A Tutorial Introduction on the New SystemC Verification Standard

C. Norris Ip
ip@cadence.com

Stuart Swan
stuart@cadence.com

Cadence Design Systems, Inc.

Abstract

This paper describes how SystemC 2.0 and the new SystemC Verification Standard provide a robust standard for developing test benches and verification IP for SoC designs. The new SystemC Verification Standard includes features for verification, such as transaction recording and constrained randomization, which facilitate stimulus generation, visualization, debugging and analysis of a simulation run.

1. Introduction

While the SystemC 2.0 standard [1,2] can be used to perform basic verification of a design, the new SystemC Verification Standard [3] improves the capability by providing APIs for transaction-based verification, constrained and weighted randomization, exception handling, and other verification tasks. In this paper, we provide an overview of the new SystemC Verification Standard through an example of a transaction-based test bench. The following aspects of the Verification Standard are discussed:

- transaction-based verification
- data introspection
- transaction recording
- constrained and weighted randomization
- miscellaneous features of the Standard, such as HDL connection and detection of bugs and exceptions.

The basic structure of a transaction-based test bench is shown in Figure 1.

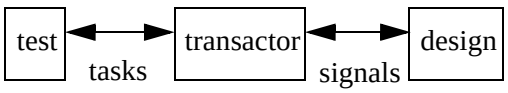

Figure 1: Transaction-Based Verification

A *transaction-based verification* methodology partitions the system into transaction-level tests, transactors, and the design [5,6]. Communication between the tests and the transactors is done through task invocation, at a level above the RTL level of abstraction. The communication between the transactors and the design is done through RTL-level signals.

Manipulation of high-level data types is an important element in this approach. A *data introspection* facility in the Verification Standard enables the manipulation of arbitrary data types in a consistent way, including C/C++ built-in types, SystemC built-in types, user-defined composite types and user-defined enumerations. As a result, arbitrary data types can be used in variable recording, transaction recording, randomization, constraints, assertions, and other high-leveling activities.

Another important aspect of transaction-based verification is *transaction recording*. The transaction recording facility in the Verification Standard allows the user to capture transaction-level activities during simulation. Through a callback mechanism, these activities can be monitored by another SystemC module at run-time, or they can be recorded into a database for visualization, debugging, and post-simulation analysis.

On the other hand, randomization allows a large amount of stimulus to be generated with less manual effort than directed tests. In order to improve coverage and focus on specific aspects of the design, constraints or weights are typically used in randomization. While many existing test benches may be using *rand()* from the C library to generate a random integer, the Verification Standard supports randomization of any data type through the use of the data introspection facility. Boolean predicates can be used as a constraint, and weights can be used as a distribution from which values are selected.

To complete the flow in a typical design environment, we envision SystemC test benches to be used for designs written in Verilog or VHDL as well. The Verification Standard provides a minimal set of HDL connection API to enable this use model. The Verification Standard also includes a debugging interface and an exception reporting API to facilitate debugging using a C++ debugger and to maintain a consistent way of reporting detection of design bugs or test bench bugs in verification models. Because of space reason, they are not described in this paper.

This paper assumes some basic knowledge about C and C++. The classes and functions in SystemC 2.0 use the prefix *sc_*, and the classes and functions in the SystemC Verification Standard use the prefix *scv_*.

2. Transaction-Based Test Bench in SystemC

It is desirable to write a test bench at the transaction level, since it captures the design intent and test scenarios at the level at which the architect thinks. Such a test bench can

be used to simulate a transaction-level design directly, and it can also be used to simulate a RTL-level design through an adaptor channel, usually known as a transactor or a bus-functional model. The description in this paper falls into the latter use model.

Using channels and interfaces in SystemC, a transaction-based test bench can be implemented as shown in Figure 2. The notation is adapted from the book *System Design with SystemC* [2], slightly modified to emphasize the use of a channel. The transaction-level interface for the transactor is implemented as a *rw_task_if* base class. Using abstract methods in C++, it can be declared as:

```
class rw_task_if : virtual public sc_interface {
public:
  typedef sc_uint<8> addr_t;
  typedef sc_uint<8> data_t;
  struct write_t {
    addr_t addr;
    data_t data;
  };
  virtual data_t read( const addr_t * ) = 0;
  virtual void write( const write_t * ) = 0;
};
```

This abstract base class specifies two abstract methods, *read()* and *write()*, and their related data types. These two methods represent the abstract level in which a test is to be written in. The class *sc_interface* is provided by SystemC to facilitate the creation of such interfaces. The template *sc_uint* and other similar templates are provided by SystemC to support data objects with different bit widths and different operator semantics.

The communication between the transactor and the design is captured in a base class with signal-level ports:

```
class pipelined_bus_ports : public sc_module {
public:
  sc_in<bool> clk;
  sc_inout<bool> rw;
  sc_inout<bool> addr_req;
  sc_inout<bool> addr_ack;
  sc_inout< sc_uint<8> > bus_addr;
  sc_inout<bool> data_rdy;
  sc_inout< sc_uint<8> > bus_data;
};
```

These ports represent the RTL-level interface in which a design communicates to its environment. The class *sc_module* is provided by SystemC to specify a module; and signal ports are created via the templates *sc_in, sc_out,* and *sc_inout,* indicating a read-only port, a write-only port, and a read-write port respectively.

A transaction-based verification methodology relies on transactors to act as the adaptors between the abstract tests and the RTL-level design. By capturing these transactors as reusable IP, new tests with complex concurrent behavior can be quickly created. In this example, a transactor is created as a class deriving from both aforementioned interfaces:

```
class rw_pipelined_transactor
: public rw_task_if, public pipelined_bus_ports {
public:
  SC_CTOR(rw_pipelined_transactor) {}
  virtual data_t read( const addr_t *);
  virtual void write( const write_t *);
  ...
};
```

The macro *SC_CTOR* in SystemC 2.0 specifies the constructor. The implementation of *read()* and *write()* convert the transaction-level operations to signal-level activities with respect to the actual pipelined protocol. The related transaction-level infactiondoth0.2(n)7.1(7(o)6.nd.3(d)-1(th0.2(tra)

```
class test : public sc_module {
public:
  sc_port< rw_task_if > transactor;
  SC_CTOR(test) { SC_THREAD(test_body); }
  void test_body();
};
```

The *sc_port* template in SystemC 2.0 creates a system-level port with an abstract interface. The methods in the channel attached to this port can be accessed through the C++ operator->, for example:

```
transactor-> write ( arg );
```

The macro *SC_THREAD* creates a new thread of execution for the *test_body* method. The implementation of *test_body* can be a directed test, a constrained random test, or a weighted random test, as described in Section 5.

It is important to note that the port of this test has the *rw_task_if* interface as the template argument, so that, by plugging in a different transactor, the test can be reused with other designs with a different bus interface. The *rw_task_if* interface can be made even more general and reusable, for example, by putting the width of address and data into template parameters instead of having a fixed value.

Finally, the code for the RTL design uses the standard SystemC RTL modeling style:

```
class design : public pipelined_bus_ports {
public:
  SC_CTOR(design) {
    SC_THREAD(addr_phase); SC_THREAD(data_phase);
  }
  void addr_phase() { while (1) ... }
  void data_phase() { while (1) ... }
  ...
};
```

The *addr_phase* and *data_phase* methods are similar to *always* blocks in Verilog, and they implement the two phases of the pipeline. The design contains the same set of ports as the transactor, although the direction of the ports are reversed. In order to keep the example simple, we have used the same base class for the signal-level ports. In practise, it is desirable to declare the ports as *sc_in* or *sc_out* and use different port declarations for the transactor and the design.

Finally, a simulation netlist can be created as follows:

```
int sc_main(int argc, char *argv[ ]) { ...
  // the modules and channels
  test t ("t");
  rw_pipelined_transactor tr ("tr");
  design duv ("duv");
  sc_clock clk ("clk",20,0.5,0,true);
  // the signals to connect the modules
  sc_signal < bool > rw, addr_req, ... ; ...
  // connecting the signals and transactors to the ports
  t.transactor = tr;
  tr ( clk.signal(), rw, addr_req, ... );
  duv ( clk.signal(), rw, addr_req, ... );
  // start simulation
  sc_start(10000); ...
}
```

The *sc_main* function is the entry point to the SystemC reference simulator. The first portion of the function instantiates the modules, the channels, and the signals. Then, it connects the modules by attaching channels or signals to the appropriate ports. In the final portion of the function, simulation is initiated through *sc_start()* with the simulation time specified in the argument.

Strictly speaking, this arrangement relies on the ability to drive the same signal (*sc_signal*) from more than one threads, since the *read()* and *write()* tasks are executed in the calling thread, and there may be more than one thread calling the tasks in the transactor. However, as described in Section 4, since explicit synchronization is performed among tasks, it is not necessary to use resolved signals. In this use model, the values are updated with the last-assignment-wins semantic, which is how the *sc_signal* class is implemented in the reference implementation.

On the other hand, using a resolved signal will give better error-detection, although it requires more code and leads to slower performance. For example, when a misbehaving design tries to drive a signal while a transactor is also driving it, collisions can be detected with a resolved signal. However, the signal must be set to high-impedance when it is no longer being driven by the current thread. A resolved signal must be a logic bit or a logic vector, so 2-state data types and arithmetic cannot be used directly, and explicit conversion must be made to an arithmetic type.

This example illustrates a preferred style for organizing transaction-based test benches in SystemC. In the remainder of this paper, implementation of the tests and transactors are discussed.

3. Data Introspection

The SystemC Verification Standard uses data introspection to enable the manipulation of arbitrary data types. It allows a library routine to extract information from data objects of arbitrary types, regardless of whether it is a C/C++ built-in type, a SystemC data type, a user-specified composite type (struct), or a user-specified enumeration. Similar techniques can be found in articles on C++ [8,9].

For example, the *rw_task_if* interface shown in Section 2 uses *sc_uint* (from SystemC) and *write_t (*a composite type). Using C++ template specialization, the Verification Standard maps these data types to an abstract interface called *scv_extensions_if*, through which the following operations can be performed on the data object.

- extraction of type information
- value access and value assignment

- randomization
- callback registration

While traditional C++ libraries typically require the user to use a similar interface class as the base class of their composite type, the Verification Standard uses template specialization to attach this interface to data objects. This style supports a wider range of data types. It enables import of legacy code without modification, and allows the same pieces of code to work on built-in types such as *int*, library types such as *sc_uint*, composite types such as a user-defined packet type with multiple fields, and enumerations such as an instruction set.

Using data introspection, a piece of code can manipulate a data object without explicit type information at compile-time. This facility can be considered as a C++ version of the Verilog PLI standard. It is a crucial building block for constrained randomization, variable recording, and transaction attribute recording.

Using the *scv_extensions_if* Interface

The abstract methods provided by *scv_extensions_if* can be classified into methods for static extensions and methods for dynamic extensions. A static extension is used for simple type information and value access or assignment. This is accomplished via the *scv_get_extensions()* function in the Verification Standard. For example, the following code extract the bit width of an integer and print its value.

```
int data;
int bitwidth = scv_get_extensions(data).get_bitwidth();
scv_get_extensions(data).print();
```

Because the same abstract interface can be used to manipulate C/C++ built-in types, SystemC types, and user-specified type, a predefined library can manipulate any of these data types, typically using the following style:

```
template<typename datatype> void process(datatype& data) {
  scv_extensions<datatype> data_ext
    = scv_get_extensions(data);
  process_core(&data_ext);
}
void process_core(scv_extensions_if *);
```

This code uses C++ template parameter deduction to find the right extension for the data object, and then pass a pointer to the extension to the actual code that process the data object. As a result, the function *process_core* can process any data object without requiring explicit type information at compile time of this function.

A dynamic extension is used when auxiliary data needs to be associated with the data object, such as a constraint or a callback function pointer. This is accomplished via the *scv_smart_ptr* template in the Verification Standard. For example, the following code generates a random value for an integer and register a callback.

```
scv_smart_ptr<int> ptr;
ptr->next(); // generate a random integer value.
ptr->register_cb(value_change_callback);
```

The *scv_smart_ptr* template is designed so that it behaves as if it is a C pointer (c.f. a smart pointer [7]).

Defining the Extensions for User-Specified Types

The Verification Standard supports C++ data types and SystemC data types without any extra work from the user. For user-specified composite types and enumerations, a template specification for the *scv_extensions* template needs to be provided to the library[1]. For example, without changing the declaration of the composite type *write_t* in *rw_task_if*, the extensions can be declared as:

```
SCV_EXTENSIONS(rw_task_if::write_t) {
public:
  scv_extensions< rw_task_if::addr_t > addr;
  scv_extensions< rw_task_if::data_t > data;
  SCV_EXTENSIONS_CTOR(rw_task_if::write_t) {
    SCV_FIELD(addr);
    SCV_FIELD(data);
  }
};
```

By providing this extension class, the data introspection interface can be used to manipulate *write_t:*

```
rw_task_if::write_t data;
int bitwidth = scv_get_extensions(data).get_bitwidth();
scv_get_extensions(data).print();
scv_smart_ptr< rw_task_if::write_t > ptr;
ptr->next(); // generate random values for both fields
ptr->register_cb(value_change_callbacks);
```

4. Transaction Recording

Transaction recording is typically used in the implementation of a transactor. The job of a transactor is to convert a high-level operation, as modeled by a function call, into signal-level communication, and vice versa. For example, the read method is implemented as follows:

```
data_t rw_pipelined_transactor::read( const addr_t * addr) {
  address_phase.lock();
  scv_tr_handle h = read_gen.begin_transaction(*addr);
  { // address phase
    scv_tr_handle h1
        = addr_gen.begin_transaction(*addr, "addr_phase", h);
    ...// address phase
    addr_gen.end_transaction(h1);
  }
  addr_phase.unlock();
```

1. The extension declaration can be automatically extracted from the C struct or C++ class definition using a script.

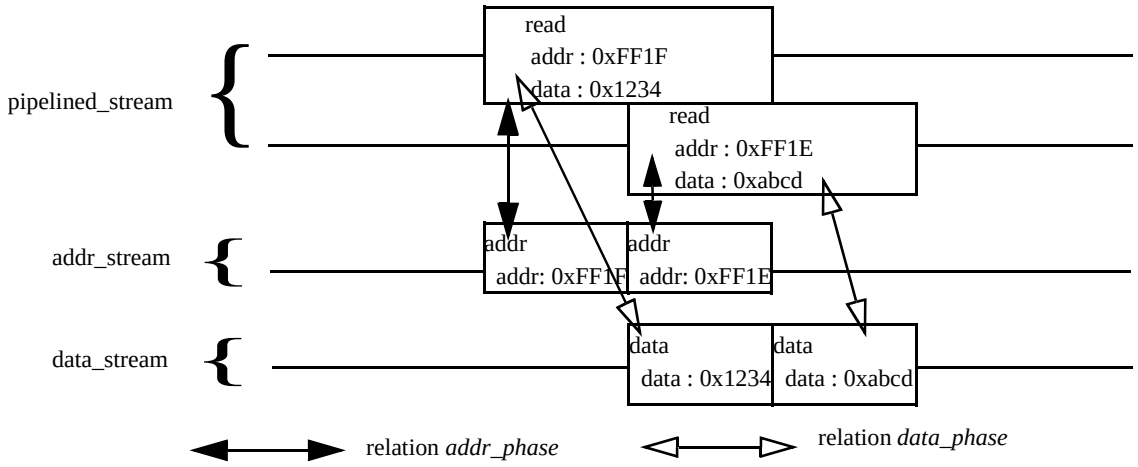

Figure 3: Transactions from Two Read Operations

```
  data_phase.lock();
  { // data phase
    scv_tr_handle h2
      = data_gen.begin_transaction("data_phase",h);
    ...// data phase
    data_gen.end(h2,data);
  }
  read_gen.end_transaction(h, data);
  data_phase.unlock();
  return data;
}
```

This method translates a call to *read()* into a series of signal activities according to the specific protocol at the signal-level interface. Two mutexes, *addr_phase* and *data_phase*, are used to coordinate the two phases in the pipelined protocol. Both of them grant access in a first-come-first-served manner. At the beginning of the address phase, *addr_phase* is locked so that another operation cannot begin. At the end of the address phase, *addr_phase* is unlocked so that, although the data phase has not finished yet, another operation can still start its address phase.

Typically each transaction-level operation generates at least one transaction, which begins when the operation has successfully arbitrated for the resource to proceed, and ends then the operation is completed. This is captured in the transaction generated by the *read_gen* generator. The *begin_transaction()* method initiates a new transaction with the supplied argument as an attribute, and the *end_transaction()* method terminates a transaction and uses the second argument as another attribute. These methods rely on the static extension of *addr_t* and *data_t* to access and record the values of the attributes.

Depending on the protocol, sub transactions can be generated to capture individual aspects of the protocol, which in this case, corresponds to the transactions from the *addr_gen* generator and the *data_gen* generator. The extra argument for the *begin_transaction* method establishes a relationship between the new transaction and the overall transaction from *read_gen.*

The transactions generated by two *read()* can be visualized as in Figure 3. Capturing the concept in this diagram, the Verification Standard provides the *scv_tr_stream* class and the *scv_tr_generator* template to describe transaction streams and transaction types. By instantiating these classes before the simulation starts, run-time efficiency is maximized. For example, *rw_pipelined_transactor* can be implemented as follows:

```
class rw_pipelined_transactor
: public pipelined_bus_ports, public rw_task_if {
  fifo_mutex address_phase;
  fifo_mutex data_phase;
  scv_tr_stream pipelined_stream;
  scv_tr_stream addr_stream;
  scv_tr_stream data_stream;
  scv_tr_generator< addr_t, data_t > read_gen;
  scv_tr_generator< addr_t, data_t > write_gen;
  scv_tr_generator< addr_t > addr_gen;
  scv_tr_generator< data_t > data_gen;
public:
  SC_CTOR(rw_pipelined_transactor)
  : pipelined_stream("pipelined_stream"),
    addr_stream("addr_stream"),
    data_stream("data_stream"),
    read_gen("read",pipelined_stream, "addr","data"),
    write_gen("write",pipelined_stream,"addr","data"),
    addr_gen("addr",addr_stream,"addr"),
    data_gen("data",data_stream, "data") { ... }
```

```
    virtual data_t read( const addr_t * addr);
    virtual void write( const write_t * write);
};
```

5. Constrained and Weighted Randomization

Constrained and Weighted random tests are an important element in verification. For example, as described in Section 3, a random value can be generated for a smart pointer by a simple call to *next()*.

Using the Verification Standard, expressions and constraints can be created using *scv_smart_ptr* as well. For example,

```
class write_constraint : virtual public scv_constraint_base {
public:
  scv_smart_ptr< rw_task_if::write_t > write;
  SCV_CONSTRAINT_CTOR(write_constraint) {
    SCV_CONSTRAINT( write->addr() < 0x00FF );
    SCV_CONSTRAINT( write->addr() != write->data() );
  }
};
```

This constraint creates two Boolean expressions regarding the fields of the variable *write*. When *next()* is executed, values that satisfy these expressions are generated for each field. Randomization for individual fields can be enabled or disabled through the *enable_randomization()* and *disable_randomization()* methods.

Declaring the constraints as classes allows them to be processed once for high-speed randomization. It also allows an object-oriented way to manage constraints, using hierarchy and inheritance. There are two ways to use these constraint classes. It can be used directly:

```
write_constraint c("c");
c.next();
scv_smart_ptr< rw_task_if::write_t> write = c.write;
```

or it can be associated with an existing smart pointer using the *use_constraint()* method:

```
write_constraint c("c");
scv_smart_ptr<rw_task_if::write_t> write;
write->use_constraint(c.write);
write->next();
```

Biased randomization using weights can be performed with a bag using the *set_mode()* method. A bag is similar to the concept of a set in mathematics, except that it can contain duplicated objects. The following example generates the value "1" 40% of the time and the value "2" 60% of the time:

```
scv_smart_ptr<int> data;
scv_bag<int > distribution;
distribution.push( 1, 40);
distribution.push( 2, 60);
data->set_mode(distribution);
for (int i=0; i<3; i++) {data->next(); ... }
```

A distribution for simple ranges can also be specified without a bag, using the *keep_only()* and *keep_out()* methods. For example, the following code generates values from 0 to 5, and from 10 to 15 with a uniform distribution:

```
scv_smart_ptr<int> data;
data->keep_only(0,15);
data->keep_out(6,9);
data->next();
```

Acknowledgement

The SystemC Verification Standard was approved by the SystemC Verification Working Group in August 2002 and is currently waiting final approval from the Steering Committee.

The authors would like to thank the SystemC Verification Working Group for creating a SystemC Verification Standard, and to thank the TestBuilder engineering team at Cadence Design Systems for creating the prototype for the standard.

References

[1] Stuart Swan, An Introduction to System Level Modeling in SystemC 2.0, white paper, www.SystemC.org.

[2] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan, System Design with SystemC, Kluwer Academic Publishers, 2002.

[3] The SystemC Verification Standard, version 1.0, to appear at www.SystemC.org.

[4] C. Norris Ip and Stuart Swan, Using Transaction-Based Verification in SystemC, white paper, www.SystemC.org.

[5] Steven Cox, Mark Glasser, William Grundmann, C. Norris Ip, William Paulsen, John L. Pierce, John Rose, Dean Shea, and Karl Whiting. Creating a C++ Library for Transaction-based Test Benches, *Forum on Design Languages*, France, September, 2001.

[6] Dhnanjay S. Brahme, Steven Cox, Jim Gallo, Mark Glasser, William Grundmann, C. Norris Ip, William Paulsen, John L. Pierce, John Rose, Dean Shea, Karl Whiting. The Transaction-Based Verification Methodology, technical report # CDNL-TR-2000-0825, Cadence Berkeley Labs, August 2000.

[7] Scott Meyers, More Effective C++, Addison-Wesley, 1996. For information about smart pointers, see item 28, smart pointers and item 29, reference counting.

[8] Nathan C. Myers, Traits: a new and useful template technique, C++ Report, June 1995. http:// www.cantrip.org/traits.html.

[9] John Maddock and Steve Cleary, C++ Type traits, Dr Dobb's Journal, October 2000. http://www.boost.org/ libs/type_traits/c++_type_traits.htm.