Generating VMM Compliant Environments with SystemVerilog FrameWorks™ Template Generator (SVF-TG)

Revision 1.0

ABSTRACT

A template-based utility, SystemVerilog FrameWorks™ Template Generator (SVF-TG) has been developed to assist in creating and maintaining VMM based environments. Using this tool, teams can create customized templates of verification components such as transactors, environment classes, and the default directory structure for a given project. SVF-TG promotes automated creation of a customized VMM-based environment, enforces a consistent look and feel, and enables rapid development and maintenance of the verification code across multiple sites and cultural barriers.

This paper describes the main features of SVF-TG and the various templates that create a complete VMM environment. By typing a command line, the user can create a customized shell of an entire environment that compiles out of the box using VCS. Later, as the project requirements evolve, the team leaders can customize the templates as needed and then automatically merge the resulting output with existing code.

Table of Contents

List of Figures

1. Introduction

A typical verification team today spans multiple sites, and often contains code from various sources. Creation and maintenance of such verification environments is a software engineering challenge. Approaches such as VMM and languages such as SystemVerilog help address this challenge by providing common methodology guidelines and powerful language features such as object-oriented programming.

Applying VMM guidelines to the verification code establishes a common vocabulary among verification engineers. However, a project has its own conventions and choices on top of the recommended VMM guidelines. Unfortunately, such guidelines are hard to communicate, maintain, and often easy to misinterpret. At best, implementing these guidelines is a manual and often a highly error prone process. Even harder is to change the guidelines once a team has already started coding. Imagine the team deciding to change its class naming convention after realizing it conflicts with the naming convention of an external vendor IP.

To accelerate the creation and simplify the maintenance of advanced verification environments, we have developed a tool, SystemVerilog FrameWorks™[1]. This is essentially a toolkit for the verification engineer that assists her in creating a verification environment from scratch and maintaining it, including providing verification templates that follow VMM guidelines. In addition, it helps in generating customizable functional coverage reports, provides utilities such as basic scoreboarding, shutdown managers, register handling, etc.

In this paper, we focus on the first component, namely, the template-generator named SystemVerilog FrameWorks™ Template Generator (SVF-TG). While SVF-TG is not restricted to generate only VMM environments, it provides a default implementation for generating a customizable VMM-based shell environment that compiles out of the box using VCS. In summary, SVF-TG allows a verification team to

a. Define conventions on how to write the various verification environment components and hook them up.
b. Capture these conventions and successful practices in an executable form.
c. Maintain changes over time, including merging of output from newer guidelines into existing code.

The guidelines are captured in a set of template files for each kind of verification component such as driver, monitor, environment, configuration, and so on. These templates are completely customizable. Once the environment shell is generated, the verification engineer can add the application-specific verification code.

Figure 1 shows the basic operation of SVF-TG. We welcome the reader to generate a customized VMM environment shell using SVF-TG.

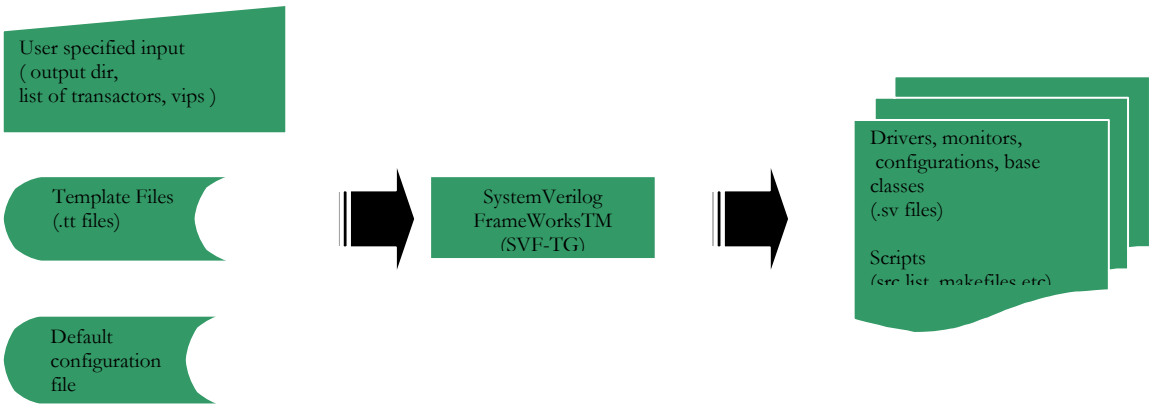

Figure 1. SystemVerilog FrameWorks™ (SVF-TG) tool

2. Challenges of creating and maintaining VMM based verification environments

We identify here a few of the challenges in creating and maintaining a VMM based environment.

Overcoming initial learning curve
Given the numerous guidelines, it is often daunting for even experienced verification engineers to adopt all aspects of VMM. By having a reference environment shell, people can start running in no time. A tool such as SVF-TG can help by generating a sample code and build scripts organized in a reasonable directory structure, thus allowing the team to hit the ground running.

Working with teams at mixed skill-levels
Not every verification engineer is skilled or interested in developing verification components based on object-oriented concepts. However, by creating the environment from scratch automatically, SVF-TG enables engineers to start coding the application specific functions without necessarily being familiar with the advanced testbench features of SystemVerilog.

Following company specific requirements
In addition to the VMM recommended guidelines, teams often decide upon company specific ones, including exceptions to VMM. As an example, the team may decide to add an input latch on each header file to avoid compilation problems. SVF-TG can automate this process significantly, including automatically creating names for the macros. See Figure 2, where the copyright notice and the adding of the SNUG_AXI_IF macro as an input latch are examples of typical company conventions.

```
//-------------------------------------------------------------------------
//
//                Confidential and Proprietary Information
//
// The information and descriptions contained herein embody confidential and
// proprietary information which is the property of Paradigm Works, Inc. Such
// information and descriptions may not be copied, reproduced, disclosed to
// others, published or used, in whole or in part, for any purpose other than
// that for which it is being made available without the express prior written
// permission of Paradigm Works, Inc.
//
// Paradigm Works, Inc.,                Copyright (c) 2001-2006
//-------------------------------------------------------------------------
/*************************************************************************
 * Author:              $Author:$
 * File:                $File:$
 * Revision:            $Revision:$
 * Date:                $Date:$
 *************************************************************************
 * Interface file axi_if.sv for vip axi in snug verification environment
 *************************************************************************/
`ifndef SNUG_AXI_IF
`define SNUG_AXI_IF

interface axi_if(input wire clk);
  logic [15:0] rdata;
  logic [15:0] wdata;
  logic write;

  // Add your own signals

  // For the DUT
  modport DUT( input wdata, output rdata, input write);

  // For the testbench
  modport TB( input rdata, output wdata, output write);

endinterface: axi  if
```

Figure 2. Example of enforcing project convention using SVF-TG

Supporting common practice

To improve the communication among various teams, VMM offers a common vocabulary and common base classes such as vmm_xactor. However, not everything in practice can be captured using common base classes. For example, a team may decide to name of methods, key variables, instances of various components etc by using the name of the VIP as a prefix. Even trickier is to capture the information on how to connect these components together.

Avoiding evils of cut and paste

When following the guidelines, one often takes existing examples, copies them over, and modifies them. Ignoring the tedium involved in processing a potentially large number files, it is still an error-prone manual process. Once implemented, users cannot easily modify these guidelines as the affected code is potentially scattered all over and hard to locate. SVF-TG helps in avoiding a majority of such errors by automating many of the steps involved.

Communicating guidelines and conventions across multiple sites
Communicating the guidelines to team members can be difficult, especially when done across multiple sites and to people at different skill levels and cultural backgrounds. If not communicated accurately, it is likely that the team will not adhere to these guidelines. However, if the team leaders can capture these guidelines in an executable form, others can very easily generate the code as recommended, thus increasing the odds of adherence.

Capturing changes and applying lessons learned from previous projects
If captured in an executable form, these guidelines can be applied to existing or new projects. For example, team leaders can create or modify templates based on a pilot project. SVF-TG can then be used to generate new code that follows the guidelines captured in these templates and merge with existing code automatically.

3. SystemVerilog FrameWorks™ (SVF-TG)

The following sections describe specific features that we found helpful in creating company-wide verification environments, across multiple sites and groups. First, we describe a typical verification environment example. Next, we describe various components generated by SVF-TG, including the directory structure, transactors, top-level environment objects. We conclude this section by describing the process of customizing the templates and maintaining these environments even when some of the underlying guidelines change.

3.1 Example verification environment

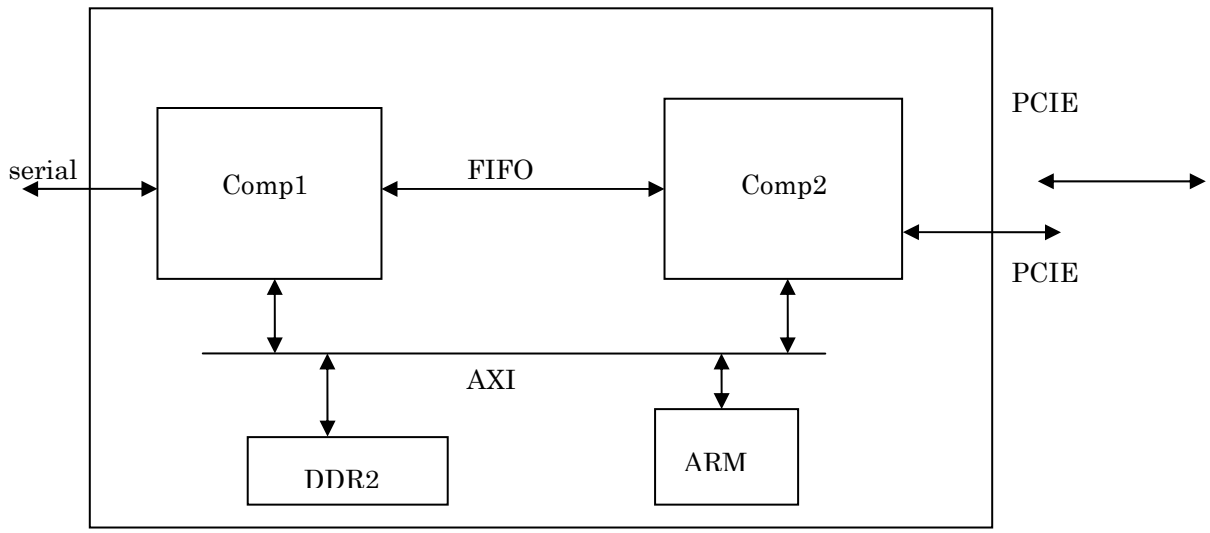

Figure 3. A Typical SOC Application

As an example, consider a typical application as shown in Figure 3. The DUT considered is an optical networking termination (ONT) device. There are two PCIE ports in the downstream direction, and one serial port in the upstream direction. There is a host interface using AMBA-AXI

protocol. There are some DDR2 ports for memory access. An internal FIFO interface is used to connect the internal components comp1, comp2.

Emulating a typical verification project based on VMM, consider that there are two verification environment classes derived from *vmm_env*. Assume *comp1* environment addresses unit-level verification for the comp1 component, while the *sys* environment verifies the entire SOC.

3.2 Creating VMM environment using SVF-TG

```
Description:
        The svfInstallVmmDirs tool takes as input a set of templates and will generate
        a SystemVerilog VMM directory structure and test environment.

Options:
  -env              -env env1 -env sys
  -help             Print this help
  -installDir       Target installation directory
  -newttdir          Directory of the new template directory
  -projName         Name of project
  -testGroup        -te "<env name> <group name> <testName> <testName>"
                    <envName> = #any# means all environments have these tests
  -ttdir            Template directory
  -version          Print version information and exit.
  -vip              -vip vipA -vip vip2[n] ...
```

Figure 4. Command line options of SVF-TG (svfVmmInstallDirs)

Figure 4 shows the command line options of SVF-TG. The name of the executable is *svfVmmInstallDirs*. The tool provides options for specifying which environments need to be generated, as well as their associated VIPs and tests. The following sections show how these command line options are used.

3.2.1 Directory structure generation

```
svfInstallVmmDirs                          # Name of the executable
        -p snug                            # Name of project
        -vip "pcie[2]"                     # 2 instances of the PCI-E vip
        -vip ser                           # one instance of ser vip
        -vip fifo                          # one instance of the fifo vip
-       -vip axi                           # one instance of the axi interface
        -env comp1                         # comp1 environment
        -te "comp1 grp1 t1 t2"             # comp1 environment has tests t1, t2 in test group grp1
        -env sys                           # sys environment
        -te "sys sys_comp1 sys_c1_test"    # sys environment has test sv_c1_test in test group sys_comp1
        -te "sys sys sys_test"             # sys environment has test sys_test in test_group sys
        -te "#all# basic csr_test reset_test"          # All envs have tests csr_test and reset_test in the basic test group
```

Figure 5. An example command line invocation of SVF-TG

Figure 5 shows a command line invocation of the VMM compliant directory structure for the example in Figure 1.

Two main verification environments, namely *comp1* and *sys,* are being created for the project *snug*, as specified by the switches *–env* and *–p* respectively. The VIPs needed for this project are specified by the *–vip* switches. For example, *-vip "pcie[2]" –vip ser* implies that two instances of the *pcie* vip and one instance of the *ser* vip are needed. One can specify some tests for each environment using the *–te* switch. For example, the option *-te "comp1 grp1 t1 t2"* defines tests *t1* and *t2* that are grouped under *grp1* directory for the *comp1* environment. Tests common to all environments can also be specified using the keyword *#all#*. For example, *-te "#all# basic csr_test reset_test"* generates the test group *basic* with tests *csr_test* and *reset_test* for all environments.

Executing this command line generates an entire VMM–based environment for the given example. The tool also generates a directory structure report, portions of which are shown in Figure 6.

```
DIRECTORY STRUCTURE DESCRIPTION
Automatically generated by SVF-TG
============================
SNUG/
  verif/                    Top verification directory
    bin/                    Scripts for this project
    doc/                    Verification documentation
    common/                 Common directory used by the entire project
      doc/                          Documentation for the common dir
      include/                      Common include directory used by the entire project
        snug_defines.sv                     SNUG project specific definitions
      src/                  SNUG project common source file directory
        snug_log.sv                 SNUG base class extended from vmm_data
        snug_data.sv                SNUG base  class extended from vmm_data
        snug_scoreboard.sv          SNUG base scoreboard class extended from vmm_xactor
        snug_notify.sv              SNUG base class extended from vmm_notify
        snug_xactor.sv              SNUG base class extended from vmm_xactor
        snug_env.sv                 SNUG base class derived from vmm_env
        snug_shutdown_mgr.sv        SNUG standalone class derived from pw_svf_shutdown_mgr class
        src.list                    Build list for common dir
    vips/                   Directory containing all external and internal Verification IPs (VIPs) for project
      pcie/                         Distribution for VIP pcie
        …
      ser/                          Distribution for VIP ser
        …
    comp1/                  Top-level directory for comp1 verification environment
      env/                          Verification environment files
        …
      tests/                        Top level tests for this environment. This contains multiple test groups.
        grp1/                               Group of tests categorized under grp1 for comp1 environment
        …
    sys/                            Top-level directory for sys verification environment.
```

Figure 6 Directory structure report (partial) automatically generated by SVF-TG

This automatically generated directory structure reports the details of each class, and its location with respect to the top of the hierarchy. Note that while VMM does not explicitly state any specific directory structure, the above organization scheme is derived from RVM's specification[5], and is a reasonable way to organize the code. For example, the directory *SNUG/verify/common/src* contains all the business unit-specific classes, as well as any other project specific class such as the *snug_scoreboard* and *snug_shutdown_manager*. All the vips go into the *vips* directory, and all the environments and their associated tests go to the directories with the same name under *verif* directory. Note that the entire generated environment shell compiles out of the box, and teams can

start adding their application specific code immediately, with the entire infrastructure already taken care of.

The following sections describe some more details on the generated vip and the env templates.

```
...
vips/                    Directory containing all external and internal Verification IPs (VIPs) for project
    pcie/                    Distribution for VIP pcie
        doc/                     Documentation for pcie VIP
        include/                 Include directory for pcie VIP
            pcie_defines.sv          pcie specific defines
            pcie_if.sv               pcie interface definitions
        src/                     Source files for pcie VIP
            pcie.sv                  Main file for pcie
            pcie_cfg.sv              Configuration file for pcie
            pcie_data.sv             Data class for pcie extended from snug_data
            pcie_checker.sv          Checker class for pcie extended from snug_xactor
            pcie_gen.sv              Generator class for pcie extended from snug_data_atomic_gen
            pcie_drv.sv              Driver class for pcie extended from snug_xactor
            pcie_mon.sv              Monitor class for pcie extended from snug_xactor
            pcie_xtr.sv              Main transactor class for pcie extended from snug_xactor
            pcie_sc_gen.sv           Scenario generator file for pcie extended from
vmm_scenario_gen
```

Figure 7 Vip directory generated by SVF-TG

3.2.2 VIP generation

Figure 7 shows a portion of the directory structure report that describes the source file templates generated for the pcie VIP. In addition to the placeholders for common definitions in the include directory, it creates a bunch of shell files for the various VIP components in the *src* directory, including a *src.list* file for easy integration with build scripts. We now briefly describe some of the code generated by the tool based on the default templates provided.

```
//-------------------------------------------------------------------------
//
//                Confidential and Proprietary Information
//
<deleted>
//
// Paradigm Works, Inc.,                Copyright (c) 2001-2006
//-------------------------------------------------------------------------
/*************************************************************************
 * Author:              $Author:$
 * File:                $File:$
 * Revision:            $Revision:$
 * Date:                $Date:$
 *************************************************************************
 * Interface file pcie_if.sv for vip pcie in snug verification environment
 *************************************************************************/
`ifndef SNUG_PCIE_IF
`define SNUG_PCIE_IF

interface pcie_if(input wire clk);
  logic [15:0] rdata;
  logic [15:0] wdata;
  logic write;

  // Add your own signals

  // For the DUT
  modport DUT( input wdata, output rdata, input write);

  // For the testbench
  modport TB( input rdata, output wdata, output write);

endinterface: pcie_if
```

Figure 8 pcie_if.sv generated by SVF-TG

Figure 8 is a simple shell file generated as a placeholder for the interface file for the pcie VIP. While the generated code is relatively simple, it demonstrates several customizations. First, the copyright notices are added automatically. Second, source control specific tags are added, as well as some default comments. Finally, input latches using compile time constants such as *SNUG_PCIE_IF* are created automatically so that the include files are not multiply compiled even when included multiple times in the code.

Figure 9 shows the low-level driver code generated for the pcie VIP. The user is expected to add the application specific code under the sections marked *// Add your own*. Note also that the constructor

arguments are automatically defined based on the name of the VIP *pcie*.

```
….
/********************************************************************************
 * Top level driver file pcie_drv.sv for snug verification environment
 ********************************************************************************/
`include "vmm.sv"

typedef class pcie_drv;
class pcie_drv_callbacks extends vmm_xactor_callbacks;
  virtual task pre_drv(pcie_drv drv,  pcie_data tr);
  endtask // pre_drv
  virtual task post_drv(pcie_drv drv, pcie_data tr);
  endtask // post_drv
endclass

class pcie_drv extends snug_xactor;
  pcie_cfg  cfg;
  pcie_data_channel in_chan;

  function new(string instance, int stream_id=-1, pcie_cfg cfg=null,   pcie_data_channel in_chan=null);
            …deleted for clarity….
  endfunction // new

  protected virtual task main();
    begin
      // Common Code
      fork
            super.main();
      join_none

      // Add your own

     // Commom code
      while(1) begin
            super.wait_if_stopped();
            transmit_t();
      end
    end
  endtask // main

  task transmit_t();
    pcie_data      tr=new;
    begin
      // Common code
      if (in_chan.level()==0) begin
            notify.indicate(XACTOR_IDLE);  notify.reset(XACTOR_BUSY);
      end
      in_chan.get(tr);
      notify.indicate(XACTOR_BUSY);  notify.reset(XACTOR_IDLE);

      `vmm_callback(pcie_drv_callbacks,pre_drv(this,tr));

      // Add your own, Do the actual driving

      // Common code
```

Figure 9. pcie_driver.sv generated by SVF-TG

```
// pcie data generator
`include "vmm.sv"

typedef class pcie_gen;
class pcie_gen_callbacks extends vmm_xactor_callbacks;
  virtual task pre_inst_gen(pcie_gen gen);
  endtask:pre_inst_gen
  virtual task post_inst_gen(pcie_gen gen, pcie_data tr, ref bit drop);
  endtask:post_inst_gen
endclass

class pcie_gen extends vmm_xactor;
  pcie_gen_callbacks          gen_cb;
  pcie_cfg                    cfg;
  int                         object_id;
  pcie_data_channel           out_chan;
  pcie_data                   randomized_obj;

  function new(string instance, int stream_id = 1, pcie_cfg cfg=null,  pcie_data_channel out_chan=null);
    …
  endfunction // new

  task main();
   fork
        super.main();
   join_none

   // raise objection to shutdown manager
   while (cfg.stop_after_n_insts>0 && object_id<cfg.stop_after_n_insts) begin
        bit drop;
        pcie_data tr;

        `vmm_callback(pcie_gen_callbacks,pre_inst_gen(this));

        // randomize blueprint
        if (!randomized_obj.randomize())
          `vmm_error(log,"Failed to randomize blueprint");

        if (log.start_msg(vmm_log::DEBUG_TYP,vmm_log::DEBUG_SEV))
          randomized_obj.display("randomized_obj");

        if (!$cast(tr,randomized_obj.copy()))
          `vmm_error(log,"Failed to cast randomized_obj to pcie_data");

        `vmm_callback(pcie_gen_callbacks,post_inst_gen(this,tr,drop));
        object_id++;
```

Figure 10 pcie_gen.sv generated by SVF-TG

Next, Figure 10 shows how a team can define its own atomic generator instead of the one defined using *`vmm_atomic_gen*. In this example, it is shown how the generator uses the various *callback* and the *snug_shutdown_manager* classes. One can thus ensure that all team members develop their specific atomic generators in the recommended manner.

3.2.3 Generation of the top level environment

```
sys/                    Top-level directory for sys verification environment. Other environments can exist in parallel
    env/                    Verification environment files
        sys_env.sv          Testbench environment implementation that hooks up various components for this environment
        top.sv              top-level testbench for current environment
        env.list                    Build list for the entire sys environment
        sys_dut_cfg.sv              Dut configuration descriptor for sys environment
        sys_reg_map.sv              Register Map for current environment
        sys_env_cfg.sv              Configuration descriptor for sys environment, including DUT configuration
    tests/              Top level tests for this environment. This contains multiple test groups.
        sys_comp1/              Group of tests categorized under sys_comp1 for sys environment
            sys_c1_test/            Directory for test sys_c1_test in group sys_comp1 for sys environment
                sys_c1_test.sv              test file for test group sys_comp1 in sys environment
            logs/                   Simulation output log files for test group sys_comp1 in sys environment
            cvr/                    Functional coverage databases for test group sys_comp1 in sys environment
        sys/                    Group of tests categorized under sys for sys environment
            sys_test/               Directory for test sys_test in group sys for sys environment
                sys_test.sv             test file for test group sys in sys environment
            logs/                   Simulation output log files for test group sys in sys environment
            cvr/                    Functional coverage databases for test group sys in sys environment
        basic/                  Group of tests categorized under basic for sys environment
            csr_test/               Directory for test csr_test in group basic for sys environment
                csr_test.sv             test file for test group basic in sys environment
            reset_test/             Directory for test reset_test in group basic for sys environment
                reset_test.sv           test file for test group basic in sys environment
            logs/               Simulation output log files for test group basic in sys environment
            cvr/                Functional coverage databases for test group basic in sys environment
```

Figure 11 Directory structure generated for sys environment by SVF-TG

Figure 11 shows the files that are generated for the system-level environment, including the sample test files as well as directories for dumping logs and coverage data. The corresponding class extended from *vmm_env* is defined in *sys_env.sv*, while the tests are in the *sys/tests* directory by default. Note that both generic tests that are common to all environments have a placeholder in the *sys/tests/basic* directory hierarchy, while *sys* specific tests are in *sys/tests/sys* directory hierarchy.

We now describe the generated code for the *sys_env* class. This class is derived from the *vmm_env* base class and represents the top-level testbench object for the *sys* environment. Portions of the generated code are shown in Figure 12 and are explained further. First, note that all the relevant components are instantiated, assembled and hooked up by default. Second, the tool is able to handle multiple instances of the same VIP and defines them in an array as defined in the template file. By automatically naming, declaring, and hooking up these objects, and calling their relevant methods such as *start_xactor* throughout the class definition as stipulated by VMM, the tool saves a lot of typing and potentially avoids mistakes and omissions inherent in a manual process.

```
`include "pcie_if.sv"
`include "ser_if.sv"
`include "fifo_if.sv"
`include "axi_if.sv"
…
class PW_sys_env extends snug_env;
…
  static vmm_log       log = new ("SNUG","ENV");
  sys_env_cfg                cfg;
  sys_reg_map          reg_map;
  snug_scoreboard            sys_sb;

  // Various vips
  pcie_xtr                   pcie_inst[2];
  virtual pcie_if.TB   pcie_port[2];
  ser_xtr                    ser_ins;
  virtual ser_if.TB          ser_port;
  fifo_xtr                   fifo_inst;
  virtual fifo_if.TB   fifo_port;
  axi_xtr                    axi_inst;
  virtual axi_if.TB    axi_port;

…
            // ****************************** Build the xactors *************************
            cfg.pcie_cfg[1].pcie_port = this.pcie_port[1];
            pcie_inst[1] = new("sys_pcie_xactor 1",  "SYS_PCIE_1", 0,  cfg.pcie_cfg[1]);

            cfg.pcie_cfg[0].pcie_port = this.pcie_port[0];
            pcie_inst[0] = new("sys_pcie_xactor 0",  "SYS_PCIE_0", 0, cfg.pcie_cfg[0]);

            cfg.ser_cfg.ser_port = this.ser_port;
            ser_inst = new("sys_ser_xactor", "SYS_SER", 0,  cfg.ser_cfg);

            cfg.fifo_cfg.fifo_port = this.fifo_port;
            fifo_inst = new("sys_fifo_xactor", "SYS_FIFO",  0, cfg.fifo_cfg);

            cfg.axi_cfg.axi_port = this.axi_port;
            axi_inst = new("sys_axi_xactor", "SYS_AXI", 0, cfg.axi_cfg);
…

  // Start up the rest of the transactors
  fork
    begin
          `vmm_debug(this.log,"Started transactor pcie[0]...");
      pcie_inst[0].start_xactor();
          `vmm_debug(this.log,"Started transactor pcie[1]...");
      pcie_inst[1].start_xactor();
          `vmm_debug(this.log,"Started transactor ser...");
      ser_inst.start_xactor();
          `vmm_debug(this.log,"Started transactor fifo...");
      fifo_inst.start_xactor();
          `vmm_debug(this.log,"Started transactor axi...");
```

Figure 12 sys_env.sv generated by SVF-TG

3.2.4 Predefined templates

The SVF-TG uses a set of template files that act as the boilerplate for generating the code. Figure 13 shows a list of such templates. The templates are written using the template-toolkit format[3], which is widely used and is highly customizable.

Svf_pw_header.tt	Project specific header files included in top of each file
Svf_proj_defines_sv.tt	Project specific defines
Svf_proj_notify_sv.tt	Project specific notify class
Svf_proj_scoreboard_sv.tt	Project base scoreboard class
Svf_proj_xactor_sv.tt	Project base transactor class
Svf_proj_cfg_sv.tt	Project DUT configuration class
Svf_proj_env_cfg_sv.tt	Project configuration class, including DUT and environment
Svf_proj_regmap_sv.tt	Project register map class
Svf_proj_data_sv.tt	Project base data class
Svf_proj_env_sv.tt	Project base env class
Svf_proj_log_sv.tt	Project base log class
Svf_common_src_list.tt	The build file for the common directory
Svf_vip_defines_sv.tt	Defines specific to each VIP
Svf_vip_data_sv.tt	Data specific to each VIP
Svf_vip_gen_sv.tt	VIP specific generator
Svf_vip_if_sv.tt	VIP specific interface
Svf_vip_mon_sv.tt	VIP specific monitor
Svf_vip_sc_gen_sv.tt	VIP specific scenario generator
Svf_vip_cfg_sv.tt	VIP specific configuration
Svf_vip_xtr_sv.tt	VIP specific tranactor collection
Svf_vip_drv_sv.tt	VIP specific driver
Svf_vip_checker_sv.tt	VIP specific checker
Svf_vip_sv.tt	Top level VIP code
Svf_vip_src_list.tt	VIP specific build file
Svf_sample_test_sv.tt	Sample test
Svf_env_sv.tt	Environment
Svf_top_sv.tt	top.sv file i neach environment

Figure 13. Templates used to generate VMM environment

In Figure 14, we show one of the template files, Svf_vip_if.tt. This file is used for generating the interface shell files for all the vips. One way to view this file is to see it as a super macro, where each macro parameter is enclosed in *[% %]* symbols. In addition, there are directives that allow one to include other macros, do string processing, and create conditional loops.

As an example, the parameter *[% vip %]* is the name of the vip in use, and is supplied as a parameter by SVF-TG when the template is expanded to generate the code. The template *Svf_pw_header.tt* has the copyright information, and is included in each template. Further parameters such as *[% fileName %]* and *[% projName %]* are also available. To make changes to the generated code, the user can change the contents of these template files, and the tool ensures that these changes are reflected for all the components that are generated.

```
/********************************************************************************
 * Interface file [% fileName %] for vip [% vip %] in [% projName %] verification environment
 ********************************************************************************/

`ifndef [% projName FILTER upper %]_[% vip FILTER upper %]_IF
`define [% projName FILTER upper %]_[% vip FILTER upper %]_IF
`include "vmm.sv"

interface [% vip %]_if(input wire clk);
  logic [15:0] rdata;
  logic [15:0] wdata;
  logic write;

  // Add your own signals

  // For the DUT
  modport DUT( input wdata, output rdata, input write);

  // For the testbench
  modport TB( input rdata, output wdata, output write);

endinterface: [% vip %]_if
```

Figure 14. File template for generating pcie_if.sv used by SVF-TG

In addition to the templates for the individual files, a template also exists for the directory structure, portions of which are shown in Figure 15. The extracted portion says how one can specify the directory structure in the form of a template that generates an .xml description of the directory structure, which is then used by SVF-TG to generate the hierarchy. It is possible to change this template as well, so that the files can be organized differently than what is provided by default. A common scenario is how one plans to organize the log and coverage files for tests. Using this approach, once can easily specify a different directory template which will cause the log and coverage directories to be relocated as desired.

```
[% FOREACH env IN envList -%]
   <blkdir name="[% env %]">
     <descr>Top-level directory for [% env %] verification environment. </descr>
     <envdir name="env">
              <descr>Verification environment files </descr>
              <file name="[% env %]_env.sv" envName="[% env %]">
                       <descr>Testbench environment implementation that hooks up various components </descr>
                       <ttfile>  Svf_env_sv.tt </ttfile>
              </file>
              <file name="top.sv" envName="[% env %]">
                       <descr>top-level testbench for current environment </descr>
                       <ttfile> Svf_top_sv.tt </ttfile>
              </file>
              <file name="env.list" context="[% env %]" envName="[% env %]">
                       <descr>Build list for the entire [% env %] environment </descr>
                       <ttfile> Svf_env_list.tt </ttfile>
              </file>

[ deleted ]...

              <file name="[% env %]_env_cfg.sv" envName="[% env %]">
```

Figure 15 Template for VMM directory structure

3.3 Maintaining VMM environments and incorporating guideline changes

It is relatively straightforward to generate the whole shell environment at the beginning of project. However, it becomes difficult to add changes to the basic templates once the project is midway, after significant code has been added to the generated shell. The following sections demonstrate how SVF-TG can assist in incorporating such changes.

```
`ifndef [% projName FILTER upper %]_[% vip FILTER upper %]_IF
`define [% projName FILTER upper %]_[% vip FILTER upper %]_IF
`include "vmm.sv"
interface [% vip %]_if(input wire clk);
  logic [15:0] rdata;
  logic [15:0] wdata;
  logic write;

  // Add your own signals

  // For the DUT
  modport DUT( input wdata, output rdata, input write);

  // For the testbench
  modport TBDRV( input rdata, output wdata, output write);
  modport TBMON( input rdata, input wdata, inut write);
```

Figure 16. Customizing the interface definitions to add separate driver and monitor modports

3.3.1 Making changes to templates down the road

Consider the verification code for the interface for the pcie vip. By default, each interface is expected to have at least two modports, DUT and TB. The generated shell file, pcie_if.sv, is shown in Figure 17(a). Suppose the team defined a new signal, err, specific to this interface as shown in Figure 17(b).

Now suppose down the road, for better reuse, the team decided that instead of one TB modport, they would like to create two modport declarations: TB_DRV and TB_MON, representing the driver and monitor interfaces respectively. The changed template file is seen in Figure 16. The result of re-running the tool by pointing to the newer template is shown in Figure 17(c). The tool both preserves existing user changes as well as points out where conflicts may occur with existing code-base. For example, the declaration of the "err" signal is preserved. However, the modport declarations themselves need to be changed, and the tool clearly specifies the old template output, the changed version, and the automatically merged result. Clearly, this conflict has to be resolved by the engineer manually, but by explicitly pointing out all the conflict areas, the verification engineer is guaranteed to address all the changes necessary to incorporate the new guidelines.

```
interface pcie_if(input wire clk);
  logic [15:0] rdata;
  logic [15:0] wdata;
  logic write;

  // Add your own signals

  // For the DUT
  modport DUT( input wdata, output rdata, input write, output err);

  // For the testbench
  modport TB( input rdata, output wdata, output write, input err);
```

a. Generated from original template

```
interface pcie_if(input wire clk);
  logic [15:0] rdata;
  logic [15:0] wdata;
  logic write;

  // Add your own signals
  logic err;

  // For the DUT
  modport DUT( input wdata, output rdata, input write, output err);

  // For the testbench
  modport TB( input rdata, output wdata, output write, input err);

endinterface: pcie_if
```

b. Modified from original template generated output

```
interface pcie_if(input wire clk);
  logic [15:0] rdata;
  logic [15:0] wdata;
  logic write;

  // Add your own signals
  logic err;

  // For the DUT
  modport DUT( input wdata, output rdata, input write, output err);

  // For the testbench
<<<<<<< CURRENT
  modport TB( input rdata, output wdata, output write, input err);
||||||| OLDTMPL
  modport TB( input rdata, output wdata, output write);
=======
  modport TBDRV( input rdata, output wdata, output write);

  modport TBMON( input rdata, input wdata, inut write);
>>>>>>> NEWTMPL
```

c. Merged result of existing user code and new template

Figure 17. pcie_if.sv as generated from original template

4. Conclusion

Creating and maintaining VMM based verification environments across multiple sites and groups is challenging. The SVF-TG tool allows teams to capture the project specific guidelines in an executable form so that complete and detailed verification environment shells can be customized and generated from scratch. In addition, the tool is able to handle changes in the guidelines during the life of a project by automatically merging newer code with pre-existing code and pointing out where there are conflicts that can only be resolved manually. Users are welcome to try a limited version of this tool at [6].

5. References

[1] SystemVerilog FrameWorks™ User Guide Version 1.0
[2] Verification Methodology Manual for System Verification by Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale
[3] http://template-toolkit.org/
[4] Risk reduction in a verification upgrade. Ambar Sarkar, Brian Bailey, http://edn.com/article/CA6396963.html
[5] Reference Verification Methodology User Guide, Version 8.6.4, April 2005
[6] http://svf-tg.paradigm-works.com