Using Transaction-Based Verification in SystemC

C. Norris Ip
ip@cadence.com

Stuart Swan
stuart@cadence.com

June 2002, Cadence Design Systems, Inc.

Keywords : Functional Verification, System-Level Verification, C++, TestBuilder, SystemC, TestBuilder-SC

Abstract

SystemC is a new modeling language based on C++ for hardware and system-level design modeling. This paper examines how a basic transaction-based test bench may be created in the SystemC version 2.0 standard. An example with a pipelined bus interface is presented to illustrate what can be done easily with the latest SystemC version 2.0 distribution [1].

The TestBuilder team is currently creating a TestBuilder-SC library to fill in several missing pieces in SystemC 2.0, in order to support advanced transaction-based verification in a real design development process. While TestBuilder [3] was designed for direct connection to an HDL simulator, TestBuilder-SC acts as a verification layer on top of SystemC. The API in TestBuilder-SC has been proposed to the SystemC's verification working group to be adopted as a standard.

1. Introduction

Recently SystemC has released version 2.0 that is capable of capturing System-Level Designs. Together with the basic facilities in version 1.0, it seems that we may be able to complete a design from System-Level down to RTL level in a seamless flow.

However, one important element in creating a product is verification. Does the current SystemC standard contain enough features for an effective functional validation of a real design?

In order to answer this question, we have gone through the exercise of creating a transaction-based test bench for a simple design with a pipelined bus. While we were able to create a basic transaction-based test bench using the current SystemC 2.0 standard, we have also identified several aspects that are not convenient or effective in supporting a realistic verification effort.

This paper reviews the design and the test benches that we used to evaluate a SystemC-based verification environment. After the review of this system, we describe the missing verification aspects, and how the existing TestBuilder concepts can be adapted into a SystemC verification layer to address these missing verification aspects in the current SystemC 2.0 standard.

We have been driving the standardization effort in the SystemC verification working group, gathering requirements and feedback from the group members. While we plan to release a new library called TestBuilder-SC as an open-source library, we are also submitting this library as a proposal to the SystemC standard, and our implementation as the reference implementation.

This paper explains some basic SystemC and TestBuilder concepts as we describe the design and the transaction-based test bench. For a more detailed description of SystemC and TestBuilder concepts, please check out the papers by Stuart Swan [4] and by Cox et al. [5].

2. Basic Transaction-Based Test Benches

SystemC 1.0 provides a set of modeling constructs that are similar to those used for RTL and behavioral modeling within an HDL such as Verilog or VHDL. Similar to HDLs, users can construct structural designs in SystemC 1.0 using modules, ports, and signals. SystemC 2.0 enables modeling of systems both at and above the RTL level of abstraction. It introduces a set of features for generalized modeling of communication and synchronization, using channels, interfaces, and events [4].

The different levels of abstraction in SystemC 1.0 and 2.0 provide the basis upon which transaction-based test benches can be created. As shown in Figure 1a, a transaction-based verification methodology partitions the system into transaction-level tests, transactors, and the design. 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 signal manipulation at the RTL/Signal level.

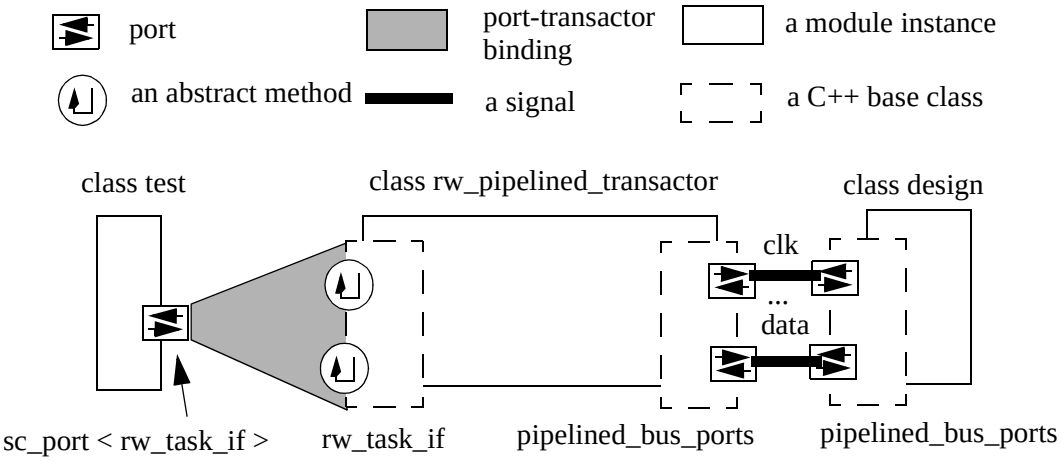
Let's look at a transaction-based test bench for a design with a pipelined bus interface, shown in Figure 1b. The complete code can be obtained upon request.

Using abstract methods in C++, the task-based interface can be declared as:

```
class rw_task_if : virtual public sc_interface {
public:
  typedef sc_uint<64> addr_t;
```

Figure 1a : transaction-based verification

Figure 1b : an example with a pipelined bus




```
  typedef sc_uint<64> data_t;
  struct write_t {
    addr_t addr;
    data_t data;
  };
  virtual data_t read( addr_t * ) = 0;
  virtual void write(write_t * ) = 0;
};
```

This abstract base class specifies two abstract methods, *read* and *write*, and their related data types, representing 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 another 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;
};
```

The signals in this class represent the RTL-level interface in which a design under verification communicates to its environment. The class *sc_module* is provided by SystemC to specify a block with ports; 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 concrete 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(addr_t *);
  virtual void write(write_t *);
private:
  fifo_mutex addr_phase;
  fifo_mutex data_phase;
};
```

The macro *SC_CTOR* is provided by SystemC to signify the constructor of a module. The class *fifo_mutex* is a mutex that grant access in a first-come-first-serve manner.

The job of a transactor is to convert a transaction as modeled by a function call into signal-level communication, and vice versa. For example, the *read* method is implemented as:

```
data_t rw_pipelined_transactor::read(addr_t * addr) {
  addr_phase.lock();
  bus_addr = *addr;
  addr_req = 1;
  wait ( addr_ack->posedge_event() );
  addr_req = 0;
  wait ( addr_ack->negedge_event() );
  addr_phase.unlock();

  data_phase.lock();
  wait ( data_rdy->posedge_event() );
  data_t data = bus_data.read();
  wait ( data_rdy->negedge_event() );
  data_phase.lock();
}
```

This method translates a read transaction into a series of signals according to the specific protocol at the signal-level interface. Because a pipelined bus is used, two mutexes, *addr_phase* and *data_phase*, are used for the two phases. At the beginning of the address phase, the corresponding mutex is locked so that there is at most one address communication on the bus at any given time. At the end of the address phase, the corresponding mutex is unlocked so that, although the data phase has not finished yet, another call to this transactor can still start its address phase. The three methods, *posedge_event()*, *negedge_event()*, and *read()*, are provided by SystemC to denote a positive edge transition of a signal, a negative edge transition of a signal, and the access of the value in a port. The procedure *wait()* in SystemC suspends the thread of execution until the event specified in the argument occurs.

With the detailed protocol abstracted by the transactor, a test can be written in a form independent of the actual signal-level interface:

```
class test : public sc_module {
public:
  sc_port< rw_task_if > transactor;
  SC_CTOR(test) { SC_THREAD(main); }
  void main();
};
void test::main() {
  // simple sequential tests
  for (int i=0; i<3; i++) {
    rw_task_if::addr_t addr = i;
    rw_task_if::data_t data = transactor->read(&addr);
    cout << "received data : " << data << endl;
  }
  // simple concurrent tests
  rw_task_if::addr_t addr[ 2 ]; addr[ 0 ] = 0; addr [ 1 ] = 1;
  rw_task_if::data_t data[ 2 ];
  SC_FORK
    sc_spawn_method( &data[ 0 ], transactor[ 0 ],
                     &rw_task_if::read, &addr[ 0 ]),
    sc_spawn_method( &data[ 1 ], transactor[ 0 ],
                     &rw_task_if::read, &addr[ 1 ])
  SC_JOIN
  cout << "received data : " << data[ 0 ] << ","
                             << data[ 1 ] << endl;
}
```

The first half of the test generates a series of three read transactions, starting a new one after the previous one has completed. The second half generates two read transactions in parallel, so that they exercise the pipeline. The macro *SC_THREAD* creates a new thread of execution for the method *main* during simulation. The *sc_spawn_method* function is similar to *create* in pthread, creating a new C++ thread and executing the method specified in the third argument for the object in the second argument. The first argument is the object to store the return value, and the last argument is the argument to the method.

The constructs, *SC_FORK*, and *SC_JOIN*, together with *sc_spawn_method*, are dynamic spawning enhancements proposed to SystemC and their implementation is distributed as an example in the SystemC 2.0 distribution kit.

It is important to note that the port of this test has the *rw_task_if* interface as the template argument. Because of this, the test can be reused with other designs with a different bus interface, by plugging in a appropriate transactor.

To complete the example, the code for the design and the netlist is shown below:

```
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) ... }
private:
  ...
};
```

```
int sc_main(int argc, char *argv[ ]) {
  ...
  // the static structures in this simulation
  test t ("t"); // the test
  rw_pipelined_transactor tr ("tr"); // the transactor
  design duv ("duv"); // the design under verification
  sc_clock clk ("clk",20,0.5,0,true); // a clock

  // the signals to connect the static data structures
  sc_signal < bool > rw;
  sc_signal < bool > addr_req;
  ...

  // connecting the signals and transactors to
  // the ports of the modules
  t.transactor = tr;
  tr ( clk.signal(), rw, addr_req, ... );
  duv ( clk.signal(), rw, addr_req, ... );

  // start simulation
  sc_start(10000);
  ...
}
```

The design implementation contains the same set of signals for the pipelined interface, with two C++ threads to respond to the two phases of the pipeline.

This example illustrates a preferred style for transaction-based verification in SystemC. In the remainder of this paper, we will discuss how to use TestBuilder-SC to extend this example for a more effective verification effort.

3. Advanced Transaction-Based Test Benches

Although existing SystemC features are sufficient for creating a basic test bench, there are still many missing pieces that are necessary for an effective verification effort. These include:

- constrained randomization for effective generation of random stimulus.
- creation of a event and transaction database for effective debugging and coverage analysis.
- complex synchronization and assertions for effective coordination among concurrent tests and detection of illegal behaviors.
- infrastructure and scripts for interaction with HDL simulators and other tools.

TestBuilder was developed to address similar issues in verification for HDL simulators such as NC-Sim. However, because it was developed independent of SystemC, TestBuilder and SystemC do not operate together seamlessly.

In September 2001, a prototype for connecting TestBuilder 1.2 and SystemC was made available in www.testbuilder.net. While it is in general very useful to our customers, the translations between data types and events are not very efficient. Some functionality is duplicated in both libraries with different APIs and use models; this could lead to confusion and deepen the learning curve.

As a result, the TestBuilder team has started a project to create another version of TestBuilder directly on top of SystemC. It is called TestBuilder-SC and is designed to have the same look and feel of SystemC, but fill in the hole so that our customers would be able to perform transaction-based verification with linkage to analysis tools offered by Cadence.

Using our experience in developing TestBuilder, and customer feedback on the use models of TestBuilder and SystemC, the TestBuilder team was able to design the interface for TestBuilder-SC in a relatively short amount of time, and is in the processing of implementing an open-source library for external distribution.

3.1 TestBuilder-SC

Taking the pipelined bus design as an example, let's discuss how TestBuilder-SC would be able to improve the verification process.

Handling arbitrary data types.

TestBuilder-SC contains a data introspection facility to analyze third-party data types and user-specified composite types.

For example, the *rw_task_if* interface uses a SystemC data class *sc_uint<64>* and a composite type *write_t*. We would like to extract information from these types and manipulate them without asking the user to modify the source code.

The data introspection facility uses partial specialization of templates to analyze a data type. Two templates are developed for this purpose:

```
template<typename T> class tb_extensions;
template<typename T> class tb_smart_ptr;
```

The *tb_extensions* template extends arbitrary data types to have a standard access interface, called *tb_extensions_if*. This standard access interface allows a piece of code to extract type information, perform value access and value assignment to a variable without requiring the code to have explicit type information at compile time.

The *tb_smart_ptr* template extends the arbitrary data type to have extra storage for callback registration, randomization, and other purposes.

This facility can be considered as a C++ version of the Verilog PLI standard. The ability to handle arbitrary data types enables the import of legacy code and facilitates the reuse of the same code for multiple libraries. It is a crucial

basic building block for constrained randomization, variable recording, and transaction attribute recording.

Constrained randomization.

Constrained random tests are an important element in a state-of-the-art verification environment. Using the data introspection facility discussed previously, TestBuilder-SC supports constrained randomization on arbitrary data types. For example, random write requests for our *rw_pipelined_transactor* transactor can be generated by:

```
tb_smart_ptr < rw_task_if::write_t > write;
for (int i=0; i<3; i++) {
  write->next(); // generate a random value
  transactor->write(&write);
  cout << "send data : " << write->data << endl;
}
```

The *tb_smart_ptr* template is designed to have very similar use model as a C/C++ pointer. Fields and methods are accessed by the overloaded operator->. The *next()* method generates a new random value, and the field access returns the value of the *data* field to *cout.*

Expressions and constraints can be created using *tb_smart_ptr* as well. For example, the following code specifies an address range and a relationship between the address and the data:

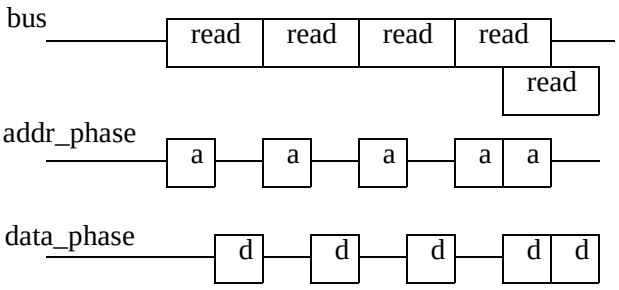
```
class write_constraint : public tb_constraint_base {
public:
  tb_smart_ptr< rw_task_if::write_t > write;
  TB_CTOR(write_constraint) {
    TB_CONSTRAINT(  write->addr() < 0x00FF );
    TB_CONSTRAINT( write->addr() != write->data() );
  }
};
```

This constraint creates two Boolean expressions regarding the fields of the variable *write*. TestBuilder-SC first translates these expressions to Binary Decision Diagrams (BDDs), and then generates only values that satisfy these expressions.

Creation of a event and transaction database.

SystemC can be viewed as a hybrid language which contains elements of both procedural and hardware description languages. While the procedural portions of a SystemC description can be analyzed and debugged using a typical C++ debugger, the hardware aspects are best analyzed and debugged using a waveform tools.

For example, the activities in the pipelined bus can be captured in a series of transactions as shown in the following figure.



This diagram succinctly displays the activities in terms of transactions, with three non-overlapping back-to-back reads and two overlapping reads. The related address and data values are displayed in the corresponding child-transactions in the address phase and data phase.

Furthermore, a coverage tool may generate a report that five read transactions have been executed, with one occurrence of simultaneous activities on both the address bus and the data bus.

A TestBuilder-SC test bench for SystemC can generate a database from which these tools can extract the relevant information. A transaction is generated through the concept of a fiber, with each fiber corresponding to a line in the waveform display. For example,

```
addr_phase.lock();

tb_tr::transaction_type<rw_task_if::addr_t>
  addr_phase_handler("addr phase", addr_phase_fiber);
tb_tr::transaction h =
  addr_phase_handler.begin_transaction(*addr);

bus_addr = *addr;
addr_req = 1;
wait ( addr_ack->posedge_event() );
addr_req = 0;
wait ( addr_ack->negedge_event() );
addr_phase.unlock();

addr_phase_handler.end_transaction(h);
```

Three lines of code (shown in italics) are added to the address phase in *rw_pipelined_transactor::read()* to capture the address phase of a read transaction. The first line creates a *addr phase* transaction type in the fiber *addr_phase_fiber.* The second line begins a transaction via the *begin_transaction()* method, and the third line ends a transaction via the *end_transaction()* method. The value in variable *addr* is added in the beginning of the transaction as an attribute. The attribute type, specified in the parameter of the template *tb_tr::transaction_type*, can be any arbitrary data type. The implementation of *begin_transaction()* uses

the data introspection facility to extract the name and the values of the attribute. The overall read transactions and the transactions for the data phase can be generated similarly.

The database may also contain data for value transitions in variables. This can be done by registering a value change callback to a variable via *tb_smart_ptr* instantiation.

Complex synchronization and assertions.

A realistic design typically has multiple interfaces and complex protocols, and operates in a complex concurrent environment. Capturing the environment of such a design in a test bench requires special features in a tool.

In order to globally coordinate traffic generation among the interfaces, and to check whether the outputs from the design occur in the right sequence, complex synchronization is necessary. The ability to efficiently wait on multiple events such as signal transitions and semaphore waiting (in a sequential, conjunctive, or disjunctive manner) is very important. Similarly a user should be able to create a monitor to verify that a complex protocol is being met throughout the simulation.

TestBuilder-SC provides a set of synchronization classes and the ability to compose them into event expressions. These synchronization classes extend the existing SystemC synchronization class to have better support for verification purposes. The event expression provides extra expressive power to model complex protocols efficiently. Temporal assertions enable the test bench to detect illegal events in simulation, and are fully integrated with the transaction recording facility in TestBuilder-SC.

Infrastructure and scripts.

The SystemC standard describes the API with which a user specifies the design and the test bench. However, a complete verification environment requires more than just the design and test bench description.

TestBuilder-SC includes several scripts for detecting utilities in a customer's C++ working environment, configuring your SystemC and TestBuilder-SC installation, and building and running simulations.

TestBuilder-SC objects support an abstract interface called *tb_object_if*, which contains various abstract methods that you can call in a debugger and other tools.

Although TestBuilder-SC is different from TestBuilder, the TestBuilder team is working on an interoperability guideline so that existing TestBuilder IP can still be used in a SystemC/TestBuilder-SC environment.

Finally, a complete verification environment also requires the ability to simulate both HDL descriptions and SystemC/TestBuilder-SC descriptions in the same simulation run. The TestBuilder team is currently working on an intermediate solution based on SystemC reference implementation, with a high-speed proprietary version being developed in the NC team.

4. Summary

This tutorial illustrates the use of SystemC and TestBuilder-SC to create effective test benches. Through the use of system-level features in SystemC 2.0, a transaction-based test bench can be created. However, in order to support realistic verification efforts, extensions to SystemC 2.0 are needed.

We have identified several areas that need improvement, and worked with the TestBuilder engineering team to come up with a proposal.

Through the use of constrained random tests, more tests can be created, leading to higher confidence in a design, with better utilization of the simulation cycles in a customer's server farm. Through the use of recording API, a database is generated in SystemC simulation, with sufficient data for Cadence's verification tools to facilitate the debugging and coverage analysis process.

Because many of our tools use a C++ library in one form or another for design and test bench capture, SystemC has the potential to be the unifying language with which various tools in a verification environment can talk to each other.

5. Acknowledgement

The authors would like to thank the many people who contributed to the development of SystemC and the TestBuilder team for designing and developing both TestBuilder and TestBuilder-SC open-source libraries.

6. References

[1] Functional Specification for SystemC 2.0, available at www.SystemC.org.

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

[3] Functional Specification and User Guide for TestBuilder 1.3, available at www.testbuilder.net.

[4] Stuart Swan, An Introduction to System Level Modeling in SystemC 2.0, white paper, available at 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.