding how to drive clocks and resets as well as active vs. passive mode settings for underlying BFMs. This information is used to create the following content:

- **Classes**: Top level test, top level virtual sequence.
- **Package**: Top level test package, top level sequence package, top level parameters package.
- **Modules**: hdl_top, hvl_top
- **Compilation flow**: File list and Makefile

6.2 YAML Format

Nearly all of the potential content in the bench YAML file is optional. The file is primarily intended to indicate top-level hierarchy and trigger the creation of the appropriate bench-level output. All properties below are optional unless noted otherwise.

6.2.1 Test Bench Variables

Name	Type	Description
`top_env`	string	(Required) Specify the name of the top-level environment to instantiate in this bench. YAML definition for this enviornment must be provided.
`top_env_params`	List of parameter_use_schema	List of parameters to apply at the instantiation of the top-level environment
`parameteres`	List of parameter_def_schema	List of parameters to be defined within the top-level bench
`veloce_ready`	True|False	Defaults to True. Produce emulation-ready code when set to "True"
`use_coemu_clk_rst_gen`	True|False	Defaults to "False". If True, the bench will utilize more complex but more capable clock and reset generation capabilities
`clock_half_period`	string	Time duration of a half-period of the clock. Example: '6ns' or '6'
`clock_phase_offset`	string	Time duration before first clock edge. Example: '25ns' or '25'
`reset_assertion_level`	True|False	Assertion level of reset signal driven by test bench
`reset_duration`	string	Time duration reset is asserted at start of simulation. Example: '100ns'
`active_passive`	List of active_passive_schema	Specify active/passive mode of operation for any underlying BFMs. Default is "ACTIVE"
`interface_params`	List of interface_param_schema	Structure describing how any underlying BFMs should be parameterized
`imports`	List of import_schema	List indicating all of the packages that should be imported by this bench package

additional_tops	List of string	List extra top-level modules to be instantiated within the test bench
mtlb_ready	True|False	Defaults to False. The generated bench will generate files allowing it to work with Matlab
use_bcr	True|False	Defaults to False. If true, calls to make vrun will use BCR underneath other than the default make flow

6.2.2 Schema Definitions

The following structures (schemas) can be used to populate information underneath the top-level properties listed in the table above.

6.2.2.1 parameter_use_schema

Description	Used as part of a component schema, defines a parameter name/value pair for the component's instantiation
Structure	name: "<name>" value: "<value>"
Example	See use in the component_schema example.

6.2.2.2 parameter_def_schema

Description	Defines a single parameter. All arguments except 'value' are required. If 'value' is not specified the parameter will not have a default value defined.
Structure	name: "<name>" type: "<type>" [value: "<value>"]

Example	parameters :
	  - name: "ADDR_WIDTH"
	    type: "int"
	    value: "16"

6.2.2.3 import_schema

Description	Defines a single package import
Structure	name: "<name>"

Example	imports :
	  - name: "my_pkg"
	  - name: "my_other_pkg"

6.2.2.4 active_passive_schema

Description	Specifies if the given BFM (specified by "bfm_name") is ACTIVE or PASSIVE for this test bench. If left unspecified the BFM will be ACTIVE.
Structure	bfm_name: "<name>" value: "ACTIVE|PASSIVE"

Example	active_passive :
	  - bfm_name: "mem_agent"
	    value: "ACTIVE"
	  - bfm_name: "dma_agent"
	    value: "PASSIVE"

6.2.2.5 interface_param_schema

Description	Specifies how a given BFM should be parameterized when instantiated within the bench
Structure	bfm_name: "<name>" value: [parameter_use_schema]

Example	interface_params :
	  - bfm_name: "control_plane_in"
	    value:
	    - name: "ADDR_WIDTH"
	      value: "16"
	    - name: "DATA_WIDTH"
	      value: "32"

Chapter 7

Global Data YAML Structure

7.1 Description

The global data structure provides information that can be used across all other types of objects (interfaces, environments, benches, etc).

7.2 YAML Format

All global data structures are optional. All global schemas must reside underneath a top-level keyword called "global".

7.2.1 Schema Definitions

The following structures can be used to define global data for a generation operation.

7.2.1.1 header

Description	Used to define a global header that is placed at the top of all files that inherit the base template file (all SystemVerilog files). Given that headers are frequently multi-line strings, it is recommended to format this entry as a YAML "block scalar", using the pipe ("|") symbol, as shown in the example below.
Structure	header: "<string>"

Example	uvmf :
	  global:
	    header: |
	      // Header that will be used across all files
	      // (c) My Company

7.2.1.2 flat_output

Description	Can either take a value of "True" or "False", defaulting to "False". When "True", all generated source for a given bench/environment/interface will be placed in a single flat directory. When "False", most files will be placed in a subdirectory called "src" underneath the directory reserved for a given bench/environment/interface.
Structure	`flat_output: ["True"|"False"]`

Example	`uvmf :`
	`  global:`
	`    flat_output: "True"`

7.2.1.3 vip_location

Description	This variable can be used to control where generated interface source code is placed. The default value, if left unspecified, is "verification_ip". The directories defined by "interface_location" and "environment_location" will exist underneath this directory.
Structure	`vip_location: "<string>"`

Example	`uvmf :`
	`  global:`
	`    vip_location: "my_vip"`

7.2.1.4 interface_location

Description	This variable can be used to control where generated interface source code is placed. The default value, if left unspecified, is "interface_packages". This directory's parent structure is defined by the "vip_location" variable.
Structure	`interface_location: "<string>"`

Example	`uvmf :`
	`  global:`
	`    interface_location: "my_interface_source"`

7.2.1.5 environment_location

Description	This variable can be used to control where generated environment source code is placed. The default value, if left unspecified, is "environment_packages". This directory's parent structure is defined by the "vip_location" variable.
Structure	`environment_location: "<string>"`

Example	`uvmf :`
	`  global:`
	`    environment_location: "my_environment_source"`

7.2.1.6 bench_location

Description	This variable can be used to control where generated bench source code is placed. The default value, if left unspecified, is "project_benches".
Structure	`bench_location: "<string>"`

Example	`uvmf :`
	`  global:`
	`    bench_location: "my_bench_source"`