UVM Framework Code Generator YAML Reference

Revision 3.6h

UVMF Code Generator YAML Reference Table of Contents

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. The script utilizes the established Python code generation API in order to operate. See the API reference manual for more details.

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

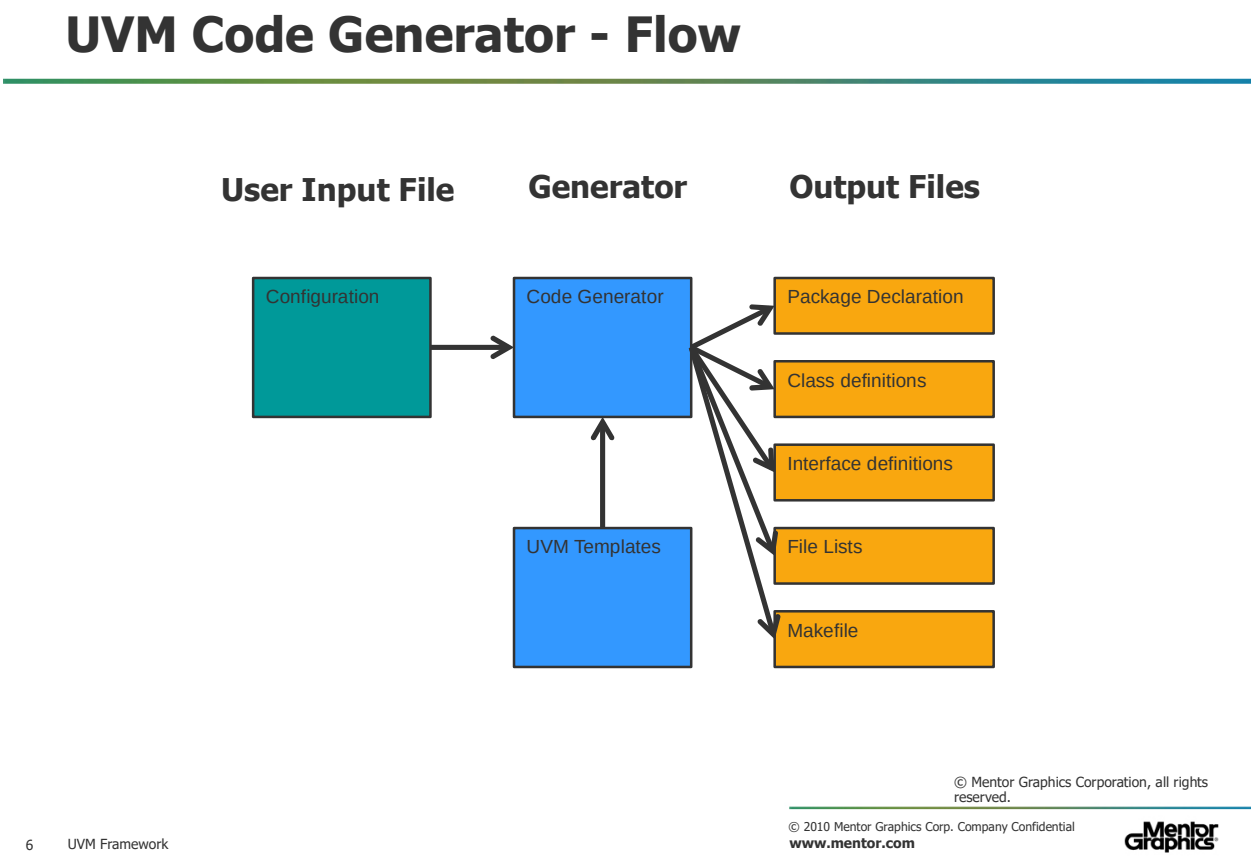
Interface Examples	**Description**
`mem_if_cfg.yaml`	User input file for generating an interface package named mem_pkg. This interface is used in block_a, block_b and chip environments and test benches
`pkt_if_cfg.yaml`	User input file for generating an interface package named pkt_pkg. This interface 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	
`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.

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`, under which the uvmf_gen API is used to produce output. Files generated can include all classes, packages, BFM's, 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.



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>
…
```

The contents within each named interface, environment or bench are described in sections below. The information can be spread across multiple files or in a single file.

2 Interface YAML Structure

2.1 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.

BFM's: Driver and monitor.

Compilation flow: File list and Makefile

2.2 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.

2.2.1 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 resets 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.
`vip_lib_env_variable`	string	Name of environment variable that will point to the location of the source for this interface package.
`veloce_ready`	True|False	Generated code is Veloce ready/friendly
`infact_ready`	True|False	Produce additional files and content to enable operation with inFact intelligent stimulus

		generation
imports	List of import_schema	List of package names to be imported for use within this interface
parameters	List of parameter_def_schema	List of parameter definitions for use with this interface
hdl_typedefs	List of typedef_schema	List of typedefs to be associated with the HDL side of this interface
hvl_typedefs	List of typedef_schema	List of typedefs to be associated with the HVL side of this interface
ports	List of port_schema	List of ports to be defined within the wire bundle for this interface
transaction_vars	List of transaction_schema	List defining all of the transaction variables associated with this interface
transaction_constraints	List of constraint_schema	List defining all of the constraints to be applied against transaction variables
config_vars	List of config_var_schema	List defining the configuration variables associated with this interface
config_constraints	List of constraint_schema	List defining the constraints to be applied against the constraint variables
response_info	response_schema	Structure defining how this interface should act when configured as a responder
dpi_define	dpi_define_schema	Structure defining DPI source associated with this interface

2.2.2 Interface Schema Definitions

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

2.2.2.1 import_schema

Description	Defines a single package import
Structure	`{ name: '<name>' }`

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

2.2.2.2 parameter_def_schema

Description	Defines a single parameter definition. All arguments are required.
Structure	`{ name: '<name>', type: '<type>', value: '<value>' }`
Example	`parameters :` `- { name: "ADDR_WIDTH", type: "int", value: "16" }`

2.2.2.3 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]" }`

2.2.2.4 port_schema

Description	Defines a single port definition for use in an interface wire bundle. All arguments are required
Structure	`{ name: '<name>', width: '<width>', dir: '<dir>' }`

Example	`ports :`
	`- { name: "rdata", width:"32", dir: "input" }`
	`- { name: "wdata", width:"32", dir: "output" }`

2.2.2.5 transaction_schema

Description	Defines a single transaction to be placed within an interface's sequence item definition. All arguments are required unless surrounded with square brackets.

Structure	`name: '<name>'`
	`type: '<type>'`
	`[ israndd: ''True''|''False'' ]`
	`[ iscompare: ''True''|''False'' ]`
	`[ unpacked dimension: '<dim>' ]`

Example	`transaction_vars :`
	`- name: "data"`
	`type: "bit [15:0]"`
	`israndd: "True"`
	`iscompare: "True"`
	`unpacked_dimension: "[1000]"`
	`- name: "latency"`
	`type: "int"`
	`israndd: "True"`
	`iscompare: "False"`

2.2.2.6 config_var_schema

Description	Defines a configuration variable to use in the given interface. All arguments are required unless denoted with square brackets. Default for 'israndd' is 'True'.

Structure	name: '<name>' type: '<type>' [israND: (''True''|''False''] }
Example	config_vars : - name: "block_a_cfgVar" type: "bit [3:0]" israND: "True"

2.2.2.7 constraint_schema

Description	Defines a constraint to be applied to the transaction variables for the given interface. All arguments are required
Structure	{ name: '<name>', value: '<value>' }
Example	transaction_constraints : - name: "address_word_align_c" value: "{ address[1:0]== 2'b00; }"

2.2.2.8 response_schema

Description	Defines how an interface behaves when configured as a responder. All arguments are required.
Structure	operation: '<logical_operation>' data: [<array of variables>]
Example	response_info : operation: "~txn.wr" data : ["data","data_parity"]

2.2.2.9 dpi_schema

Description	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.
Structure	`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>]`

Example	`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: "(unsigned int variable1, unsigned int variable2)"`
	`  sv_args:`
	`  - { name: "variable1", type: "int unsigned", dir: "input" }`
	`  - { name: "variable2", type: "int unsigned", dir: "input" }`
	`  - name: "good_bye_world_from_interface"`
	`  return_type: "void"`
	`  c_args: "(unsigned int variable3, unsigned int variable4)"`
	`  sv_args:`
	`  - { name: "variable3", type: "int unsigned", dir: "input" }`
	`  - { name: "variable4", type: "int unsigned", dir: "input" }`

2.2.2.10 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

3 Utility Component YAML Structure

3.1 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. Currently only 'predictor' and 'coverage' components are supported in this section and these are used when generating environments in which they are instantiated.

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 predictors' analysis_exports the predictor broadcasts a transaction out of each of the predictors' 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.

3.2 YAML Format

Top-level properties for all utility components are listed in the table below. The expectation is that individual utility components will have different overall structure but because 'predictor' components are the only defined structure at this time, only that YAML format is defined.

3.2.1 Top-Level Properties

Name	Type	Description
`type`	"`predictor|coverage`"	Indicates what type of utility component definition this is. The only supported value for this property is currently "predictor".
`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.

3.2.2 Schema definitions

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

3.2.2.1 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 Environment YAML Structure

4.1 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

4.2 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. 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.

4.2.1 Top-Level Properties

Name	Type	Description
agents	List of	Ordered list of underlying agents

	component_schema	(interfaces) to instantiate within this environment. The YAML definition for agents must be provided as part of the script run.
parameters	List of parameter_def_schema	Top-level parameters to be defined for this environment
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) 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
subenvs	List of component_schema	List of sub-environments to instantiate within this environment. YAML definitions for each sub-environment must be provided.
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 will be connected together.
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_constraints	List of constraint_schema	Defines constraints associated with the configuration variables for this environment
imp_decls	List of	Defines the names of imp_decl macros

	imp_decl_schema	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 is identical.
`typedefs`	typedef_schema	Specifies typedefs to be defined in this environment. See `typedef_schema` for more details.

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 component_schema

Description	Defines a component. 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>'`
	`[parameters: <parameter use schema> ]`

Example	`agents:`
	`- name: ''control_plane_in''`
	`type: ''mem''`
	`initiator_responder: ''INITIATOR''`
	`parameters:`
	`- { name: ''ADDR_WIDTH'', value: ''CP_IN_ADDR_WIDTH'' }`
	`- { name: ''DATA_WIDTH'', value: ''CP_IN_DATA_WIDTH'' }`

4.2.2.2 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	`agents:`
	`- name: ''control_plane_in''`
	`type: ''mem''`
	`parameters:`
	`- { name: ''ADDR_WIDTH'', value: ''CP_IN_ADDR_WIDTH'' }`
	`- { name: ''DATA_WIDTH'', value: ''CP_IN_DATA_WIDTH'' }`

4.2.2.3 parameter_def_schema

Description	Defines a single parameter definition. All arguments are required.
Structure	`{ name: '<name>', type: '<type>', value: '<value>' }`
Example	`parameters :` `- { name: "ADDR_WIDTH", type: "int", value: "16" }`

4.2.2.4 import_schema

Description	Defines a single package import
Structure	`{ name: '<name>' }`

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

4.2.2.5 qvip_subenv_schema

Description	Defines a QVIP sub-environment to instantiate
Structure	`{ name: '<name>', type: '<type>' }`

Example	`qvip_subenvs:` `- { name: "qvip_env", type: "axi4_2x2_fabric_qvip" }`

4.2.2.6 tlm_port_schema

Description	Defines a TLM port/export to instantiate. The "type" field defines the transaction type that the component will be parameterized to 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"`
	`type: "mem_transaction"`
	`connected_to: "control_plane_in.monitored_ap"`

4.2.2.7 tlm schema

Description	Defines a TLM connection within the environment. The "driver" field should be a reference to a port emitting items and the "receiver" should be a reference to an export/imp consuming items.
Structure	`driver: '<driving_port>', receiver: '<receiving_port>'`

Example	`tlm_connections :`
	`- driver: "control_plane_in.monitored_ap"`
	`receiver: "block_a_pred.control_plane_in_ae"`
	`- driver: "secure_data_plane_in.monitored_ap"`
	`receiver: "block_a_pred.secure_data_plane_in_ae"`

4.2.2.8 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 sub-environment and underlying agent (hierarchy denoted with an underscore), 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. 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.
Structure	`driver: '<driving_port>', ap_key: '<key>', receiver: '<receiving_port>'`

Example	`qvip_connections :`
	`- driver: "qvip_env_mgc_axi4_m0"`
	`ap_key: "trans_ap"`
	`receiver: "block_a_pred.control_plane_in_ae"`
	`- driver: "qvip_env_mgc_axi4_m1"`
	`ap_key: "trans_ap"`
	`receiver: "block_a_pred.secure_data_plane_in_ae"`

4.2.2.9 config_var_schema

Description	Defines a configuration variable to use in the given interface. All arguments are required unless denoted with square brackets. Default for 'isrand' is 'True'.

Structure	`name: '<name>'`
	`type: '<type>'`
	`[isrand: (''True''|''False'']` }

Example	`config_vars :`
	`- name: "block_a_cfgVar"`
	`type: "bit [3:0]"`
	`isrand: "True"`

4.2.2.10 constraint_schema

Description	Defines a constraint to be applied to the transaction variables for the given interface. All arguments are required
Structure	`{ name: '<name>', value: '<value>' }`

Example	`transaction_constraints :`
	`- name: "address_word_align_c"`
	`value: "{ address[1:0]== 2'b00; }"`

4.2.2.11 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"`

4.2.2.12 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. Any more will be ignored.

Structure	`use_adapter: 'True' | 'False'`
	`use_explicit_prediction: 'True' | 'False'`
	`maps:`
	`- { name: '<map name>', interface: '<interface name>' }`

Example	`register_model :`
	`use_adapter: "True"`
	`use_explicit_prediction: "True"`
	`- { name: "bus_map", interface: "control_plane_in" }`

4.2.2.13 typedef_schema

Description	Defines a typedef. All arguments are required.
Structure	`{ name: '<name>', type: '<type>' }`
Example	`typedefs :` `- { name: "addr_t", type: "bit [15:0]" }`

5 Test Bench YAML Structure

5.1 Description

The test bench YAML data structure contains information about an 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.

Packages: top level test package, top level sequence package. Top level parameters package.

Modules: hdl_top, hvl_top.

Compilation flow: File list and Makefile

5.2 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.

5.2.1 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 environment must be provided.
`top_env_params`	List of parameter_use_schema	List of parameters to apply at the instantiation of the top-level environment
`parameters`	List of parameter_def_schema	List of parameters to be defined at the top-level
`veloce_ready`	True|False	Produce emulation-ready code when set to "True"
`infact_ready`	True|False	Test bench generated is inFact ready. Makefile contains variables, switches, and arguments to run inFact.
`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 utilities.
`clock_half_period`	string	Time duration of half period. Example: '6ns', or '6'
`clock_phase_offset`	string	Time duration before first clock edge. Exaple: '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', or '100'
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 additional_top_schema	List extra top-level modules to be instantiated within the test bench

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 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	`agents:`
	`- name: ''control_plane_in''`
	`type: ''mem''`
	`parameters:`
	`- { name: ''ADDR_WIDTH'', value: ''CP_IN_ADDR_WIDTH'' }`
	`- { name: ''DATA_WIDTH'', value: ''CP_IN_DATA_WIDTH'' }`

5.2.2.2 parameter_def_schema

Description	Defines a single parameter definition. All arguments are required.
Structure	`{ name: '<name>', type: '<type>', value: '<value>' }`
Example	`parameters :` `- { name: "ADDR_WIDTH", type: "int", value: "16" }`

5.2.2.3 import_schema

Description	Defines a single package import
Structure	`{ name: '<name>' }`

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

5.2.2.4 active_passive_schema

Description	Specifies if the given BFM (specified by bfm_name) is ACTIVE or PASSIVE for this testbench. If left unspecified an agent 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"}`

5.2.2.5 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" }`

5.2.2.6 additional_top_schema

Description	Specifies an extra top-level module to be instantiated within the bench
Structure	`{ name: '<name>' }`

Example	`additional_tops:`
	`- { name: "extra_top0" }`
	`- { name: "extra_top1" }`