

UVMF App Note

Refactoring and Reusing Code Using the Generator

Jonathan Craft – Senior AE Consultant – October 2018

Introduction

When starting from a blank slate, if one follows these steps, the UVMF code generator can easily be used to produce UVMF interface, environment and bench code:

1. Write YAML configuration for all desired structures
2. Invoke the UVMF code generator using the configuration files
3. Test generated code with a dry-run before making edits
4. Make necessary manual edits. Searching for the term “UVMF_CHANGE_ME” in generated code can be used as a guide to where changes may be required:

```
// UVMF_CHANGE_ME : Implement response protocol signaling.
// templates also do not generate protocol specific response signaling. Use the
// following as examples for transferring data between a sequence and the bus
// In wb_pkg - wb_memory_slave_sequence.svh, wb_driver_bfm.sv

task do_response(                                output bit [DATA_WIDTH-1:0] data ,
          output bit [DATA_WIDTH-1:0] dst_address ,
          output bit [STATUS_WIDTH-1:0] status   );
    @(posedge pclk_i);
    @(posedge pclk_i);
    @(posedge pclk_i);
    @(posedge pclk_i);
    @(posedge pclk_i);
endtask
```

The following list is not comprehensive but should provide a general idea of what will need to be done:

- a. Interfaces
 - i. do_transfer task in driver BFM
 - ii. do_monitor task in monitor BFM
- b. Environments
 - i. Fill in details of write functions in predictors
 - ii. Add utility sequences
- c. Benches –
 - i. Instantiate DUT
 - ii. Define new sequences and tests that invoke them

Once you have started making edits to generated code, however, changes to the structure that was in place originally may need to be modified. In many cases it will be easier to modify the YAML configuration files and regenerate the desired component's files than apply those same changes to the generated output by hand. Some common reasons for taking these actions might include:

- Many of these types of changes will precipitate throughout an entire bench. For example, changes to the port list of an interface will not only impact that interface definition but also the way that interface is instantiated and connected at the bench level.

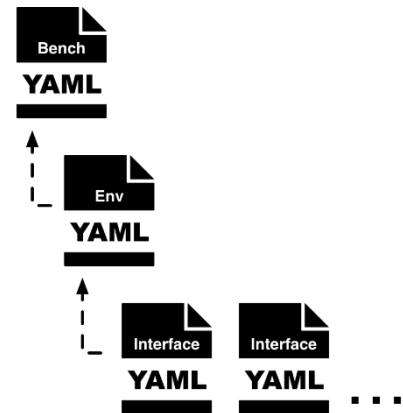


This document describes some strategies to minimize the amount of repeated work when the need arises to re-run the generator against code that has already seen manual edits.

Script Behavior

When running the generator script one is required to provide all of the YAML for the desired top-level. For example, when generating a new UVMF bench the user must supply YAML descriptions for the bench itself along with the bench's top-level environment and all sub-components underneath that environment. This is true even if only the bench needs to be regenerated.

By default, the generator script will produce code for all YAML configuration that was provided. Due to the requirement discussed above, this can be troublesome if only a bench needs to be regenerated but all code underneath the *verification_ip* directory can remain unchanged.



The default behavior can be modified with a command-line switch. Use the `-g <component_name>` or `--generate=<component_name>` switch to instruct the script to only produce code associated with the given component.

Note: While it is legal to use the same component name for an interface, environment and bench that will mean that the `--generate` switch will not be able to differentiate between them. Any component whose name matches what is passed in with the switch will be produced.

Another default behavior worth noting is that for safety, no new code will be produced if it will overwrite existing code.

```
[jcraft@localhost] 469> yaml2uvmf.py ./yaml_files/mem_if_cfg.yaml
Generating /work/framework/templates/python/examples/uvmf_template_out/
Generating /work/framework/templates/python/examples/uvmf_template_out/
Generating /work/framework/templates/python/examples/uvmf_template_out/
Generating /work/framework/templates/python/examples/uvmf_template_out/
Generating /work/framework/templates/python/examples/uvmf_template_out/
Generating /work/framework/templates/python/examples/uvmf_template_out/
```

Figure 2: Generation of New Files - Nothing is overwritten by default

Messages indicating that the production of a given file is being “Skipped” will be displayed instead of messages indicating a file is being “Generated”. This behavior can also be modified via the `-o` or `--overwrite` switch, which will force all files to be regenerated regardless of existing files. Obviously, this can be dangerous so use with caution.

```
[icraft@localhost] 470> yam12uvmf.py ./yaml_files/mem_if_cfg.yaml
Skipping /work/framework/templates/python/examples/uvmf_template_ou
Skipping /work/framework/templates/python/examples/uvmf_template_ou
Skipping /work/framework/templates/python/examples/uvmf_template_ou
Skipping /work/framework/templates/python/examples/uvmf_template_ou
Skipping /work/framework/templates/python/examples/uvmf_template_ou
Skipping /work/framework/templates/python/examples/uvmf_template_ou
Skipping /work/framework/templates/python/examples/uvmf_template_ou
Skipping /work/framework/templates/python/examples/uvmf_template_ou
```

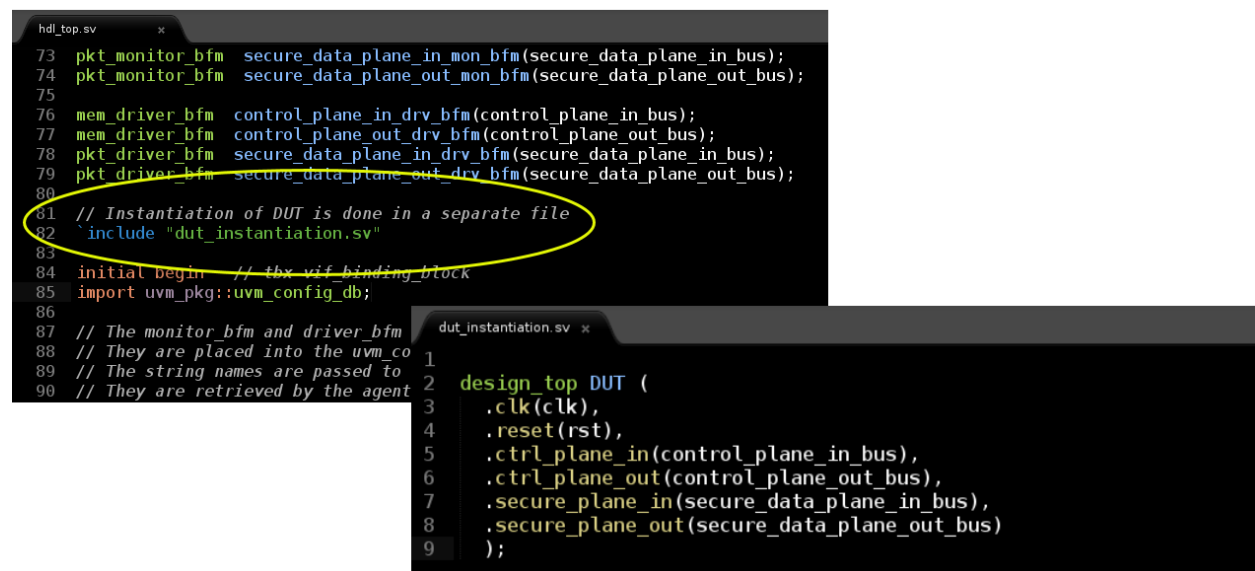
Figure 3: Skipping Existing Files

General Suggestions

The following items will likely be useful regardless of which strategy you choose to employ when regenerating or reusing code

DUT Instantiation – Use `include file

One of the key items that must be done manually after generating a bench is to instantiate the DUT underneath the `hdl_top` module. Instead of directly instantiating and connecting the DUT within `hdl_top` it can help to keep that code in a separate file that is included in the appropriate place within the `hdl_top.sv` file. This way, if the bench requires regeneration only a single `include line will need to be manually added to the output instead of the entire instantiation.



```
hdl_top.sv
73 pkt_monitor_bfm secure_data_plane_in_mon_bfm(secure_data_plane_in_bus);
74 pkt_monitor_bfm secure_data_plane_out_mon_bfm(secure_data_plane_out_bus);
75
76 mem_driver_bfm control_plane_in_drv_bfm(control_plane_in_bus);
77 mem_driver_bfm control_plane_out_drv_bfm(control_plane_out_bus);
78 pkt_driver_bfm secure_data_plane_in_drv_bfm(secure_data_plane_in_bus);
79 pkt_driver_bfm secure_data_plane_out_drv_bfm(secure_data_plane_out_bus);
80
81 // Instantiation of DUT is done in a separate file
82 `include "dut_instantiation.sv"
83
84 initial begin // tbx vif binding block
85 import uvm_pkg::uvm_config_db;
86
87 // The monitor bfm and driver bfm
88 // They are placed into the uvm_co
89 // The string names are passed to
90 // They are retrieved by the agent

dut_instantiation.sv
1
2 design_top DUT (
3 .clk(clk),
4 .reset(rst),
5 .ctrl_plane_in(control_plane_in_bus),
6 .ctrl_plane_out(control_plane_out_bus),
7 .secure_plane_in(secure_data_plane_in_bus),
8 .secure_plane_out(secure_data_plane_out_bus)
9 );
```

Figure 4: Use separate file to instantiate the DUT

Save Old Output

As stated earlier, the code generator's default behavior is to skip files if they already exist. Therefore, in order to regenerate something the user will need to move or delete the existing code or use the `-o` switch to force the script to overwrite existing files. All of these options could result in significant loss of

time and effort if the old code is lost and it had many manual changes applied. For this reason it is strongly recommended to copy the entire directory structure that was generated previously to a safe location before regenerating or use a version control system to preserve earlier changes. This way, any accidental overwrites can be recovered and the old files can be used as reference when reapplying manual changes.

Regeneration Techniques

Consider the following methods when regenerating a part of an existing test bench.

Focused Regeneration

The `-g` or `--generate` switch can be used to produce a specific component within a larger bench. The `-o` or `--overwrite` switch must also be used in order to force the new files to overwrite the old. As suggested above, it would be good practice to save a copy of what will be overwritten before regeneration.

Delete or Move Existing Code

By moving only the code associated with the component that you wish to regenerate but leaving everything else in place you can use the default behavior of the script to your advantage. All of the remaining code will have its code generation skipped but the missing code will be generated.

```
interface_packages
|-- wb_pkg.sv
|-- wb_pkg_hdl.sv
|-- src
    |-- wb_configuration.svh
    |-- wb_driver.svh
    |-- wb_driver_bfm.sv
    |-- wb_if.sv
    |-- wb_monitor.svh
    |-- wb_monitor_bfm.sv
    |-- wb_sequence_lib.svh
    |-- wb_transaction.svh
    |-- wb_transaction_coverage.svh
    |-- wb_typedefs.svh
    |-- wb_typedefs_hdl.svh
    |-- reg2wb_adapter.svh
```

Limited Configuration

If the desired changes are limited to an interface it may be possible to only provide that interface's YAML configuration as input to the generator, along with a `-o` switch to overwrite existing code. Care must be taken, however, as many changes to an interface will have ripple effects to environments and benches that involve that interface, so it is quite likely that this approach will not work effectively.

Use a Graphical Merge Tool

Utilities such as BeyondCompare, Meld, or TkDiff can be used to view the differences between two files and resolve them into a single file. When reincorporating earlier changes into a newly regenerated file this approach can be quick and convenient.



Figure 5: Graphical Merge Tool

Note: By default, there is timestamp information in the header for all generated output that supports comments which could trigger a diff tool to highlight all files as being different. Some diff tools allow you to specify certain file content patterns to be ignored during analysis which can help to reduce this type of noise.

Reuse Techniques

In addition to making changes to an existing component during the course of normal development it may be desirable to simply reuse the code associated with a component when developing a different bench. This will likely occur both when developing a higher-level bench that incorporates block-level structures (environments, interfaces) as well as when developing a completely different bench that uses some of the same protocols as an earlier environment. In all of these situations you will likely have a mixture of new code to generate that leverages existing interfaces and environment code.

As stated earlier, any invocation of the generator requires that all YAML configuration be provided to the script from the top-level down. If reusing existing code this will often mean that you will need to provide YAML definitions for components that are already complete and ready for use.

The techniques described below will look similar to those described earlier but as a mirror image. Instead of regenerating perhaps one or two components across an entire environment or bench you may find yourself generating the lion's share of the code while avoiding some key areas being reused.

Note: All code that is considered reusable should reside in a `verification_ip` directory. While it is possible to define multiple `verification_ip` locations for the sake of simplicity this document assumes a single area. The same concepts can be applied to multiple locations.

Leverage Default Behavior

Even though you will have to provide accurate YAML definitions for code that is being reused from another bench, the default generator behavior of skipping existing code will allow you to run the generator against a mixture of new and existing YAML but only generate the new code. If going this route do not use the `--overwrite` switch.

Overwrite New with Old

If starting from an empty directory you can generate all new code and simply copy the reusable component code over top of what the generator produced.