SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier

John Aynsley
Doulos
john.aynsley@doulos.com

Abstract- **The OVM and VMM methodologies each provide powerful, flexible and intuitive frameworks for the construction of SystemVerilog verification environments. However, many SystemVerilog users also have models written in C, C++, or sometimes SystemC. Furthermore, the emergence of the SystemC TLM-1 and TLM-2.0 transaction-level modeling standards is having an impact on communication styles within SystemVerlog verification environments. This paper offers practical guidance on using the SystemVerilog Direct Programming Interface (DPI) to integrate existing C, C++ and SystemC code into an OVM- or VMM-based SystemVerilog testbench without jeopardizing portability from one simulator to another. This is achieved by presenting a set of simple, robust guidelines for creating portable DPI code.**

1. Introduction

The requirement to call a C/C++ reference model from a SystemVerilog test bench is not uncommon. A contemporary SystemVerilog test bench would typically be based on either the OVM or the VMM functional verification methodology or sometimes on a homebrew variant of these. One of the tasks performed by such a test bench is to compare the actual behavior of the design-under-test (DUT) against the behavior of a functional reference model, which might originally have been coded in C, C++, or SystemC. A C/C++ algorithm or reference model would necessarily be untimed, but a SystemC model could include timing information.

Against this backdrop, the question of how to call a C/C++ or SystemC reference model from a SystemVerilog test bench is frequently raised. Unfortunately, although each of the above-mentioned languages is defined by a formal standard, the interface between the languages is not standardized with the same precision as the languages themselves. In principle, the SystemVerilog Direct Programming Interface (DPI) permits procedural function calls between SystemVerilog and C. In practice, differences between implementations and the lack of any standard support for calls between SystemVerilog and SystemC mean that any solution needs to be tool-dependent.

1.1 Portability

The goal of this paper is to present a very practical answer to the question of how to use the SystemVerilog DPI to communicate between an OVM or VMM SystemVerilog test bench and a C/C++, or SystemC reference model in a way that is, as far as possible, portable between simulators. The intent is to provide a solution that actually works, today! To this end, we endeavor to offer a set of relatively simple coding guidelines tried-and-tested with current simulator releases. As a consequence, we are forced down the line of addressing practical issues of simulator support for the DPI and the procedural interface between SystemVerilog and SystemC.

The DPI supports simple function calls between SystemVerilog and C. Simulation tool vendors also offer the facility of mixed-language simulation, such as the ability to instantiate a SystemC module from a top-level SystemVerilog module. But mixed-language instantiation alone is insufficient to address the issue of building procedural interfaces between the languages. Tool vendors offer proprietary (non-standard) solutions to the problem of making object-oriented method calls (that is, calling class member functions as opposed to global functions) between SystemVerilog and C++ and of passing transactions between SystemVerilog and SystemC. Anyone wishing to use these facilities is force to chose between "selling their soul" to the EDA vendor in the sense of getting locked into a proprietary solution, or of investigating and using a restricted set of features that form the "lowest common denominator" between tools. We take the latter approach in this paper.

1.2 Transaction-Level Modeling

The SystemC community has developed the TLM-1.0 and TLM-2.0 standards for Transaction Level Modeling in the context of architectural exploration and creating so-called *virtual platform* models of a hardware platform for early software execution. Meanwhile, for functional verification, SystemVerilog test benches written to the OVM or VMM standards also make use of transaction-level modeling (TLM) for internal communication. This means using function calls to pass around objects that encapsulate the attributes of a transaction. The transactions themselves are specific to the interfaces or protocols being modeled. In order to exploit this commonality between the domains of modeling and verification OVM has adopted the TLM-1.0 standard for communication and the latest version of VMM has added internal communication features inspired by TLM-2.0. In this paper, we look to develop an approach to communication between SystemVerilog and C/SystemC that is consistent and interoperable with these TLM standards.

OVM and VMM test benches are transaction-oriented. On the other hand, although C/C++ reference models can also be transaction-oriented, they will sometimes be completely untimed or may perform some transformation on an entire dataset without having any regard for how and when that data is presented in the hardware implementation or the test bench. For example, a reference model for an FFT may transform a dataset from the time domain to the frequency domain. Integrating such a reference model into a test bench requires that the mismatch between transaction-by-transaction processing and batch processing be addressed. There may also be

differences in the degree of pipelining present in the design-under-test and the reference model. Although the solution to many of these issues will be application-specific, in this paper we pick out some specific aspects concerning communication across the SystemVerilog/C interface using the DPI. In particular, we look at the choice between using simple function calls versus TLM-style interfaces, and how to synchronize communication across the DPI.

2. Goals and Assumptions

The goals and assumptions of this work are as follows:

i. There exists a test bench coded in SystemVerilog using OVM or VMM.

ii. There exists an untimed reference model coded in C, C++ or SystemC to be called from the test bench. (Timed SystemC models are also considered as an extension.)

iii. The SystemVerilog DPI is to be used to communicate between SystemVerilog and the reference model

iv. The goal is to achieve portability between simulators by using features that have relatively mature and robust implementations in all simulators.

v. The simulators in question are the production releases of SystemVerilog simulators from Cadence, Mentor and Synopsys current as of November 2009.

vi. The goal of portability is to be achieved by creating a simple and robust set of guidelines that can be easily followed

3. Test Bench and Reference Model

A SystemVerilog test bench typically exercises an RTL model of a Design-Under-Test (DUT) by pin wiggling, that is, by making low level assignments to individual Verilog wires with more-or-less precise timing (the timing could be accurate to the picosecond or merely clock-cycle-accurate). In the OVM and VMM methodologies, the pin wiggling is usually encapsulated within driver and monitor components that communicate with the rest of the test bench using transactions. In VMM this is termed the command layer, in which transactions are simple and atomic. These atomic transactions are generated from higher layers of the test bench which combine these atomic transactions into more complex higher-level transactions according to the details of the interface or protocol being modeled.

In OVM and VMM, transactions may be sent to a so-called scoreboard which collects functional coverage information and checks for functional correctness. It is within the scoreboard that a test bench may need to invoke a reference model to calculate the expected values of the DUT or to analyze the actual values generated by the DUT. The scoreboard is typically expected to receive both stimulus sent to the DUT and the actual response from the DUT transaction-by-transaction at some appropriate abstraction level.

As observed in the introduction, a C/C++ reference model may not be structured to receive transactions one-by-one. Rather, the programming interface to the reference model may consist of a single function call that carries with it an entire dataset, or the dataset may be read from an external file. Thus it may be necessary for the scoreboard to collect a set of incoming transaction and then pass them to the reference model for batch processing. Theoretically this buffering could occur on the SystemVerilog or the C side of the DPI, but for practical reasons explained below we found it most straightforward to batch the data on the C side.

4. Transactions in TLM-1.0 and TLM-2.0

The Open SystemC Initiative (OSCI) TLM-1.0 and TLM-2.0 standards define semantics for passing transactions as function arguments and for managing the lifetimes of transaction objects. These standards can inform our decisions concerning the best way to pass information between SystemVerilog and C. In particular, the solution we chose should be consistent with the use of the TLM standards on either the SystemVerilog or SystemC sides of the DPI interface.

4.1 OVM and TLM-1.0

OVM communication is based on TLM-1.0. The most significant aspect of TLM-1.0-style communication is that each transaction is strictly unidirectional. This is referred to as “effective pass-by-value”, implying that although the C++ implementation does not literally use pass-by-value in every case (sometimes pass-by-reference is used), the transaction object should not be modified by the target. To use a metaphor, TLM-1.0 transactions are thrown over-the-wall from initiator to target, and any communication in the reverse direction requires a separate transaction. Some TLM-1.0 users have chosen to pass pointers to data within the transaction object for simulation efficiency, but agree that any attempt to access shared memory using these pointers is outside the scope of the TLM interface (i.e. entirely the user’s responsibility).

In TLM-1.0, the semantics of blocking transport are to send one unidirectional request transaction from initiator to target, and to receive one unidirectional response transaction back on return. This is fundamentally different from blocking transport in TLM-2.0 where a single transaction object carries both request and response.

In OVM, the TLM-1.0 request/response semantics are particularly evident in the interface between the ovm_sequencer and the ovm_driver. The sequencer sends a request to a driver. The driver implements the transaction by communicating with lower levels of the protocol stack, perhaps by wiggling pins, then when it is done sends a response back to the sequencer.

4.2 VMM and TLM-2.0

VMM communication is based on the vmm_channel, and version 1.2 of the VMM library adds TLM-2.0-style interfaces. Both vmm_channel and TLM-2.0 are based on the idea that transaction objects are passed by reference, and the lifetime of a transaction object may extend across multiple function calls. VMM permits a transaction to remain in a vmm_channel while it is worked on by the target, and the target is permitted to modify the state of the transaction object in order to return a response back to the initiator. The target must explicitly signal the completion of the transaction back to the initiator. VMM and TLM-2.0 are quite similar in these respects.

TLM-2.0 provides two transport interfaces: blocking (b_transport) and non-blocking (nb_transport). With b_transport, the entire transaction is completed within a single method call. With

nb_transport, the progress of a transaction may be described using multiple nb_transport method calls going back-and-forth between initiator and target.

4.3 Consequences for the DPI

For our purposes, the practical consequence of the above discussion is that any communication between a SystemVerilog test bench and a C reference model should be clearly structured into a request and a response (however those may be implemented), and the lifetime and completion of the transaction need to be explicitly considered. A request is passed from SystemVerilog to C, and a response returned back to SystemVerilog.

The easiest case is that of an untimed C/C++ reference model which is called with a dataset, does some calculation, and returns the answer from the very same function call. In this case request, response, and completion occur in a single function call. The request and response do *not* need to be represented as explicit transaction objects, as will be explained below. The important thing is that the distinction between request and response is kept clear, and the completion time is well-defined.

In the case that transactions need to batched to build a dataset for the C/++ reference model, this should be done on the C-side of the interface.

Things get a lot more complicated if the reference model is a SystemC module that provides a blocking interface, because we can get caught up in issues of synchronization between the SystemVerilog and SystemC schedulers. Whatever, the SystemC target needs to send a response back to the SystemVerilog initiator when it is ready, and once more we need to design the interface such that we have a very clear idea of when the transaction is complete.

5. Using the DPI with C

Here we give a brief overview of the DPI just to highlight the pertinent aspects.

The standard SystemVerilog DPI permits function calls between SystemVerilog and C, that is, C functions may be called from SystemVerilog, and SystemVerilog tasks and functions may be called from C.

5.1 Features of the DPI

In order to call a C function from SystemVerilog, the SystemVerilog code must have an *import declaration* which specifies the name, return type, and arguments of the C function. For example

```
// SystemVerilog
import “DPI-C” function int my_func(string s);

initial
  i = my_func(“Hello world\n”);

// C
int my_func( const char* s);
```

In order to call a SystemVerilog task or function from C, the SystemVerilog code must have an *export declaration* which specifies only the name of the task/function. For example

```
// SystemVerilog
export “DPI-C” task my_task;

task my_task;
  #10;
endtask

// C
{ my_task(); }
```

DPI calls can have input, output, and inout arguments, and a return value in the case of a function, all of which support a limited set of data types.

An imported DPI task/function can call an exported DPI/task function, and in the case of a task, that exported task is permitted to execute timing controls (delays or waits). Thus a nested call to a DPI export from a DPI import can suspend the execution of a SystemVerilog process.

A DPI import that calls a DPI export or that makes PLI or VPI calls must be declared as a *context import*. This declaration ensures that the compiler passes information concerning the SystemVerilog context, that is, the location in the SystemVerilog module hierarchy, through the DPI. In the absence of context information, a C function is obliged to call svSetScope() to set the SystemVerilog context.

A principle feature of the DPI is that DPI calls have the same semantics as regular SystemVerilog task/function calls when viewed from SystemVerilog. Moreover, DPI calls are *transparent* when viewed from the SystemVerilog side. This means that DPI task/function arguments use the native SystemVerilog data format, and that nested import -> export call chains that pass SystemVerilog arguments back into SystemVerilog through an intervening C function will be indistinguishable from native SystemVerilog calls that have the same functionality. This makes the DPI very straightforward to use from the SystemVerilog side.

From the C side, things are a little more complicated. Although simple scalar data types have a common representation between SystemVerilog and C, the internal representation of more complicated types is simulator-specific. Thus although SystemVerilog data items of more complex types (such as nested structs and arrays) are guaranteed to pass through the DPI transparently, their C representations are not in general expected to be portable between tools.

Generally, *small* data types have portable representations when passed as DPI arguments. *Small* data types include bit, byte, int, logic, string, and enums (by passing the integral type associated with the enum). Logic vectors, that is, packed arrays of type logic, have a canonical representation in which the two bits that represent the value 0,1,Z,X are split across two separate words using the natural word length of the host. The DPI offers a standardized C programming interface to access and manipulate this canonical representation, with the effect that passing logic vectors as DPI arguments is portable provided that the standard API is used on the C side.

5.2 Practical DPI Restrictions

In principle, nested user-defined structs and arrays composed of the above-mentioned *small* data types may be passed as DPI arguments.

In practice, we found that this capability was not fully supported by all current simulators.

In principle, the DPI offers the ability to pass so-called open array argument in which the size of the array is left unspecified on the SystemVerilog side, and provides a C programming interface to manipulate such open arrays on the C side. Again, we found that not all current simulators provide robust support for this facility.

Aside from current tool limitations, the DPI also has some inherent limitations in the sense that it does not permit SystemVerilog class objects to be passed as DPI arguments, and does not permit SystemVerilog class methods or C++ class member functions to be called directly through the DPI in either direction.

5.3 DPI Guidelines: SystemVerilog-to-C

We were able to create simple, robust, portable DPI code to communicate between SystemVerilog and C by adhering to the following guidelines:

i. Pass only *small types* as DPI arguments, that is, do not pass user-defined structs, open arrays, or multi-dimensional arrays

ii. Logic vectors, i.e. packed arrays of type logic, may be passed as DPI arguments and accessed using the standard C API with the proviso that conditional compilation may be necessary to pick out the appropriate class properties in a portable way. Some simulators refer to the two words of the canonical representation using class properties .a and .b while others use .aval and .bval. For example:

```
// SystemVerilog
#include "svdpi.h"

int print ( svLogicVecVal* v ) {
  int i;
  for (i = 0; i < 8; i++) {
    #ifdef CADENCE
      printf("%d%d", v->a % 2, v->b % 2);
      v->a = v->a >> 1;
      v->b = v->b >> 1;
    #else
      printf("%d%d", v->aval % 2, v->bval % 2);
      v->aval = v->aval >> 1;
      v->bval = v->bval >> 1;
    #endif
  }
  printf("\n");
}
```

iii. In order to pass structs across the DPI, break them down into *small* arguments and re-assemble the struct on the C side if desired.

iv. Declare any imported tasks as *context* tasks.

It should be noted that not all current simulators suffer from the same limitations, and it was found that each simulator had its own idiosyncrasies; none were without fault. However, the explicit goal of this paper was to find a simple, robust, portable approach that works with all simulators. We found that the chosen approach shielded us from having to struggle with the detailed differences between the simulator implementations, and was thus more productive. This same philosophy is taken throughout.

5.4 DPI Guidelines: SystemVerilog-to-C++

The standard SystemVerilog DPI, i.e. "DPI-C", only supports function calls between SystemVerilog and C. However it is very straightforward to make C++ calls simply by forcing the C++ compiler to use C linkage for both imported and exported functions as follows:

```
// C++
#include "svdpi.h"

extern "C"
int imported_function_called_from_SystemVerilog() { … }

extern "C"
int exported_function_called_from_CPP();
```

The only caveat is that the DPI does not permit C++ member functions (methods) to be called.

DPI code such as this can be made portable across all current simulators. All simulators provide the standard DPI header "svdpi.h".

6. Calling an Untimed C/C++ Reference Model

In this section we explore the issues involved in calling an untimed reference model from an OVM or VMM SystemVerilog test bench using a Fast Fourier Transform (FFT) algorithm as an example. The test bench passes around transactions that each represent a single 16-bit sample in the time domain. The intent is to use the FFT reference model to convert a set of samples from the time domain to the frequency domain for analysis. The samples are collected by the scoreboard as the test bench executes, a batch of samples is sent to the FFT reference algorithm, and the scoreboard logs the results of the analysis. The transaction stream in the test bench actually consists of multiple interleaved sample streams, where each individual sample is tagged with a stream number.

The FFT algorithm is primarily coded in C but makes occasional use of C++ features such as stream I/O, so it was necessary to use a C++ compiler and to force the compiler to use C linkage for the DPI functions as described above.

In the original C++ program, the main() function initialized the dataset to be transformed by reading the data from an external text file. We replaced this with two DPI functions, one to initialize the state of the arrays used during the FFT transform, and another to pass individual samples from the SystemVerilog test bench to the C++ model:

```
// SystemVerilog
import "DPI-C" function void initialize_fft(input int n_points);
import "DPI-C" function void transfer_sample(
                        input logic [3:0] tag, logic signed [15:0] data);

// C++
#include "svdpi.h"
extern "C"
void initialize_fft(int n_points);
```

```
extern "C"
void transfer_sample(svLogicVecVal* tag, svLogicVecVal* data);
```

In keeping with the guidelines described above, only *small types* are passed through the DPI. Passing logic vectors is straightforward provided that conditional compilation is used on the C side for portability, again as described above.

As an alternative approach, we investigated batching the data on the SystemVerilog side and passing the entire dataset across the DPI in a single function call, but this approach was fraught with practical difficulties. We attempted to pass a multi-dimensional array as a DPI argument, but trying to unravel the differences between the simulator implementations proved to be an unproductive use of time:

- Simulator 1 required a single level of pointer indirection in the C API
- Simulator 2 required a second level of pointer indirection in the same supposedly standard C API
- Simulator 3 just crashed

The samples are batched on the C side of the DPI (i.e. stored in a static array variable between DPI calls), and the FFT algorithm called when the dataset associated with each sample stream is full. Both of the DPI calls are necessarily non-blocking, that is, the C++ code executes immediately in the context of the calling SystemVerilog process with no simulation time passing.

In terms of the earlier discussion concerning TLM, the call to transfer_sample() passes a request from the test bench to the reference model using input arguments. Although not necessary for this particular example, it is very straightforward to pass a response on return from the DPI function call by using inout or output arguments or the function return value. Using our approach of only passing *small types*, the distinction between request and response can be kept very clean. This approach works fine when called from an OVM or a VMM driver or scoreboard.

Using this approach, it was possible to use precisely the same DPI calls and C++ code with an OVM test bench and simulators from Cadence and Mentor, and with a VMM test bench and the Synopsys simulator. While this study was limited to the methodology/vendor combinations as stated, this same approach should work successfully with other combinations.

7. Using the DPI with SystemC

All current simulators support mixed-language designs, and specifically permit SystemC modules to be instantiated directly from SystemVerilog. All simulators support the ability to make pin-level connections across languages, but unfortunately there is no standard for procedural communication between SystemVerilog and SystemC aside from the DPI.

The native SystemVerilog DPI does not explicitly support SystemC. However, it is possible to use the standard DPI with SystemC in the sense that SystemC is just another C++ application.

SystemC modules typically provide a procedural interface either by implementing a SystemC interface or by offering a SystemC *export*. Using this procedural interface requires the ability to make C++ method calls, which is not possible using the standard DPI. Two vendors, Cadence and Mentor, both support an extended version of the DPI called "DPI-SC" aimed at overcoming this restriction. Synopsys achieves similar functionality by extending the capability of "DPI-C" and by offering the TLI, or Transaction-Level Interface.

If "DPI-SC" were used to make direct SystemC interface method calls from SystemVerilog, it may well be necessary to modify the argument types of the method calls, for two reasons. Firstly, SystemC methods often pass transaction objects, which is not possible with the DPI. Secondly, we may want to restrict the DPI arguments to *small types* for portability. As a consequence, it is not necessarily desirable to make direct SystemC interface method calls from SystemVerilog, and in general a better approach may be to instantiate the original target SystemC module from a wrapper module on the SystemC side, where that wrapper adheres to our guidelines for portable DPI use.

As a further practical difficulty, although "DPI-SC" is supported by both Cadence and Mentor, there are differences of detail between the two implementations that would restrict portability when making class method calls. (However, the mutual implementation of calls to global C++ functions seems more robust, as described below.)

7.1 Timing across SystemVerilog and SystemC

DPI calls from SystemVerilog run in the context of the SystemVerilog scheduler, and C functions are intrinsically non-blocking; a DPI call can only suspend execution by making a call back to a DPI task exported from SystemVerilog. But SystemVerilog and SystemC each have their own schedulers, and there is no standard programming interface for controlling the interactions between those schedulers. Specifically, a DPI call from SystemVerilog to SystemC *must not block*, or at least if it does call sc_core::wait(), the code cannot be expected to be portable.

Calling into SystemC models from SystemVerilog is simplest if the SystemC models are untimed. In that case we can use the same paradigm as for any untimed C++ model, that is, a DPI call from SystemVerilog passes a request to an untimed model, and a response is passed back on return from that or from a subsequent DPI call.

7.2 DPI Guidelines: SV-to-SC, untimed, one instance

We were able to create simple, robust, portable DPI code to communicate between SystemVerilog and a single instance of an untimed SystemC module by adhering to the following guidelines (in addition to those given above):

i. Use "DPI-C", not "DPI-SC"

ii. Use "DPI-C" imports only, not exports.

iii. Cadence, and only Cadence, requires a dummy SystemVerilog module stub corresponding to the SystemC module:

```
// SystemVerilog
`ifdef NCSC
   module scwrap ()
      (* integer foreign = "SystemC"; *);
   endmodule
`endif
```

iv. Put the SystemC module class definition in a header file named module_name.h and any member function definitions in a file module_name.cpp

v. Define any static variables or static members in the module_name.cpp file, not in the header file.

vi. Include the following conditional compilation directives in the module_name.cpp file for any SystemC module:

```
// SystemC
#ifdef CADENCE
NCSC_MODULE_EXPORT( module_name );
#endif

#ifdef MENTOR
SC_MODULE_EXPORT( module_name );
#endif
```

vii. Avoid intrusive changes to the original SystemC module by instantiating it from a SystemC wrapper module that implements the DPI funtionality.

viii. Within the wrapper module, create a static data member that stores a pointer to the C++ module instance *this*, and initialize the pointer from the constructor. This technique only works for a single module instance.

ix. Have the C side DPI function assemble a transaction from its arguments as necessary, pass the transaction to the corresponding method of the SystemC wrapper, and set any output arguments on return:

```
// C++
scwrap* scwrap::instance = 0;  // Static member

extern "C"
void entry(unsigned char cmd, int addr, int* data)
{
  bus_t tx;
  tx.cmd = cmd;
  tx.addr = addr;
  tx.data = *data;
  scwrap::instance->execute( &tx );
  *data = tx.data;
}
```

7.3 DPI Guidelines: SV-to-SC, multiple instances

The above approach can easily be extended to handle multiple instances of the same SystemC module. Instead of having a single instance pointer in the wrapper pointing to the module itself, have a vector of pointers:

```
// SystemC
class scwrap: public sc_module
{
public:
  scwrap(sc_module_name n);

  scmod* m_scmod;
  static std::vector<scwrap*> instance;
  static int count;
  int id;
};
```

The module constructor populates the vector of instance pointers:

```
// C++
scwrap::scwrap(sc_module_name n)
: sc_module(n)
{
  m_scmod = new scmod("m_scmod");
  id = ++count;

  instance[id] = this;
}
```

The DPI function can now use the id member to send calls to the correct target instance:

```
// C++
extern "C"
void entry(int id, unsigned char cmd, int addr, int* data)
{
  bus_t tx;
  tx.cmd = cmd;
  tx.addr = addr;
  tx.data = *data;
  scwrap::instance[id]->m_scmod->execute( &tx );
  *data = tx.data;
}

extern "C"
const char* get_path(int id)
{
  return scwrap::instance[id]->name();
}
```

On the SystemVerilog side, the id is passed to the DPI call in order to differentiate between the SystemC module instances. A get_path() DPI function is provided so that the SystemVerilog test bench can identify which id corresponds to which instance. The test bench can maintain a mapping between hierarchical paths and ids to make DPI calls more convenient:

```
// SystemVerilog
import "DPI-C" function void entry(
          input int id, input byte cmd, input int addr, inout int data);
import "DPI-C" function string get_path(input int id);

int path_to_id[string];
…
virtual function void end_of_elaboration;
   path_to_id[get_path(1)] = 1;
   path_to_id[get_path(2)] = 2;
endfunction
```

Unfortunately, differences between simulator implementations cause portability issues again. The SystemC name() method returns the hierarchical path in a different format in each case! Conditional compilation can be used to patch up the differences:

```
// SystemVerilog
`ifdef CADENCE
    id = path_to_id["top.m_hier.m_scwrap_1"];
`endif
`ifdef MENTOR
    id = path_to_id["/top/m_hier/m_scwrap_1"];
`endif
`ifdef SYNOPSYS
    id = path_to_id["top_m_hier_m_scwrap_1"];
`endif
entry(id, tx.cmd, tx.addr, tx.data);
```

What can one say?

Using this approach, with minor use of conditional compilation it was possible once again to use precisely the same DPI calls and C++ code with an OVM test bench and simulators from Cadence and Mentor, and with a VMM test bench and the Synopsys simulator.

7.4 DPI Guidelines: Timed SystemC Models

As mentioned at the outside, one premise of this paper was the requirement to integrate untimed reference models into a SystemVerilog test bench. However, as an extension, we now consider timed SystemC models.

It is increasingly the case that timed SystemC models conform to the OSCI TLM-2.0 standard. As described above, TLM-2.0 supports both blocking and non-blocking transport interfaces for processing regular transactions. Even models that do not conform to the TLM-2.0 standard typically use these same concepts.

To handle a timed SystemC model, because the DPI call itself must not be blocked by the SystemC scheduler (as discussed above), the response may need to be returned at a later time using a separate DPI call from SystemC to SystemVerilog. Hence we require both DPI imports and DPI exports between SystemVerilog and SystemC.

The approach we have adopted is to use two DPI calls, a DPI import function that sends a request to the SystemC model (by passing *small types* as function arguments), and a DPI export function that sends a response back to the SystemVerilog test bench at a later time (again by passing *small types* as function arguments). Both can be simple non-blocking function calls. This mechanism has similarities with the TLM-2.0 non-blocking interface, and like that interface, is able to support multiple simultaneous pipelined transactions. However, our approach does not require that either the SystemVerilog or the SystemC model be TLM-2.0-compliant.

As a purely practical matter, linking SystemVerilog and SystemC applications with function calls in both directions seems best handled by switching to "DPI-SC" for Cadence and Mentor, while "DPI-C" is sufficient for Synopsys. As mentioned above, currently there are portability issues when making SystemC method calls using "DPI-SC", but the implementation of global C++ function calls through the "DPI-SC" interface (or "DPI-C" for Synopsys) seem to be robust for all simulators. Synopsys users also have the option of using the TLI, but a premise of this paper is the desire to create portable code and to minimize reliance on vendor-specific features.

To call global C++ functions at "DPI-SC" imports requires the use of vendor-specific function registration macros as follows:

```
// SystemC
#ifdef CADENCE
  #define NCSC_INCLUDE_TASK_CALLS
  #define CDN_OR_MEN
#endif

#ifdef MENTOR
  #define MTI_BIND_SC_MEMBER_FUNCTION
  #include "sc_dpiheader.h"
  #define CDN_OR_MEN
#endif

// In module constructor
#ifdef MENTOR
   SC_DPI_REGISTER_CPP_FUNCTION(entry);
   SC_DPI_REGISTER_CPP_FUNCTION(get_path);
#endif

// At file scope
#ifdef CADENCE
   NCSC_REGISTER_DPI(entry)
   NCSC_REGISTER_DPI(get_path)
#endif
```

In order to call a DPI export function from C++, we need to set the SystemVerilog scope. The programming interface to achieve this (svGetScope, svSetScope) is part of the SystemVerilog language standard. The scope can be determined from within the DPI import function, and then used subsequently for the return call to the DPI export:

```
// C++
static svScope  calling_scope;

// DPI import
extern "C"
void entry(int id, char cmd, int addr, int* data) {
  …
  calling_scope = svGetScope();
}

…
{
   svSetScope(calling_scope);

   // Call to DPI export
   tx_done(id, tx->cmd, tx->addr, tx->data);
}
```

In order to deal with multiple outstanding transactions, the SystemC wrapper module maintains a pool of SystemC threads. Incoming transactions are allocated to the next free thread, and a thread remains tied to a transaction instance until the response has been returned to the SystemVerilog test bench and the transaction completed. The thread can then be returned to the pool and reused. Each thread has an associated event which, when notified, causes the thread to resume and process a new incoming transaction.

Each incoming transaction is handled by the SystemC wrapper as follows:

i. Assemble a new transaction object from the DPI arguments
ii. Associate the transaction object with the next free thread
iii. Notify an event to cause the thread to resume

Each SystemC thread executes the following loop:

i. Wait for an event notification
ii. Call the blocking method of the SystemC module instance to implement the functionality requested by the test bench
iii. On return from the blocking method call the DPI export, unpacking any response attributes from the transaction and passing them as DPI arguments
iv. Delete the transaction object

On the SystemVerilog side, the implementation of the DPI export method can handle the response as follows:

i. Check the response attributes passed as arguments
ii. Use the id argument to tie up with the request, for example by executing a Verilog event trigger ->done_ev[id]
iii. The transactor that generated the request can either block waiting for the response or can generate several pipelined transactions without waiting

There are some key points to note from the above: the actual DPI calls are all non-blocking; the blocking method of the target SystemC module is called from a SystemC thread in the wrapper; a thread pool is needed to support multiple pipelined transactions; and an id argument is used to map global DPI calls to the module hierarchy (see the description of multiple instances above).

On the SystemVerilog side, the lifetime of each transaction will extend beyond the call to the DPI import. Hence it is essential that the test bench creates a new transaction object per transaction rather than reusing the same object, which may still be "alive". (This is considered good practice anyway in both OVM and VMM.) The DPI export call that indicates the completion of the transaction is necessarily a global function call, rather than being a class method call to the OVM component or VMM transactor that initiated the transaction, so the programmer must contrive some mechanism to communicate between the incoming DPI call and the initiator, based on the id argument that is common to the outgoing and incoming DPI calls. From the SystemVerilog side, the incoming DPI export call is effectively asynchronous with respect to the operation of the test bench. The initiating component/transactor can either suspend until it is notified of the completion of the transaction, or can continue to generate new transactions.

Garbage collection needs to be considered. Garbage collection in SystemVerilog is implicit, but on the C++ side the lifetime of transaction objects needs to be carefully thought through such that objects can be either deleted or re-used when they die.

Once again, with this approach it was possible to use precisely the same C++ code with an OVM test bench and simulators from Cadence and Mentor, and with a VMM test bench and the Synopsys simulator. On the SystemVerilog side, the DPI calls differed only in that Cadence and Mentor used "DPI-SC" and Synopsys "DPI-C".

8. Conclusions

We set out to find a way of using the SystemVerilog DPI to call C/C++ and SystemC reference models in a way that was simple, robust, and portable between simulators. On the whole, the results were very positive. We found an approach by which we were able to create a single body of C/C++ or SystemC code that is portable between all current simulators, albeit by imposing some restrictions on the DPI features used.

It is true that the lowest common denominator between DPI implementations uncovered by this work is substantially below the full level of capability of any of the individual simulators considered, and also true that this level will change over time as vendors improve their implementations. Users must make their own decision concerning the value of portability between simulators versus exploiting the full capabilities of their chosen vendor.

8.1 Summary of the Common Approach

i. Pass only *small types* and logic vectors as DPI arguments, that is, do not pass user-defined structs, open arrays, or multi-dimensional arrays

ii. For untimed reference models (C/C++ or SystemC), use "DPI-C" imports only, that is, do not use DPI exports or "DPI-SC"

iii. For SystemC reference models, use a SystemC wrapper module that can assemble or convert function arguments and re-direct incoming DPI calls to the appropriate SystemC module instance

iv. Have the DPI calls identify the target SystemC module instance using an id argument

v. For timed SystemC models, use "DPI-SC" (Cadence and Mentor) or "DPI-C" (Synopsys)

vi. For timed SystemC models, have the SystemC wrapper maintain a pool of SystemC threads to call any blocking SystemC method

vii. For timed SystemC models, have the SystemC wrapper call a DPI export on completion of the transaction

viii. Have all DPI calls in either direction be non-blocking

ix. Use conditional compilation as necessary to handle differences in directives, headers, macro names, and hierarchical path names between simulators

8.2 Example Files

Example files to accompany this paper can be downloaded from www.doulos.com/knowhow/systemverilog/

9. References

[1] IEEE Std 1800-2005 "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language,"

[2] IEEE Std 1666-2005 "IEEE Standard SystemC Language Reference Manual"

[3] OSCI TLM-2.0 Language Reference Manual, version JA32, 2009

[4] Bergeron, Cerny, Hunter and Nightingale. 2005 Verification Methodology Manual for SystemVerilog. Springer ISBN-10: 0-387-25538-9

[5] VMM Standard Library User Guide, Version 1.2, November 2009

[6] OVM Class Reference, Version 2.0.2, June 2009

