Integrating SystemC Models with Verilog and SystemVerilog Models Using the SystemVerilog Direct Programming Interface

Stuart Sutherland
Sutherland HDL, Inc.
stuart@sutherland-hdl.com

ABSTRACT

The Verilog Programming Language Interface (PLI) provides a mechanism for Verilog simulators to invoke C programs. One of the primary applications of the Verilog PLI is to integrate C-language and SystemC models into Verilog simulations. But, Verilog's PLI is a complex interface that can be daunting to learn, and often slows down simulation performance. SystemVerilog includes a new generation of a Verilog to C interface, called the "Direct Programming Interface" or "DPI". The DPI replaces the complex Verilog PLI with a simple and straight forward import/ export methodology.

- *How does the SystemVerilog DPI work, and is it really easier to use than the Verilog PLI?*
- *Can the SystemVerilog DPI replace the Verilog PLI for integrating C and SystemC models with Verilog models?*
- *Is the SystemVerilog DPI more efficient (for faster simulations) that the Verilog PLI?*
- *Is there anything the SystemVerilog DPI cannot do that the Verilog PLI can?*

This paper addresses all of these questions. The paper shows that if specific guidelines are followed, the SystemVerilog DPI does indeed simplify integrating SystemC models with Verilog models.

Table of Contents

1.0 What is SystemVerilog?

SystemVerilog is *not* a new Hardware Description Language (HDL). SystemVerilog is a standard set of extensions to the existing Verilog HDL. These extensions address two essential needs in today's design projects: modeling more hardware logic with fewer lines of code, and writing more verification logic with fewer lines of code. For a more complete discussion of the SystemVerilog extensions to Verilog, refer to the paper presented at the 2003 SNUG-Europe conference, "*HDVL += (HDL & HVL): SystemVerilog 3.1, The Hardware Description AND Verification Language*" [6]. The complete definition of the SystemVerilog extensions can be found in the "*Accellera SystemVerilog Language Reference Manual*" [2]. Accellera plans to donate the SystemVerilog standard to the IEEE 1364 Verilog standards group in June of 2004. It is expected that all the SystemVerilog extensions to the Verilog language will become part of the next IEEE Verilog standard.

In general, this paper uses the term "*Verilog*" to mean the Verilog HDL *with* the SystemVerilog extensions. The code examples in this paper do not distinguish which constructs are Verilog and which constructs are SystemVerilog extensions to Verilog.

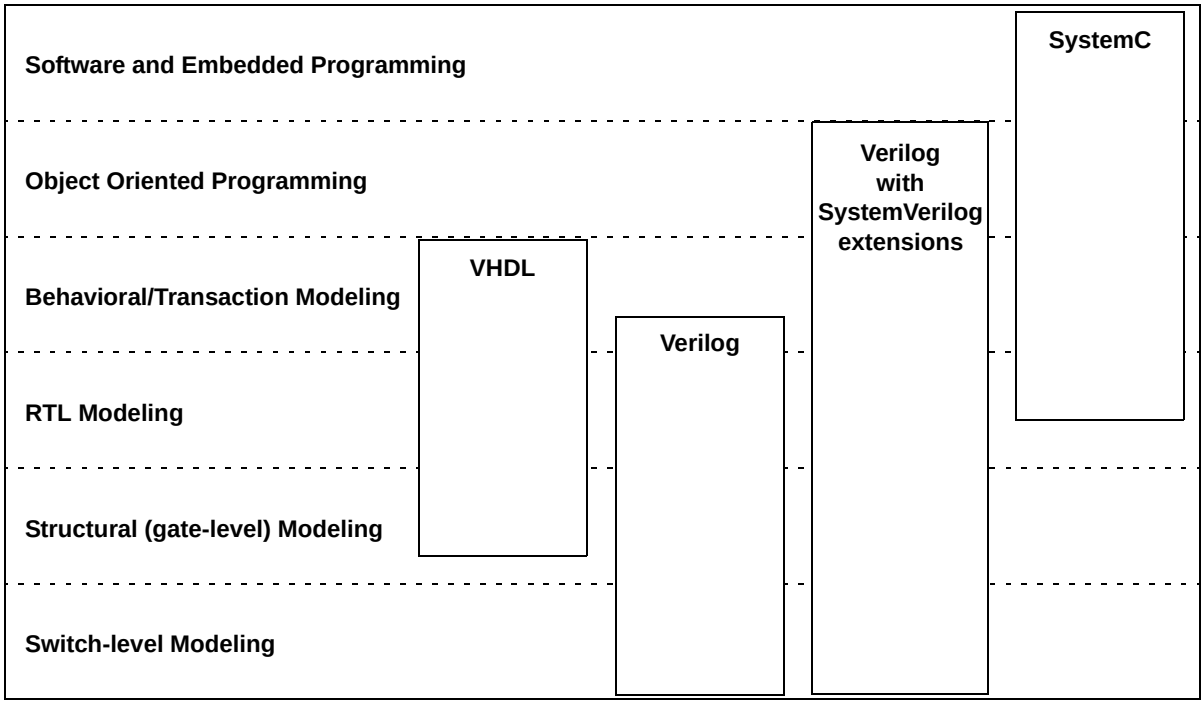
2.0 Why integrate SystemC models with Verilog models?

Today's hardware design paradigms typically involve creating models of an intended design before the hardware is actually built. Simulations are then run on these models to verify that the design behavior is as intended. The most common modeling languages today are SystemC, VHDL, and Verilog. Each language has unique strengths, weaknesses, advantages and disadvantages (and often strong user preferences).

Figure 1 provides a high-level comparison of where Verilog, Verilog with the SystemVerilog extensions, VHDL and SystemC fit in the hardware modeling paradigm. Three key points should be observed in this chart:

- SystemVerilog does not replace SystemC — each language has unique capabilities.
- SystemVerilog bridges a gap between Verilog and SystemC.
- The overlap between SystemVerilog and SystemC makes it easier to mix models written in each language within the same simulation.

Figure 1. Design language overlap



This paper does not compare or promote the use of any particular modeling language. It is simply assumed that, for whatever reason, designs will often comprise a mix of modeling languages. The question posed and answered in this paper is how can the SystemVerilog Direct Programming Interface be used to handle a mix of SystemC and Verilog models in the same simulation.

2.1 Standard and proprietary interfaces

The standard application interface between Verilog and the C language is the Verilog Programming Language Interface (PLI). Because the PLI is part of the Verilog standard, a Verilog to SystemC integration is portable to all IEEE compliant simulators. However, the PLI has a number of disadvantages, which are outlined later in this paper. To overcome the disadvantages of the Verilog PLI, many simulator companies provide proprietary mechanisms to integrate Verilog models and SystemC models in the same simulation. These proprietary methods overcome the shortcomings of the Verilog PLI, but at the cost of portability. Source code in either or both the Verilog models and the SystemC models must be customized for each proprietary simulator used within a company.

3.0 An overview of the SystemVerilog DPI

The SystemVerilog Direct Programming Interface (DPI) allows Verilog code to call the names of C functions as if the function were a native Verilog task or function (a task or function defined in the Verilog language). This is done by importing the C function name into the Verilog language, using a simple **`import`** statement. The import statement defines that the function uses the DPI interface, and contains a prototype of the function name and arguments. For example:

```
import "DPI" function real sin(real in); // sine function in C math library
```

This import statement example defines the function name `sin` for use in Verilog code. The data type of the function return is a `real` value (double precision) and the function has one input, which is also a `real` data type. Once this C function name has been imported into Verilog, it can be called the same way as a native Verilog language function. For example:

```
always @(posedge clock) begin
   slope <= sin(angle);  // call the "sin" function from C
end
```

The imported C function must be compiled and linked into the Verilog simulator. This process will vary with different simulators. With the Synopsys VCS simulator, the process is very straightforward; the C source file name, or a pre-compiled object file name, is listed along with Verilog source code files as part of the `vcs` compile command.

With the SystemVerilog DPI, the Verilog code is unaware that it is calling C code, and the C function is unaware that it is being called from Verilog.

DPI compared to PLI. The ability to import a C function and then directly call the function using the DPI is much simpler than the Verilog PLI. With the PLI, users must define a *user-defined system task* or *user-defined system function.* The user-defined system task/function name must begin with a dollar sign ($). For example, the sine function above might be represented in Verilog with `$sine`. This function is then associated with a user-supplied C function known as a *calltf routine*. The calltf routine can then invoke the `sin` function from the C math library. The calltf routine, and any functions it calls, must then be compiled and linked into the Verilog simulator. Once these steps have been performed, the `$sine` system function can then be called from within Verilog in the same way as a regular function. When simulation executes the call to `$sine`, it will invoke the C calltf function that can then call the C `sin` function.

```
always @(posedge clock) begin
   slope <= $sine(angle);  // call the sine PLI application
end
```

At first glance, it might seem that within the Verilog code, there is very little difference between using the PLI function and an imported DPI function. Indeed, within the Verilog code, there is no significant difference; in both cases, the Verilog code is calling a task or function. Where the difference becomes apparent—and it is a significant difference—is in what it takes to define the PLI user-defined system task or system function. Using the DPI, the Verilog code can directly call the `sin` function from the C math library, directly pass inputs to the function, and directly receive a return value from the C function. Using the PLI, several steps are required to create a user-defined system task that indirectly calls and passes values to the sin function. This indirect process is described in more detail in the following section.

4.0 Verilog PLI capabilities

In order to fully discuss the strengths and weaknesses of the SystemVerilog DPI, it is necessary to understand the capabilities of the Verilog PLI, and the C libraries that are part of the PLI standard.

The Verilog PLI is a simulation interface. It provides run-time access to the simulation data structure, and allows user-supplied C functions a way to access information within the simulation

data structure. This user-supplied C function can both read and modify certain aspects of the data structure. The Verilog PLI does not work with Verilog source code. It only works with a simulation data structure. The PLI is not designed for use with tools other than simulation, such as synthesis compilers.

4.1 How the PLI calls C functions

The Verilog PLI provides a set of C language functions that allow programmers to access the data structure of a Verilog simulation. These C functions are bundled into three PLI libraries, referred to as the *TF library,* the *ACC library* and the *VPI library.* The PLI access to the simulation data structure is dynamic, in that it occurs while simulation is running. The access is also bidirectional. Through the PLI, a programmer can both read information from the simulator's data structure, and modify information in the data structure.

The Verilog PLI allows programmers to extend the Verilog language though the creation of *system tasks and system functions*. The names of these user-defined system tasks and functions must begin with a dollar sign ($). A task in Verilog is analogous to a subroutine. When a task is called, the simulator's instruction flow branches to a subroutine. Upon completion of the task, the instruction flow returns back to the instruction following the task call. Verilog tasks do not return values, but can have input, output and inout (bidirectional) formal arguments. A function in Verilog is analogous to functions in most languages. When a function is called, it executes a set of instructions, and then returns a value back to the statement that called the function. Verilog tasks and functions can be defined as part of the Verilog language.

Using the Verilog PLI, a programmer must first define a system task/function name, such as `$sine`. The programmer then creates a C function referred to as a *calltf routine,* which will be associated with the `$sine` task/function name. When simulation executes the statement with the `$sine` system function call, the simulator calls the calltf routine associated with `$sine`. This calltf routine is a layer between `$sine` and the `sin` function in the C math library. The arguments to $sine are not passed directly to the calltf routine. Instead, the calltf routine must call special PLI functions from a PLI library to read the input argument of `$sine`. The calltf routine can then call the `sin` function, passing the input to the function. The calltf routine will receive the return value from the `sin` function, and then call another special PLI function to write this return value back to the `$sine` function. A final step when using the Verilog PLI is to bind the user-defined system task/function name with the user-defined calltf routine. This step is different for each simulator, and can range from editing a special table file to editing and compiling a complex C language file.

The PLI libraries allow a calltf routine to do more than just work with the arguments of a system task or function. The libraries also allow a calltf application to search for objects in the simulation data structure, modify delays and logic values in the data structure, and synchronize to simulation activity and simulation time.

4.2 The PLI TF, ACC and VPI libraries

The Verilog PLI provides a set of C language functions that allow programmers to access the data structure of a Verilog simulation. These C functions are bundled into three PLI libraries, referred to as the *TF library,* the *ACC library* and the *VPI library.*

The TF library is the oldest generation of the Verilog PLI. This library serves two primary purposes: First the TF library allows C programs to read the values of system task/function arguments and to write values back to those arguments. Second, the TF library allows C programs to synchronize to simulation events and simulation time.

The ACC library extends the TF library to allow C programs to analyze the structural aspects of a simulation data structure. For example, a C program can traverse the simulation data structure to find all instances of ASIC cells, and then modify delay information for each cell instance.

The VPI library both supersedes and extends the older TF and ACC libraries. With the VPI library, users are given full access to the entire simulation data structure. This includes the arguments of system tasks and functions (replacing the TF routines) and the structural objects of a design (replacing the ACC routines). In addition, the VPI library can access all RTL and behavioral statements that make up a design. The VPI library can also access the simulator's event scheduler, and add or remove scheduled events, which both replaces and enhances the synchronization capabilities of the TF and ACC libraries. The SystemVerilog standard extends the VPI library to also support all SystemVerilog extensions to the Verilog language.

The PLI access to the simulation data structure is dynamic, in that it occurs while simulation is running. The access is also bidirectional. Through the PLI libraries, a programmer can both read information from the simulator's data structure, and modify information in the data structure.

4.3 PLI indirect access

The PLI libraries serve as a protecting layer between the simulation data structure and a user's C program. For example, when Verilog code calls a PLI application, function arguments are not directly passed to the C program, and the C program cannot directly return values back to Verilog. With the Verilog PLI, a programmer must first define a system task function name, such as `$sine`. The programmer then creates a C function referred to as a *calltf routine*, which will be associated with the `$sine` task/function name. When simulation executes the `$sine` system function call, the simulator calls the calltf routine associated with `$sine`. The arguments to `$sine` are not passed directly to the calltf routine. Instead, the calltf routine must call special functions from a PLI library to read the input argument of `$sine`. The calltf routine can then call the C `sin` function. The calltf routine will receive the return value from the `sin` function, and then call another special PLI function to write this return value back to the `$sine` function. By not allowing the user's C code to directly pass values to and from Verilog simulations, the PLI protects the Verilog simulation from any harmful affects of an ill-behaved C program.

The PLI libraries allow a calltf routine to do more than just work with the arguments of a system task or function. The libraries also allow a calltf application, or C functions that are called from a calltf application, to search for objects in the simulation data structure, modify delays and logic values in the data structure, and synchronize to simulation activity and simulation time. Common applications of the Verilog PLI in today's engineering environments include: commercial and proprietary waveform viewers, commercial and proprietary design debug utilities, RAM/ROM program loaders, scan chain vector loaders, custom file readers and writers, power analysis, circuit tracing, C model interfaces, co-simulation environments, parallel process distribution, and much, much more. The ways in which the PLI can be used to extend Verilog simulators is limited only by the imagination of programmers.

5.0 A more detailed look at the SystemVerilog DPI

As introduced at the beginning of this paper, a key feature of the DPI is that it allows Verilog code to directly call C functions, without the complexity of creating and defining a system task or function name and an associated calltf routine. With the DPI, values can be directly passed to the C functions, and received directly back from the C functions. These direct calls and directly passing values to and from C functions can be done without the use of any procedural interface libraries. This makes the DPI much more straightforward and easy to use than the PLI's TF, ACC, and VPI interfaces. It should be noted, however, that the DPI does provide its own procedural libraries to aid in the conversion of complex data types (discussed in section 5.4).

The DPI standard has its origins in two proprietary interfaces, the VCS *DirectC* interface from Synopsys, and the SystemSim *Cblend* interface from Co-Design Automation (later acquired by Synopsys). These two proprietary interfaces were originally developed to work with their respective simulator, and not as a standard that would work with any simulator. The Accellera SystemVerilog standards committee merged the capabilities of the two donated technologies together, and defined the DPI interface semantics in such a way as to ensure the DPI could work with any Verilog simulator.

5.1 The DPI import declaration

The DPI import declaration defines a prototype of the C function name, arguments and function return type. A C function can be imported as either a Verilog task, or as a Verilog function. A task in Verilog can have input, output and bidirectional inout arguments, but does not have a return value. Tasks are called as programming statements from Verilog procedural code. Functions differ from tasks in that they have a return value, as well as input and output arguments. Functions can be called anywhere an expression can be used in Verilog code.

Two example import declarations are:

```
import "DPI" function real sin(real in); // sine function in C math library
import "DPI" task file_write(string data); // user-supplied C function
```

The DPI import declaration allows a local name to be used to represent the C function name. This local name can be useful if the C function name would conflict with other names in the Verilog model. For example, the `sin` C function above could be given the name of `sine_func` in Verilog:

```
import "DPI" sine_func = function real sin(real in);
```

The DPI import declaration can be anywhere a native Verilog function can be defined. This includes modules, interfaces, program blocks, packages, and the compilation-unit space. The imported task or function name is local to the scope in which it is imported. It is legal to import the same C function in multiple scopes, such as in multiple modules, as long as each DPI import declaration of the same C function name has exactly the same prototype. If an imported C function is to be used in multiple scopes, a good coding style is to define the import declaration in a SystemVerilog package. The package can be used in any number of modules, interfaces and program blocks, without having to duplicate the DPI import declaration.

5.2 Function formal arguments

Imported C functions can have any number of formal arguments, including none. By default, each formal argument is assumed to be an input into the C function. The DPI import declaration can override this default, and explicitly declare each formal argument as an **`input`**, **`output`** or bidirectional **`inout`** argument. In the following example, a square root function is defined to have two arguments: a double precision input, and a 1-bit output value that represents an error flag.

```
import "DPI" function real sqrt(input real base, output bit error);
```

From the Verilog code perspective, input and output formal arguments appear the same as with a task or function defined in the Verilog language. That is, the behavior is as if input argument values are copied into the task or function when it is called, output argument values are copied out of the task or function when it returns, and inout arguments are copied in at the call, and copied out at the return. This copy-in and copy-out behavior does not impose the actual implementation of the DPI by Verilog simulators. It merely describes the effect that is seen within the Verilog code. Implementation might use pointers or other methods to optimize simulation performance.

A formal argument that is declared as an input can only have values passed into the C function. The C function should not modify its copy of the argument value. This can be assured by declaring the input arguments of the C function as **`const`** variables.

5.3 Function return values

A C function that is imported as a function into Verilog can have any return value type that is legal in the C language, including pointers. SystemVerilog adds a special **`chandle`** data type for importing C functions that return a pointer data type.

5.4 Data type restrictions

On the Verilog side, the DPI restricts the data types that can be passed to a C function's formal arguments, and that can be used as the return types of imported functions. The legal data types are: **`void`**, **`logic`**, **`bit`**, **`byte`**, **`shortint`**, **`int`**, **`longint`**, **`real`**, **`shortreal`**, **`chandle`**, and **`string`**. These data types can also be used as function return types.

Verilog vectors of **`reg`**, **`logic`** and **`bit`** data types can be defined with arbitrary widths, from 1-bit wide to 1 million bits wide. When passed in or out of a C function, these data types require special handling. How these vectors are represented in C can be complex, and is beyond the scope of this paper. In brief, the DPI includes a library of special C functions to aid in the conversion of Verilog vectors to a representation in C. For example, one way in which simple Verilog vectors can be represented in C is as an array of integers, where each integer represents 32 bits of the Verilog vector. The 4-state logic of Verilog **`reg`** and **`logic`** data types are encoded as an array of integer pairs in C. The mapping of Verilog vectors into arrays of integers makes it more difficult to work with Verilog vectors within C.

Verilog structures and arrays can also be passed into and out of imported C functions. When using structures and arrays, the DPI imposes several limitations. Briefly, unpacked Verilog structures and arrays that use data types compatible with C are easy to pass in and out of C. They are represented as equivalent structures and arrays in C. More complex Verilog structures and arrays

are more difficult to work with using the DPI interface, as there is no equivalent representation of this data in the C language. Another consideration is that Verilog arrays can start with any arbitrary address, whereas C arrays always begin with address 0. The SystemVerilog LRM [2] discusses how complex structures and arrays can be passed to and from C functions using the DPI.

Caution! The DPI places the burden on the user of the DPI to match the data types used in the C language with equivalent data types on the Verilog side (at the import statement). For example, an **`int`** on the C function side should be declared as an **`int`** in the DPI import prototype. A **`double`** on C function side should be declared as a **`real`** in the DPI import statement, etc. The DPI does not provide a mechanism for the C function to test what type of value is on the Verilog side. The C function simply reads or writes to its arguments, unaware that it was actually called from the Verilog language. If the Verilog prototype does not match the actual C function, the C function might read or write erroneous values.

This strict and potentially unforgiving declaration requirement is very different than with the Verilog PLI. The PLI provides mechanisms for the C function to test the data types of system task/ function arguments. The PLI application can then adapt how values are read or written based on the data types used on the Verilog side. In addition, The PLI functions that read or write values perform automatic conversions of values from one data type to another.

5.5 Pure, context and generic C functions

The DPI allows C functions to be classified as *pure functions*, *context functions*, or *generic functions*.

Pure C functions. With a pure function, the results of the function must depend solely on values that are passed into the function through formal arguments. An advantage of pure functions is that simulators may be able to perform optimizations that improve simulation performance. A pure function cannot use global or static variables, cannot perform any file I/O operations, cannot access operating system environment variables, and cannot call any other functions. Only non-void functions with no output or inout arguments can be specified as pure. Pure functions cannot be imported as a Verilog task. An example of declaring an imported C function as pure is:

```
import "DPI" pure function real sin(real in); // function in C math library
```

Context C functions. A context C function can use global and static variables, and can call other C functions. Context functions are aware of the Verilog hierarchy scope in which the function is declared. This allows an imported C function to call functions from the PLI TF, ACC or VPI libraries, which makes it possible for DPI functions to take advantage of PLI features such as writing to the simulator's log file and files that are opened from within Verilog source code. An example declaration of a context-dependent task is:

```
import "DPI" context task print(input int file_id, input bit [127:0] data);
```

Generic C functions. By default, the DPI considers an imported C function to a *generic function*. (The SystemVerilog standard does not use the term "generic"). A generic C function can be imported as either a Verilog function or a Verilog task. The task or function can have input, output

and inout arguments. Functions can have a return value, or be declared as void. A generic C function can call other functions, but cannot call any function from the Verilog PLI libraries that require a sense of context.

Caution! It is the user's responsibility to correctly declare an imported function pure or context. By default, DPI functions are assumed to be generic. A call to a C function that was incorrectly declared as pure may return incorrect or inconsistent results, and can cause unpredictable run-time errors, even crashing the simulation. Similarly, if a C function accesses the Verilog PLI libraries or other API libraries, and that function is not declared as a context function, unpredictable simulation results can occur, or the simulation may crash.

5.6 Referencing PLI libraries from DPI functions

A C function that is imported as either a context function or a context task is allowed to call functions from the Verilog TF, ACC and VPI PLI libraries. This has at least two important advantages: First, by utilizing the PLI libraries, a DPI function can access information within the simulation data structure that was not passed into the imported C function. For example, a DPI function could write to the simulator's log file, or to files that had been opened by the Verilog code. Second, the DPI import methodology is much simpler than the complex PLI mechanism of defining a system task/function name. Using the DPI, a PLI application can be imported as a Verilog-like function, without the complexity of defining a system task/function name (e.g. $sine shown earlier in this paper) and then having to bind the system task/function name into the Verilog simulator.

Limitations. There is an important limitation to calling PLI library routines from a C function that has been imported using the DPI. Each instance of a PLI user-defined system task or function has a unique ID (referred to as an instance handle). If, for example, the same user-defined system task is used in two different modules, each usage is completely unique within the simulation data structure. A large number of routines in the PLI libraries depend on this instance-specific characteristic. Calls to a DPI function do not have unique instance handles. If the same DPI function is called from two or more locations in the Verilog source code, there is no way within the function to distinguish one call from another.

Another important difference between the DPI and the PLI lies in what the two interfaces classify as the task or function scope. A context DPI application uses the scope where the task or function DPI import statement is declared; *not* the scope from where the task or function is called. This matches Verilog native tasks and functions, where a task or function scope is the module in which the task or function is defined and not the scope from where it is called. The scope context in the PLI, on the other hand, is the scope from which a system task or system function is called. The PLI libraries expect context scope to be the scope of the call, not the scope of the DPI import declaration. This limits which functions a DPI application can call from the TF, ACC and VPI PLI libraries.

5.7 Exporting Verilog tasks and functions

In addition to importing functions from C, the DPI allows Verilog tasks and functions to be exported to C (or potentially other foreign languages). This allows code within the C language to execute code written in Verilog. Through a combination of imported C functions and exported

Verilog tasks and functions, data can be easily passed back and forth between Verilog and C, and modified by whichever language is best suited for the design need.

An export declaration is similar to a DPI import declaration, except that only the name of the Verilog task or function is specified. The formal arguments of the Verilog task or function are not listed as part of the DPI export declaration. For example:

```
export "DPI" adder_function;
```

Optionally, a different name can be given to the task or function within the C language, as in:

```
export "DPI" adder = adder_function;  // called "adder" within C
```

A Verilog task or function can only be exported from the same scope in which it is defined, and only one DPI export declaration for a task or function is allowed. The formal arguments of an exported task or function must adhere to the same data type rules as with DPI import declarations.

In Verilog, a function can be called from other functions or tasks, whereas a task can be called from other tasks but not from other functions. This restriction is also true for exported tasks or functions. An exported Verilog function can be called from a C function that has been imported as a context function or context task. An exported Verilog task can only be called from a C function that is imported as a context task.

Both exported functions and exported tasks have a return value that is unique to the DPI. The return value of an exported task or function is an `int` value, which indicates whether or not a disable statement is active on the current task execution.

An important advantage of exporting Verilog tasks is that tasks can consume simulation time through the use of nonblocking assignments, event controls, delays, and wait statements. This provides a way for a DPI-based C function to synchronize activity with simulation time. When a C function calls an exported Verilog task that consumes time, execution of the C function will stall until the Verilog task completes execution and returns back to the calling C function.

The ability for C functions to call Verilog tasks and functions is unique to the DPI. There is no equivalent to exporting tasks and functions in the Verilog PLI standard.

6.0 Interfacing to SystemC models

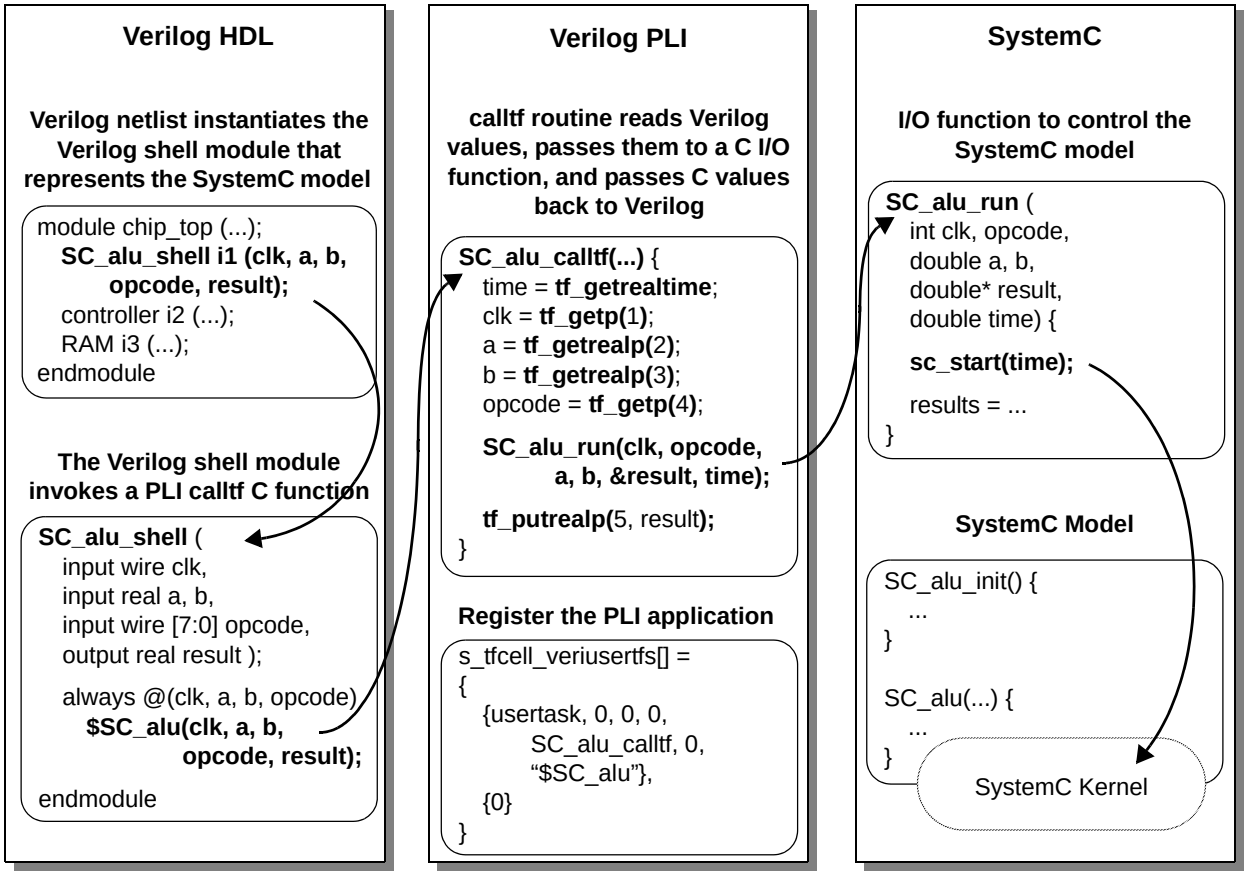
6.1 Using the Verilog PLI to interface to SystemC models

The Verilog PLI can be used to integrate SystemC models with Verilog simulations. Using the TF, ACC and VPI libraries, there are many different ways the communication between Verilog models and SystemC models can be accomplished. One common approach is to:

- Create a Verilog shell module to represent the input and output ports of the SystemC model.
- Within the shell module, call a PLI application that serves as an I/O communication channel between Verilog signals and variables in the SystemC model
- Create a C-language function that holds and transmits values between the PLI application and the SystemC model, using SystemC calls to advance time in the SystemC kernel.

Figure 2 illustrates these major components for this PLI-based interface between Verilog and SystemC. The code shown is not complete, but shows the context of the communication involved. Any of the three PLI libraries provide the functionality required to accomplish this interface. This example uses the TF library.

Figure 2. Using the Verilog PLI to interface Verilog to SystemC



Performance considerations. Using the PLI to interface Verilog models with C and SystemC models is portable to any IEEE 1364 compliant Verilog simulator. However, the PLI is an intermediate layer between Verilog and SystemC. This layer requires values to be passed to and from the SystemC model using calls to the PLI libraries. The indirect passing of values is overhead which can, and does, impact the run-time performance of simulation. The PLI also has many capabilities that are not required when interfacing Verilog models to SystemC models. These unused capabilities can also impact simulation performance.

6.2 Using the DPI to interface Verilog and SystemC models

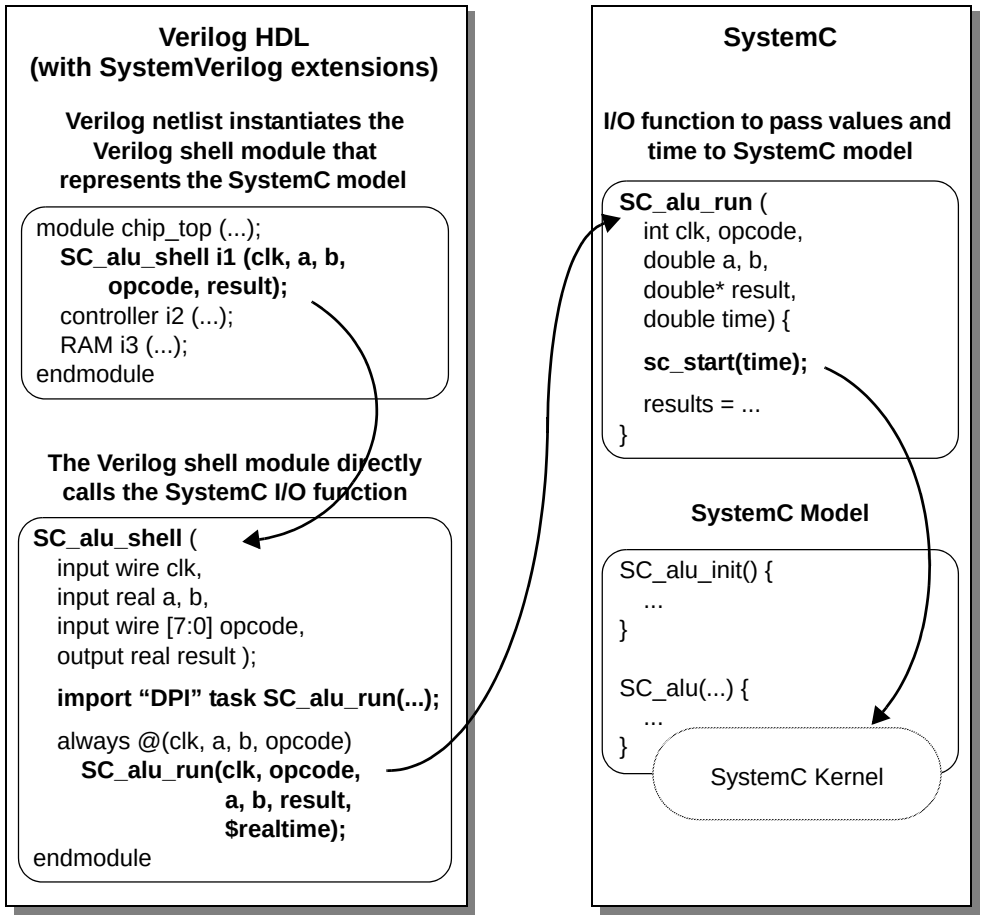
The SystemVerilog DPI allows Verilog values to be directly passed to and from C functions. This direct interface can significantly reduce the complexity of interfacing Verilog models to SystemC models. As with the PLI, there are many ways this interface can be accomplished. A simple method is similar to the approach illustrated with the Verilog PLI:

- Create a Verilog shell module to represent the input and output ports of the SystemC model.

- Create a C-language function that holds and transmits values between the DPI application and the SystemC model, using SystemC calls to advance time in the SystemC kernel.
- Within the Verilog shell module, directly call the C function that controls the SystemC model

Figure 3 illustrates this method of using the SystemVerilog DPI to interface between Verilog models and SystemC models. The code shown in this figure is not complete, but is sufficient to show the context for the communication involved.

Figure 3. Using the SystemVerilog DPI to interface Verilog to SystemC



6.3 Data type considerations when using the DPI

Both Verilog (with the SystemVerilog extensions) and SystemC provide a large number of data types and ways to represent design data. What data types are used requires some up-front planning in order to easily transfer data between models in these languages.

Table 1 lists the fundamental data types of Verilog with the SystemVerilog extensions and SystemC. This table is not all comprehensive, but is complete enough to show that there are a number of common data types in each language, as well as a number of unique data types.

Table 1: Partial list of SystemVerilog and SystemC data types

Verilog (with SystemVerilog extensions)	**SystemC**
data types that are equivalent in both languages	
byte	char
shortint	short int
int	int (32-bit), sc_int
longint	long long, sc_bigint
real	double
int unsigned	unsigned int (32 bit), sc_uint
longint unsigned	unsigned long long, sc_biguint
string	char*
structures	structures
unions	unions
enum (using default int type and values)	enum
chandle (not used within SystemVerilog, only used to store and pass C pointers)	void*
similar data types that must be manually mapped between languages	
object handle	object handle
bit (2-state vector of arbitrary size)	sc_bit (2-state vector of arbitrary size)
logic, reg (4-state vector of arbitrary size)	sc_logic (4-state vector of arbitrary size)
unpacked arrays, 1 or more dimensions	unpacked arrays, 1 or more dimensions
dynamic arrays, associative arrays	dynamic arrays, associative arrays
data types unique to SystemC (must be manually converted to SystemVerilog)	
	sc_int<>, sc_bgint<>, ... (sized integers)
	sc_fixed, sc_ufixed, ... (fixed point)
data types unique to SystemVerilog (must be manually converted to SystemC)	
packed arrays with more than 1 dimension	
queues	
integer, time (4-state, arbitrary size)	
wire, wand, wor, tri, triand, trior, trireg, pullup, pulldown (net types with strength)	

This table shows that the SystemVerilog extensions to the Verilog HDL provide a common set of data types with SystemC. When these overlapping types are used, data can be easily transferred

from a model written in one language to a model written in the other language. There is no need to perform complex data conversions.

Vectors of `bit` and `logic` types can be represented in either SystemVerilog or SystemC, but the representation of these values is not the same. For example, a Verilog vector can be big endian or little endian and does need a bit 0. SystemC assumes vectors are little endian with the least-significant bit being bit 0. To allow for these and other differences, additional coding is required to pass bit and logic vectors between the two languages. The SystemVerilog DPI provides a library of conversion functions that can aid in passing values of these similar, yet different, data types between the languages. These conversions are not simple, and can make it much more difficult to interface SystemC models into Verilog simulations.

It is also possible to transfer data between the languages using data types that are unique to one or the other language. To do this, however, additional coding is required to map values from one language's data types to the other language's data types. In addition, both SystemVerilog and SystemC allow users to create user-defined data types. Extra coding is also required to pass values of user-defined types between languages.

6.4 Synchronizing SystemC time with Verilog time

The preceding examples used the SystemC `sc_start()` function to synchronize SystemC time with Verilog time. Using this method, a Verilog simulation first advances in time to some point. When the Verilog code calls the C function that communicates with the SystemC model, `SC_alu_run()` in the previous examples, the current Verilog simulation time is passed in. The SystemC simulation is then advanced to the same time as Verilog. That is, the Verilog simulation is controlling the time of the SystemC model simulation.

SystemVerilog's DPI provides another way for synchronizing time between Verilog models and SystemC models. With the DPI, Verilog tasks can be exported to C. A C program can then call the Verilog task as if it were a C void function. The C program is not aware that the function it called is actually a Verilog task. Verilog tasks can have timing controls, including nonblocking assignments, delays (e.g. `#10ns`), event controls (e.g. `@(posedge clock)`) and level sensitive delays (e.g. `wait (reset == 0)`). When a SystemC model calls an exported Verilog task, execution of the SystemC model will suspend until the task is completed. The Verilog simulation will have advanced in time based on the task's time controls.

For example, a Verilog task can be written that delays until a positive edge of clock occurs. This task can then be called from a SystemC model. The SystemC simulation can then advance past Verilog time to its next clock cycle, and then stall until Verilog simulation catches up by calling the exported Verilog task with a clock delay. The execution of the SystemC model will be delayed until the Verilog simulation has advanced to its next positive edge of clock.

7.0 Conclusions

SystemVerilog is a major set of extensions to the IEEE 1364-2001 Verilog standard. These extensions include a Direct Procedural Interface (DPI). SystemVerilog is an Accellera standard, and is publicly available for use with any EDA software product. Many EDA companies are actively adding the SystemVerilog extensions to their products. Synopsys has already released

versions of VCS (for simulation) and HDL Compiler (for synthesis) that support much of the SystemVerilog standard. Accellera will be donating the SystemVerilog extensions to the IEEE 1364 standards committee in June of 2004. It is expected that these extensions will become a part of the next version of the IEEE 1364 standard.

Prior to the introduction of the SystemVerilog DPI standard, the Verilog PLI was the only portable means to simulate SystemC models and Verilog models together. While fully capable of performing this interfacing, using the Verilog PLI has several disadvantages. Learning how to write even small PLI applications can be a daunting task for many engineers. The PLI has many capabilities that are not needed to interface Verilog with SystemC, but that add to the complexity of using the PLI. The PLI is a protecting layer between Verilog simulation data structures and C code, which adds overhead to simulation performance and complexity to the code required to interface Verilog with SystemC.

To overcome the disadvantages of the Verilog PLI, many simulator companies provide proprietary methods to integrate Verilog models and SystemC models in the same simulation. These proprietary methods simplify the interface for users, and can improve simulation performance. However, these proprietary methods are not portable from one simulator to another. Source code in either or both the Verilog models and the SystemC models must be customized for each simulator used within a company.

The SystemVerilog Direct Programming Interface provides an ideal solution for integrating Verilog models with SystemC models. The DPI import declarations allow Verilog code to directly call C code without the complexity and overhead of the Verilog PLI. Interfacing Verilog and SystemC becomes a simple and straightforward process. Because the DPI is a standard, integrating Verilog and SystemC models using the DPI is portable to all simulators that adhere to the SystemVerilog standard. SystemVerilog also extends Verilog with a number of C-like data types. These data types provide commonality between Verilog models and SystemC models, which can make it much easier to transfer data between models.

The most efficient yet portable method for integrating HDL-based models and SystemC models is to use Verilog with the SystemVerilog extensions. The SystemVerilog DPI can integrate Verilog and SystemC models in a manner very similar to using the Verilog PLI, but in a much more straight forward manner.

8.0 References

[1] *"IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey, 2001. ISBN 0-7381-2827-9.

[2] *"SystemVerilog 3.1a, draft 6: Accellera's Extensions to Verilog"*, Accellera, Napa, California, 2004.

[3] *"SystemC 2.0.1 Language Reference Manual Revision 1.0"*, Open SystemC Initiative, San Jose, California, 2004. http://www.systemc.org

[4] *"SystemC Version 2.0 Users Guide"*, Open SystemC Initiative, San Jose, California, 2004. http://www.systemc.org

[5] *"The Verilog PLI Handbook, second edition"*, Stuart Sutherland, Kluwer Academic Publishers, Boston, Massachusetts, 2002. ISBN 0-7923-7658-7.

[6] "*HDVL += (HDL & HVL), SystemVerilog 3.1 The Hardware Description AND Verification Language*", Stuart Sutherland and Don Mills, paper presented at Synopsys 2003 SNUG Europe conference. http://www.snug-univer-

sal.org/papers/papers.htm.

[7] "*The Verilog PLI Is Dead (maybe) — Long Live the SystemVerilog DPI*", Stuart Sutherland, paper presented at Synopsys 2004 SNUG San Jose conference. http://www.snug-universal.org/papers/papers.htm.

[8] "*SystemVerilog APIs*", Doug Warmke, presentation at DAC 2003 Accellera SystemVerilog Workshop, Session 4. http://www.systemverilog.org/pdf/4_DPIOverview.pdf.

[9] "*Interfacing SystemC to HDL using PLI/VPI & FLI*", on-line example by AnsLab. http://www.anslab.co.kr.

[10] *"Designers May Find C++ to Their Liking"*, Mike Baird, EEDesign, 03 Oct. 2001, www.eedesign.com/article/showArticle.jhtml?articleId=16503703

9.0 Glossary of acronyms

ACC — Access interface; the second generation of the Verilog PLI.

DPI — Direct Programming Interface; part of the Accellera SystemVerilog extensions to the Verilog language.

HDL — Hardware Description Language. The two most popular HDLs are Verilog and VHDL.

OVI — Open Verilog International; A not-for-profit organization to promote Verilog standards and the use of Verilog. In 2001 merged with VHDL International (VI) to form Accellera.

PLI — Programming Language Interface; part of the IEEE Verilog language.

PLI 1.0 — A nickname for the TF and ACC interfaces. Originally an OVI standard.

PLI 2.0 — A nickname for the VPI interface. Originally an OVI standard.

TF — Task/Function interface; the oldest generation of the Verilog PLI.

VPI — Verilog Programming Interface; the most current generation of the Verilog PLI.

10.0 About the author

Mr. Stuart Sutherland is a member of the Accellera SystemVerilog technical subcommittee that is defining SystemVerilog, and is the technical editor of the SystemVerilog Reference Manual. He is also a member of the IEEE 1364 Verilog Standards Group, where he serves as co-chair of the PLI task force. Mr. Sutherland is an independent Verilog consultant, who specializes in providing expert training on the Verilog HDL, SystemVerilog and PLI. He can be reached by e-mail at stuart@sutherland-hdl.com. His web site is www.sutherland-hdl.com.