Developing C++ Testbench Components Using UVM Phase and Agent Concepts

JinHaeng Cho
Scot Hildebrandt

NVIDIA
Westford MA, USA

www.nvidia.com

ABSTRACT

The UVM provides many useful libraries and as such is becoming the new de facto testbench standard. A majority of new testbench development adopts UVM and many companies want to convert non-UVM testbenches to UVM. However, converting non-UVM testbenches into UVM may not be cost effective due to various reasons such as: legacy infrastructure, familiarity of language, external IP support, etc. Also, many companies are using C++ to develop architectural or proof of concept models before the actual design phase starts, and it is desirable to reuse those C++ model in the design verification environment.

This paper introduces a technique on developing C++ testbench components using the UVM phase and agent concepts. This approach makes it easier not only to reuse such C++ models in the design verification phase, but also to convert them into UVM components. This method also has the advantage over a pure UVM testbench in terms of fast experimental turnaround time created by reduced simulator compile time.

Table of Contents

Table of Figures

1 Introduction

As chip design is getting more complex, there is the desire to verify complete functionality at the unit level testbench, with the upper level testbench verification more focusing on connectivity and inter operation between units. With this trend, it is more important to have a unified methodology and component reusability across testbenches. UVM is being quickly adopted as this de facto verification methodology standard. However, there are cases where some C++ code could be ported over or instead made to interact with newly developed UVM components. This paper describes a technique to write C++ models that can be easily converted to, or simulated with, UVM components as an intermediate step toward a pure UVM testbench.

2 Justification for using C++

A single testbench implementation language is normally preferred over multiple mixed languages since it can simplify flows and unify methodology. This testbench was developed as the projects' specification was in its early stages, so it was desired to have proof of concept modelling and fast prototyping. Also there was an architectural model, which employed a C++ only environment, and there was the possibility to use these new unit testbench components in this architectural model environment. In addition, this project was planning to co-simulate with external VIP, which is written in C++ from a cycle-based simulator environment.

We explored a testbench architecture that met the above requirements and also could be easily expanded to other industry standard verification methodologies such as UVM. The solution was to use C++ for testbench component development but utilize UVM concepts. By using C++ components, we had other benefits noted below.

2.1 Build time reduction

One of the common complaints of VCS in the early development stage is the re-compiling of the executable after design or verification components are changed. Synopsys has improved the flow over the years so that the re-compilation time can be reduced dramatically with the use of the incremental compilation feature. Unfortunately our build flow doesn't work properly for incremental compilation and as such, re-compilation time is one of our painful areas. When using C++ components, we compile these C++ components separately and create a shared object file that we then link to the VCS executable. Unless there are changes that affect the Verilog side (one example is DPI) or C++ function interfaces, we need only recompile the C++ components and the VCS executable can be used without recompilation. In our case, C++ re-compilation only takes seconds.

2.2 Light simulation overhead

The SystemVerilog language and UVM library has many useful runtime features such as periodic garbage collection, null object referencing, etc. However these features do not come free but add some run time overhead so that simulation performance is worse when compared to a testbench with C++ components. In addition, the use of DPI calls to cross the SystemVerilog and

C++ language boundary is very efficient and as such that we can achieve much faster simulation time in our testbench.

2.3 Early readiness for DUT verification

In many design verification flows, full DUT verification is delayed due to testbench component availability. In our model we developed the testbench components for proof of concept but with later reuse for design verification in mind. With this we could quickly ramp up the full DUT verification environment.

3 C++ components with UVM phase and agents

The benefit of developing testbench components using UVM is that the simulation imparts features such as multi-threading, delay, event, randomization, etc. which are all available from the SystemVerilog language itself. However, C++ itself doesn't provide such simulation features so the developer needs to implement them or use existing libraries. SystemC could have been a possible solution but it requires stiff ramp up and requires additional infrastructure work to implement and utilize. In the following sections, we will describe how to utilize UVM like structures in C++ code for many simulation features. We collectively called these created C++ classes CSIM (C++ SImulation Module). The reuse of BFM and C++ testbench components will be referred to as CSIM components throughout this document.

3.1 CSIM classes

Many of upper level UVM components such as agents, drivers, and monitors are derived from base UVM classes. The CSIM components also have some base classes and most of BFM modules inherit one or multiple of those base classes as needed.

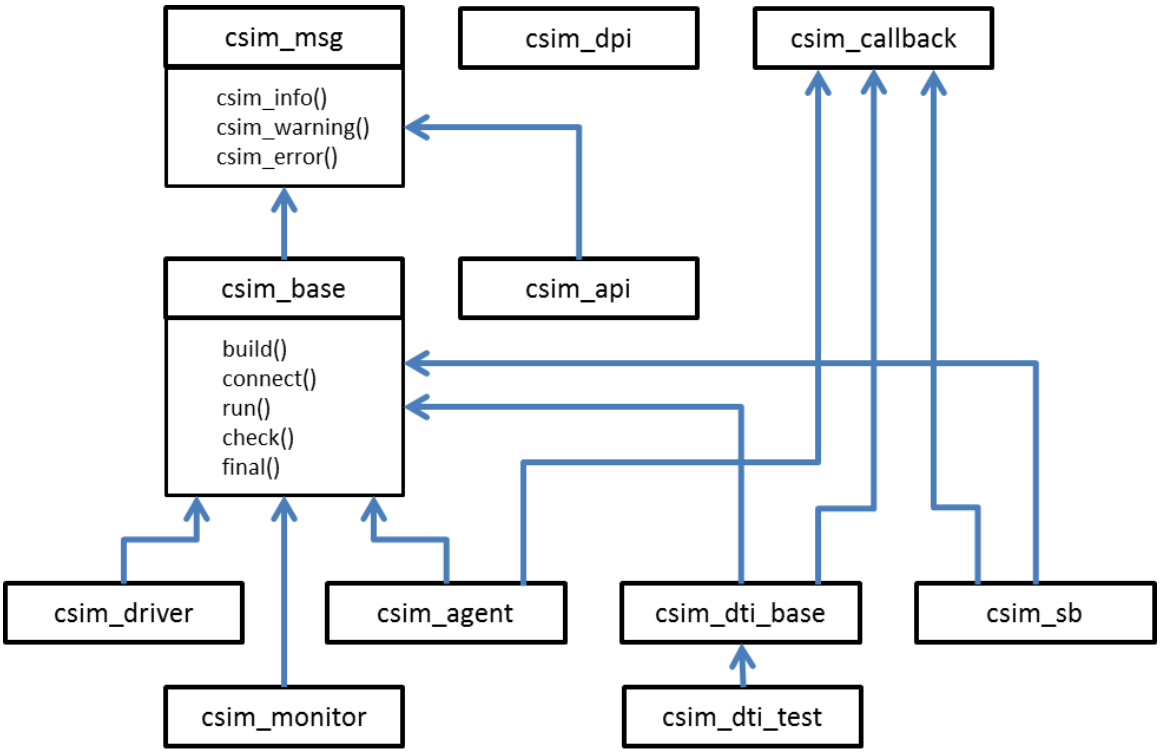

Figure 1 CSIM class hierarchy

3.1.1 Class csim_msg

The csim_msg class serves as the base class for the messaging system. It provides the macros for INFO, WARNING, and ERROR types of messages, along with verbosity control.

3.1.2 Class csim_base

The csim_base class is the main common class across all CSIM components. It has all UVM like phase functions, objection control, clock, and reset interfaces. It inherits the csim_msg class and most of the CSIM components are derived from csim_base.

3.1.3 Class csim_callback

The csim_callback class is used as callback mechanism for packet and credit events. This class provides functions that pass packets or credits as items. This is mostly used by the BFM agent or scoreboard, and registers it to the monitor so it can pass the received packets or credits to the agent or scoreboard.

3.1.4 Class csim_agent

The csim_agent class is the main bus functional model. It can be considered equivalent to the UVM agent, sequencer and agent environment classes combined as it creates the driver and monitor and interacts with stimulus.

3.1.5 Class csim_driver

The csim_driver class is used to drive packets to the interface. When used in the Verilog co-simulation environment, the driver pulls a packet from the csim_agent and unpacks the packet to interact with the SystemVerilog side interface driver. In a C++ only environment, this driver will send the packet as is to the other consumer by calling the appropriate function.

3.1.6 Class csim_monitor

The csim_monitor class is used to collect packets from the interface. When used in Verilog co-simulation environment, the monitor packs the interface signals from SystemVerilog through DPI into a packet. When a packet is collected, it sends the packet to the consumer who registered the callback to the monitor.

3.1.7 Class csim_dpi

The csim_dpi class is the channel utilized to communicate with SystemVerilog side functions. It implements C++ side import DPI functions and calls SystemVerilog side export DPI functions.

3.1.8 Class csim_api

The csim_api class is collection of functions to support external VIPs and provides functions to communicate with SystemVerilog and other CSIM components.

3.1.9 Class csim_packet

The csim_packet is similar to the uvm_sequence_item and is used to pass packet information among csim components.

3.1.10 Class csim_dti_base

The csim_dti_base class is similar to the uvm_base_sequence and is used as the base class for all stimulus. The acronym DTI stands for Directed Test Interface and gives the user full control of sequence generation, including random traffic generation capability. This base class has common functions among all stimulus such as: testbench initialization and control, status, statistics and callback functions.

3.2 CSIM phase and testbench construction

The csim_base class has functions that are equivalent to the UVM phases. Each phase's functions are named as do_* and are called through DPI from the corresponding UVM phase. For example, do_build() is called on the build_phase and do_connect() is called on the connect_phase. The functionality of each do_*() is similar to the UVM phases. In UVM, the evaluation order for build_phase(top-down) and connect_phase(bottom-up) are opposite, but in CSIM the evaluation order is determined by the order the CSIM component is pushed into the

global CSIM instance vector in the csim_base class. For the build phase, it is naturally top-down order since the parent registers itself before the child component. For the connect phase, it will be also evaluated in top-down order due to the instance registration order so the user needs to be careful about dependencies in the connect phase. Below are list of currently supported phases and brief descriptions.

- do_build() : called on the build_phase and instantiates csim components from parent to child
- do_connect() : called on the connect_phase and registers callback for packet and credit events
- do_run() : called on the run_phase and spawns threads
- do_check() : called on the check_phase and does the end of simulation check
- do_final() : called on the final_phase and reports the end of simulation status and statistics

Below is a diagram showing the testbench top-level classes used to construct the testbench components and phase control. Two of left side blocks are SystemVerilog(SV) classes and the right two blocks are C++ classes.

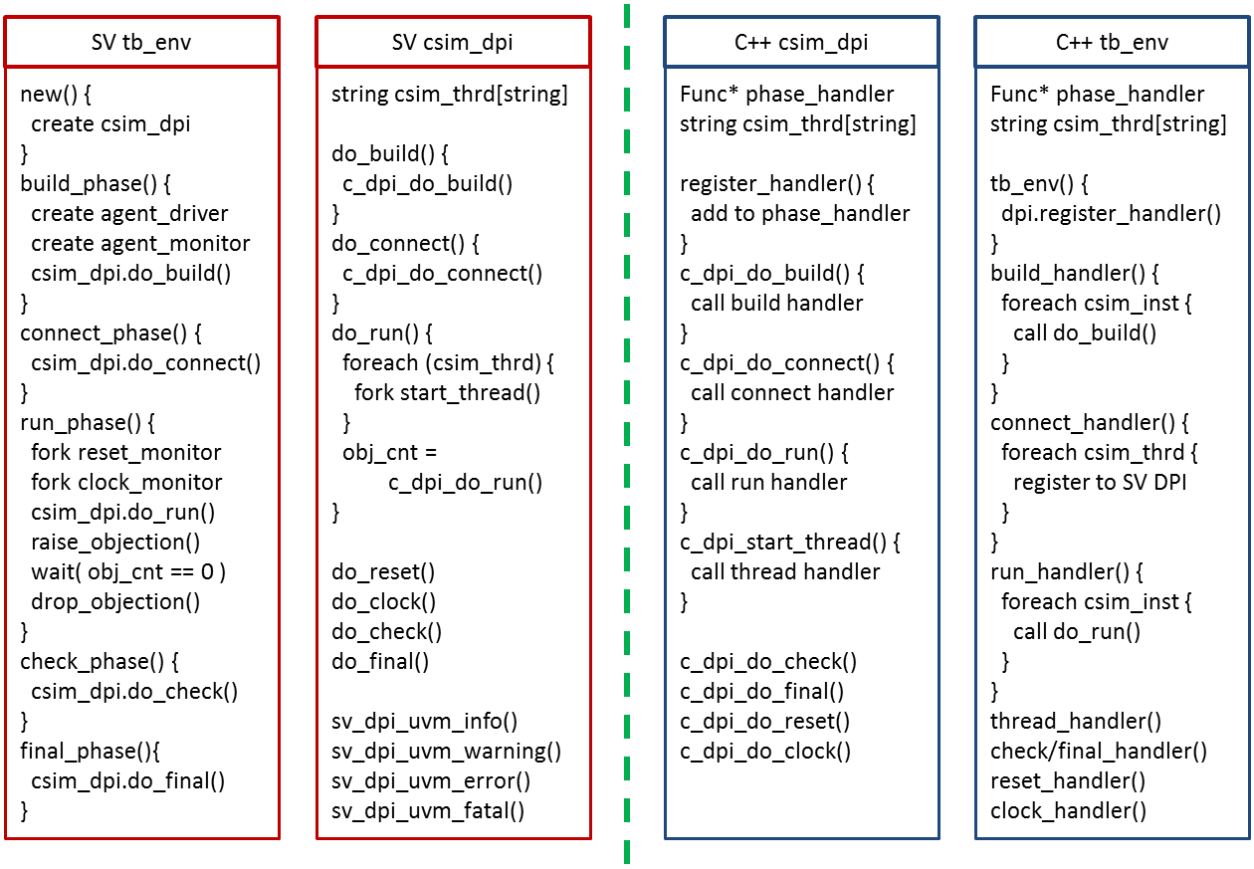

Figure 2 CSIM testbench construction

3.3 Multi-threading

Unlike SystemVerilog, C++ doesn't have multi-threading support in the language itself. The user needs to implement such a feature themselves or utilize a library such as SystemC. In this testbench, we decided to use the SystemVerilog language to create multi-threading in the C++ side.

In SystemC, there are two main mechanisms to spawn a thread, SC_METHOD and SC_THREAD, and the function body is executed whenever trigger condition is met (SC_METHOD) or until it exits (SC_THREAD). In CSIM we provide a similar, but limited, version of such a mechanism. For the SC_METHOD style thread the CSIM has do_reset() and do_clock() function in the csim_base class. The prototype for do_reset() and do_clock() follows:

void do_reset(uint *rst_sel*, uint *inst_id*, bool *is_asserted*);
void do_clock(uint *clk_sel*, uint *inst_id*, bool *is_posedge*);

Where *clk_sel* is the ID of the clock and *inst_id* is the ID for the instance that is using *clk_sel* and *is_posedge* is clock edge information. The do_clock() is called per each clock source through DPI from the SystemVerilog side. In the C++ side each component that inherits csim_base class may implement do_clock() if they have logic that operates on a certain clock. A single component can react to multiple different clock sources by calling different functions from do_clock() after parsing the clk_sel information. The do_reset() operates the same way, but the source of the event is the reset signal in Verilog.

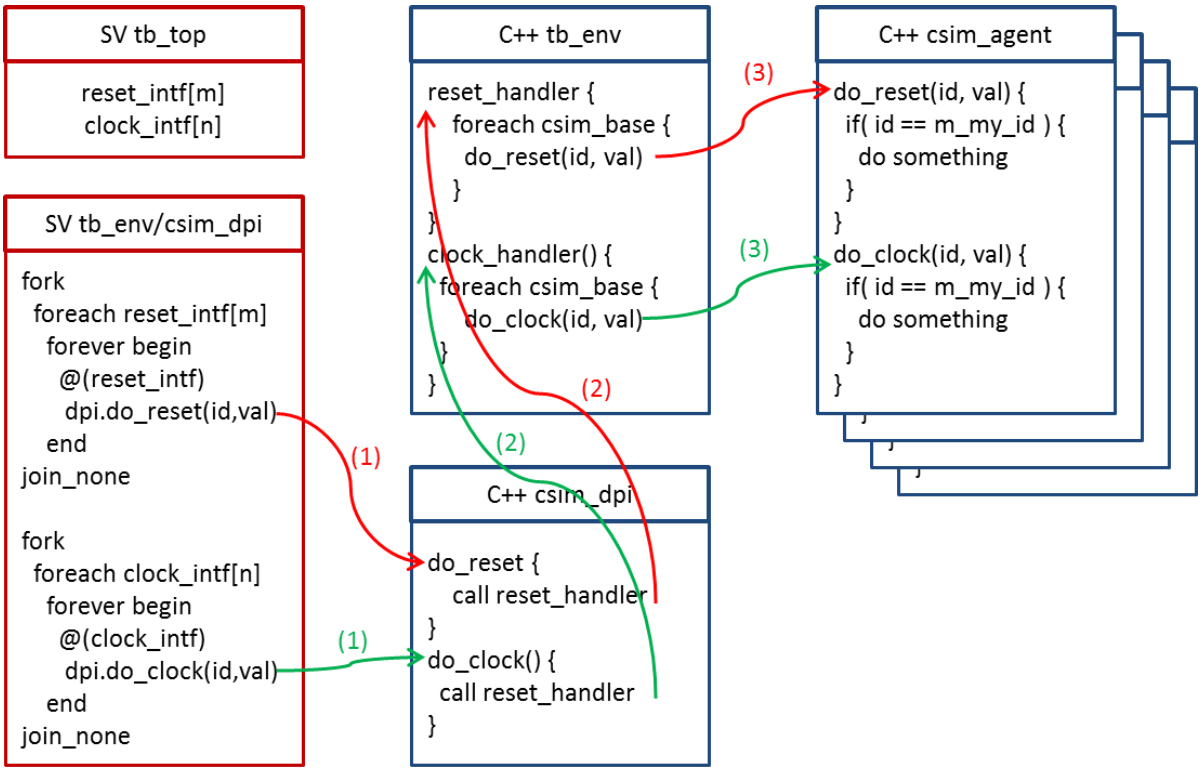

Figure 3 CSIM reset and clock thread construction flow

For the SC_THREAD style thread, the csim components register functions by function and instance name to a thread vector list either in the build or connect phases. The thread registration information is stored in SystemVerilog and during the run phase, SystemVerilog starts the thread by forking a task that calls DPI with the registered thread name information. During the CSIM construction, each csim_base derived component registers itself into a global static vector in csim_base. When start_thread() is called from SystemVerilog, with the function and instance name, the CSIM instance with the matching instance name from the stored static vector calls its own start_thread() with thread function name and the function is executed until it exits. Similar to the SC_THREAD, the registered thread function can call time-consuming functions through DPI. One example of such a time consuming DPI call is wait_ns(delay_time), which simply waits the given time in the SystemVerilog domain and returns.

Figure 4 illustrates how a CSIM thread is registered and started during the simulation. The number in the diagram indicates the order of function calls. Thread registration is requested (step 1-1) and thread information is collected into the csim_thread_vec (step 1-2) on the C++ side. Then the csim_thread_vec information is transferred to System Verilog data structure (steps 2-1 and 2-2) and the CSIM thread is started during the run_phase (steps 3-1 through 3-4).

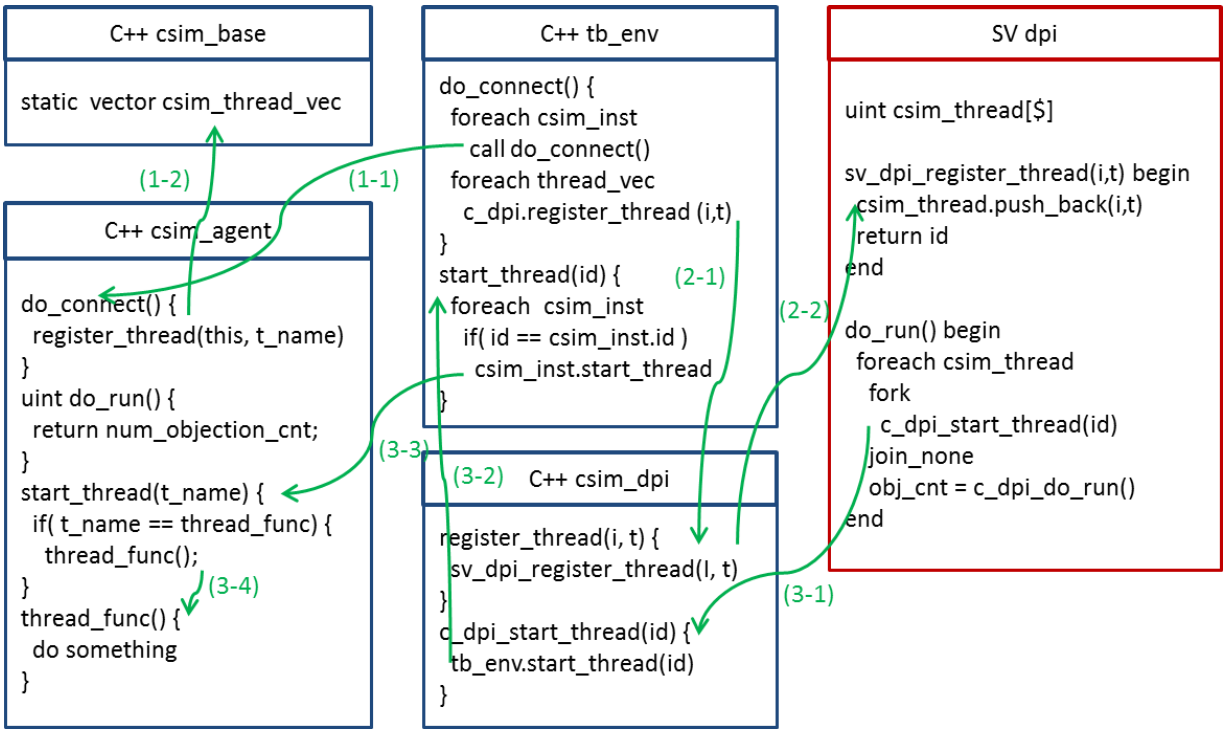

Figure 4 CSIM thread registration

3.4 Stimulus generation

The CSIM environment has a csim_packet that is equivalent to a uvm_sequence_item and its csim_dti_base is analogous to a uvm_sequence. Each test case inherits the csim_dti_base class and implements the main stimulus body based on the scenario it will cover. Unlike SystemVerilog, there is no built in randomization feature in the csim_packet, so we created

another class called csim_pkt_gen that has all the randomization knobs to generate a random or controlled random csim_packet. Each test can use the csim_pkt_gen object to create certain packets by setting control knobs or simply asking for specific packets. The stimulus main function is normally registered as a thread so that it can utilize time consuming functions. By using the csim_api and a SystemVerilog interface, the test can drive and monitor Verilog signals and also can wait for events from Verilog. The user can register multiple threads in a test that can either run independently of each other or be coordinated by a user defined synchronization scheme.

We collected all the different test cases under a single directory and compiled them together to create a single shared object file and linked this into the VCS executable. One or multiple test cases can be selected from the command line by passing the test name to the VCS executable through plusargs similar to how UVM selects tests for run_test(). If a user needs to modify an existing test case or add a new test case, only the DTI test directory needs to be modified and recompiled versus recompiling the whole VCS executable.

Figure 5 below illustrates typical stimulus flow. The stimulus has a clock() function that is called on every clock edge associated with it on the SystemVerilog side through DPI. Also it registers a thread function which is called during run_phase and run as a separate SystemVerilog thread until it returns. The thread can call blocking SystemVerilog tasks through DPI and remains suspended until the SystemVerilog action completes.

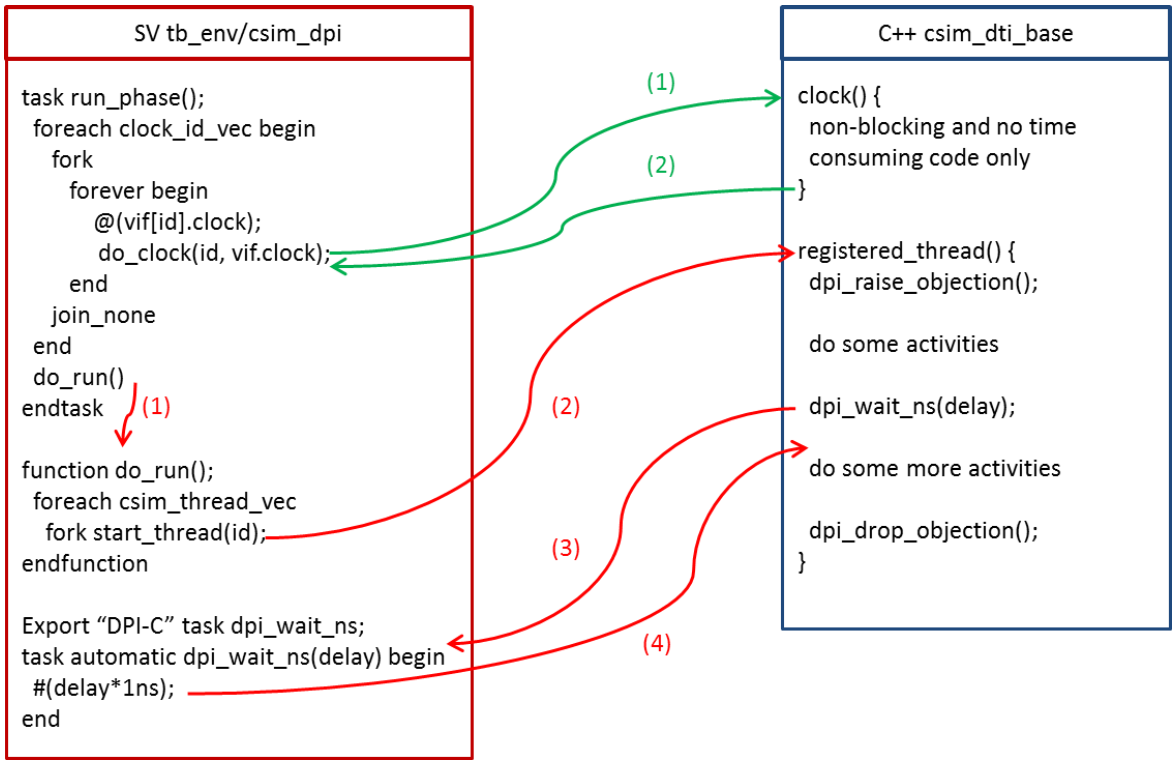

Figure 5 CSIM stimulus flow

3.5 Testbench configuration

In the current CSIM implementation, we didn't utilize uvm_config_db and uvm_resource_db to collect command line options and then pass the configuration information to testbench components. We instead used the publically available *sknob* library. The *sknob* library reads command line strings, parses them, and then stores the parsed result into its centralized storage. Our C++ and SystemVerilog components use *sknob* functions to retrieve option switch values to configure the testbench. This *sknob* library supports regular expressions and some randomization so that we can apply the same option value across multiple instances or different values to the same configuration variable in multiple instances of the same class. When used to evaluate a random string, the returned value is cached, allowing the user to obtain the same return result across multiple callers, be they C++ or SystemVerilog.

During the build and connect phases, the testbench environment components get topology configuration information through *sknob* and instantiate agents as directed. Also other static configuration information such as buffer size, credits, and internal delays are initialized in those phases.

3.6 Driving and collecting packets

As the CSIM model can be utilized in two different environments, both a C++ only environment and a combined C++/SystemVerilog co-simulation environment, the packet driver and monitor are structured as two layers. The C++ layer is responsible for interface protocol and implements the packet pack/unpack state machine. The SystemVerilog layer is a simple signal driver and monitor so that it updates interface signals from a C++ driver request and sends observed interface signal values to a C++ monitor. In the C++ only environment, the SystemVerilog interface DPI call can be replaced with function interfaces to other C++ code.

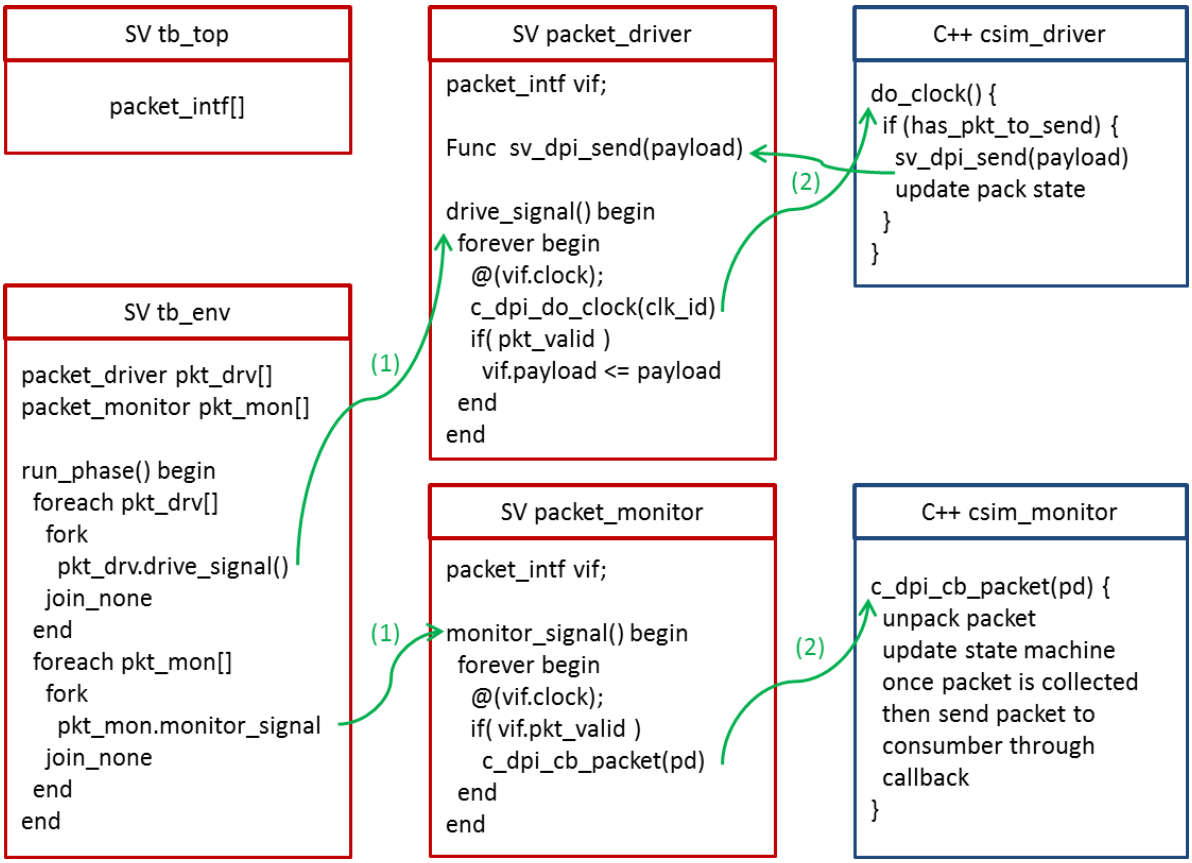

Figure 6 CSIM Driver and Monitor

4 API for external VIP

The only API usage model currently implemented was created to support external VIP that came in the form of a cycle-based, 2-state, C++ model. This VIP generates stimulus and provides checker functions and needs to drive and sample signals on the Verilog side. The external VIP is provided as shared object files and compiled into the VCS executable by linking against those shared objects.

The main requirements to use this VIP are

- Provide a clock event to advance the simulation cycle in the VIP
- Provide a standardized C++ side 2-state drive and monitor interface
- Attach signal ports in C++ to SystemVerilog interfaces to drive and sample Verilog signals
- Provide simulation phase and status information
- Provide unified messaging functions

Utilization of this VIP doesn't require any special feature additions outside of the interfacing API. The VIP has its own simulation phases and they are mostly mapped as sub-phases within CSIM's and UVM's run_phase. Whenever CSIM calls the API to advance the VIP's simulation cycle, it collects VIP's simulation phase information and if it is in a running state, CSIM raises an objection so that simulation will continue until VIP declares an end of simulation condition.

Through the API's messaging communication, the VIP can report errors to CSIM. CSIM will exit the simulation after shutting down the VIP with some phase handshakes through the API.

5 Conclusions

The initial testbench was developed well ahead of any RTL design being written. It only took a few months to create initial CSIM components and the CSIM model in the testbench only environment. By modelling the components in C++, we could quickly implement it as high-level abstract functions and iterate many changes to prove the validity of the specification. Also, even without RTL design, the testbench started with C++ and Verilog co-simulation, and a C++ model was instantiated as a DUT model. By having this testbench, we could develop and verify all the components including stimulus well ahead of a true DUT so that when the design was available, we could quickly create testbench with DUT and start full regression mode immediately.

We successfully integrated cycle-based C++ external VIP into our testbench for stimulus and checking. Our testbench integrated against this external VIP was used in the qualification regression for the design IP delivery to our partner company.

This approach also proved very efficient in terms of bug turn time, especially in testbench components. We could iterate many changes in short time due to fast testbench recompilation and simulation time.

6 Future work

One of the areas we didn't take advantage of a SystemVerilog feature is in randomization. SystemVerilog provides powerful built in randomization with its constraint solver but since we generated stimulus in C++, we needed to implement our own simple randomization mechanism. In the future, we plan to define packet classes in SystemVerilog and use rand variable for the packet fields. The packet class will have constraints for each packet field that we will use to create new packets that are automatically randomized based on those constraints. We can then pass the SystemVerilog packet class to the C++ agent through DPI and the rest of the flow will remain the same.

One disadvantage of our approach over UVM is in the area of debugging. Since DVE and Verdi don't trace C++ internal variables, we mostly relied on print statement in C++ to track the C++ components' internal state. Since the C++ compilation time was so fast and didn't require VCS recompilation, it was not an issue to add some debugging messages and rerun the simulation. Sometimes an attached debugger such GDB was used in conjunction with an interactive DVE session for tracing down stack dump sources. Adding more runtime debug aid, such as exporting C++ variable values to Verilog to allow dumping of the values, might be an area of improvement.

Another area we will explore is utilizing the UVM Connect library or the UVM-ML open architecture. In the next project, we plan to integrate an architectural model, written in SystemC, into the DUT testbench to validate the design implementation. The UVM Connect library

appears to provide the TLM level functionality needed to connect the SystemC architectural model to our UVM compatible DUT testbench.

7 References

[1] Kathleen A Meade, Sharon Rosenberg : A Practical Guide to Adopting the Universal Verification Methodology(UVM), Second Edition

[2] Sknob : http://sourceforge.net/projects/sknobs/

[3] SystemC : http://accellera.org/downloads/standards/systemc

[4] UVM Connect : https://verificationacademy.com/topics/verification-methodology/uvm-connect

[5] UVM-ML Open Architecture : http://forums.accellera.org/files/file/65-uvm-ml-open-architecture/