http://www.doulos.com/knowhow/systemc/tutorial/

SystemC Tutorial

This tutorial is taken from material in the introductory chapters of the Doulos SystemC Golden Reference Guide.

The first part, below, covers a brief introduction to SystemC, and then an example of a simple design. The subsequent parts cover Debugging, and Hierarchical Channels. The final part covers Primitive Channels and the Kernel.

A Brief Introduction to SystemC

The SystemC Class Library has been developed to support system level design. It runs on both PC and UNIX platforms, and is freely downloadable from the web.

The class library is being released in stages. The first stage, release 1.0 (presently at version 1.0.2) provides all the necessary modelling facilities to describe systems similar to those which can be described using a hardware description language, such as VHDL. Version 1.0 provides a simulation kernel, data types appropriate for fixed point arithmetic, communication channels which behave like pieces of wire (signals), and modules to break down a design into smaller parts.

In Release 2.0 (presently at version 2.0.1), the class library has been extensively re-written to provide an upgrade path into true system level design. Features that were “built-in” to version 1.0, such as signals, are now built upon an underlying structure of channels, interfaces, and ports. Events have been provided as a primitive means of triggering behaviour, together with a set of primitive channels such as FIFO and mutex. Version 2.0 allows much more powerful modelling to be achieved by modelling at the level of transactions.

Version 2.1 added a number of features including the ability to spawn processes after simulation has started, and extra callbacks into the operation of the simulation kernel.

In 2005 the language was standardised as IEEE 1666-2005. Version 2.2 of the reference implementation of the class library is currently available and has been updated to comply with the IEEE standard

In future, Version 3.0 of the class library will be extended to cover modelling of operating systems, to support the development of models of embedded software.

It is also possible to provide additional libraries to support a particular design methodology. Examples of this include the SystemC Verification Library (SCV).

The SystemC Class Library has been developed by a group of companies forming the Open SystemC Initiative (OSCI). For more information, and to download the freely available source code, visitOSCI.

Modules and Processes

This section contains a complete simple design to demonstrate the use of modules and processes in SystemC. For simplicity, it is very low level - not the style of coding you would normally expect in a system level design language!

The points demonstrated are

- Creating hierarchy
- The sc_signal primitive channel
- (Specialized) ports
- Processes (SC_METHOD, SC_THREAD, SC_CTHREAD)
- A simple test bench

SystemC Background

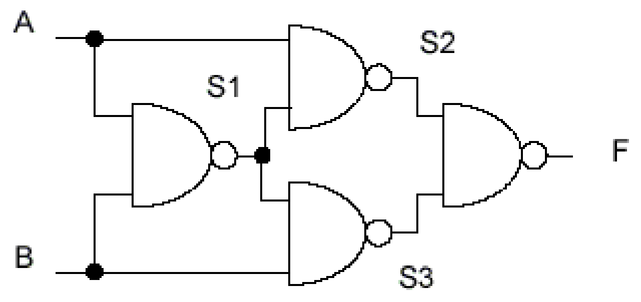
Why look at Modules and Processes? The reason is that SystemC is intended to cope with both hardware and software, and to allow large systems to be modelled.

Processes are small pieces of code that run concurrently with other processes. Virtually all the high-level system level design (SLD) tools that have been developed use an underlying model of a network of processes. SystemC provides processes to support the construction of networks of independent (concurrent/parallel) pieces of code.

SLD needs to deal with large designs. To cope with this, it is common to use hierarchy. Hierarchy is implemented in SystemC by using the module, a class that can be linked to other modules using ports. Modules allow a piece of design to be worked on separately. Modules may contain processes, and instances of other modules.

The Example Design

The design consists of an EXOR gate implemented with four NAND gates. Again, it is important to note that this is not a typical design style - but it is nice and simple to understand. The design looks like this



The first step is to model the NAND gate. A NAND gate is a combinational circuit; its output is purely a function of the values at the input. It has no memory, and requires no clock. Because of this, the model can use the simplest kind of SystemC process, an SC_METHOD.

SC_METHODs are simply C++ functions. Because of that, the SystemC class library has to make them behave like processes. In particular

- The SystemC class library contains a simulation kernel - a piece of code that models the passing of time, and calls functions to calculate their outputs whenever their inputs change.
- The function must be declared as an SC_METHOD and made sensitive to its inputs.

Here is the code for the NAND gate, in one file, nand.h

```
#include "systemc.h"
SC_MODULE(nand2)          // declare nand2 sc_module
{
  sc_in<bool> A, B;       // input signal ports
  sc_out<bool> F;         // output signal ports

  void do_nand2()         // a C++ function
  {
    F.write( !(A.read() && B.read()) );
  }

  SC_CTOR(nand2)          // constructor for nand2
  {
    SC_METHOD(do_nand2);  // register do_nand2 with kernel
    sensitive << A << B;  // sensitivity list
  }
};
```

Hierarchy in SystemC is created using a class `sc_module`. `sc_module` may be used directly, or may be "hidden" using the macro `SC_MODULE`. The example `SC_MODULE` above creates an `sc_module` class object called `nand2`.

Next are declared input and output ports. In general, a port is declared using the class `sc_port`. For instance, input ports using `sc_signal` would be declared

```
sc_port<sc_signal_in_if<bool>,1> A,B;
```

but as you can see, this is a lot of typing. For convenience, it is also possible to create and use specialized ports. `sc_in` is an example of a specialized port for the `sc_signal` class.

The ports may be of any C++ or SystemC type - the example uses `bool`, a built-in C++ type.

Next, the function that does the work is declared. The input and output (specialized) ports include methods `read()` and write() to allow reading and writing the ports. `A` and `B` are read, the NAND function is calculated, and the result is written to `F` using the write() method.

Note that you can often get away without using the `read()` and write() methods, as the `=` operator and the type conversion operators have been overloaded. So you could write

```
F = !(A && B);
```

but it is a good habit to use `read()` and `write()` as it helps the C++ compiler disambiguate expressions.

After the function `do_nand2()` is written, there is a constructor for the `sc_module` instance `nand2`. SystemC provides a shorthand way of doing this, using a macro `SC_CTOR`. The constructor does the following

- Create hierarchy (none in this case)
- Register functions as processes with the simulation kernel
- Declare sensitivity lists for processes

It is also possible to initialize anything that required initialization here - for instance, a class data member could be initialized.

In the example above, the constructor declares that `do_nand2` is an `SC_METHOD`, and says that any event on ports `A` and `B` must make the kernel run the function (and thus calculate a new value for `F`).

Hierarchy

The EXOR gate is built up from four copies (or instances) of the NAND gate. This is achieved by using the EXOR gate constructor to connect the NAND gate instances. Here is the code for the EXOR gate

```
#include "systemc.h"
#include "nand2.h"
SC_MODULE(exor2)
{
  sc_in<bool> A, B;
  sc_out<bool> F;

  nand2 n1, n2, n3, n4;

  sc_signal<bool> S1, S2, S3;

  SC_CTOR(exor2) : n1("N1"), n2("N2"), n3("N3"), n4("N4")
  {
    n1.A(A);
    n1.B(B);
    n1.F(S1);

    n2.A(A);
    n2.B(S1);
    n2.F(S2);

    n3.A(S1);
    n3.B(B);
    n3.F(S3);

    n4.A(S2);
    n4.B(S3);
    n4.F(F);
  }
};
```

The start looks very similar to the NAND gate, but note that it includes the file `nand2.h`. This allows access to the module containing the NAND gate.

The module exor2 is created, and ports are declared. Note that it is allowed to re-use the names `A`, `B`and `F`, as this is a different level of the hierarchy.

The original diagram shows some "pieces of wire" to connect the NAND gates. These are created by declaring`sc_signal`s `S1`,`S2` and `S3`.`sc_signal` is a class with a template parameter specifying the type of data the signal can hold - `bool` in this example. `sc_signal` is an example of a primitive channel, a built-in channel within the SystemC class library. It behaves like a signal in VHDL.

The constructor for the EXOR gate is more complex than that for the NAND gate, as it must have four instances of `nand2`. After the port declarations, four instances of `nand2` are declared: `n1`, `n2`, `n3` and `n4`. A label must be given to each instance. The four labels `"N1"`, `"N2"`, `"N3"` and `"N4"` are passed to the constructors of the instances of `nand2` by using an initializer list on the constructor of exor2.

Finally, the ports are wired up. This is done in the constructor as shown.

Test bench

To test the design, there is a stimulus generator. This is another module, very similar to the above. The only significant point is that it uses a thread (`SC_THREAD`), a kind of process that can be suspended. Here is the code for `stim.h`

```
#include "systemc.h"
SC_MODULE(stim)
{
  sc_out<bool> A, B;
  sc_in<bool> Clk;

  void StimGen()
  {
    A.write(false);
    B.write(false);
    wait();
    A.write(false);
    B.write(true);
    wait();
    A.write(true);
    B.write(false);
    wait();
    A.write(true);
    B.write(true);
    wait();
    sc_stop();
  }
  SC_CTOR(stim)
  {
    SC_THREAD(StimGen);
    sensitive << Clk.pos();
  }
};
```

Note the final call to `sc_stop()` which makes the simulation stop. The monitor code looks very similar, and is omitted - it is in a file `mon.h`.

Here is the top level - it is inside a file main.cpp that includes all the submodules described above

```
#include "systemc.h"
#include "stim.h"
#include "exor2.h"
#include "mon.h"

int sc_main(int argc, char* argv[])
{
  sc_signal<bool> ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS,0.5);

  stim Stim1("Stimulus");
  Stim1.A(ASig);
  Stim1.B(BSig);
  Stim1.Clk(TestClk);

  exor2 DUT("exor2");
  DUT.A(ASig);
  DUT.B(BSig);
  DUT.F(FSig);

  mon Monitor1("Monitor");
  Monitor1.A(ASig);
  Monitor1.B(BSig);
  Monitor1.F(FSig);
  Monitor1.Clk(TestClk);

  sc_start();  // run forever

  return 0;

}
```

The header files for the modules are included, top-level signals declared to do the wiring, and a clock created using `sc_clock`; then each module is instanced and connected.

After that, calling `sc_start()` starts the simulation and it runs forever (or rather until it encounters the call to `sc_stop()` in the stimulus module).

Here is the output from this example

```
      Time A B F
       0 s 0 0 1
     10 ns 0 0 0
     20 ns 0 1 1
     30 ns 1 0 1
```

```
      40 ns 1 1 0
```

If you look at it, you will notice something very odd - the first line at time 0 says that `F` is 1 (true), while `A` and `B` are 0 - not a very convincing EXOR gate! By 10 ns, everything is as it should be. What is going on at time 0?

Simulation

The SystemC library contains a simulation kernel. This decides which processes (software threads) to run. At time 0, all `SC_METHOD`s and `SC_THREAD`s will run in an undefined order, until they suspend. Then `SC_CTHREAD`s will run when a clock edge occurs.

The problem above is due to a combination of circumstances

- The `sc_clock` statement results in a rising edge at time 0, so both the monitor and stimulus processes will run (in an undefined order, it is not known which will run first)
- Variables in C++ do not always have a defined initial value (unless they are declared static). So the data value held by `F` happens to be starting at 1 (true)
- The `do_nand2` `SC_METHOD` runs at time 0, and schedules `F` to update, but `F` is a signal, which cannot update instantaneously, so the value 1 is still present when the monitor process runs.

To prove this is the case, it is possible to modify the `sc_clock` statement to delay the first edge of the clock, as follows

```
  sc_clock TestClk("TestClock", 10, SC_NS,0.5, 1, SC_NS);
```

The final `1, SC_NS` arguments specify a 1 ns delay before the first clock edge occurs. Now time has passed, so `F` will be updated. Here is the corresponding output

```
       Time A B F
       1 ns 0 0 0
      11 ns 0 0 0
      21 ns 0 1 1
      31 ns 1 0 1
      41 ns 1 1 0
```

Now you can see that `F` is always correct.

Conclusions

That concludes a quick tour of modules and processes. You have seen the importance of understanding the concurrent nature of the SystemC simulation kernel, together with the behaviour of the `sc_signal` primitive channel.

You have also seen some basic examples of instancing lower level modules within a top-level module, and how `sc_main` is used.

Debugging

This tutorial shows you some basic debugging techniques, including the use of waveform tracing. First, it shows some basic text based methods, then extends the example from the Modules and Processes tutorial to use waveform tracing.

Text-Based Debugging

Even without waveforms, some simple techniques can be used to find out what is going on inside your design. These include the use of the `name()` method and careful placement of text output statements.

SystemC provides overloaded stream insertion operators for the built-in data types, so you can just use statements such as

```
cout << mydata << endl;
```

and it will work, even for SystemC data types.

Here is the code of the mon.h from the previous chapter

```
#include "systemc.h"
#include
SC_MODULE(mon)
{
    sc_in<bool> A,B,F;
    sc_in<bool> Clk;

  void monitor()
  {
    cout << setw(10) << "Time";
    cout << setw(2) << "A" ;
    cout << setw(2) << "B";
    cout << setw(2) << "F" << endl;
    while (true)
    {
      cout << setw(10) << sc_time_stamp();
      cout << setw(2) << A.read();
      cout << setw(2) << B.read();
      cout << setw(2) << F.read() << endl;
      wait();    // wait for 1 clock cycle
    }
  }

  SC_CTOR(mon)
  {
    SC_THREAD(monitor);
    sensitive << Clk.pos();
  }
};
```

- Note that `cout` works fine here, as the `read()` method returns values of type `bool`, which is a built in C++ data type; but it would have worked fine for SystemC data types such as `sc_logic`, `sc_int`, and so on.
- Secondly, note the use of `sc_time_stamp()` to print out the current simulation time.

Another use for `cout` is to find out how your design is built-up before simulation starts. Will the NAND gates of the previous chapter be constructed before the EXOR gate? They should be, but a simple way to prove it is to add a line inside each constructor printing out a message. E.g.

```
SC_CTOR(nand2)
{
  cout << "Constructing nand2" << endl;
```

Here is the corresponding output after adding statements like that

```
Constructing stim
Constructing nand2
Constructing nand2
Constructing nand2
Constructing nand2
Constructing exor2
Constructing mon
```

The disadvantage of this is that it does not show which instance is referred to. This can be overcome by using the `name()` method. The `name()` method is defined for most things in SystemC. We can simply say

```
cout << "Constructing nand2 " << name() << endl;
```

instead, and now the output is

```
Constructing stim
Constructing nand2 exor2.N1
Constructing nand2 exor2.N2
Constructing nand2 exor2.N3
Constructing nand2 exor2.N4
Constructing exor2
Constructing mon
```

Note that `name()` returns what the user specifies in the labels of the instances. Other methods you might want to look at are `dump()` and `print()` (have a look at `sc_object` in the Doulos SystemC Golden Reference Guide to see examples).

Waveform Tracing

The SystemC library supports waveform tracing. This requires adding statements to the top level of your system (inside `sc_main`). Here is the top level from the Modules and Processes tutorial with waveform tracing (to a vcd, Value Change Dump, file).

```
#include "systemc.h"
#include "stim.h"
#include "exor2.h"
#include "mon.h"
```

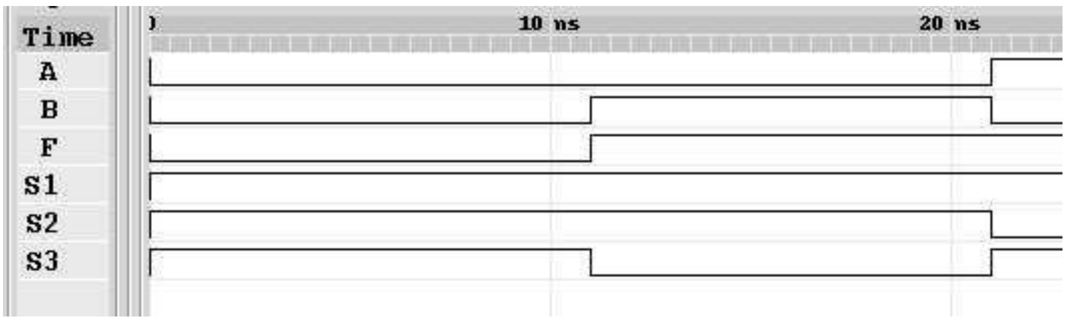
```
int sc_main(int argc, char* argv[])
{
  sc_signal<bool> ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS,0.5, 1, SC_NS);

  ... instance of stim

  exor2 DUT("exor2");
  DUT.A(ASig);
  DUT.B(BSig);
  DUT.F(FSig);

  ... instance of mon

  sc_trace_file* Tf;
  Tf = sc_create_vcd_trace_file("traces");
  ((vcd_trace_file*)Tf)->sc_set_vcd_time_unit(-9);
  sc_trace(Tf, ASig  , "A" );
  sc_trace(Tf, BSig  , "B" );
  sc_trace(Tf, FSig  , "F" );
  sc_trace(Tf, DUT.S1, "S1");
  sc_trace(Tf, DUT.S2, "S2");
  sc_trace(Tf, DUT.S3, "S3");

  sc_start();  // run forever
  sc_close_vcd_trace_file(Tf);
  return 0;
}
```

Note the sequence of operations

- Declare the trace file
- Create the trace file
- (Optionally specify the trace time unit)
- Register signals or variables for tracing
- Run the simulation
- Close the trace file

Note also that it is possible to trace hierarchically (e.g. `DUT.S1`).

Here are the resulting waveforms



Conclusion

Some debugging techniques have been demonstrated, based on outputting text and on waveform tracing.

For more difficult problems you might need to use a C++ debugger (for instance gdb/ddd on UNIX), or a dedicated SystemC debugger.

Hierarchical Channels

Hierarchical channels form the basis of the system-level odelling capabilities of SystemC. They are based on the idea that a channel may contain quite complex behaviour - for instance, it could be a complete on-chip bus.

Primitive channels on the other hand cannot contain internal structure, and so are normally simpler (for instance, you can think of `sc_signal` as behaving a bit like a piece of wire).

To construct complex system level models, SystemC uses the idea of defining a channel as a thing that implements an interface. An interface is a declaration of the available methods for accessing a given channel. In the case of `sc_signal`, for instance, the interfaces are declared by two classes `sc_signal_in_if` and `sc_signal_out_if`, and these declare the access methods (for instance `read()` and `write()`).

By distinguishing the declaration of an interface from the implementation of its methods, SystemC promotes a coding style in which communication is separated from behaviour, a key feature to promote refinement from one level of abstraction to another.

One additional feature of SystemC is that if you want modules to communicate via channels, you must use ports on the modules to gain access to those channels. A port acts as an agent that forwards method calls up to the channel on behalf of the calling module.

Hierarchical channels are implemented as modules in SystemC: in fact, they are derived from `sc_module`. Primitive channels have their own base class, `sc_prim_channel`.

To summarise

- Separating communication from behaviour eases refinement
- An interface declares a set of methods for accessing a channel
- A hierarchical channel is a module that implements the interface methods
- A port allows a module to call the methods declared in an interface

Writing hierarchical channels is a big topic, so this chapter will just show a basic example, a model of a stack.

Define the interface

The stack will provide a read method and a write method. These methods are non-blocking (they return straight away without any waits). To make this usable, each method returns a `bool` value saying if it succeeded. For instance, if the stack is empty when read, `nb_read()` (the read method) returns false.

These methods are declared in two separate interfaces, a write interface `stack_write_if`, and a read interface, `stack_read_if`. Here is the file containing the declarations, `stack_if.h`.

```
#include "systemc.h"
class stack_write_if: virtual public sc_interface
{
public:
  virtual bool nb_write(char) = 0;  // write a character
  virtual void reset() = 0;         // empty the stack
};

class stack_read_if: virtual public sc_interface
{
public:
  virtual bool nb_read(char&) = 0;  // read a character
};
```

The interfaces are derived from `sc_interface`, the base class for all interfaces. They should be derived using the keyword `virtual` as shown above.

The methods `nb_write`, `nb_read`, and `reset` are declared pure virtual - this means that it is mandatory that they be implemented in any derived class.

Stack Channel

The stack channel overrides the pure virtual methods declared in the stack interface. Here is the code

```
#include "systemc.h"
#include "stack_if.h"

// this class implements the virtual functions
// in the interfaces
class stack
: public sc_module,
  public stack_write_if, public stack_read_if
{
private:
  char data[20];
  int top;                 // pointer to top of stack

public:
  // constructor
  stack(sc_module_name nm) : sc_module(nm), top(0)
  {
  }

  bool stack::nb_write(char c)
  {
    if (top < 20)
    {
      data[top++] = c;
      return true;
    }
    return false;
  }

  void stack::reset()
  {
    top = 0;
  }

  bool stack::nb_read(char& c)
  {
    if (top > 0)
    {
      c = data[--top];
      return true;
    }
    return false;
  }

  void stack::register_port(sc_port_base& port_,
                            const char* if_typename_)
  {
    cout << "binding    " << port_.name() << " to "
         << "interface: " << if_typename_ << endl;
  }
};
```

There is some local (private) data to store the stack, and an integer indicating the current location in the stack array.

The `nb_write` and `nb_read` functions are defined here. Also there is a `reset()` function which simply sets the stack pointer to 0.

Finally, you will notice a function called `register_port` - where does this come from?

It is defined in `sc_interface` itself, and may be overridden in a channel. Typically, it is used to do checking when ports and interfaces are bound together. For instance, the primitive channel `sc_fifo` uses register_port to check that a maximum of 1 interface can be connected to the FIFO read or write ports. This example just prints out some information as binding proceeds.

Creating A Port

To use the stack, it must be instanced. In this example, there are two modules, a producer and a consumer. Here is the producer module

```
#include "systemc.h"
#include "stack_if.h"

class producer : public sc_module
{
public:

  sc_port<stack_write_if> out;
  sc_in<bool> Clock;

  void do_writes()
  {
    int i = 0;
    char* TestString = "Hallo,     This Will Be Reversed";
    while (true)
    {
      wait();             // for clock edge
      if (out->nb_write(TestString[i]))
        cout << "W " << TestString[i] << " at "
             << sc_time_stamp() << endl;
      i = (i+1) % 32;
    }
  }

  SC_CTOR(producer)
  {
    SC_THREAD(do_writes);
      sensitive << Clock.pos();
  }
};
```

The producer module declares a port that interfaces to the stack. This is done with the line

```
sc_port<stack_write_if> out;
```

which declares a port that can be bound to a `stack_write_if`, and has a name `out`.

You can bind more than one copy of an interface to a port, and you can specify the maximum number of interfaces that may be bound. For instance, if we wanted to allow 10 interfaces to be bound, we could declare the port as

```
sc_port<stack_write_if,10> out;
```

If you leave out the number, the default is 1.

To actually write to the stack, call the method `nb_write` via the port:

```
out->nb_write('A')
```

This calls `nb_write('A')` via the `stack_write_if`. You must use the pointer notation `->` when doing this.

The consumer module that reads from the stack looks very similar, except it declares a read port

```
sc_port<stack_read_if> in;
```

and calls nb_read, e.g.

```
in->nb_read(c);
```

where c is of type char.

Note that nb_read and nb_write both return the value true if they succeed.

Perhaps the most interesting thing about this is that the functions nb_write and nb_read execute in the context of the caller. In other words, they execute as part of the SC_THREADs declared in the producer and consumer modules.

Binding Ports and Interfaces

Here is the code for sc_main, the top level of the design

```
#include "systemc.h"
#include "producer.h"
#include "consumer.h"
#include "stack.h"

int sc_main(int argc, char* argv[])
{
  sc_clock ClkFast("ClkFast", 100, SC_NS);
  sc_clock ClkSlow("ClkSlow", 50, SC_NS);

  stack Stack1("S1");

  producer P1("P1");
  P1.out(Stack1);
  P1.Clock(ClkFast);

  consumer C1("C1");
  C1.in(Stack1);
  C1.Clock(ClkSlow);

  sc_start(5000, SC_NS);

  return 0;
}
```

Note how the write interface of the stack is implicitly bound in the line

```
P1.out(Stack1);
```

This example is a bit odd (but perfectly legal!) as the stack itself does not have ports. It is simply causing the first stack_write_if to be bound in this line. If there were more than one stack_write_if, they would be bound in sequence.

This is the result of running the code:

```
binding    C1.port_0 to interface: 13stack_read_if
binding    P1.port_0 to interface: 14stack_write_if
W H at 0 s
R H at 0 s
W a at 100 ns
R a at 150 ns
W l at 200 ns
// and so on for 5000 ns
```

The messages about binding are coming from the `register_port` function in `stack.h`. The names of the ports (`C1.port_0`...) are fabricated by the SystemC kernel, as are the interface names (`13stack_read_if`...).

Conclusion

The example shown above is quite simple, and yet there is a lot to learn and understand. It gives you just a quick tour through the process of defining and writing your own hierarchical channels. The key points to remember are

- Hierarchical channels allow complex bus models
- Communication and behaviour are separated using interfaces
- Interfaces may be accessed from outside a module using ports
- When you call a method in a channel via an interface, you do not have to know how it is implemented - you only need to know the declaration of the interface call
- When methods defined in the channel are executed, they are executed in the context of the caller

Primitive Channels and the Kernel

This section describes a little of the operation of the SystemC simulation kernel, and then relates this to the behaviour of primitive channels.

Simulation Kernels

Most modelling languages, VHDL for example, use a simulation kernel. The purpose of the kernel is to ensure that parallel activities (concurrency) are modelled correctly. In the case of VHDL, a fundamental decision was made that the behaviour of the simulation should not depend on the order in which the processes are executed at each step in simulation time. This section starts by describing what happens in VHDL, because SystemC mimics some key aspects of the VHDL simulation kernel, but also allows other models of computation to be defined. Once VHDL is understood, it is possible to extend the discussion to incorporate the more general simulation kernel of SystemC.

For instance, suppose in SystemC there are two `SC_THREAD`s, both sensitive to a trigger.

```
SC_THREAD(proc_1);

sensitive << Trig.pos();

SC_THREAD(proc_2);

sensitive << Trig.pos();
```

When the trigger changes from low to high, which process will run first? More importantly, does it matter? In the analogous situation using VHDL, you really do not care. This is because in VHDL, communication between processes is done via signals, and process execution and signal update are split into two separate phases.

The VHDL simulation kernel executes each process in turn, but any resulting changes on signals do not happen instantaneously. The same is true of SystemC and `sc_signal`. To be precise, the assignments are scheduled to happen in the future, meaning when all currently active processes have been evaluated and have reached a point where they need to suspend and wait for some event to occur.

It is possible that no simulated time has passed. If that is the case, and there are pending updates for signals, the processes that react to those signals will run again, without time having passed. This is known as a "delta cycle" and has the effect of determining unambiguously the order in which communicating processes are to execute in cases where no simulation time is passing.

SystemC can do this too, but can also model concurrency, communication and time and in other ways.

Non-determinism

When signals in VHDL or `sc_signal`s in SystemC are used to communicate between processes, the simulation is deterministic; it will behave the same on any simulation tool.

In SystemC however, the language allows non-determinism. For instance, suppose a variable declared in a class is accessed from two different `SC_THREAD`s, as described above. Here is an example

```
SC_MODULE(nondet)
{
  sc_in Trig;

  int SharedVariable;
  void proc_1()
  {
    SharedVariable = 1;
    cout << SharedVariable << endl;
  }

  void proc_2()
  {
    SharedVariable = 2;
    cout << SharedVariable << endl;
  }

  SC_CTOR(nondet)
  {
    SC_THREAD(proc_1);
    sensitive << Trig.pos();
    SC_THREAD(proc_2);
    sensitive << Trig.pos();
  }
};
```

In this example, which `SC_THREAD` will run first is undefined - there is no way of telling which will run first.

For hardware modelling, this is unacceptable. But for software modelling, this might represent a system where a shared variable is used, and where non-determinism is not important - all that is necessary is to guarantee that two processes cannot simultaneously access the variable.

Software engineers use concepts such as mutual exclusion (mutex) or semaphores to cope with such situations.

Events and Notifications

After looking at the background, it is now possible to summarise the operation of the SystemC simulation kernel.

The SystemC simulation kernel supports the concept of delta cycles. A delta cycle consists of an evaluation phase and an update phase. This is typically used for modelling primitive channels that cannot change instantaneously, such as `sc_signal`. By separating the two phases of evaluation and update, it is possible to guarantee deterministic behaviour (because a primitive channel will not change value until the update phase occurs - it cannot change immediately during the evaluation phase).

However SystemC can model software, and in that case it is useful to be able to cause a process to run without a delta cycle (i.e. without executing the update phase). This requires events to be notified immediately (*immediate notification*). Immediate notification may cause non-deterministic behaviour.

Notification of events is achieved by calling the `notify()` method of class `sc_event`.

There are three cases to consider for the notify method.

- `notify()` with no arguments: immediate notification. Processes sensitive to the event will run during the current evaluation phase
- `notify()` with a zero time argument: delta notification. Processes sensitive to the event will run during the evaluation phase of the next delta cycle
- `notify()` with a non-zero time argument: timed notification. Processes sensitive to the event will run during the evaluation phase at some future simulation time

The `notify()` method cancels any pending notifications, and carries out various checks on the status of existing notifications.

The behaviour of the simulation kernel can now be described

1. Initialization: execute all processes (except `SC_CTHREAD`s) in an unspecified order.
2. Evaluation: select a process that is ready to run and resume its execution. This may cause immediate event notifications to occur, which may result in additional processes being made ready to run in this same phase.
3. Repeat step 2 until there are no processes ready to run.
4. Update: execute all pending calls to `update()` resulting from calls to `request_update()` made in step 2.
5. If there were any delta event notifications made during steps 2 or 4, determine which processes are ready to run due to all those events and go back to step 2.

6 If there are no timed events, simulation is finished.

7 Advance the simulation time to the time of the earliest pending timed event notification.

8 Determine which processes are ready to run due to all the timed events at what is now the current time, and go back to step 2.

Note the functions `update()` and `request_update()`. The kernel provides these functions specifically for modelling primitive channels such as `sc_signal`. `update()` actually runs during the update phase if it was requested by calling `request_update()` during the evaluation phase.

A primitive channel

So how do you write a primitive channel? It is actually surprisingly straightforward! Firstly, all primitive channels are based on the class `sc_prim_channel` - you can think of this as the primitive channel equivalent of `sc_module` for hierarchical channels.

Here is the code of a FIFO channel, which is a drastically simplified version of the "built-in" `sc_fifo` channel. It is simplified in that it only provides blocking methods, and it is not a template class (it only works on type `char`).

Here is the interface, `fifo_if.h`

```
#include "systemc.h"
class fifo_out_if :  virtual public sc_interface
{
public:
  virtual void write(char) = 0;          // blocking write
  virtual int num_free() const = 0;      // free entries
protected:
  fifo_out_if()
  {
  };
private:
  fifo_out_if (const fifo_out_if&);      // disable copy
  fifo_out_if& operator= (const fifo_out_if&); // disable
};

class fifo_in_if :  virtual public sc_interface
{
public:
  virtual void read(char&) = 0;          // blocking read
  virtual char read() = 0;
  virtual int num_available() const = 0; // available
                                         // entries
protected:
  fifo_in_if()
  {
  };
private:
  fifo_in_if(const fifo_in_if&);            // disable copy
  fifo_in_if& operator= (const fifo_in_if&); // disable =
};
```

Basically, there is a read and write method, both of which are blocking, i.e. they suspend if the FIFO is empty (read) or full (write).

Here is the code for the first part of the channel.

```
#include "systemc.h"
#include "fifo_if.h"

class fifo
: public sc_prim_channel, public fifo_out_if,
  public fifo_in_if
{
protected:
  int size;                  // size
  char* buf;                 // fifo buffer
  int free;                  // free space
  int ri;                    // read index
  int wi;                    // write index
  int num_readable;
  int num_read;
  int num_written;

  sc_event data_read_event;
  sc_event data_written_event;

public:
  // constructor
  explicit fifo(int size_ = 16)
  : sc_prim_channel(sc_gen_unique_name("myfifo"))
  {
    size = size_;
    buf = new char[size];
    reset();
  }

  ~fifo()                    //destructor
  {
    delete [] buf;
  }
```

Notes

- The channel is derived from `sc_prim_channel`, not `sc_module`

- The constructor automatically generates an internal name so the user does not have to specify one
- The constructor uses dynamic memory allocation, so there must also be a destructor to delete the claimed memory.
- There are two `sc_event` objects created. These are used to signal to the blocked read and write processes when either space becomes available (if a write is blocked) or data becomes available (when a read is blocked).

The next few functions are used to calculate if space is available and how much is free. The algorithm uses a circular buffer, accessed by the write index (`wi`) and the read index (`ri`).

```
int num_available() const
{
  return num_readable - num_read;
}

int num_free() const
{
  return size - num_readable - num_written;
}
```

Here is the blocking write function. Note that if `num_free()` returns zero, the function calls `wait(data_read_event)`. This is an example of dynamic sensitivity. The thread from which write is called will be suspended, until the data_read_event is notified.

```
void write(char c)         // blocking write
{
  if (num_free() == 0)
    wait(data_read_event);
  num_written++;
  buf[wi] = c;
  wi = (wi + 1) % size;
  free--;
  request_update();
}
```

Once the process resumes after `data_read_event`, it stores the character in the circular buffer, and then calls `request_update()`. `request_update()` makes sure that the simulation kernel calls `update()` during the update phase of the kernel.

Here is the reset function that clears the FIFO.

```
void reset()
{
  free = size;
  ri = 0;
  wi = 0;
}
```

And here is the read function. Behaviour is similar to the write function, except this time the process blocks if there is no space available (the FIFO is full).

```
  void read(char& c)        // blocking read
  {
    if (num_available() == 0)
      wait(data_written_event);
    num_read++;
    c = buf[ri];
    ri = (ri + 1) % size;
    free++;
    request_update();
  }
```

For convenience, here is a "shortcut" version of read, so we can use

```
  char c = portname->read();
```

syntax.

```
  char read()                // shortcut read function
  {
    char c;
    read(c);
    return c;
  }
```

And finally the `update()` method itself. This is called during the update phase of the simulation kernel. It checks to see if data was read or written during the evaluation phase, and then calls `notify(SC_ZERO_TIME)` as appropriate to tell the blocked `read()` or `write()` functions that they can proceed.

```
  void update()
  {
    if (num_read > 0)
      data_read_event.notify(SC_ZERO_TIME);
    if (num_written > 0)
      data_written_event.notify(SC_ZERO_TIME);
    num_readable = size - free;
    num_read = 0;
    num_written = 0;
  }
};
```

The top level of the design looks very similar to the hierarchical channel. Here it is (from `main.cpp`)

```
#include "systemc.h"
#include "producer.h"
#include "consumer.h"
#include "fifo.h"
```

```
int sc_main(int argc, char* argv[])
{
  sc_clock ClkFast("ClkFast", 1, SC_NS);
  sc_clock ClkSlow("ClkSlow", 500, SC_NS);

  fifo fifo1;

  producer P1("P1");
  P1.out(fifo1);
  P1.Clock(ClkFast);

  consumer C1("C1");
  C1.in(fifo1);
  C1.Clock(ClkSlow);

  sc_start(5000, SC_NS);

  return 0;
}
```

Note how there is no need to give the FIFO primitive channel a name, due to the use of `sc_gen_unique_name()`.

Conclusions

This chapter has shown a glimpse of writing a primitive channel. There are many more details, and you might want to look at the source code for `sc_signal` or `sc_fifo` to see more information.

A particular thing to note is the use of dynamic sensitivity - note how the blocked read and write functions are actually overriding the static sensitivity to the clock signal.