Augmenting a C++/PLI/VCS Based Verification Environment with SystemC

Dr. Ambar Sarkar

Paradigm Works Inc.
ambar.sarkar@paradigm-works.com

ABSTRACT

Due to increased popularity of high-level verification languages (HVLs) such as Vera/e/SystemC, many verification teams are increasingly curious about adopting HVLs in their own functional verification. However, unless starting from scratch with a new verification environment, it is often a difficult task to have a HVL based approach coexist with a PLI based approach.

The author recently had the opportunity of augmenting an existing C++/PLI/VCS based verification environment with SystemC based verification components. While it sounds simple, there were quite a few technical challenges to overcome. For example, how would one make the transactors in the existing environment communicate with the SystemC transactors? This paper describes such challenges and implemented solutions.

Table of Contents

Table of Figures

1.0 Introduction

Due to increased popularity of high-level verification languages (HVLs) such as Vera/e/SystemC, many verification teams are increasingly curious about adopting such HVLs in their own functional verification flow. For verification environments already using a C++/PLI[1] / VCS[2] based approach, SystemC[3] with SCV[4] offers an excellent path to migrate to a HVL while preserving many of the legacy verification environment. Using a HVL for the first time typically implies using newer set of tools, and often results in a different way of writing verification code.

For example, if using a C++/PLI /VCS based approach, one cannot directly manipulate or observe Verilog signals from the C code. Typically, transaction requests from the test are queued. These queued requests are forwarded by the verification environment to the Verilog side. This forwarding mechanism is the PLI code, which has both Verilog and C/C++ components. The low level code that interacts directly with the DUT is written in Verilog and acts as a bus functional model (BFM), whereas the C code ends up implementing some sort of queuing data structure that arbitrates requests from the C side to the Verilog BFM.

On the other hand, test code written in SystemC/SCV can both manipulate and observe signals in the DUT directly. Majority (or all) of the code can be being written using the HVL. From a verification point of view, writing most or all of the verification code in HVL is preferable as a lot of features such as templates and verification components such as scoreboards can be employed at all levels of abstractions.

However, unless starting from scratch with a new verification environment, it is often a difficult to have a HVL based approach coexist with a legacy PLI based approach as the two approaches depend on different mechanisms of communicating between the test code and the RTL. The legacy and the HVL transactors end up being quite different, especially at the lower levels of abstraction. Considering SystemC is really C++, it comes as a surprise to many when they just hope that they can immediately start writing transactors in SystemC that will coexist with their preexisting C++ transactors written for a legacy PLI based environment. While it sounds simple to use a SystemC transactor within a legacy C++/PLI environment, there are a few technical challenges to overcome.

2.0 Typical PLI based Environments

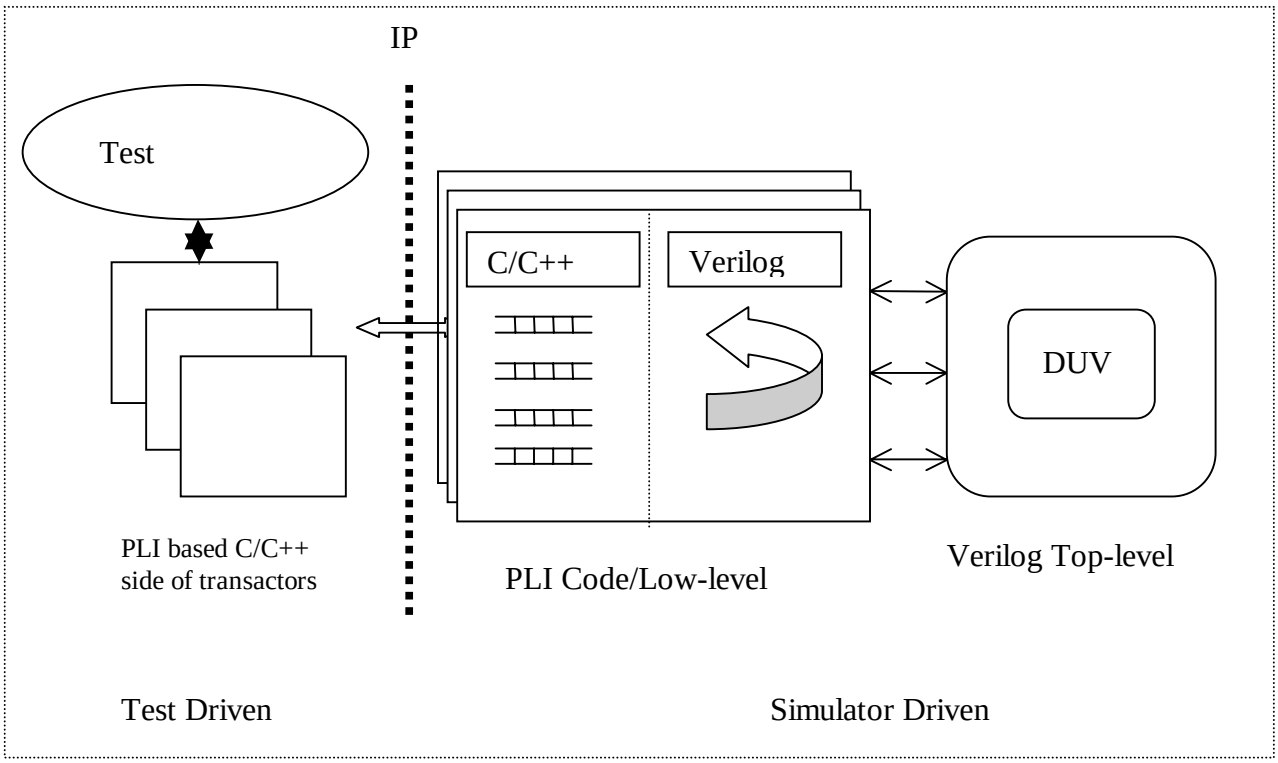

Figure 1.0 Typical C++/PLI/VCS Architecture

Figure 1.0 describes a typical PLI based verification environment architecture. There are two main timing domains in the environment, the simulator driven and the test driven. The test driven domain contains the test code and the C/C++ portions of the transactors. The simulator driven domain executes the Verilog code, as well as any PLI code that exists for the low-level transactors or BFMs. The test code and the PLI code run in separate threads or processes, and use inter-process communication (IPC) methods such as messages or shared-memory to communicate to the Verilog side.
A transactor typically has two components, the C++ side, and the PLI/Verilog side. The C++ side forwards transaction requests to and accepts responses from the Verilog side over the IPC. The requests are typically queued up and executed by the Verilog side which polls these queues per clock cycle.

2.1 Issues with PLI based approaches

There are several issues that arise when using a PLI based approach.

2.1.1 Indirect interaction with the Verilog

Under most implementations, the test code cannot directly manipulate or observe the Verilog signals. The transactions are specified in a high-level manner, and these requests are queued and sent one at a time to the Verilog side of the transactor, which implements the low level details.

For example, when performing a register write, the test code specifies the address and the data, while the Verilog side takes care of driving the actual Verilog signals and implements the low-level bus protocol. The C side cannot observe or drive the Verilog signals directly without significant coding effort. This often creates a problem in cases where fine grained control of stimulus is needed.

2.1.2 Unavailability of HVL features at all levels of abstraction

Since the direct interaction with the DUT has to be done in Verilog,, the lower-level transactor code is written in Verilog as well. This means many of the features of high-level languages such as pointers, standard template libraries, reentrant tasks etc are unavailable when implementing low-level transactors.

2.1.3 Needs work to synchronize time between C++ and Verilog domains

Since the C++ side does not have any explicit notion of simulation time, the verification environment often provides tasks that are called by the C side. However, it is still quite clumsy to synchronize events on the C side with those on the Verilog side.

2.1.4 Follows Verilog-on-top model

Since most HVLs already have the notion of simulation time built into them, the test code is the topmost object in the verification hierarchy and drives the entire simulation by advancing simulation time. In case of PLI models, it is the Verilog side that advances time and is the topmost element in the verification hierarchy. Not only it is much simpler to write tests using a HVL on top model, but most HVLs based environments are easier to compile with the HVL on top model.

3.0 Integrating SystemC

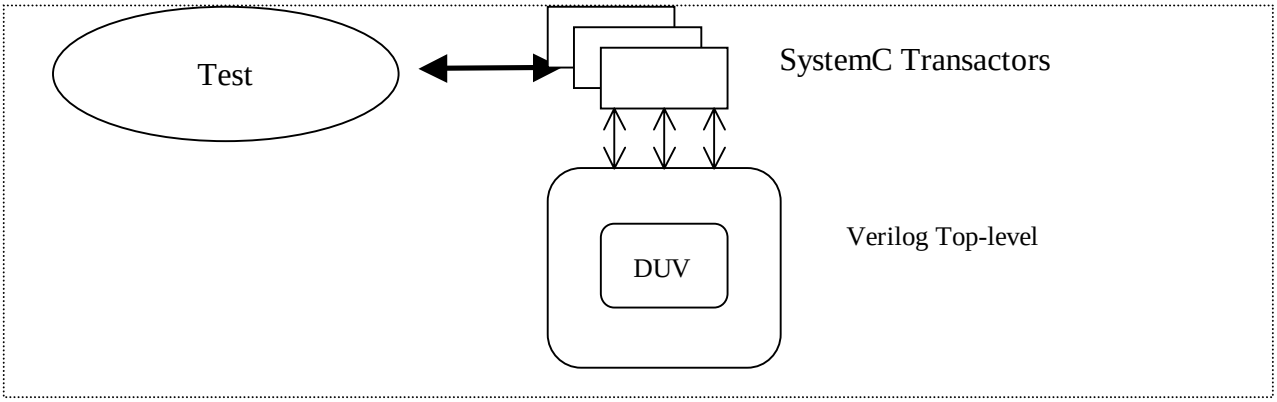

Figure 2.0 Typical SystemC-on-top Architecture

Figure 2.0 above shows a typical SystemC based verification environment architecture. Since one can directly observe and manipulate Verilog signals from SystemC, the transactors written in SystemC can be directly hooked up with the DUT signals. All levels of abstractions can be coded in SystemC. Both the test code and the transactors can execute with the Verilog simulator in a time synchronized manner. The goal is to somehow integrate this architecture on top of the legacy C++/PLI based verification environment.

In Figure 3.0 below, a combined approach that integrates SystemC transactors in a PLI based environment is shown. Note that the SystemC transactors exist in the Verilog time domain, whereas the legacy transactors do not. All legacy transactors and test code can be executed as before.The SystemC transactors coexist with the legacy transactors, and can communicate with both the DUT and the legacy transactors. This combined approach allows one to now write test code that can directly manipulate or observe the DUT signals while preserving the legacy transactors. By virtue of its direct interaction with SystemC transactors, the legacy test environment now has been enhanced to interact directly with the Verilog time domain in a synchronized manner.

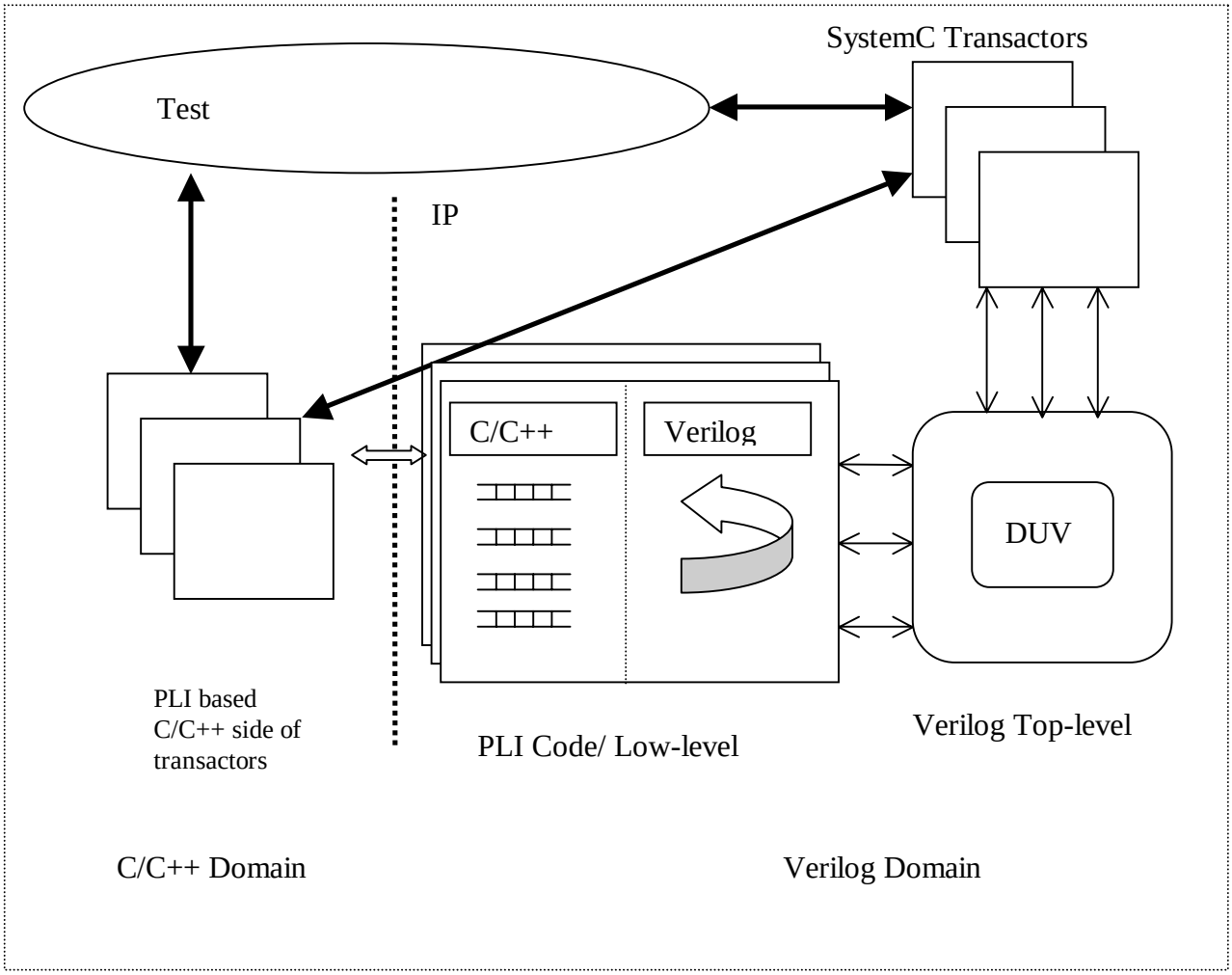

Figure 3.0 Typical C++/PLI/VCS Architecture With SystemC

To create the verification environment shown in Figure 3.0, several challenges need to be addressed. First, the new SystemC code needs to be compiled and linked in with the existing environment. Second, the SystemC transactors need to be instantiated within the existing environment, and they need to communicate with the test environment, including the legacy transactors. Any error reporting from SystemC side has to be assimilated into the legacy framework as well. In addition, one has to make sure that any Verilog files produced during

SystemC compilation by VCS are ignored by typical assertion tools. The following sections describe these challenges and their solutions in detail.

3.1 Linking OSCI SystemC Reference Implementation Using VCS

By default, VCS does not include an integrated SystemC simulator. While future versions of the tool may change this, for now, a reference implementation has to be downloaded, modified, compiled, and installed.

The reference implementation is available from OSCI. The implementation is available in the form of source code under GPL. The source code is available for download. However, after downloading, a couple of files in the source code hierarchy need to be replaced with files provided by Synopsys. These changes are required to integrate the SystemC simulator with VCS. The location of the files are available in the VCS users manual in the section[???].
Once the files are changed, the SystemC simulator can be compiled and installed according to the instructions that come with the SystemC release. Root permission is not needed, but it is preferable to install it in a common site-wide location to avoid having multiple installations for different projects.

In addition to the reference SystemC implementation from OSCI, the SystemC Verification Library (SCV) is also required for verification. However, no changes are required in the source code hierarchy and the library can be installed in a straightforward manner according to the instructions that come with the distribution.

3.2 Compiling SystemC With PLI-based transactors

The transactors written in SystemC cannot be just compiled and linked with the existing C++ objects. In addition to generating the object files from the transactors, one needs to generate wrapper code that implements the communication between Verilog and SystemC.

In VCS, the syscan utility is provided for this purpose. The typical invocation of syscan utility is as follows:

```
syscan -cpp <cpp>  -Mdir=<work_csrc_path>  <srcfiles> -cflags <cflags>

where
     <cpp>              is the path to the SystemC compatible C++ compiler
     <work_csrc_path>   is the path to the output directory for the VCS
                        compilation and should be the same as the
                        one used for the legacy environment
     <srcfiles>         list of files that declare and define the SystemC
                        modules
     <cflags>           any relevant C compilation flags including those used
                        in the legacy environment
```

The syscan utility generates the wrapper code, and compiles the relevant SystemC files and links the resulting object files with the objects generated from the other verilog files.

To ensure that the SystemC code can work with the legacy code, the following should be kept in mind.

C++ compiler used. The use of SystemC may require one to use a different version of the C++ compiler than the one used in the legacy environment. For example, if the legacy version used gcc version 2.95.3 on Red Hat Linux platforms, it is necessary to upgrade the C++ compiler to at least version 2.96. This also implies that the entire legacy code needs to be recompiled with the new compiler version. This may result in several new compile and runtime warnings which need to be addressed and may potentially require some source code changes in the legacy environment.

Output of syscan utility. The syscan utility generates and compiles the wrapper code in Verilog by processing the SystemC class declarations. The –Mdir option specifies the output directory where the relevant output is generated. The same output directory as the one used in compiling the legacy Verilog code should be specified. Typically, this should be the path to the work.csrc directory used in the legacy environment.

Incompatibility of legacy header files with syscan. Typically, the new SystemC transactors may need to call the utilities developed in the legacy environment. Often, just including the header files in the SystemC code suffices. However, sometimes the legacy header files may give warnings or even errors when compiled with the syscan utility, even though it is legal C/C++ code. In such cases, separate header files(.h) and corresponding source files(.cc) need to be written that serve as proxy implementations and eliminate the need for including the incompatible header files.
For example, suppose the legacy function `Register::Write(uint addr)` is needed and is defined in the header file *Register.h*. However, Register.h may turn out to be incompatible with syscan. In such cases, a proxy header and source files need to be defined:

scvproxy.h

```
#ifndef __SCVPROXY_H__
#define __SCVPROXY_H__

#include “scv.h”

void RegisterWrite(uint addr);
….

#endif
```

scvproxy.cc

```
#include “scvproxy.h”
#include “Register.h”
void RegisterWrite() {
  Register::Write(addr);
}
….
```

The file *scvproxy.h* can be included in any of the SystemC transactor that needs to call `Register::Write()`. The file *scvproxy.cc* should be added to the list of C++ files compiled in the legacy environment.

3.3 Makefile Issues

Any time the SystemC transactor code changes, the wrapper code needs to be generated, and all the SystemC files recompiled. However, the SystemC transactor code needs to be compiled as a part of the Verilog compilation step and not the C side compilation. As a result, extra dependencies of the following form need to be added to the makefile dependencies to make sure the wrapper code is regenerated and the related files recompiled.

For example, suppose ScXactor.cc contains the definition of the transactor ScXactor. The following dependency rule will be needed:

```
<work_csrc_path>/sysc/ScXacor/ScXactor.v: <path to ScXactor.cc>
   syscan -cpp <cpp>  -Mdir=<work_csrc_path> $<:$(<F:%.cc=%) -cflags <cflags>

where
   <cpp>                 is the path to the SystemC c++ compiler
   <work_csrc_path>      is the path to the output directory of the VCS
                         compilation for the legacy environment
   <cflags>              any relevant c compilation flags including those used
                         in the legacy environment
```

3.4 Instantiating SystemC Objects

The SystemC transactors are instantiated in the Verilog domain, and as a result, the handles to these transactors are not available on the C side by default. In other words, one cannot create an instance of the SystemC transactor from the C side directly. Instead, a mechanism has to be implemented that passes the handle from the Verilog side to the C side. The basic idea here is to use the C++ static method definition to pass the pointer to the SystemC transactors from the Verilog to the C domain.

In the given example, `ScXactor` is the SystemC transactor. The file *ScXactor.h* and *ScXactor.cc* contain the definition of the transactor `ScXactor`. The static function `ScXactor::GetHandle()` can be used from anywhere in the test environment to get a handle to the instance of the transactor.

ScXactor.h

```
// ScXator module is declared here
SC_MODULE(ScXactor)
{
 public:

   sc_in <sc_logic>    clk;
   sc_inout <sc_lv<16> >    data;


   SC_CTOR(ScXactor):
      clk("clk"),
      data("data")
   {
      m_pScXactor = this;
   }
```

```
   void DoFoo();

   // Access this instance of the ScXactor
   static ScXactor  * GetHandle();

private:

   static ScXactor  *m_pScXactor;
};
```

ScXactor.cc

```
// The following are needed to instantiate static members of this class
// Initialize this static member to NULL
// This will get initialized only when the constructor is called
ScXactor * ScXactor::m_pScXator = 0;

ScXactor *ScXactor::GetHandle() {
   return  m_pScXactor;
}
```

top.v

```
   ScXactor  ScXactor(
       .clk(clk),
 .data(data))
```

```
Accessing ScXactor from C/C++ side:

#include “ScXactor.h”

ScXactor *pXtr;

void foo() {
…
  // Make sure this is called after time has progressed by at least 1
  // unit
  pXtr = ScXactor::GetHandle();
  if (!pXtr) {
      // Print error
      exit();
  }
  pXtr->DoFoo();
}
```

3.5 Multiple Instantiations

The approach outlined in section 3.4 assumes that there is only one instance of the `ScXactor`. However, it is common to require multiple instantiations of the same transactor class. The different instances can be differentiated by passing a parameter in the port list. In the example below, the port id is used for this purpose. Based on the value assigned during the transactor instantiation, the transactor instance identifies itself as being transactor 0 or transactor 1.

Unfortunately, the simple approach of checking the id port in the constructor code and assigning the handles accordingly as shown below does not work:

ScXactor.h

```
// Channel implementing the transactor
SC_MODULE(ScXactor)
{
 public:

sc_in <sc_logic>      id;
sc_in <sc_logic>    clk;
sc_inout <sc_lv<16> >    data;


   SC_CTOR(ScXactor):
      id("id"),
      clk("clk"),
      data("data")
   {
     // The following would not work
      if (id == 0)
            m_pScXactor0 = this;
      if (id == 1)
            m_pScXactor1 = this;

   }

   void DoFoo();

   // Access this instance of the ScXactor
   static ScXactor  * GetHandle(int which);

private:

   static ScXactor  *m_pScXactor0;
   static ScXactor  *m_pScXactor1;
};
```

ScXactor.cc

```
// The following are needed to instantiate static members of this class
// Initialize this static member to NULL
// This will get initialized only when the constructor is called
ScXactor * ScXactor::m_pScXator0 = 0;
ScXactor * ScXactor::m_pScXator1 = 0;

ScXactor *ScXactor::GetHandle(int which) {
   return  ((which)?m_pScXactor1:m_pScXactor0);
}
```

top.v

```
ScXactor  ScXactor0(
      .id (1'b0),
      .clk(clk),
.data(data));
ScXactor  ScXactor1(
      .id (1'b1),
      .clk(clk),
.data(data));
```

It appears that the initialization of the SystemC/Verilog wrapper code is not completed by the simulator by the time the constructor executes. As a consequence, the SystemC ports cannot be accessed during initialization. The code in the earlier section will result in either a system crash or hang.

The workaround is to write a separate thread that accesses the ports a little bit after the transactor is constructed. In the following example, the thread `detect_id_thread` reads the id port 1 ns later to identify the instance.

ScXactor.h

```
SC_MODULE(ScXactor)
{
 public:

  sc_in <sc_logic>      id;
  sc_in <sc_logic>      clk;
  sc_inout <sc_lv<16> >    data

  SC_CTOR(ScXactorT):
    id("id"),
    clk("clk"),
    data("data"),
    {
      // The following workaround is needed in case of
      // multiple instantiations. The thread will eventually
      // read the id port and set the pointers to
      // transactor instances accordingly
      SC_THREAD(detect_id_thread);
      …
    }

  void detect_id_thread();
…
};
```

ScXactor.cc

```
// The following are needed to instantiate static members of this class
// Initialize this static member to NULL
// This will get initialized only when the constructor is called
ScXactor * ScXactor::m_pScXator0 = 0;
ScXactor * ScXactor::m_pScXator1 = 0;

ScXactor *ScXactor::GetHandle(int which) {
   return  ((which)?m_pScXactor1:m_pScXactor0);
}

// The following thread reads the assigned value to the id port
// and sets the instance pointers accordingly
void ScXactorT::detect_id_thread()
{
  // Wait for a small time to make sure the SystemC reads the
  // value of id
```

```
  wait(1, SC_NS);

 if (id == 0)
   m_pScXactor0 = this;
 else
   m_pScXactor1 = this;
}
```

The following shows how to access individual ScXactor instances.

```
#include "ScXactor.h"

void foo() {
  ScXactor *pXtr;
  // Make sure this is called after time has progressed by at least 1 unit
  pXtr = ScXactor::GetHandle(0);
  if (!pXtr) {
      // Print error
      exit();
  }
  pXtr->DoFoo();
}
```

3.6 Communicating Between PLI and SystemC Transactors

In the integrated environment, both kinds of transactors can call each other. For example, the legacy transactor can call the SystemC transactors to directly manipulate and observe verilog signals. Similarly, the SystemC transactors can call the legacy transactors to reuse legacy code.

Unfortunately any legacy call that can potentially advance simulation time cannot be called directly from the SystemC transactor code. For example, if the legacy transactor implemented a read method that caused simulation time to advance, the SystemC transactor would be unable to successfully execute that call. Instead, either a core-dump or a hang will occur.

As a workaround, the same call can be implemented using asynchronous calls. In the example below, instead of calling the blocking call to execute a read using the legacy transactors, one can use a combination of non-blocking call and polling:

Instead of calling the following from the SystemC side:

```
read_data = LegacyTransactor.read_blocking(addr);
```

Use the following:

```
LegacyTransactor.read_non_blocking(addr);
while (!LegaacyTransactor.read_done()) {
     wait();
}
```

3.7 Error Reporting

SystemC/SCV provides extensive support for error messaging. On the other hand, a separate reporting infrastructure may also exist within the legacy environment. So, any SystemC simulator

error and warning messages will need to be forwarded to the pre-existing error reporting framework. This is accomplished by overriding the default report handler class `scv_report_handler` to redirect SystemC error messages to the legacy reporting framework.

The following is an example of how to override the `scv_report_handler` class.

```
class ProjectReportHandler: public scv_report_handler {
public:
  static void
    report(
         scv_severity severity,
         scv_msg_type msg_type,
         const char * msg,
         const char *file, int line
         ) ;
};

void
ProjectReportHandler::report(
                       scv_severity severity,
                       scv_msg_type msg_type,
                       const char * msg,
                       const char *file, int line
                       )
{
   string tmStmp;

   tmStmp = sc_time_stamp().to_string();
   switch (severity) {
   case SCV_INFO:
      LEGACY_INFO("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(),  msg);
      break;
   case SCV_WARNING:
      LEGACY_WARNING("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(), msg);
      break;
   case SCV_ERROR:
      LEGACY_ERROR("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(), msg);
      break;
   case SCV_FATAL:
      LEGACY_FATAL("%s:Time: %s: %s\n", msg_type, tmStmp.c_str(), msg);
      break;
   default:
      LEGACY_WARNING("Unknown SCV message type seen at time %s when printing
\n%s\n", tmStmp.c_str(),
             msg);
   }
}
```

And finally, replace the standard scv report handler with an instance of project_report_handler:

```
ProjectReportHandler  project_report_handler;
scv_report_handler::set_handler(project_report_handler);
```

3.8 Using Randomization

The SCV library supports constrained randomization by providing a rich set of related classes, macros, and utilities. The call to randomization is handled by the SCV constraint solver, which reports any failures using the default scv_report_handler, which means they are sent to the standard output or standard error. Such failures must be detected by the legacy verification environment's error reporting infrastructure since otherwise obscure and hard to isolate testbench bugs may show up when executing tests. To make sure these failures are detected, the default SystemC error reporting must be overridden using the approach as described in Section 3.7.

3.9 Using Assertions

Using assertion languages suh as SVA/0in/PSL to embed additional functional checks and improve functional coverage is becoming more commonplace. These assertions are typically provided as comments and either embedded or compiled along with the RTL design. As discussed earlier, the *syscan* utility generates wrapper code that automatically connects the SystemC modules with the Verilog simulator. Some of this wrapper code is in Verilog. Care must be taken that wrapper code the SystemC module instantiations are not included for parsing by the assertion tool. Otherwise many warnings and potentially parse errors may result when running the assertion tools. For example, if using a tool such as 0in, one should enclose the SystemC transactor instantiations in the Verilog code in the following `translate_on/off` macros:

```
 // 0in translate_off
 ScXactor  ScXactor0(
      .id (1'b0),
      .clk(clk),
.data(data));

 ScXactor  ScXactor1(
      .id (1'b1),
      .clk(clk),
.data(data));
 // 0in translate_on
```

The wrapper code generated is ignored by adding the related SystemC module names to the list of modules to be skipped by the assertion tool:

```
+skipmodules+ScXactor+…
```

4.0 Conclusions and Recommendations

It is quite feasible to incorporate SystemC support into a legacy verification environment based on PLI/C++ approach. While it is relatively straightforward to write SystemC code and use many of the advanced features in SystemC, it takes some effort to incorporate the ability to have the SystemC code interact with the DUT directly and thus take full advantage of the HVL features of SystemC. The effort required is reasonable, and the author has successfully followed the solutions outlined in this paper in a real verification environment.

5.0 Acknowledgements

I would like to thank Ronald Goodstein of First Shot Logic Simulation and Design for reviewing this paper and offering me valuable feedback and constructive criticisms.

6.0 References

[1] Stuart Sutherland: *"The Verilog PLI Handbook"*, Kluwer Academic Publishers
[2] Synopsys : VCS/VCSi User Guide Version 7.1.2.
[3] OSCI: SystemC Version 2.0 User's Guide. Update for SystemC 2.0.1. Open SystemC Initiative. www.systemc.org.
[4] OSCI WG Verification: SystemC Verification Standard Specification V1.0p1. www.systemc.org.