Novel GUI Based UVM Test Bench Template Builder

Vignesh Manoharan
Aeva Inc.
555 Ellis St. Mountain View, CA 94043
+1-858-717-2486, vignesh@aeva.ai

***Abstract*- Adoption rate of Universal Verification Methodology (UVM) is increasing day by day across industry and the need for building new Verification Intellectual Property (VIP) or testbench is in great demand. Writing effective and structured UVM testbench from scratch is cumbersome most of the time and following a standard structure with provision for better re-usability across projects is also challenging. What if the time taken for initial development cycle is reduced to minutes instead of days with the help of a Graphic User Interface (GUI) to build the verification component templates? This abstract presents an overview about the GUI interface used to develop the individual UVM components or the entire VIP templates loaded with features to customize and configure as per the user requirements.**

I. Introduction

Most often there is not standard structure, template or outline that user follows while developing any of the VIP components or verification environment codes. In general, engineers tend to carry forward the existing files from other projects and try to edit the information, which leads to non-standard code development across projects. Although there are some existing template generators which uses command line or spreadsheets, it is noticed that with increased number of parameters and complex requirements, using these generators have become more cumbersome than helpful.

This GUI based UVM template generator allows the user to create any component template dynamically in matter of minutes with lots of customization. This approach also brings in standardization of code development across projects and business groups as well as helps to bring up VIP's initial development cycle at a faster pace.

II. Deep Dive Into UVM Template Generator Operation

A. How the tool is implemented and what is supports

The tool is built using Python Tkinter framework to create the GUI layouts in grid fashion mechanism. All text processing and editing are done using python scripting. The tool helps in:

- Building pure UVM template codes
- Building single UVM components or complete UVM testbench and architecture
- Building Multi Agent, Multi Monitor, Multi Scoreboard based Environments
- Building Multi-Environments based flow targeting complex SOC's [System on Chip] scenarios
- Integrating Agents, Monitors, Scoreboards into already existing Environment and helps in integration between environments
- All the codes generated from this tool uses 'Natural Docs [3]' formatting for easier documentation

B. Creating Single UVM Components

The moment user launches the tool, the GUI pops-up with two options, namely 1. create "Single UVM Component", 2. "Single & Multi Env VIP" as shown in Figure 1.

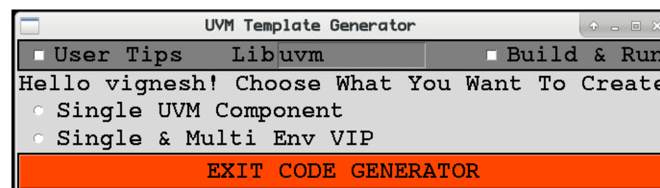


Figure 1. Initial Tool Layout

Once the user clicks the single component radio button, tool lists out multiple objects and component options to be created namely: sequence_item, agent, environment etc. as shown in Figure 2. The user can choose whichever component or object they want and create the corresponding templates by clicking the 'Generate Code' button. Based on the component the user chooses, the tool displays required customization options.

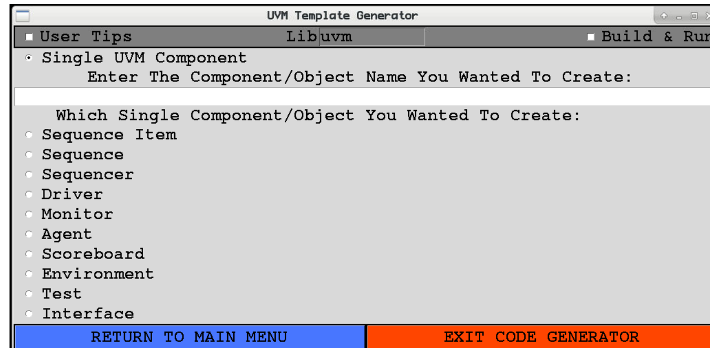


Figure 2. Single UVM Component Tool Layout

Interface Creation: The tool provides the user with multiple options to develop an interface file such as:

- Creating a default interface with an empty shell
- A user defined interface via the GUI as shown in Figure 3
- Loading a spreadsheet

For example, if the user wants a simple interface with 5 to 10 signals, then the user can choose to create using a user defined interface option. To do this, all the users needs to do is enter the necessary signal details in the required entry widget, which includes signal name, type, packed/un-packed element, clocking block, modport details and click "Done interface config", to generate the necessary code. If the interface contains many signals, the tool provides the option to load interface details using a spreadsheet method and then generate the interface code.

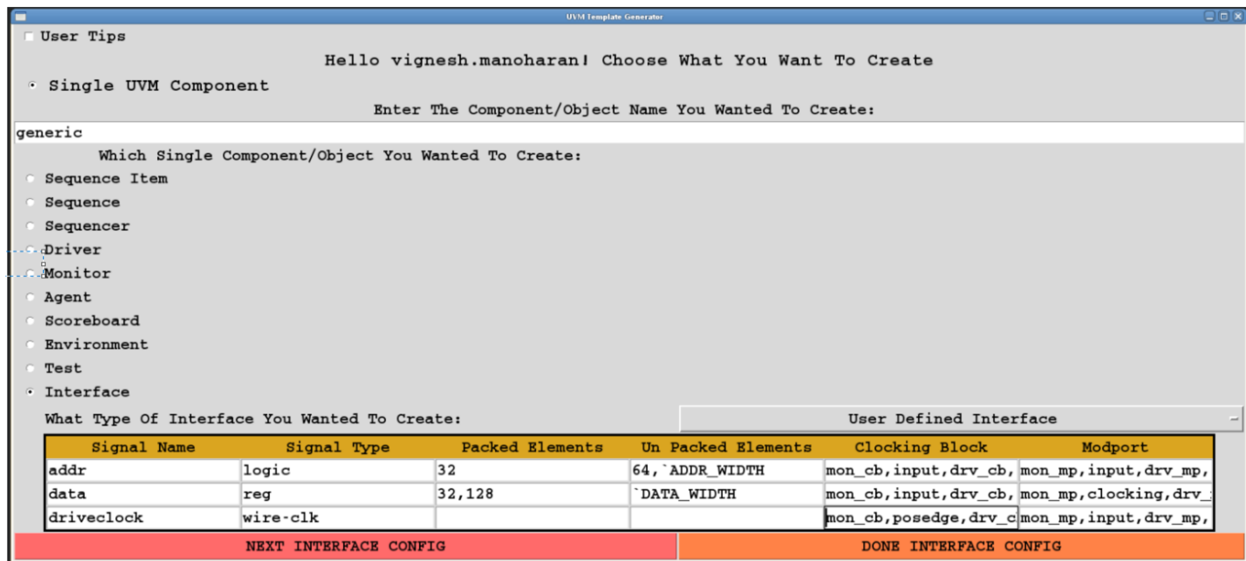
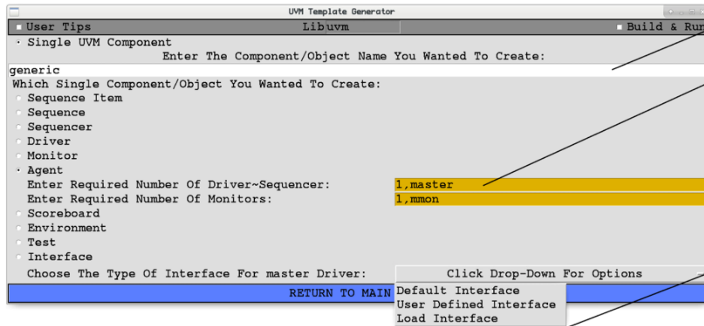


Figure 3. User Defined Interface Details Filled via GUI

Agent Creation: When the user wants to create an agent, the tool further provides options for the user to enter the number of driver-sequencers or monitors they want with required names as shown in Figure 4. The tool generates the necessary code templates which are compile clean and ready to use. As soon as the code is generated, the tool goes to the default/initial layout.



a. Enter the name of the Agent.
b. Once you click the Agent radio-button, the tool provides the user with following options:
   i. How many driver-sequencer you wanted to create inside the agent.
   ii. How many monitors you wanted to create inside the agent.
c. The user is also provided with the option to add explicit names for individual components. And the format followed is provided in the highlighted entry box.
   i. Enter no of drivers, 1st driver name, .. nth.
   ii. 2, apple, mango
d. Now when the user clicks the Agent component, and enters the driver names, tool automatically allows the user to create the respective interface for each driver mentioned.
e. The user is provided with three options to define the interface,
   i. Default interface,
   ii. User defined interface,
   iii. Load interface

Figure 4. Component Specific Customization Layout

C. Creating Complete UVM VIP

The moment user clicks the “Single & Multi Env VIP” from the initial layout, the tool provides a couple of options as shown in the Figure 5, namely 1. GUI Approach, 2. Load Spreadsheet Approach

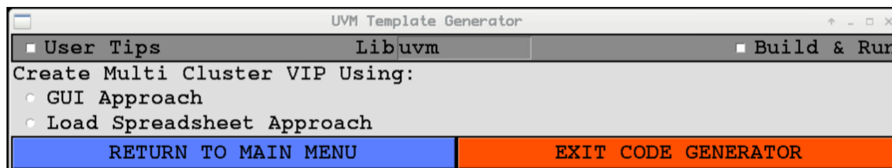


Figure 5. Complete UVM VIP Development tool Layout

In the GUI mode, the tool provides the user with entry widgets to enter details about a complete environment. The environment can contain n-instances of sub-environments, n-instances of agents, environment level monitors, scoreboards and interface files. The complete GUI based approach tool layout is shown in Figure 6.

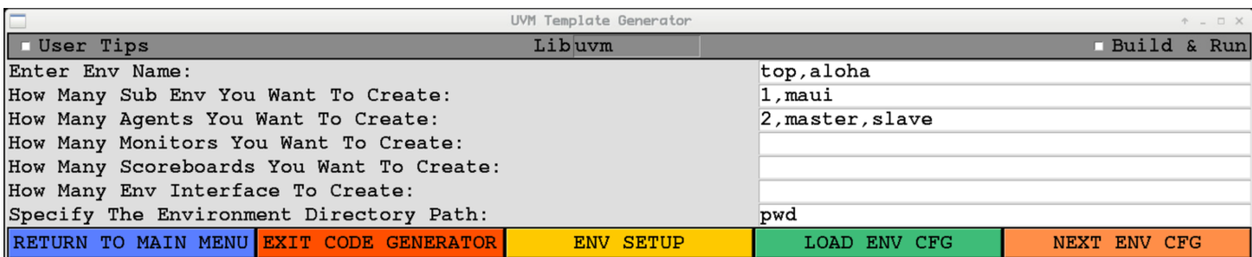


Figure 6. GUI Approach tool layout with partial filled in data

As the user starts filling in the details about the environment to be created, tool intuitively brings up the required widgets to provide necessary details. For example, when the user starts filling the details about the agent, the tool provides input widgets to enter the details about driver, agent level monitor and the interface information as shown in Figure 7.

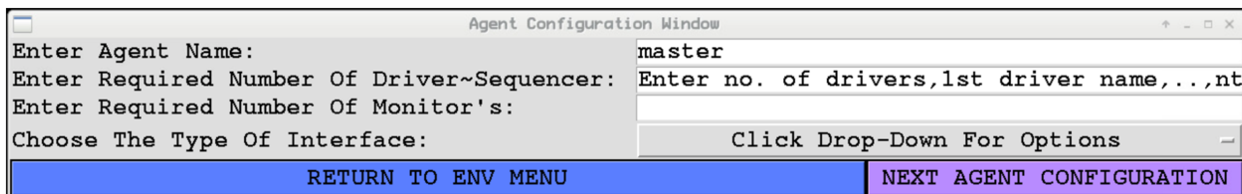


Figure 7. Popped up tool window for acquiring agent details

Once the user enters all the required information about the environment to be built, the user will click the "Env Setup" button found at the bottom of the tool window. By clicking that button, the tool generates a matrix table with all the monitors, scoreboards and provide the option for user to make the necessary connection as shown in Figure 8.

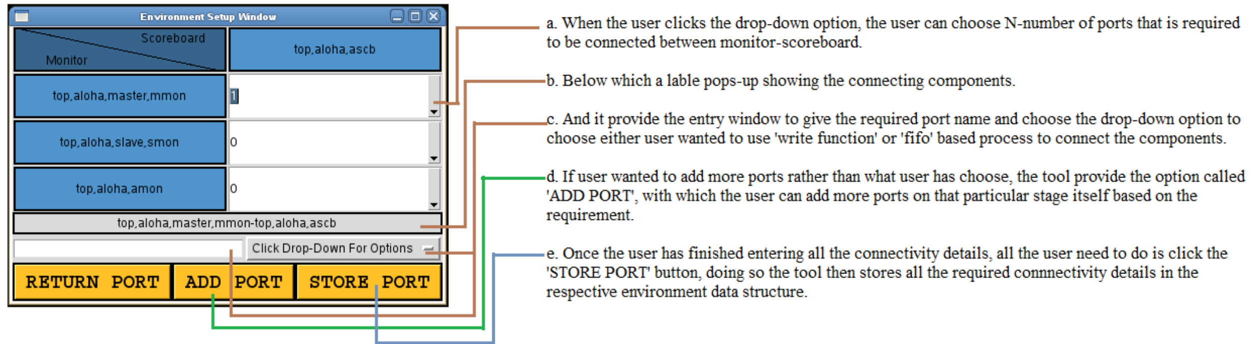


Figure 8. Monitor-Scoreboard Connectivity Matrix table

Once the user has provided the required details about all the environments and the monitor-scoreboard connectivity information, the user needs to click the “Done Env Cfg” button as shown in Figure 9 to instruct the tool that the user has confirmed all the testbench setup and it is safe to move ahead.

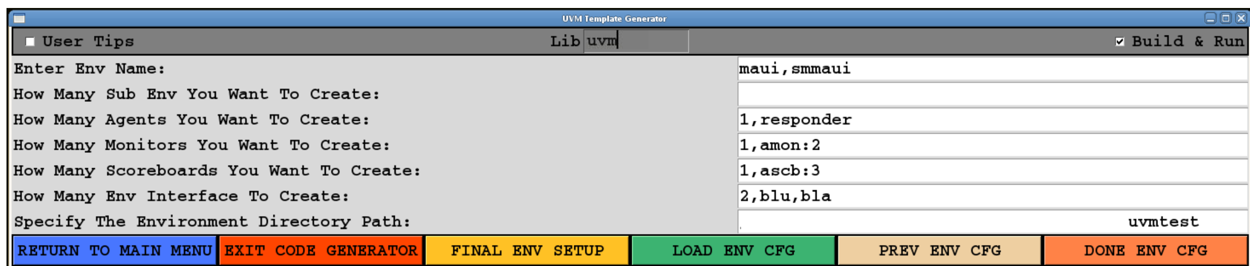


Figure 9. Environment configuration confirmation layout

After confirming the environment configuration, the user then clicks the “Generate Code” button as shown in Figure 10. This will instruct the tool to build the testbench codes, necessary files, and directory structures.

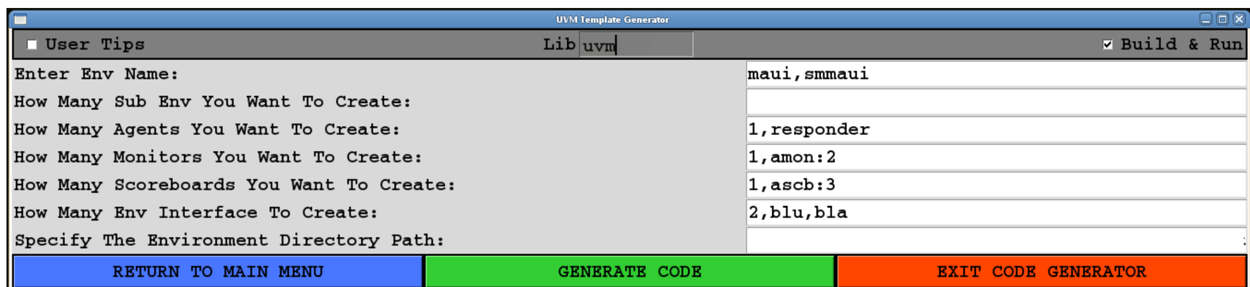


Figure 10. Final tool layout before proceeding to generate code

By clicking the "Generate Code" button, the tool generates:

- Agent related files, which includes agent, driver, sequencer, monitor, and agent config files
- Environment file, environment configuration, monitors and scoreboards templates
- A basic test and testbench top to quickly run and check the setup
- Sequence item, sequence, environment, and test package file with all the required files included in the precise order needed so that it compiles clean
- Necessary include files and environment source scripts
- An important file called “Environment Configuration Dump” xlsx file. This file contains all the details about the components that are built inside the environment

In the complete UVM VIP tool window, when the user chooses the “Load Spreadsheet Approach”, the tool pops up with the layout as shown in Figure 11.

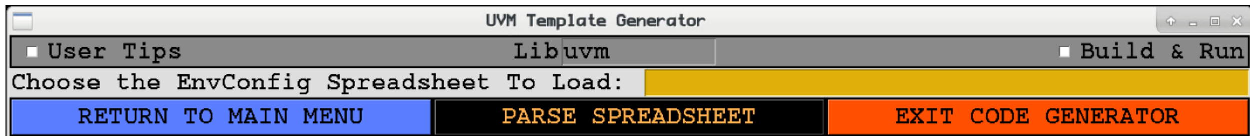


Figure 11. Load spreadsheet tool layout

In this mode, the user can enter the details in a tool understandable spreadsheet format as shown in the Figure 12.

EnvironmentNo	Environment	ParentEnvi	SubEnviror	Monitor	Scoreboar	Agent	AgentConfig	MonScbCon	EnvIntfDetails	Directory	EnvCfgFilePath
0	aloha	top	1,maui	1,amon:2	1,ascb:3	2,master,slave	master	aloha,master,mmon-aloha,ascb-aoo,2	1,aintf	<Path>	
							2,red,ruby		1		
							1,mmon				
							1-1				
							slave				
							1,blue				
							1,smon				
							1				
1	maui	aloha									<Path>/maui_env/guidocs/maui_env_cfg_file.xlsx,maui_env_cfg

a. Agent Name.
b. Driver-Sequencer Details,
c. Monitor Details,
d. Interface Details

Figure 12. Spreadsheet example which the tool understands

By selecting the necessary file to be parsed and clicking the “Parse Spreadsheet” button, the tool reads through the xlsx file and automatically fills with in the environment details in the GUI layout. The user can then browse through the configuration in the tool before generating the final UVM template code.

D. Novel Stitch, Create & Stitch Feature

The art of developing a testbench doesn’t happen in a single day but is a continual long-term process. For example, on day 1, the user might just need to build the environment skeleton. On day 2, the user might end up adding few other components namely agents and environment level monitors. Later the user adds the required scoreboard and connectivity. How does this tool take care of such cases? Well, the tool provides couple of novel features namely, “Stitch”, “Create & Stitch” modes which helps in incremental testbench development process.

Stitch Mode: This mode comes in handy if the user has already created an environment with the tool which takes care of generating different kinds of clock sources and now the user wants to build a block level bench which is going to take care of register programming. The user needs to launch the tool, generate the required block level testbench skeleton and then, using the “Stitch mode”, the user can stitch the other sub-environments into this block level environment. The user needs to enter the required number of sub-env's wanted to be stitched and add the sub-env's names appended with “__s” as shown in the Figure 13.

a. To add a new environment into an existing environment, user need to place the cursor into the sub-env entry window.
b. Enter the number of sub-env, followed by name appended by “__s”. For e.g., clock__s as shown.

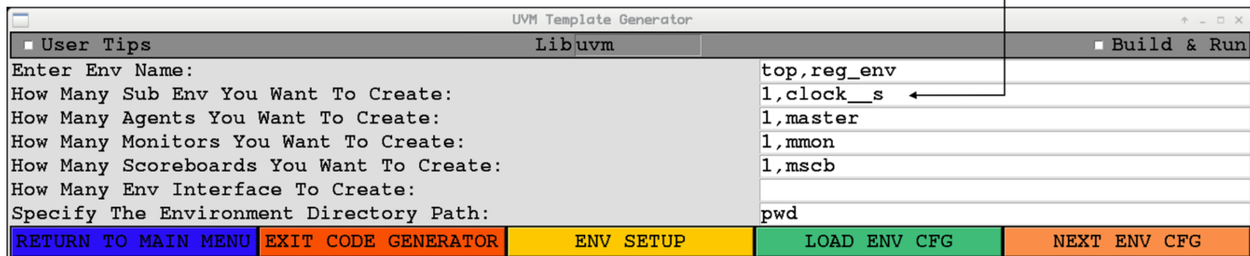


Figure 13. Novel Stitch method for adding existing environment

By doing this, the tool will understand that the user is trying to “Stitch” an already created environment into an existing environment. The tool automatically takes care of instantiating and editing the necessary files to include the individual elements. It does not re-generate the files but smartly updates the content.

Create & Stitch Mode: If, in the above block level environment, the user wants to add a new agent, the user can do so by launching the tool and loading the block level environment using “Load Spreadsheet” Mode. Next, the user can add the details about new agent i.e., number of agents followed by the name of agent appended with “__c”, as shown in Figure 14 and then clicking the "Generate Code" button. The tool knows the user is adding new components onto an existing block level environment, so the tool generates only the new components and then stitches them onto the already existing environment in all the necessary places.

a. To add a new agent into an existing environment, user need to place the cursor into the agent’s entry window.
b. Enter the number of new agents, followed by name appended by “__s”. For e.g., slave__s as shown.

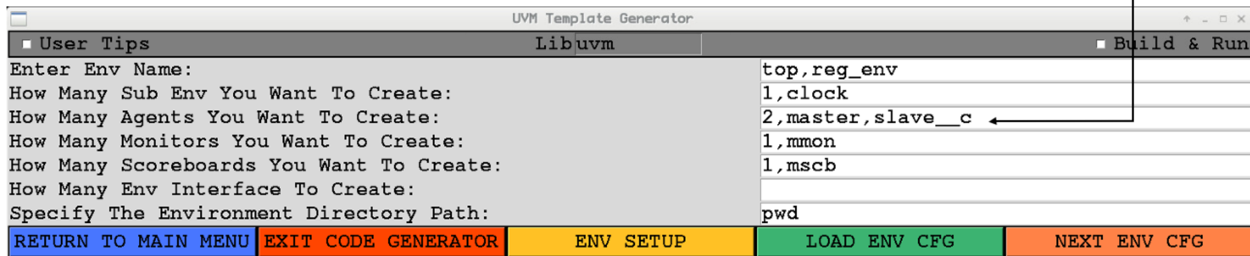


Figure 14. Novel Create and Stitch method for adding new agents into existing environment

Thus, these novel techniques cleverly help the users in building the testbench over the course of project. The granularity level at which the work can start from can be:

- Adding single or multiple port connectivity’s between monitor and scoreboard
- Adding new environment level monitor’s and scoreboard’s in already existing environment with updated connectivity
- Adding driver/monitor inside already existing agents and adding new agents into existing environment
- Adding an environment using create & stitch or just stitch process into an already existing environment

III. Code Snapshots

Let’s look at some of the important code snippets that are being auto generated by the tool. All the codes shown below are stripped down version of complete code which has proper comments and indentation. Figure 15 shows the code snippet for agent component. All the required components such as driver, sequencer, and monitor are instanced correctly, and they are completely controlled by agent level configuration.

```
 1 `ifndef INC_MASTER_AGENT_SV
 2 `define INC_MASTER_AGENT_SV
 3
 4 class master_agent extends uvm_agent;
 5  int unsigned              master_agent_id;
 6  master_agent_config       master_agent_cfg;
 7  master_sequencer          master_sqr;
 8  master_driver             master_drv;
 9  master_monitor            master_mon;
10
11 `uvm_component_utils_begin(master_agent)
12 `uvm_field_int(master_agent_id, UVM_ALL_ON)
13 `uvm_component_utils_end
14 endclass :master_agent
```

```
 1 function void master_agent::build_phase(uvm_phase phase);
 2  super.build_phase(phase);
 3
 4  if (!uvm_config_db#(master_agent_config)::get(this, "*", "master_agent_config", master_agent_cfg))
 5  begin
 6   `uvm_error(get_type_name(), "master_agent_config object is not found in config_db!");
 7  end
 8
 9  if (master_agent_cfg.master_is_active) master_mon = master_monitor::type_id::create("master_mon",this);
10
11  if (master_agent_cfg.master_is_active) begin
12   master_drv = master_driver::type_id::create("master_drv",this);
13   master_sqr = master_sequencer::type_id::create("master_sqr",this);
14  end
15 endfunction: build_phase
16
17 function void master_agent::connect_phase(uvm_phase phase);
18  super.connect_phase(phase);
19  if (master_agent_cfg.master_is_active) begin
20   master_drv.seq_item_port.connect(master_sqr.seq_item_export);
21  end
22 endfunction: connect_phase
23
24 `endif // INC_MASTER_AGENT_SV
```

Figure 15. Agent template code

The code snippet in Figure 16 shows the code implementation for an environment component. As you can see, the required dynamic agent components and scoreboards are instanced. They are created based on a top-level environment configuration file, thus giving user the flexibility to control the components as needed from top level test. The tool automatically takes care of the connectivity between the components, passing the leaf level configuration to the respective components obtained from the environment configuration.

```
`ifndef INC_GENERIC_ENVIRONMENT_SV
`define INC_GENERIC_ENVIRONMENT_SV

class generic_environment extends uvm_env;
 mon_monitor                 mon_mon;
 scb_scoreboard              scb_scb;
 master_agent                master_agt;
 slave_agent                 slave_agt[];
 generic_environment_config generic_environment_cfg;

 `uvm_component_utils_begin(generic_environment)
 `uvm_field_int(generic_env_id, UVM_ALL_ON)
 `uvm_component_utils_end
endclass: generic_environment

function void generic_environment::build_phase(uvm_phase phase);
 ...
 slave_agt = new[generic_environment_cfg.no_of_slave_agt];
 for (int i = 0; i < generic_environment_cfg.no_of_slave_agt; i++) begin
  $sformat(agent_inst_name, "slave_agt[%0d]", i);
  if (generic_environment_cfg.slave_agent_cfg[i].is_active) slave_agt[i] = slave_agent::type_id::create(agent_inst_name, this);
  uvm_config_db#(int)::set(this, {agent_inst_name, "*"}, "slave_agt_id", i);
 end
endfunction: build_phase

function void generic_environment::connect_phase(uvm_phase phase);
 if (generic_environment_cfg.scb_is_active && generic_environment_cfg.master_agent_cfg.master_is_active) begin
  master_agt.master_mon.master_abc_analysis_port.connect(scb_scb.scb_master_abc_analysis_export);
 end
endfunction: connect_phase
`endif //INC_GENERIC_ENVIRONMENT_SV
```

Figure 16. Environment template code

The code snippet in Figure 17 shows the different options which were provided to the user while connecting the monitor to the scoreboard, i.e., either using FIFO based approach on the left side or using a user defined write function method. Based on the user choice, the tool automatically builds the required codes and makes the necessary connectivity between the components. The tool uses a built-in tree algorithm to identify which environment the respective monitor-scoreboard connectivity needs to be placed in a multi-env scenario.

```
`ifndef INC_MASTER_SCOREBOARD_SV
`define INC_MASTER_SCOREBOARD_SV

class master_scoreboard extends uvm_scoreboard;
 generic_environment_config generic_environment_cfg;
 uvm_analysis_export #(master_master_sequence_item_base) master_master_master_ab
 local uvm_tlm_analysis_fifo #(master_master_sequence_item_base) master_master_m

 `uvm_component_utils_begin(master_scoreboard)
 `uvm_component_utils_end
endclass: master_scoreboard

function master_scoreboard::new(string name = "master_scoreboard", uvm_component
 super.new(name, parent);
 master_master_master_abc_analysis_export = new("master_master_master_abc_analys
 master_master_master_abc_analysis_fifo = new("master_master_master_abc_analysis
endfunction: new

function void master_scoreboard::connect_phase(uvm_phase phase);
 super.connect_phase(phase);
 master_master_master_abc_analysis_export.connect(master_master_master_abc_analy
endfunction: connect_phase

`endif //INC_MASTER_SCOREBOARD_SV
```

```
`ifndef INC_SLAVE_SCOREBOARD_SV
`define INC_SLAVE_SCOREBOARD_SV

`uvm_analysis_imp_decl(_slave_master_def_scoreboard)

class slave_scoreboard extends uvm_scoreboard;
 generic_environment_config generic_environment_cfg;
 uvm_analysis_imp_slave_master_def_scoreboard #(master_sequence_item_base, slave_s

 extern virtual function void write_slave_master_def_scoreboard(master_sequence_it

 `uvm_component_utils_begin(slave_scoreboard)
 `uvm_component_utils_end
endclass: slave_scoreboard

function slave_scoreboard::new(string name = "slave_scoreboard", uvm_component par
 super.new(name, parent);
 slave_master_def_analysis_export = new("slave_master_def_analysis_export", this);
endfunction: new

function void slave_scoreboard::write_slave_master_def_scoreboard(master_sequence_
endfunction: write_slave_master_def_scoreboard

`endif //INC_SLAVE_SCOREBOARD_SV
```

Figure 17. Code difference between FIFO based and Write function-based connectivity in Scoreboard

Figure 18 shows the code snippet for interface code. This interface is generated using load spread-sheet method, where the user can define the details about each signal to be used. The user can also include the necessary signals for the clocking blocks and add them in the modport based on the requirement. The tool provides the user option to generate interface with/without clocking block and modport by using the right switches.

One of the important, auto generated code feature is the package file. The user usually tends to commit a lot of mistakes while including the files in the package leading to missing files or including the files in wrong order. Figure 19 shows the auto generated code snippet of the package file. The tool knows the entire list of files that is being created

and knows the exact order in which the files are supposed to be included starting from interface, configuration objects, sequence items, sequences, leaf level components containing agents and its counterparts, environment and top-level generic test for compilation and simulation to validate the developed codes.

```
1 interface generic_interface (input wire driveclock) ;
2  logic [63:0] addr [31:0] [63:0];
3  reg [31:0] data [15:0] [1023:0];
4  reg enable;
5
6  clocking mon_cb@(posedge driveclock);
7   input       addr;
8   input       data;
9   input       enable;
10 endclocking: mon_cb
11
12 clocking drv_cb@(negedge driveclock);
13  output      addr;
14  output      data;
15  output      enable;
16 endclocking: drv_cb
17
18 modport mon_mp(
19  input driveclock, clocking mon_cb
20 );
21
22 modport drv_mp(
23  input driveclock, clocking drv_cb
24 );
25 endinterface: generic_interface
```

Figure 18. Interface code snippet

```
1 package generic_env_package;
2  import uvm_pkg::*;
3 `include "uvm_macros.svh"
4
5  import generic_seq_item_package::*;
6
7 `include "slave_agent_config.sv"
8 `include "master_agent_config.sv"
9
10 `include "generic_environment_config.sv"
11
12 `include "slave_monitor.sv"
13 `include "master_monitor.sv"
14
15 `include "slave_driver.sv"
16 `include "slave_sequencer.sv"
17 `include "master_driver.sv"
18 `include "master_sequencer.sv"
19
20 `include "slave_agent.sv"
21 `include "master_agent.sv"
22
23 `include "mon_monitor.sv"
24 `include "scb_scoreboard.sv"
25
26 `include "generic_environment.sv"
27 endpackage: generic_env_package
```

Figure 19. Auto generated package code

IV. Performance Evaluation

Below Table 1 shows the comparison between 'Novel GUI Based UVM testbench template builder' and other open source UVM code generators.

TABLE I

COMPARISON BETWEEN THIS TEMPLATE BUILDER AND OTHER OPENSOURCE TOOLS

Comparison Points	Easier UVM Code Generator [4]	Open Titan UVM Generator [5]	Novel GUI Based UVM testbench Template Builder [1]
License	Open source	Open source	Open source
Support GUI	No	No	Yes
Generation of UVM class code	Yes	Yes	Yes
Generation of complete UVM environment	No	Yes	Yes
Generation of multi-instance of agents, monitors, environments, etc. for complex testbench	No	No	Yes
Smart monitor and scoreboard connectivity	No	No	Yes
Incremental testbench development	No	No	Yes
Environment Integration	No	No	Yes
Open-source documentation formatting [3]	No	No	Yes

V. Summary

The UVM template generator provides the user to create any component template or the entire VIP dynamically in matter of minutes. The generator helps in standardization of code development and re-usability of the code across the projects and helps in complete integration of the verification collateral. With the template generator's unique 'Create & Stitch' feature, the tool can add new components, add connection between components or append sub environment VIPs to the already existing code, hence helping in incremental enhancement of the testbench. The tool helps in massive (99.88%) time reduction in bringing up the initial UVM VIP template development cycle [0.0083Hrs/30Sec], which is compile clean, properly commented, and ready to use as compared to the code developed by an engineer [8Hrs/1Bussiness day]. Hence, this tool indeed improved verification productivity and showcased its performance in complex streamlined products.

References

[1] Tool documentation: https://github.com/hellovimo/uvm_testbench_gen/wiki/The-Novel-GUI-Based-UVM-Template-Generator

[2] GitHub repository for tool open-source code: https://github.com/hellovimo/uvm_testbench_gen

[3] Natural Docs: https://www.naturaldocs.org/

[4] Doulos 'Easier UVM Code Generator': https://www.doulos.com/knowhow/systemverilog/uvm/easier-uvm/easier-uvm-code-generator/

[5] Open Titan Utilities: https://docs.opentitan.org/util/uvmdvgen/README/