Utilizing SystemC for Design and Verification

Co-Authored by
Alan Ma and Allan Zacharda

www.model.com

Mentor Graphics®

Introduction to SystemC

General Overview

Background and History

SystemC has its roots in the idea of using C++ for modeling hardware and software components of a system. SystemC was released to the public in September 1999 and ownership put into the hands of the Open SystemC Initiative (OSCI). Since then development of the language has been done by the Language Working Group of OSCI.

SystemC consists of a C++ class library and simulation scheduler which supports the description of hardware. SystemC "extends" the C++ language using classes to achieve the following:

1) A notion of time is introduced and implemented.
2) Processes are defined which are executed in "parallel" within the simulator.
3) Hardware data types are introduced. These include arbitrary precision integers, fixed point numbers and 4 valued logic.
4) Modules are introduced as a means of encapsulating behavior and describing hierarchy.

SystemC Development Environment

Since SystemC is C++ the development environment is the standard C/C++ development. There is a reference simulator and class library that may be downloaded from the OSCI website (www.systemc.org).

Modeling Characteristics & Levels

A number of terms are used to characterize models and to label model types used in SystemC. These terms are listed below and defined in the glossary.

Terms used to characterize models:

- UnTimed Functional (UTF)
- Timed Functional (TF)
- Bus Cycle Accurate (BCA)
- Pin Cycle Accurate (PCA)
- Register Transfer (RT) accurate

Model types:

- System Architectural Model
- System Performance Model
- Transaction Level Model (TLM)
- Functional Model
- Register Transfer Level (RTL) model

SystemC Benefits and Advantages

Some SystemC designs start as Functional Models while much of the SystemC coding done is at the TLM level. Almost always the TLM serves as an executable platform that is accurate enough to execute software on.

The number one reason for the use of SystemC is the significantly increased simulation performance at the TLM level over executable platforms modeled at the RT level using Verilog or VHDL. SystemC TL models are fast enough to serve as a software development platform allowing for early software development and for co-simulation of hardware and software. Both TL and functional models are fast enough for system level architectural modeling and analysis.

Typical numbers for a System on Chip type design range from ~1K cycles per second to upwards of ~300-400K cycles per second. Where in this range a particular design falls is largely dependent upon how the processor is modeled in the system. If an instruction set simulator (ISS) is used then performance is typically in the 1K to 10K range. If the processor is modeled as a direct-connect to the system bus then performance moves into the 100K+ range.

The second reason for SystemC use is functional verification. The same executable platform that is used to develop the software is often used for verification of the entire system. This verification occurs early on in a project and the TLM becomes a golden reference for the entire system. Because SystemC is C++, it has a number of inherent properties, such as classes, templates and inheritance, that lend themselves to verification. These capabilities are extended with the SystemC Verification Library (discussed later) making SystemC a powerful verification language as well as modeling language.

SystemC Flows and Usage

Figure 1 shows a typical SystemC based system design flow. The TLM serves often as both an executable platform for software and as a golden reference model for verification.
The path to gates from the TLM is either direct using behavioral synthesis or the TLM is hand translated to either a SystemC or HDL (Verilog or VHDL) RTL model and then synthesized.

Verification is done at the TLM level. As the TLM is translated to the RTL or gate levels the same verification may be used through the use of adapters (sometimes called transactors).

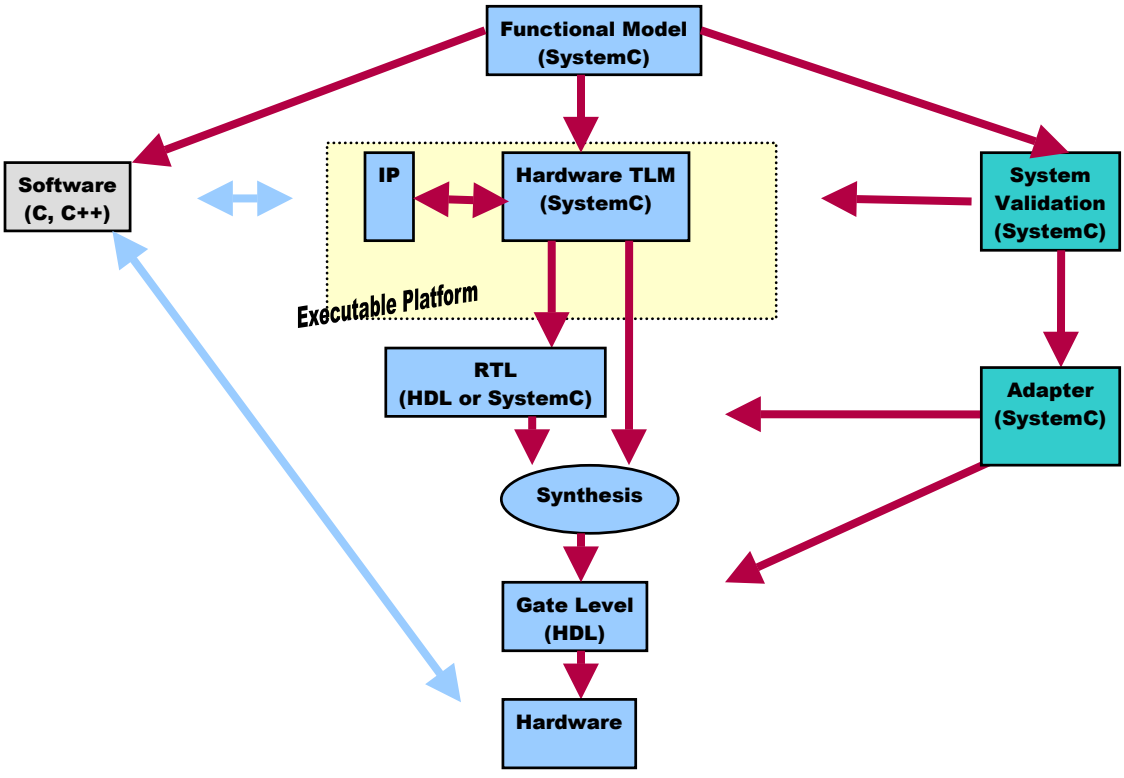

Figure 1 - SystemC Design Flow

Figure 2 focuses on the hardware design flow. From the System or TLM model one may take a path using behavioral synthesis to gates or through RTL synthesis.
The Test bench may be refined from a functional level (UTF/TF) but it is not required. With the use of adapters, testing from a functional level test bench may be accomplished on RTL and gate level models.

RTL models may be in SystemC, Verilog, or VHDL. Gate level models are typically written in Verilog or VHDL. Co-simulation is required if the SystemC test bench is used with a Verilog or VHDL design representation.

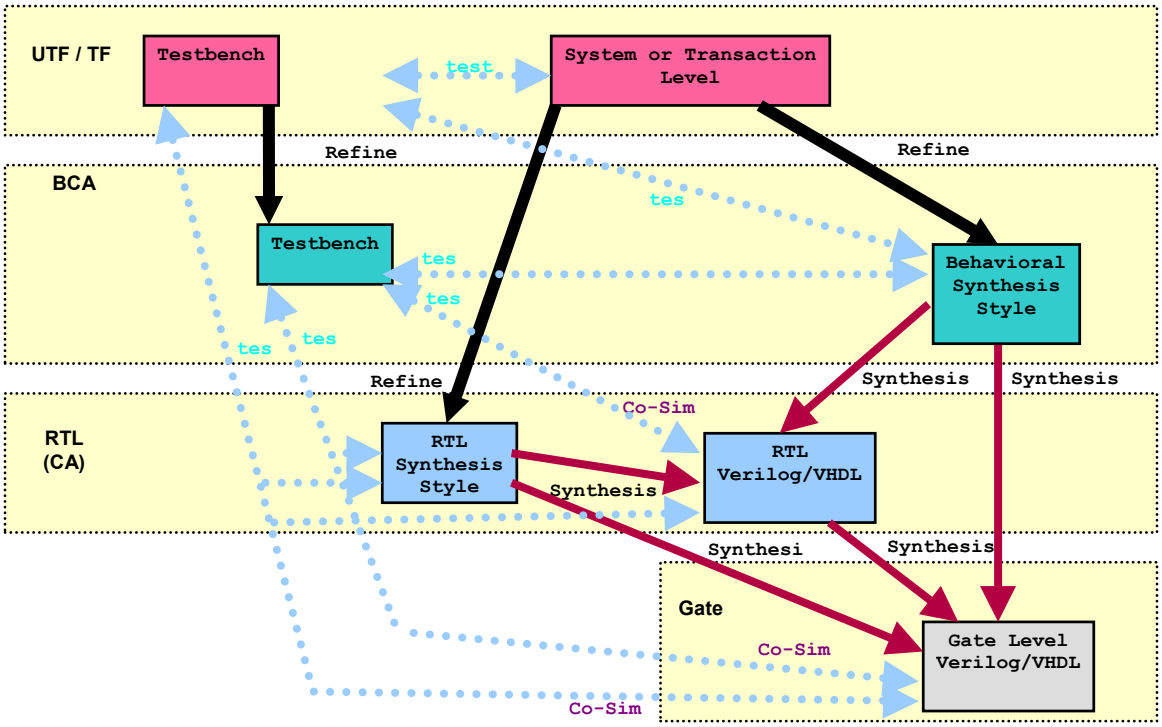

Figure 2 – SystemC Hardware Design Flow

SystemC Language Architecture

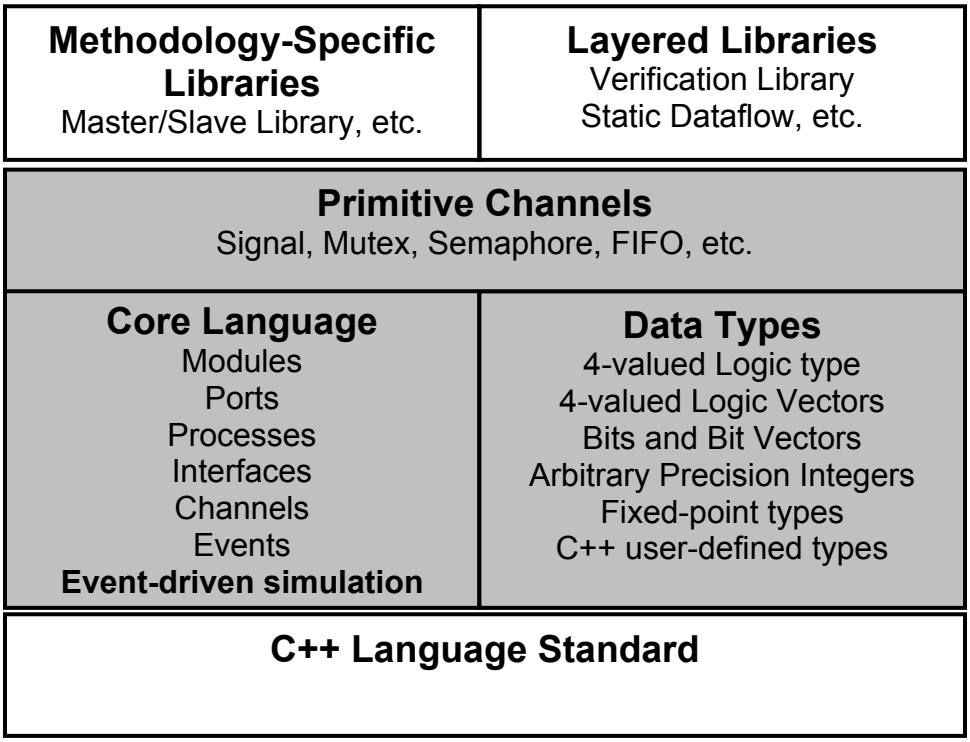

Figure 3 - SystemC Language Architecture

Figure 3 shows the SystemC language architecture. The blocks shaded with gray are part of the SystemC core language standard. SystemC is built on standard C++. The layers above or on top of the SystemC standard consist of design libraries and standards considered to be separate from the SystemC core language. The user may choose to use them or not. Over time other standard or methodology specific libraries may be added and conceivably be incorporated into the core language.

The core language consists of an event-driven simulator as the base. This simulator works with events and processes. The other core language elements consist of modules and ports for representing structure, while interfaces and channels are used to describe communication. The data types are useful for hardware modeling and certain types of software programming. The primitive channels are built-in channels that have wide use such as signals and FIFOs.

SystemC System

A SystemC system consists of a set of modules.

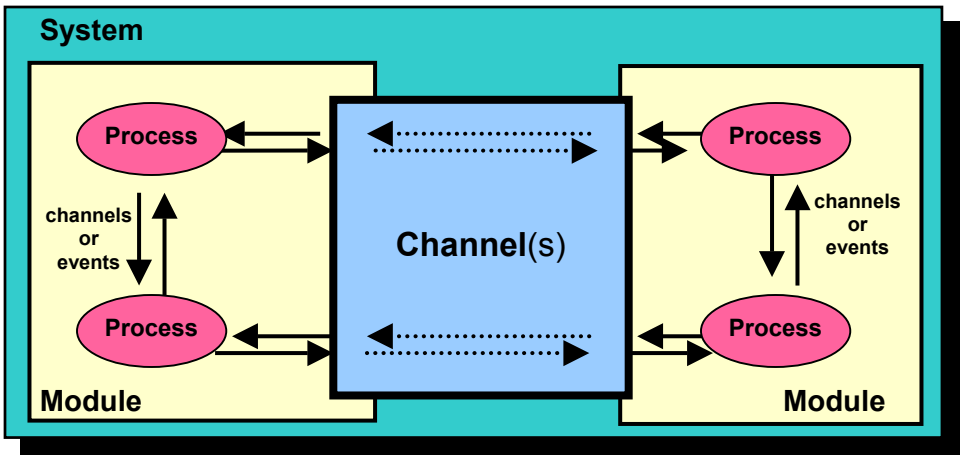

Figure 4 - SystemC System

Modules contain concurrent processes. Processes describe functionality and communicate with each other through channels or events. Inter-Module communication is through channels Modules are used to create hierarchy. The top level is not a module but is a SystemC function `sc_main()`.
Modules may be instantiated inside of `sc_main()` as well as inside of other instances of modules. Some tools hide `sc_main()` from the user so the user only sees a top level module with instances of other modules inside of it.

Basic Modeling Structure

A module may represent a system, a block, a board, a chip and so forth. The ports represent the interface, pins etc., depending upon what the module represents. See Figure 5.

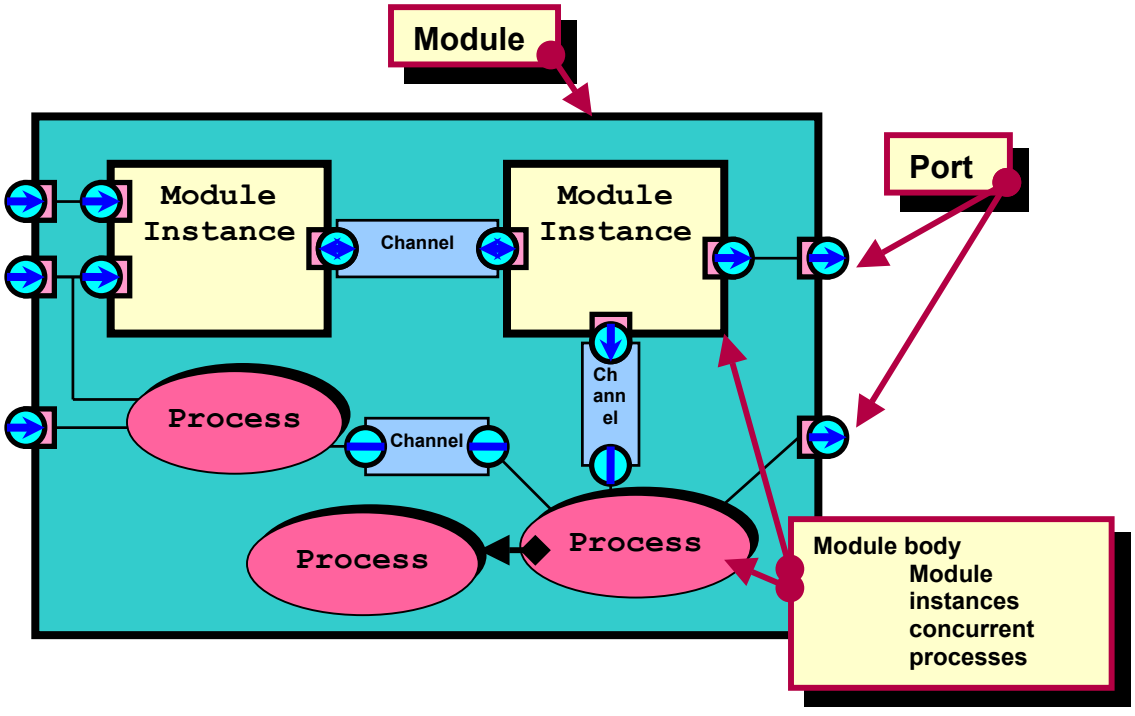

Figure 5 - SystemC Module

Module instances at a peer level are connected through their ports with a channel. Parent to child connection of module instances is done port to port. Processes communicate through channels or events.

The basic communication structure includes interfaces, channels and ports.

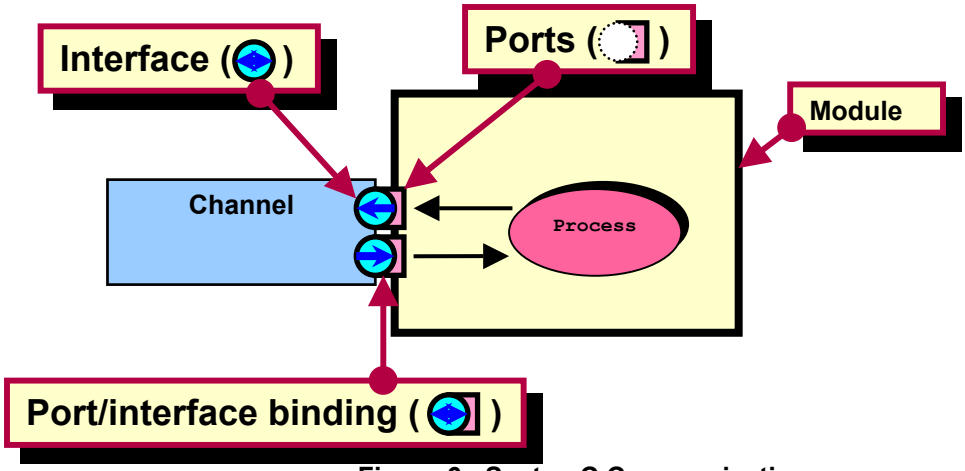

Figure 6 - SystemC Communication

An interface defines a set of access methods. It is purely functional and does not provide the implementation of the methods. It only provides the method's signature. Interfaces are bound to ports – they define what can be done through a particular port.

The channel implements one or more of an interface's methods. It is the container for communication functionality.

A port is "bound" to an interface. It is the object through which modules, and hence its processes, can access the methods of a channel. A process accesses the channel by applying the interface methods to a port. To connect a channel to a port, the channel must implement the interface the port is bound to. This allows for refinement of channels independent of ports. Ports may be bound to multiple channels.

There are two general types of channels: primitive and hierarchical.

Primitive channels have no visible structure, they contain no processes and cannot directly access other channels. Examples of primitive channels in SystemC are:

- `sc_signal<T>`
- `sc_signal_rv<N>`
- `sc_fifo<T>`
- `sc_mutex`
- `sc_semaphore`
- `sc_buffer<T>`

Hierarchical channels have structure. They appear similar to modules in that they can have processes, ports and instances of other channels or modules. They can directly access other channels.

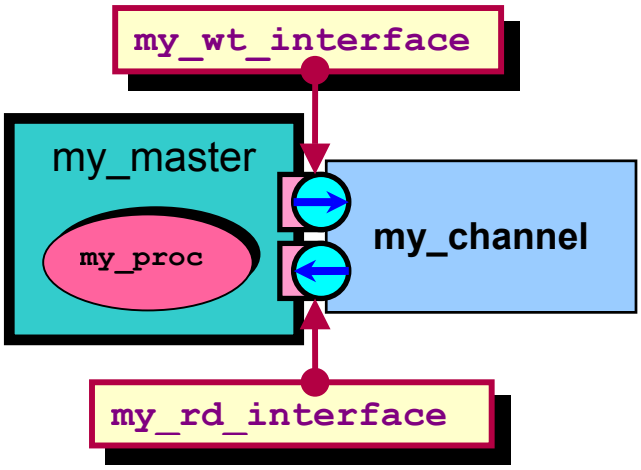

Figure 7 - Ports, Channels & Interfaces

In Figure 7 `my_proc` is a process inside of the module `my_master`. `my_wt_interface` is an interface that *defines* a method (function) called `wt()` for accessing some channel through the port it is bound to. `my_rd_interface` is an interface that *defines* a method (function) called `rd()` for accessing some channel through the port it is bound to. `my_channel` *implements* the functions `rd()` and `wt()`. When `my_proc` calls `rd()` or `wt()` the code that is actually executed is in `my_channel`.

Events and Dynamic Sensitivity

An event is the basic synchronization object. An event is used to synchronize between processes. Channels use events to implement blocking.

A process is "triggered" or caused to run based on the occurrence of events as defined by a sensitivity list which consists of one or more events. SystemC supports both static and dynamic sensitivity lists. A static list is determined before simulation begins where a dynamic need not be defined before simulation starts.

Processes may wait for an event to occur. Dynamic sensitivity coupled with ability of process to wait on an event provide for efficient simulation (faster) and simple modeling at higher levels of abstraction.

Processes

Functionality is described in processes. Processes look very much like normal C++ functions with slight exceptions. Processes are registered with the SystemC scheduler to have a particular type. A process is invoked by the scheduler based on its sensitivity list. Some processes execute when called and return control to the calling mechanism (they behave like a function). Some processes are called once, and then can suspend themselves and resume execution later (they behave like threads).

Processes are not hierarchical – you can not have a process inside another process. Modules are used to create hierarchy. Processes use channels or events to communicate with each other. They use timing control statements to implement synchronization.

SystemC supports two different types of processes:

- Methods (SC_METHOD)
- Threads (SC_THREAD)

Refinement Overview

During the SystemC design process there are two types of refinement. The first is model refinement. This is where the model *itself* is refined to a more detailed level. The second type of refinement is communication refinement. This is where the *channel* connecting the model is changed or refined.

This ability to separate model refinement from communication refinement is a powerful feature of SystemC. For one thing it allows for step-wise refinement of a design. Instead of requiring the designer to refine the entire design when moving from one level to another (Transaction to RTL for example) in one step, multiple steps may be taken. Since each step may be verified, fewer errors are in the refinement process. Another advantage is greater flexibility in the refinement and verification process.

Verification Overview

Functional verification is "proving" that the models you write reflect the specification of the thing you wish to model. In SystemC functional verification is done through simulation, applying stimulus to the Device Under Test (DUT) and verifying the response against an expected result. The DUT may be models at varying levels, from System Architectural Models (SAM) to Register Transfer Level models (RTL) to gate level models written in other languages such as Verilog or VHDL. Stimulus is usually generated within a testbench and verification of results is often achieved in the same testbench.

With SystemC, a system level approach is generally recommended for designs.
System level testing means testing the entire design all together using a single testbench for both stimulus and response checking. The advantage of this approach is each module is tested in the system context it will operate in. There is a uniform system model. This means less duplication of effort than an approach where each unit is tested individually and it encourages common testing methodologies and strategies. Additionally it is useful to the entire team and not just the individual designer.

The disadvantage of system level testing is it takes more planning and forethought. A poor system level testbench will be rejected by designers.

Modeling

Functional Modeling

Functional level modeling describes modeling done at levels above Transaction Level Modeling (TLM) and encompasses System Architectural Models (SAM) and System Performance Models (SPM). Modeling at this level is algorithmic in nature and may be timed or untimed. Models may represent software, hardware or both.

The behavior of the models at this level is described algorithmically. Typically behavior is described as generators and consumers of data. Processes may be assigned time for performance analysis purposes. This timing is not cycle accurate but rather describes the time to generate or consume data or to model buffering or data access.

The behavior of the interfaces between modules is described using communication protocols. Time may be assigned to the transport of data for performance analysis.

These type of models are used to explore architectures, for proof of algorithms and for performance modeling and analysis.

Consider an example of a simple arithmetic unit, add_sub (Figure 8) which does add and subtract operations. This simple example will be used to illustrate the different levels of modeling. It is simple yet one can use it to project how more complex designs might be accomplished using the same modeling style. The code for this example is in the Code Appendix.

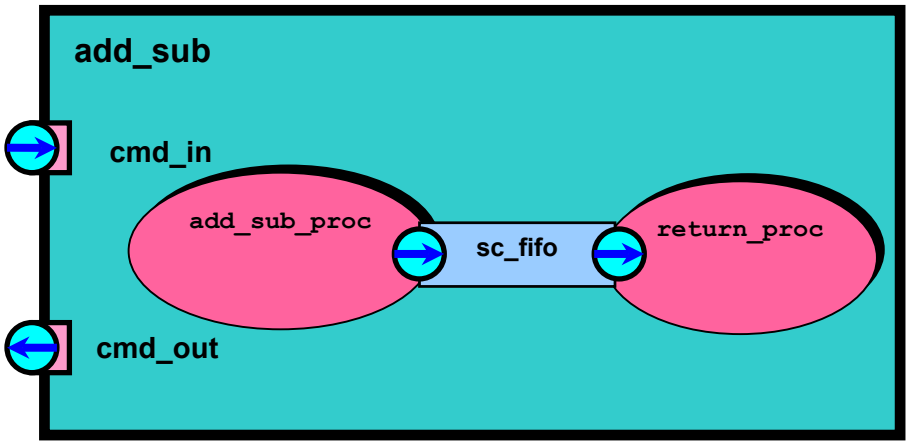

Figure 8 - add_sub Block Diagram

The module add_sub is implemented using two processes, `add_sub_proc` and `return_proc` and one internal `sc_fifo` channel. There is one input port `cmd_in` and one output port `cmd_out`.

The data type of the ports and the local channel is a user defined type `cmd`. `cmd` is a class that has four data members: `command`, `op_1`, `op_2` and `cmd_id`. The `command` data member is of type `cmd_t`, which is an enumerated type as follows:

```
enum cmd_t {ADD, SUB};
```

`op_1` and `op_2` are of type `int` and are the operands for the operation to be performed. `cmd_id` is of type `int` and is used as a unique id for each `cmd` object.

The process `add_sub_proc` reads a `cmd` from the `cmd_in` port. It looks at the `command` field to determine what operation to perform. It then performs the operation and puts the upper 32 bits of the results into the `op_1` field of the received `cmd`, the lower 32 bits into the `op_2` field and writes the resulting `cmd` into `result_buff` the local `sc_fifo` channel.

The code for this module is event based and algorithmic in nature. `add_sub_proc` waits for an event on the input port (write of a `cmd` object), processes the data and puts the result in a buffer. `add_sub_proc` has statements which represent the processing time of the add_sub unit.

`return_proc` also waits on an event on the `result_buff` (write of a `cmd` object), then reads a `cmd` object and writes it to the output port. Note that the transport of the data modeled with `sc_fifo` channels with zero time delay.

To complete our simulation model we add a simple testbench model `tb` (Figure 9). tb consists of two processes, `test_main` and `receive_cmds`. `test_main` generates `cmd` objects and sends them to `add_sub`. It also generates expected results for each `cmd` sent. `receive_cmds` receives the results from `add_sub` and verifies the data received against the expected results for correctness. The `cmd_id` field of the `cmd` object is used to identify each command for verification purposes.

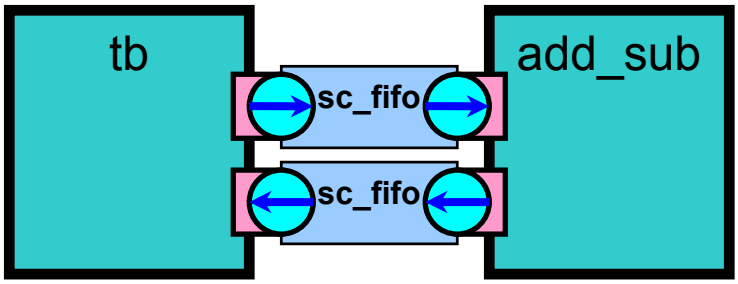

Figure 9 - add_sub System Block Diagram

The code for tb is in the Code Appendix

Transaction Level Modeling

Transaction level models typically describe only hardware. Many designs, such as typical System on Chip (SoC) designs at this level have one or more buses modeled as channels. Connecting to this central bus are master devices such as processors and slave devices such as memories, accelerators and so forth.

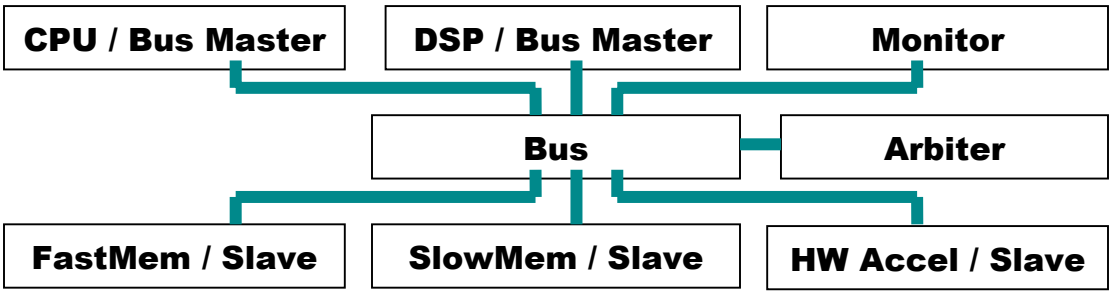

Figure 10 - Transaction Level Model Example

Data transfers are modeled as transactions such as read and write. The model behavior is typically timed algorithmic descriptions. The interfaces do not have pin level detail. They may or may not be cycle accurate depending upon the level of modeling desired.

TLM models simulate much faster than lower Register Transfer Level (RTL) models because pin level detail is not present, model descriptions are simpler and code is event based not clock-driven. Typically, TLM simulation results are 2 to 4 orders of magnitude faster than equivalent RTL models. This speed allows the use of a TLM design as an execution platform for:

- Software development
- Verification platform
- Models for IP

An example TLM bus, simple_bus(Figure 11), is available with the SystemC library from www.systemc.org. Key features are:

- Cycle based synchronization bus (simple_bus)
- Implemented as a channel
- Multiple masters
- Multiple slaves
- Arbiter port

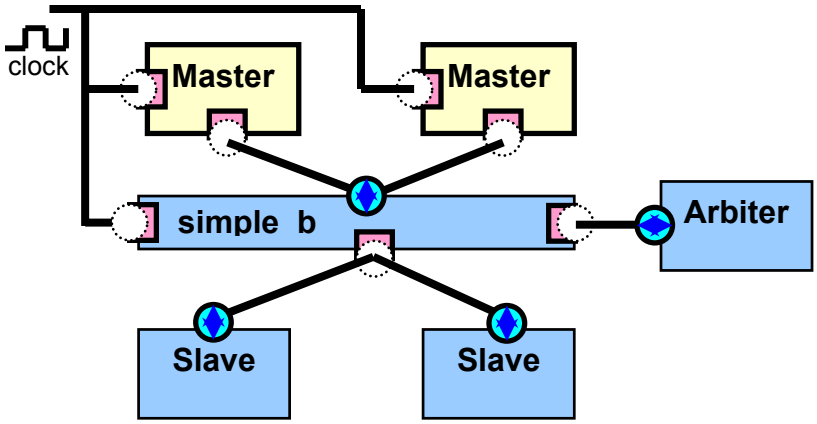

Figure 11 - simple_bus Block Diagram

A simple_bus master(Figure 12**):**

- A simple_bus master is a module that has a master port
- A Master can:
    - read or write with blocking or non-blocking behavior
    - communicate directly with a slave (for debugging purposes)
    - Reserve the bus for a subsequent transaction with a lock flag
    -

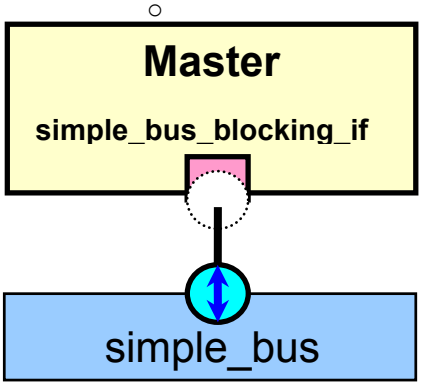

Figure 12 - simple_bus Master

A simple_bus slave(Figure 13**):**

- simple_bus slave must be a channel that implements simple_bus_slave_if
- Slave can be instantiated with a start address and end address parameters

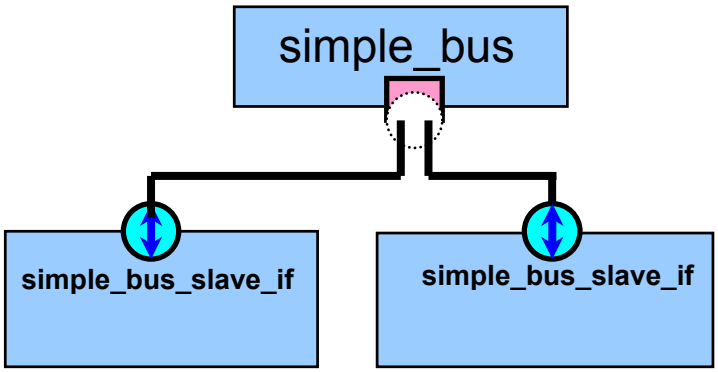

Figure 13 - simple_bus Slave

The add_sub and tb modules discussed earlier were connected using sc_fifo channels(Figure 9).

Let's connect these modules to the simple_bus to create a TLM of the add_sub system. The first problem encountered is the ports on both tb and add_sub need to be connected to channels that implement the sc_fifo_read_if and sc_fifo_write_if interfaces. simple_bus does not implement either of those interfaces. To solve this problem we introduce the idea of adapters. An adapter is simply a channel that implements multiple interfaces and "translates" or adapts between them. To connect tb and add_sub to the simple_bus we will use 4 adapters as shown in Figure 14.

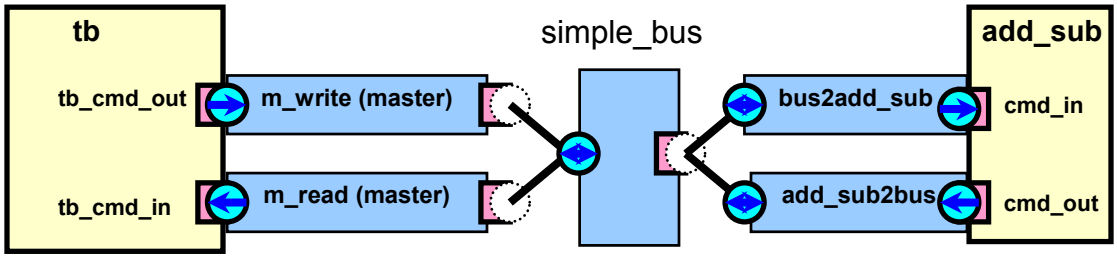

Figure 14 - add_sub With simple_bus

The bus2add_sub channel implements the simple_bus_slave_if and the sc_fifo_in_if interfaces and "translates" between them. The add_sub2bus channel implements the simple_bus_slave_if and the sc_fifo_out_if interfaces and "translates" between them.

Note that in the refinement from a functional model to a TLM model that the code in tb and add_sub does not change. In this particular example only a TLM bus was added along with adapters.

The adapters do the memory mapped responses on the simple_bus.

We could if we desired merge the adapters with our modules to create the topology shown in Figure 15. Motivation for taking this step depends upon the ultimate target level of the modules. It does not gain anything functionally. In the case of the tb one could argue is a waste of time. For the add_sub it would depend on the next step in the refinement process. If add_sub is to be made an RTL design for example. SystemC provides the flexibility to take either path for the tb. It may be merged with the adapter at the TLM level or first be refined to an RTL level and then merged with the adapter.

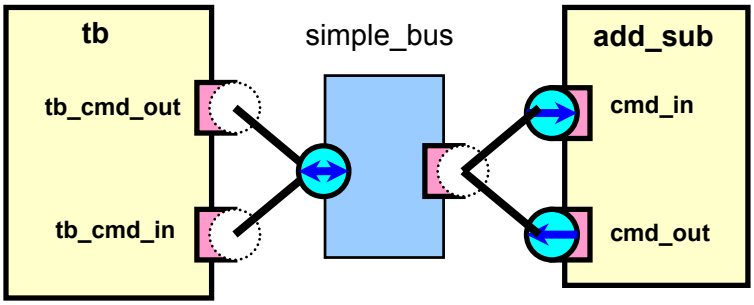

Figure 15 - add_sub With Merged simple_bus Adapters

Register Transfer Level (RTL) Modeling

RTL models describe hardware and contain a complete detailed functional description. Every register, every bus, every bit is described for every clock cycle.

In general, writing RTL *style* code in SystemC is similar to writing RTL code in either the Verilog or VHDL hardware description languages. If the targeted code is for synthesis then the same general rules apply, such as avoiding accidentally inferred latches.

Although thread processes may be used at the RT level, in general method processes are used for performance reasons. Method processes have less context switching overhead than thread processes.

Using the add_sub TLM example (Figure 16):

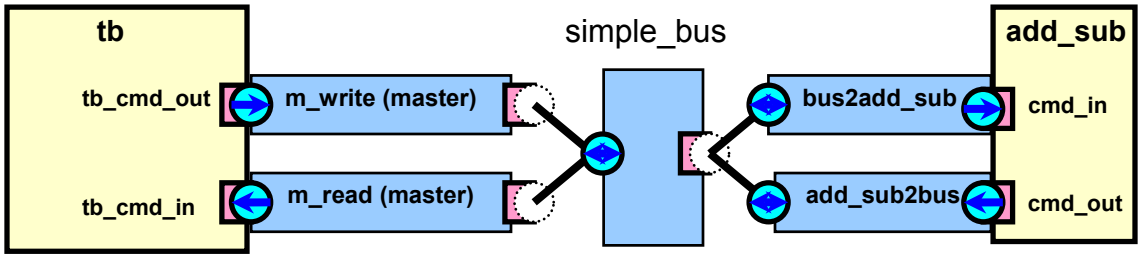

Figure 16 - add_sub With simple_bus

We could refine the `add_sub` module to the RT level and provide a new adapter between the `simple_bus` and the RTL `add_sub`. `tb` would remain identical with this resulting diagram(Figure 17):

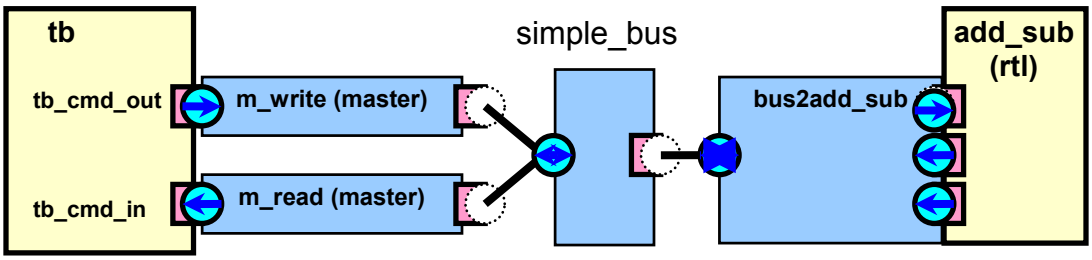

Figure 17 - New Adapter

The code for this example may be found in the Code Appendix

TLM to RTL refinement

In refining from a TLM to an RTL model there are several areas to consider.

- Timed and untimed algorithmic descriptions need to be replaced with pin cycle accurate or register transfer accurate descriptions.
- Abstract channels such as `sc_fifo` need to be replaced with hardware channels such as `sc_signal`.
- C++ data types may need to be replaced with SystemC data types. For example if a 32 bit tri-state bus is modeled, a sc_lv type would need to be used instead of an unsigned int.
- User defined types, such as the `cmd` type used in the `add_sub` example would need to be replaced with c++ or SystemC types.

Note the following refinements to the TLM model to create the RTL model in the add_sub example.

The abstract ports of type `sc_fifo_in` and `sc_fifo_out` have been replaced with hardware ports of type `sc_in` and `sc_out`.

The implicit handshaking in the `sc_fifo` channels has been replaced with an explicit request (`bus_valid`) / acknowledge (`ack`) handshake.

The user defined type `cmd` which packaged up the information for the add_submetic operation and was received and sent through the `sc_fifo_in` and `sc_fifo_out` ports has been replaced with `data_in_bus` and `data_out_bus ports` of type `unsigned int`.

Additional ports have been added for address (`addr_bus`) and for control (`rw_`) information.

Additional processes have been added. `ns_logic` and `update_state` implement an explicit finite state machine which does the handshaking of data in and out of the module. `regs` implements the writing of registers.

The `sc_fifo` buffer `results_buff` has been replaced with a memory (`mem`) for storing incoming operations and the results. There is now no buffering but rather one operation at a time can be accomplished.

Functional Verification Using SystemC

In SystemC functional verification is done through simulation, applying stimulus to the Device Under Test (DUT) and verifying the response against an expected result.

General Structure

In general a testbench consists of stimulus generation & application to the device under test and response verification as shown in the figure Figure 18.

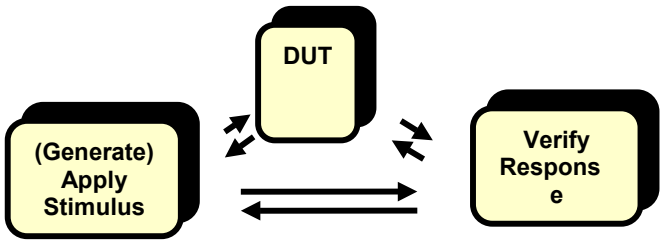

Figure 18 - General Testbench Structure

Figure 19 shows a typical SystemC system verification arrangement. The testbench consists of the Transaction Generator and the Response Checker.

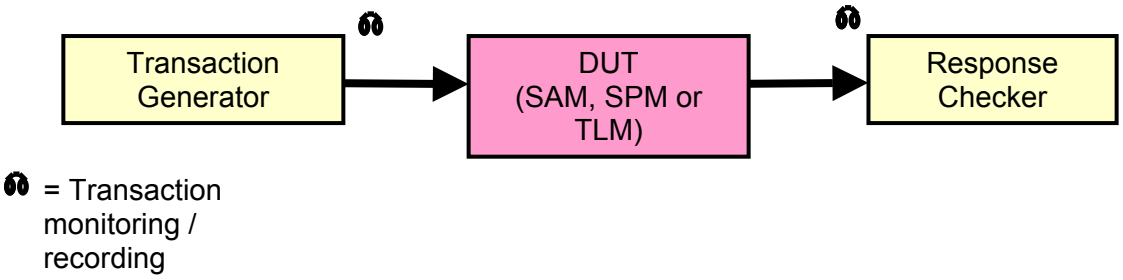

Figure 19 - Typical SystemC Testbench Structure

Stimulus Generation

A testbench generates stimulus for the device under test (DUT) in order to set up scenarios in which the DUT's behavior is monitored. The scenarios should reflect both normal and abnormal external conditions.

Testbenches should be developed early in the design cycle, so that architectural assumptions are checked, and can be modified before too much effort has been expended in refining them. At this high level of abstraction, the tests can be organized as a group of transactions, where a transaction is defined as a high-level interaction between two parts of the system that often, but not necessarily involves transport of data.

For example, in the add_sub design described earlier (Figure 9), the testbench sends command and data instructions across a channel to the `add_sub` module. These instructions represent a high-level transaction. In a single write of the channel, all the information needed to perform a command is transferred to `add_sub` from `tb`.

It is a good idea to organize the testbench in a modular fashion. For example separate out the stimulus generation code from the results checking code. Also perhaps make it so that transaction-generating code is placed in its own function. This makes it clearer as to when and what kind of transaction is being generated.
The arithmetic testbench (`tb`), contains both the stimulus generation and the response checker.

Testbench Refinement

During the SystemC design process there are two types of refinement. The first is model refinement. This is where the DUT *itself* is refined to a more detailed model.
The second type of refinement is communication refinement. This is where the *channel* connecting the DUT is changed or refined. An example of this type of refinement would be the simple bus insertion in place of the fifo channels as described earlier.

When either of these refinements occurs, the testbench interface may not properly connect to the DUT any longer. In the example of the testbench for the add_sub unit the simple bus was inserted in place of the fifo channels breaking the simulation as shown in Figure 20.

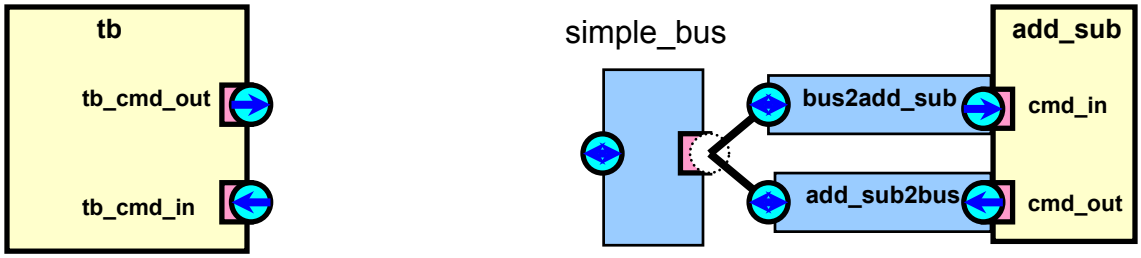

Figure 20 - Testbench with out Adapters

The choice in these situations is to either refine the testbench itself or to insert adapters. In general the better practice is to insert adapters as shown in Figure 21. This provides for better modularity and the testbench itself does not need to be refined. In the add_sub problem two adapters are inserted, the `m_write` and `m_read` adapters.

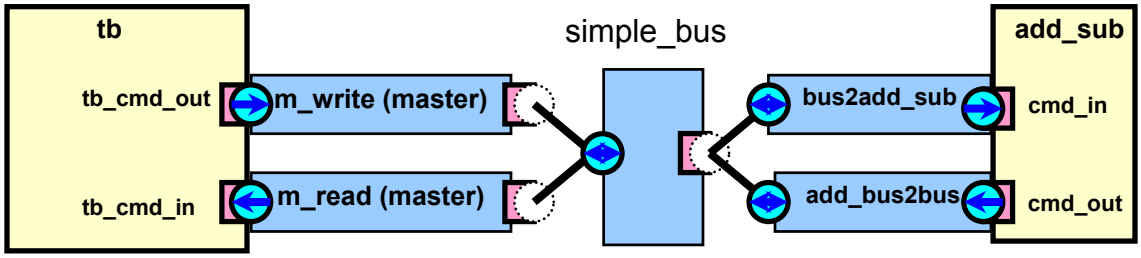

Figure 21 - Testbench with Adapters

When the DUT model itself is refined, the golden model may be used for verification as shown Figure 22. SystemC lends itself well to this approach through use of adapters with the refined DUT.

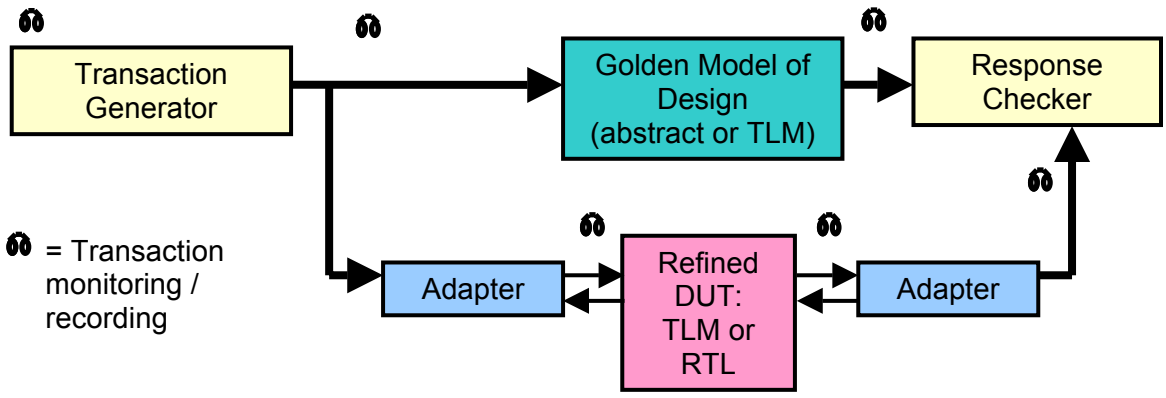

Figure 22 - Verification with Golden Model

SystemC Verification Library

In simple cases, the testbench can perform an exhaustive test by generating all possible input combinations. However, in most cases, exhaustive stimulus generation is either impossible, or highly prohibitive in terms of simulation time.

Beyond the inherent C++ features SystemC has a verification library with multiple useful features for generating stimulus and verifying results:

- Data Introspection, callbacks
- Randomization of data values with advanced seed control
- Constraints and weights for sophisticated, directed randomization
- Useful utility datatypes (scv_bag, scv_sparse_array)

Data Introspection

Data introspection allows arbitrary data types to provide information about themselves and to manipulate their contents. The verification library provides a "wrapper" class implementation for normal C++ and SystemC types. The wrapper type acts like a pointer, but has extra introspection methods.

Examples of things you can do with the introspection methods are:

- What type are you?
- What is your value?
- Set the value directly.
- Set the value through randomization.
- Register callbacks.
    - When data changes.
    - When accessed.
    - When written.

Randomization & Constraints

The verification library provides multiple stimulus generation techniques:

- Directed Tests
    - Traditional method for testbenches
- Weighted Randomization
    - Focuses stimulus on interesting cases
- Constrained Randomization
    - Enables complex and thorough tests to be developed quickly
- Combinations of the above

There are methods provided to randomize, control and constrain the randomization and set the random seed of a variable. Example functionality is:

- Generate a uniformly distributed random value.
- Keep out - do not generate values between a low and high value.
- Keep only – generate values only between a low and high value.

Summary

SystemC provides a natural way to design at many levels of abstraction. It works well for functional modeling, as well as transaction modeling, thus allowing designers to move between modeling methods while using the same language, as well as gaining the performance

advantages of higher-level design descriptions. In addition, RT-level modeling can be done in SystemC as well, or even co-simulated with traditional HDL simulation. SystemC offers the most flexible and powerful modeling evironment available. It does, however, require learning not only C/C++, but also all the layers on top of that, namely the SystemC objects and templates, and additionally the standard verification library (SCV).

Glossary of Terms

Modeling Terms

UnTimed Functional (UTF)
UTF refers to both the model interface and the model functionality. Time is not used for execution. Processes execute in zero time and the transport of data takes zero time. Time is used only perhaps as an ordering mechanism. Data types used are typically native C++.

Timed Functional (TF)
TF refers to both the model interface and the model functionality. Time is used for execution. Processes are assigned and execution time and the transport of data take some finite time. Latencies are modeled. Data types used are typically native C++.

UTF and TF may be mixed in the same model

Bus Cycle Accurate (BCA)
BCA refers to the model interface and not the model functionality. Timing is cycle accurate, usually tied to a system clock. BCA does not infer pin level detail. Typically transfer of information is modeled as transactions. Data types may include some SystemC types.

Pin Cycle Accurate (PCA)
PCA refers to the model interface and not the model functionality. Timing is cycle accurate and tied to a system clock. Accuracy of the interface description is to the pin level. Data types may include some SystemC types.

Register Transfer (RT) accurate
PCA refers to the model functionality. Everything is fully timed. Clocks are used for synchronization. There is a complete detailed functional description. Every register, every bus, every bit is described for every clock cycle. Data types may include some SystemC types.

RT accurate is synonymous with Register Transfer Level (RTL) models.

Model Types

System Architectural Model
These models are an executable specification of the system. They typically describe both hardware and software components.
The *model interface* is UTF with no pin level detail. It is typically modeling communication protocols.
The *model functionality* is UTF. Behavior is modeled algorithmically and is typically described sequentially. Concurrent behavior is possible to model but difficult because things are untimed.
These models are useful for architecture exploration and for algorithm determination and proof

System Performance Model
These models are a timed executable specification of the system. They typically describe both hardware and software components.
The *model interface* is UTF or TF with no pin level detail. It is typically modeling communication protocols. Communication behavior may be timed and may or may not be cycle accurate.
The *model functionality* is UTF or TF. Behavior is modeled algorithmically. Processes are assigned time. Concurrent behavior modeled. Timing is not cycle accurate.
These models are useful for high-level performance modeling and time budgeting.

Transaction Level Model (TLM)
These models are used for modeling an executable platform and typically describe hardware only.
The *model interface* is TF with no pin level detail and may or may not be cycle accurate. Data transfers are modeled as transactions.
The *model functionality* is TF and is not cycle accurate. Behavior is described in terms of transactions.

Functional Model
Model type that refers to modeling done at levels of abstraction above TLM. Encompasses both System Architecture and System Performance models.

System Level Model
Model type that refers to modeling done at levels of abstraction above RTL. Encompasses System Architecture, System Performance, and TLM models.

Behavioral Synthesis Model
Model which follows behavioral synthesis guidelines. It is useful for architectural analysis and implementation.
The *model interface* is cycle accurate with pin level detail.
The *model functionality* is TF and not cycle accurate. Behavior is modeled algorithmically.

Bus Functional Model
Model typically used for simulation purposes. Usually used to model processors and so forth. It is not intended for synthesis.
The *model interface* is pin cycle accurate (PCA).
The *model functionality* is typically described as transactions.

Gate Level Model
It doesn't work well in SystemC, but you can do it if you want to.

Code Appendix

Tb

const int MAX_OPS = 128;

```
SC_MODULE(tb) {
 sc_port<sc_fifo_out_if<cmd> > tb_cmd_out;
 sc_port<sc_fifo_in_if<cmd> > tb_cmd_in;
 void send_cmd(cmd_t, int, int);
 void gen_expected_results(cmd&);
 void test_main();
 void receive_cmds();
 cmd gold_rom[MAX_OPS];
 int cmd_id;
 int cmd_error;
 int num_cmds;
 sc_event cmd_received;
 SC_CTOR(tb)
   {
    num_cmds = 0;
    cmd_id = 0;  // init id
    cmd_error = 0;
    SC_THREAD(test_main);
    SC_THREAD(receive_cmds);
  }
};

void tb::test_main() {
  int i;
  wait(1,SC_NS);
//wait to get of time zero to start
  for(i = 1; i < 5; i++){
    send_cmd(ADD, i, i+1);
    send_cmd(SUB, 2*i, i);
  }
  while(num_cmds) { //any cmds outstanding?
//wait for the next cmd
    wait(cmd_received);
  }
  wait(100, SC_NS);
  cout << endl << sc_time_stamp() << "  " << name()
<< ":  All commands complete" << endl
<< "There were " << cmd_error << " errors"
<< endl << endl;

  sc_stop();  // end simulation
}

void tb::send_cmd(cmd_t command, int op_1, int op_2) {
  cmd tmp_cmd;
  tmp_cmd.command = command;//command field
  tmp_cmd.op_1 = op_1; // op_1
```

```
  tmp_cmd.op_2 = op_2; // op_2
  tmp_cmd.cmd_id = cmd_id; // cmd_id
  tb_cmd_out->write(tmp_cmd); // send cmd
  cout << sc_time_stamp() << "  " << name();
  cout << ": command sent: "<< tmp_cmd << endl;
//inc number of outstanding commands
  num_cmds++;
  gen_expected_results(tmp_cmd);  // set up expected results
}

void tb::gen_expected_results(cmd & tmp_cmd)
{
  sc_int<64> result;
// check to make sure in array bounds
  if(cmd_id < MAX_OPS)
  { // create expected results
    switch (tmp_cmd.command) {
      case ADD:
        result = tmp_cmd.op_1 + tmp_cmd.op_2;
      break;
      case SUB:
        result = tmp_cmd.op_1 - tmp_cmd.op_2;
      break;
    }
    tmp_cmd.op_1 = result.range(63,32);
    tmp_cmd.op_2 = result.range(31,0);
// save expect results
    gold_rom[cmd_id++] = tmp_cmd;
  }
}

void tb::receive_cmds(){
  cmd tmp_cmd;
// get off of time zero
  wait(1, SC_NS);
  cmd_error = false;// set error flag
  while(true){
    tmp_cmd = tb_cmd_in->read(); //wait for a cmd return
    if (tmp_cmd == gold_rom[tmp_cmd.cmd_id]) {
      cout << sc_time_stamp() << "  " << name();
      cout << ": received correct result for: " << tmp_cmd << endl;
    }
    else  {
      cmd_error++;
      cout << sc_time_stamp() << "  " << name();
      cout << ": ERROR command =  "<< tmp_cmd
           << ", should be:  "
           << gold_rom[tmp_cmd.cmd_id];
    }
 // dec num cmds outstanding
   num_cmds --;
// notify received a cmd
   cmd_received.notify();
  }
}
```

Functional Model add_sub

```
SC_MODULE(add_sub) {
  sc_port<sc_fifo_in_if<cmd> > cmd_in;
  sc_port<sc_fifo_out_if<cmd> > cmd_out;
// local channels
  sc_fifo<cmd> result_buff;
// Other methods
  void write_result (cmd&, sc_int<64>&);
// Processes
  void return_proc();
  void add_sub_proc();

  SC_CTOR(add_sub) {
  // Register process
    SC_THREAD(return_proc);
    SC_THREAD(add_sub_proc);
  }
};
// Thread process that handles
// incoming commands
void add_sub::add_sub_proc() {
  cmd tmp_cmd;
  sc_int<64> result;
  // wait to get off time zero
  wait(1, SC_NS);
  while (true) {
    // Get the next command
    tmp_cmd = cmd_in->read();
    switch (tmp_cmd.command) {
      case ADD:
        result = tmp_cmd.op_1 +
      tmp_cmd.op_2;
        wait(20, SC_NS); //time to do op
      write_result(tmp_cmd, result);
      break;
      case SUB:
        result = tmp_cmd.op_1 -
      tmp_cmd.op_2;
        wait(20, SC_NS); //time to do op
      write_result(tmp_cmd, result);
      break;
      default:
        cout << endl << sc_time_stamp() << "  "
             << name() << ":  Received an illegal command, value = "
             <<tmp_cmd.command << endl << endl;
        break;
    }
  }
}
// Thread process for returning
// the data
void add_sub::return_proc() {
while (true){
  // get a result and send it back cmd_out->write(result_buff.read());
  }
}
```

```
// function for putting result in
// cmd and in buffer
void add_sub::write_result(cmd& tmp, sc_int<64>& result) {
  tmp.op_1 = result.range(63,32);  //upper 32 bits
  tmp.op_2 = result.range(31,0);
// lower 32 bits
  result_buff.write(tmp);
// write into buffer
}
```

TLM Model add_sub

```
SC_MODULE(add_sub) {
  // Interface to testbench
sc_port<sc_fifo_in_if<cmd> > cmd_in;
sc_port<sc_fifo_out_if<cmd> > cmd_out;
  // channels
sc_fifo<cmd> result_buff;
  // Other methods
void write_result (cmd&, sc_int<64>&);
  // Processes
void return_proc();
void add_sub_proc();
  // Constructor
SC_CTOR(add_sub) {
  // Register process
    SC_THREAD(return_proc);
    SC_THREAD(add_sub_proc);
  }
};
// Thread process that handles incoming commands
void add_sub::add_sub_proc() {
  cmd tmp_cmd;
  sc_int<64> result;
  wait(1, SC_NS); // wait to get off time zero
  while (true) {
    // Get the next command
    tmp_cmd = cmd_in->read();
    switch (tmp_cmd.command) {
      case ADD:
        result = tmp_cmd.op_1 + tmp_cmd.op_2;
        wait(20, SC_NS); // time to do op
        write_result(tmp_cmd, result);
      break;
      case SUB:
        result = tmp_cmd.op_1 - tmp_cmd.op_2;
        wait(20, SC_NS); // time to do op
        write_result(tmp_cmd, result);
      break;
      default:
      cout << endl << sc_time_stamp()<< "  " << name()
      << ":  Received an illegal command, value = "
      <<tmp_cmd.command << endl << endl;
        break;
    }
  }
}
```

```
// Thread process for returning the data
void add_sub::return_proc() {
  while (true){
    // get a result and send it back
    cmd_out->write( result_buff.read() );
  }
}
// function for putting result in cmd and in buffer
void add_sub::write_result(cmd& tmp, sc_int<64>& result) {
  tmp.op_1 = result.range(63,32);
//upper 32 bits
  tmp.op_2 = result.range(31,0);
// lower 32 bits
  result_buff.write(tmp);
// write into buffer
}
```

RTL Model add_sub

```
enum state_t
{
 s0, s1, s2, s3
};
SC_MODULE(add_sub) {

  sc_in<unsigned int> data_in_bus;
  sc_in<bool> rw_;
//1 = read, 0 = write
  sc_in<sc_uint<32> > addr_bus;
  sc_in<bool> bus_valid;
  sc_out<bool> ack;
  sc_out<unsigned int> data_out_bus;
  sc_in<bool> clk;

  sc_uint<32> tmp;
  sc_uint<2> count;
// sm
  sc_signal<state_t> state, n_state;
  sc_signal<bool> valid_operands;

// channels
  sc_uint<32> mem[1024];
  sc_signal<bool> load_cmd, load_op_1;
  sc_signal<bool> load_op_2, load_result;
  sc_signal<bool> set_ack;

// Other methods
  void write_result (sc_int<64>&);

// Processes
  void ns_logic(); // comb logic
  void update_state(); // seq logic
  void return_proc();
  void add_sub_proc();
```

```
  void regs();

// Constructor
  SC_CTOR(add_sub) {
    count = 0;
// init normally with reset
    valid_operands.write(false);
    state.write(s0); // init state
   // Register process
    SC_THREAD(add_sub_proc);
    SC_METHOD(ns_logic);
      sensitive << state.default_event() << bus_valid;
    SC_METHOD(update_state);
      sensitive << clk.neg();
    SC_METHOD(regs);
      sensitive << clk.pos();
  }
};

void add_sub::update_state(){
  state.write(n_state);  // update state
}

void add_sub::ns_logic(){
  valid_operands.write(false);
//default value
  set_ack.write(false); // default value
  load_cmd.write(false);
  load_result.write(false);

  switch(state) {
   case s0:
     if(bus_valid->read() == true) {
       if(rw_ == true) { // is it a read?
         load_result.write(true);
// set for return
       set_ack.write(true);//set sent data
       } else  {// no its a write
         load_cmd.write(true);
         set_ack.write(true);//set got data
       }
     n_state.write(s1);//set the next state
     }
     else
       n_state.write(s0); // stay here
     break;
   case s1:
     if(bus_valid->read() == false) {
       n_state.write(s0); // go back to s0
       ack->write(false);  // clear ack
     }
     else {
       n_state.write(s1); // stay here
     }
     break;
   default:  n_state.write(s0); break;
   }
```

```
}
// Thread process that does computation
void add_sub::add_sub_proc() {
  sc_int<64> result;
  while (true) {
    wait(valid_operands.posedge_event() );
    switch (mem[0]) {
      case ADD:
        result = mem[1] + mem[2];
        wait(20, SC_NS); // time to do op
      write_result(result);
      break;
      case SUB:
        result = mem[1] - mem[2];
        wait(20, SC_NS); // time to do op
      write_result(result);
      break;
      default:
        cout << endl << sc_time_stamp()<< "  " << name()
             << ":  Received an illegal command, value = "
             << mem[0] << endl << endl;
        break;

    }
  }
}

// function for putting result in cmd
// and in buffer
void add_sub::write_result(sc_int<64>& result) {
  mem[6] = result.range(63,32);  //upper 32 bits
  mem[7] = result.range(31,0);  // lower 32 bits
}

void add_sub::regs(){
  if(load_cmd){
    tmp = addr_bus->read();
// get the address
    mem[tmp.range(4,2)] = data_in_bus->read();  // get the data
    if(count == 2) { // last one
      count = 0; //clear count;
      valid_operands.write(true);
    }
    else
      count++;  // inc count
//    cout << sc_time_stamp() << hex << " cmd = "<< mem[tmp.range(4,2)]
<< endl;
  }
  else
    valid_operands.write(false);

  if(load_result){
    tmp = addr_bus->read();
// get the address
    data_out_bus->write(mem[tmp.range(4,2)]);
// write the data
  }
```

```
  if(set_ack)
    ack->write(true);
  else
    ack->write(false);
}
```