By Donna Mitchell and Dan Notestein

Easing Today's Verification Language Bedlam

Creating SystemC and HDL testbenches with SCV

Writing reusable bus-functional testbench models has always been a challenge. In the past few years, several verification languages have been introduced to address that challenge. For a company like SynaptiCAD Inc. (Blacksburg, VA), which makes the TestBencher Pro graphical code generator, a diverse customer base requires that our tool be able to generate the same model in all verification languages currently in use. We have a lot of experience pushing each language to its performance limits and comparing how the languages work.

SCV (SystemC Verification) – a verification library for SystemC – is one of the newest players in the verification language space. This article describes some of the benefits we've found in using SCV to develop testbenches, as well some of the pitfalls. First, however, this caveat: SCV is still under development, so some features discussed in this paper may not be officially accepted into the standard yet.

TWO USAGE MODELS

There are two basic ways to use SCV. The first is to use it in a pure SystemC environment, where both the testbench and the design model are written in C++. The other way is to use SCV as the testbench language, while maintaining the design models in an HDL language so they can be synthesized. Ultimately, the goal here is to be able to use the same SCV testbench to test either a SystemC or an HDL mode with only minor differences in an interface file. The solution would be portable to different simulators and simulator combinations.

However, there are some technical limitations with this approach. Limited bi-directional signal support and proprietary wrapper classes can make this process more complicated than it sounds. SCV in a pure SystemC environment overcomes many of the bus-functional model design difficulties found in pure VHDL or Verilog environments. But in a mixed SCV and HDL environment, there are still some technical issues to work out.

The biggest problem with using SCV with an HDL-based design is that, without native simulator support, an SCV model cannot communicate with a bi-directional signal in an HDL model. Only SCV_INPUT and SCV_OUTPUT are supported. This is a big limitation, because one of the nice things about using a C++ bus-functional model is that you can write simple models that do things that are difficult to achieve using pure VHDL or Verilog. By mixing C++ testbench models with HDL design models, you theoretically get the best of both worlds – industry proven synthesizers for the design and flexible, high performance bus-functional models with dynamic memory allocation for the testbench.

Simulators with native support, such as the Incisive simulators from Cadence Design Systems, Inc (San Jose, CA), work around this problem by using proprietary wrapper-class libraries to provide bi-directional communication. The wrapper classes handle the communication between models in different languages. All of the code written in either C++ or HDL is elaborated and executed by the same simulation engine. This solution works, but it also means that the code is no longer portable between different simulators. This poses a particular problem for IP designers, or for those who use graphical code generators working to produce models that can be used with any simulator combination.

This lack of bi-directional support is particularly disappointing in light of SCV's parentage. The SCV architecture is based on the older Cadence TestBuilder library. With TestBuilder, you basically have the same power as SCV, but you have simulator independence. For pure SystemC design, SCV is certainly a proper choice. For mixed HDL designs and C++ testbenches, however, you need to consider the native support of your simulator and the audience that will be using your models before making a final decision between the two libraries.

SCV SIMPLIFIES BFM DEVELOPMENT

Most complex testbenches are coded using bus-functional models (BFMs) that mimic the I/O behavior of a device without modeling its internal computational abilities. The BFM-based testbench architecture we use at SynaptiCAD is in that category. (see Figure 1)

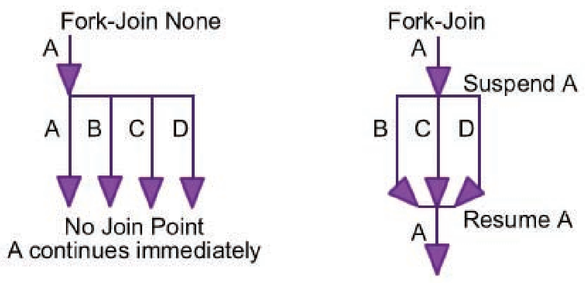

Figure 1: The SynaptiCAD bus-functional model architecture

The Transaction Manager controls a queue of transactors that can be applied to the model under test. The Transaction Generator uses the constrained random data generation features in SCV to determine which transactions to execute (for example, a microprocessor's read or write cycle), and which parameter values to use for the transaction (for example, the address and data values for a write cycle). The Transaction Monitor records which transactions are executed, including any transaction outputs generated by the model under test, and also employs the SCV data coverage features to ensure that the testbench will provide adequate functional coverage.

To maximize simulation performance, the transactors themselves should generally be modeled in the language of the design, since there is typically a performance penalty for simulation activity that occurs across simulation languages. But if you're using a mixed-language, single-kernel simulator, you can often work across language boundaries without incurring a significant penalty.

In any event, the Transaction Manager should be coded in a language that supports data structures and dynamic memory allocation and SCV C++ is a perfect choice. There is very little speed penalty for coding the testbench in this manner because the Transaction Manager makes simple function calls to the native language transactors. These function calls happen relatively infrequently, compared to the amount of activity that occurs during the execution of the transaction.

RANDOMIZED AND SIMPLIFIED

Another useful capability, added recently to SystemC version 2.1, is the ability to dynamically create processes and perform both standard fork-join type operations and fork-join-none operations. (see Figure 2)

Traditional fork-joins, like those available in Verilog, cause the spawning process/thread to wait until all the spawned processes are completed. Alternatively, fork-join-none operations allow multiple processes to be spawned off simultaneously, without causing the spawning process to wait for the spawned processes to complete. The fork-join-none capability is also available in several other verification languages, including OpenVera and e.

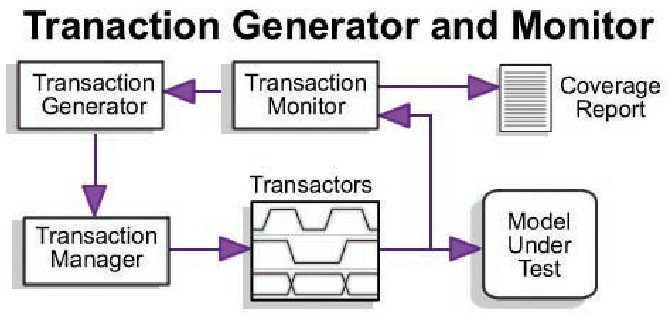

Figure 2 : SCV supports fork-join-none, allowing creation of sequences of randomized non-blocking transactions.

In a bus-functional model, fork-join-none can be used to enable multiple randomly generated transactions to be executed. Without this feature, randomized non-blocking transaction sequences require the addition of a complex set of handshaking signals. Those signals trigger the transactions without waiting for them to finish, which unnecessarily complicates the testbench architecture.

Frequently, it's necessary in developing testbenches for packet-based switched-networks such as an ATM switch, to initiate transactors connected to one part of the testbench hierarchy from another. Similarly, hierarchical transaction calls can be used in the design of a PCI bus-functional model in which the PCI Master and PCI Slave models are components of a parent PCI model. The PCI model triggers the PCI Master transactions and collects the results from of PCI Slave transactions. This is a difficult task in VHDL, because there is no native support in the language for hierarchical referencing of signals or tasks, and also because of problems that occur when signals need to be driven by multiple processes.

Verilog, however, makes it relatively simple to call transactors in other parts of the design hierarchy. Verilog doesn't suffer from the multiple driver problems associated with VHDL, although it does lack of support for dynamic memory allocation. That lack makes it almost impossible to support important testbench capabilities such as enqueing of transactions with randomly generated parameters for execution later in the testbench.

SCV is able to address the problems inherent in Verilog and VHDL. SCV makes it relatively simple to create a flexible architecture for transaction-based testbenches, as it supports dynamic memory allocation and has no multiple driver problems.

COVERAGE WITH CONSTRAINED RANDOMIZATION

Using a mix of directed and randomized test values has become a popular technique when testing large systems. It's fairly impossible these days to test all possible input cases for more than the simplest designs. The randomized data is constrained to a subset of "likely" inputs that enable reasonable coverage of system functionality. SCV offers a robust set of classes for generating constrained random data to meet this need.

For instance, SCV provides a class, scv_random that can be used to generate an independent stream of random integers. Typically, this class isn't used directly. Instead, you use a template class called scv_smart_ptr, which contains internal scv_random instantiations. To perform basic randomization, you just create a scv_smart_ptr and pass the class you're wanting to randomize as a template parameter. This allows you to call the "next" method to generate a random value.

SCV also supports a form of introspection on SCV data objects. Introspection is a relatively new concept in computer science. The strategy allows a program to gain knowledge of the properties of an initially unknown data object. An algorithm, for instance, could use introspection to determine the name and data types of data members in an object. Using introspection, the scv_smart_ptr can perform automatic randomization on the fields of a user-defined class without requiring the user to write custom code for his class.

For more complicated randomization needs, users can create a class that derives from the scv_contraint_base class. The derived class is used to specify soft and hard constraints, so that the randomly generated numbers represent legal values based on specific design criteria. Hard constraints have to be met; if the random generator cannot meet them, a runtime error will be generated. The constraint engine will also attempt to satisfy soft constraints. However, if these can't be met due to hard constraints, the soft constraints will be ignored.

In addition to placing basic constraints on data values, it's also possible to generate weighted random numbers using the scv_bag class. Using this class, the user the ability to flexibly control the probability distribution of the random numbers being generated.

SUMMARY

As designs become significantly more complicated, it is increasingly difficult to develop adequate testbenches using standard VHDL and Verilog language constructs. SCV is the latest language to address the insufficiency in HDL languages by offering capabilities such as dynamic processes, dynamic memory allocation, object oriented programming model, and built-in libraries for constrained random data generation.

SCV is the perfect choice for anyone using a design methodology that incorporates SystemC at some stage in the design, since the testbenches developed during the SystemC phase can be reused again during the HDL design stage. It is also a good candidate for developing testbenches for strictly HDL-based designs as it offers many advanced testbench capabilities not available in Verilog and VHDL.

Another benefit not to be overlooked – SCV is a C++ based language. As a result, there is a large body of existing C++ libraries that can be incorporated into your testbench development. It's also easier, therefore, to test systems whose functionality is partitioned between both the hardware and the software components.

One final point – SCV an attractive choice as a verification language even from a cost consideration. Like SystemC itself, SCV is a free open-source library and C++ development tools are extremely cheap, especially when compared to standard EDA tools in the market today.

Donna Mitchell *is a co-founder and VP of marketing of SynaptiCAD. Mitchell* received her BS and MS degrees in electrical engineering from Virginia Tech. As one of the founders of SynaptiCAD, she assisted in the design and development of SynaptiCAD's first products, Timing Diagrammer, WaveFormer Pro and TestBencher Pro.

Daniel Notestein*, is the President of SynaptiCAD and the chief software architect for SynaptiCAD's TestBencher Pro and VeriLogger Pro products. Notestein obtained his bachelor's degree in electrical engineering and minors in computer science and math from Virginia Tech and his MSEE from the University of Texas.*