Introduction to SystemC Tutorial

SystemC is essentially a C++ library used for modeling concurrent systems in C++. Along with concurrency, SystemC provides a notion of timing as well as an event driven simulations environment. Due to it's concurrent and sequential nature, SystemC allows the description and integration of complex hardware and software components.

To some extent, SystemC can be seen as a Hardware Description Language. However, unlike VHDL or Verilog, SystemC provides sophisticated mechanisms that offer high abstraction levels on components interfaces. This, in turn, facilitates the integration of systems using different abstraction levels. Further more SystemC isn't a new language, it is C++ - consequently existing software IP can be seamlessly linked into a SystemC project.

This tutorial will focus on the strong connection between C++ and SystemC by making the analogy between Hardware modeling and Object-Oriented modeling.

The first section will look at the creation of Hardware components in C++.

The second section will highlight the benefits of a SystemC implementation.

Lastly, a worked example will allow you to gain experience with the steps involved in the creation, and simulation, of a SystemC design.

Contents

I. Hardware Modeling in C++

II. Introduction to SystemC

III. SystemC Worked Example

I. Hardware Modeling with C++

Rational

This tutorial is an introduction to the use of C++ for describing hardware models. Essential constructs for the creation of SystemC models are also introduced.

Note this is a deliberately simplified description of C++. Full details on the language, coding styles; design guidelines can be found on the Essential C++ for SystemC class (more details at www.esperan.com)

Introduction

C++ implements Object-Orientation on the C language. For most Hardware Engineers, the principles of Object-Orientation seem fairly remote from the creation of Hardware components. However, ironically, Object-Orientation was created from design techniques used in Hardware designs. Data abstraction is the central aspect of Object-Orientation which incidentally, is found in everyday hardware designs through the use of publicly visible “ports” and private “internal signals”. Furthermore, the principle of “composition” used in C++ for creating hierarchical design is almost identical to component instantiation found in hardware designs. The coming sections will introduce the basics of C++ by looking at the creation of a hardware component.

The Class

The class in C++ is called an Abstract Data Type (ADT). It defines both data members and access functions (also called methods). Both data members and access functions are said to be private by default. In other words, data members and access functions are not visible from the outside world. This ADT mechanism is analogous to a package and package body in VHDL. The designer is responsible for making publicly available the essential set of access functions for manipulating an ADT.

The semantics for a C++ class declaration is as follows:

```
class counter
{
  int value;
public:
  void do_reset() { value = 0 ; }
  void do_count_up() { value++ ; }
  int do_read() { return value; }
};
```

In this example we see the declaration of an ADT called `counter` with a data member `value` and publicly available access functions: `do_reset`, `do_count_up` and `do_read`. Although this class declaration is complete, a class declaration will commonly also contain specialized functions such as constructors and a destructor. When constructors are used, they provide initial values for the ADT’s data members. This mechanism is the only allowed means for setting a default value to any data member.

A destructor is used to perform clean-up operations before an instance of the ADT becomes out of scope. Pragmatically, the destructor is used for closing previously opened files or de-allocating dynamically allocated memory.

An example of an ADT with constructors and a destructor is as follows:

```
class counter
{
  int value;
public:
  void do_reset() { value = 0 ; }
  void do_count_up() { value++ ; }
  int do_read() { return value; }
  counter() {
    cout << "This a simple constructor" << endl;
    value = 10;
  }
  counter(int arg): value(arg) {
    cout << "This a more interesting constructor" << endl;
  }
  ~counter() {
    cout << "Destroying a counter object" << endl;}
};
```

In these examples, all access functions are made visible to the outside world through the use of the `public` keyword. However, not all functions have to be made public; they can be held hidden from the rest of the world through the use of the `private` keyword.

The Object

An object is an instance of an ADT. Any number of instances can be made of a given ADT and each of these instances is initialized individually. An example of object instantiation and message passing is as follows:-

```
void main() {

counter first_counter;
counter second_counter(55);

// Message passing
first_counter.do_reset();
for (int i=0; i < 10; i++) { first_counter.do_count_up(); }
second_counter.do_count_up();

first_counter.do_reset();
second_counter.do_reset();
}
```

In this example two instances of the counter ADT are created. The `first_counter` instance uses the default constructor which does not require any arguments. The `second_counter` instance uses a more sophisticated constructor form that requires an argument (55).

Messages are sent to individual objects via the 'dot' notation which is similar to what is used for "struct" or record data structures.

The building of more complex objects can be achieved through the composition mechanism. This mechanism is akin to component instantiation in Hardware design.

The following example illustrates the principle of hierarchical design through composition.

```
class modulo_counter
{
  counter internal_counter;
  int terminal_count;
public:
  void do_reset() { internal_counter.do_reset(); }
  void do_count_up() {
    if ( internal_counter.do_read() < terminal_count ) {
      internal_counter.do_count_up();
    } else { internal_counter.do_reset(); }
  }
  int do_read() { return internal_counter.do_read(); }
  modulo_counter(int tc):
    terminal_count(tc),
    internal_counter(0) {
    cout << "A new modulo_counter instance" << endl; }
};
```

In this example a new ADT `modulo_counter` is created from an instance of a counter ADT called `internal_counter`. An additional data member `terminal_count` was added to provide the modulo value for this new ADT. The access function `do_count_up` is customized to take in considerations of the modulo counter specifications. As it can be observed, most actions are delegated to the `internal_counter` instance via message passing. It is interesting to notice how the constructor function is used to initialize both the `terminal_count` value and the instantiated object `internal_counter`. The argument `tc` is used to set the value of `terminal_count` and the value `0` is passed to the constructor of the `internal_counter` object.

Inheritance

Along with composition, C++ provides a sophisticated mechanism for reusing code called: inheritance. With inheritance designers are able to create new ADTs from existing ones by accessing all public elements of a parent class into a child class. This principle can be more appropriate than composition since it usually requires less effort to achieve the same goal. However inheritance does not replace composition but complements it. The following example illustrates the creation of a modulo counter ADT from an existing counter ADT whilst using inheritance.

```
class modulo_counter : public counter {
  int terminal_count;
public:
  void do_count_up() {
    if ( do_read() < terminal_count ) {
      counter::do_count_up(); }
    else { do_reset(); }
  }
  modulo_counter(int tc): terminal_count(tc), counter(0) {
    cout << "A new modulo_counter instance" << endl; }
};
```

In this example a new access function called `do_count_up` is created. This overrides the inherited function from the parent class `counter`. The remaining public access functions, (`do_reset`, `do_read`) found in the parent class, are now available to the child class: `modulo_counter`. The creation of the `modulo_counter` class is greatly simplified through the use of inheritance. Nevertheless, the `do_read` and `counter::do_count_up` functions had to be used in the newly created `do_count_up` function to access the private data members inside the parent class. Although this is a valid solution, C++ offers a more pragmatic alternative, consisting in using a protected encapsulation in place of a private one. Unlike private members, protected members are inherited along with public ones when a parent class is derived, however, protected data members remain hidden from the outside world in the same way as private members do. A new implementation of the `counter` and `modulo_counter` class using protected data members is as follows:

```
class counter
{
protected:
  int value;
public:
  void do_reset() { value = 0 ; }
  void do_count_up() { value++ ; }
  int do_read() { return value; }
  counter() {
    cout << "This a simple constructor" << endl;
    value = 10;
  }
  counter(int arg): value(arg) {
    cout << "This a more interesting constructor" << endl;
  }
  ~counter() { cout << "Destroying a counter object"
    << endl;}
};

class modulo_counter : public counter {
protected:
  int terminal_count;
public:
  void do_count_up() {
    if ( value < terminal_count ) { value++; }
    else { value = 0; }
  }
  modulo_counter(int tc): terminal_count(tc), counter(0) {
    cout << "A new modulo_counter instance" << endl; }
};
```

In this last example the data member `value` is directly accessed from within the `modulo_counter` class. The use of the protected encapsulation has simplified the creation of the derived class by making data members from the parent class accessible.

The principle of single inheritance can be extended to allow a child class to be built from multiple parents. This principle is referred as multiple inheritance. Multiple inheritance enables designers to rapidly create new sophisticated classes from a multitude of existing ADTs. The use of multiple inheritance commonly requires careful planning since it can create numerous undesirable side effects. An example of multiple inheritance is as follows:

```
class reg {
protected:
    int value;
public:
    void do_reset() { value = 0; }
    int do_read() { return value; }
    void do_write(int arg) { value = arg; }
    reg(): value(0) {}
};

class up_counter: virtual public reg {
public:
    void do_count_up() { value++; }
};
```

```
class down_counter: virtual public reg {
public:
    void do_count_down() { value--; }
};

class up_down_counter: public up_counter, public down_counter { };
```

Here the up_down_counter ADT is created from two parent classes: up_counter and down_counter. The up_down_counter does not require any code since it inherits all of its implementation from its parent classes. It is important to point out that both the up_counter and down_counter are inheriting virtually the register class. The virtual inheritance is used here to prevent multiple declarations of the value, do_reset, do_read, do_write inside the up_down_counter since this ADT inherits those attributes twice, though both the up_counter and down_counter inheritance.

Template Class

The template mechanism is used for creating more versatile ADTs. Templates can be used for variable types or values. An example of a simple template class is as follows:

```
template<int min, int max> class barrel_counter {
private:
    int value;
public:
    void do_reset() { value =  min; }
    void do_count_up() {
        if (value< max ) { value++; }
        else {value = min; }
    }
    int do_read() { return value; }
    barrel_counter(): value(min) { }
};


int main(int argc, char *argv[])
{
    barrel_counter<10, 50> first_counter;
    barrel_counter<0, 10> second_counter;

    for (int i=0; i < 60; i++) {
        first_counter.do_count_up();
        second_counter.do_count_up();
        cout << first_counter.do_read() << endl;
        cout << second_counter.do_read() << endl;
    }
    return 0;
}
```

This counter implementation defines two templates min and max; these variables are then used inside the class to define the boundaries of the barrel counter. This example uses templates to set variable values; alternatively templates can be used to set a variable type. An example of templates used for variable types is as follows:

```
template<class T> class adder {
private:
    T result;
public:
    T add( T a, T b ) {
        result = a + b ;
        return result ;
    }
};

int main(int argc, char *argv[]) {
    adder<int> integer_adder;
    adder<float> float_adder;

    int int_result = integer_adder.add(3, 5);
    float float_result = float_adder.add(6.7, 10.2);
    cout << int_result << endl;
    cout <<float_result << endl;
    return 0;
}
```

This example uses a variable type `T` used throughout the class to provide a versatile ADT implementation. At a later stage two objects of type `adder` are declared, however both adders use different types for its operations.

Conclusions

As we illustrated in this section, Hardware components can be modeled in C++ and to some extent the mechanisms used are similar to those used in HDLs. Additionally C++ provides inheritance as a way to complement the composition mechanism and promotes design reuse.

Nevertheless, C++ does not provide for concurrency which is an essential aspect of systems modeling. Furthermore, timing and propagation delays cannot easily expressed in C++. The SystemC library provides additional mechanisms such as processes and dedicated data types to tackle C++ modeling deficiencies.

In the next section of this tutorial we will discover how SystemC allows easier and more efficient modeling of hardware by resolving some of the deficiencies of C++.

II. Introduction to SystemC

Rational

This tutorial is aimed at Hardware and Software engineers wishing to acquire an understanding of the SystemC language and its various uses.

No prior knowledge of C++ or HDL is required for the completion of this tutorial; however, throughout this tutorial we will be drawing parallels between SystemC and other languages.

Note this is a deliberately simplified description of SystemC. Full details on the language, coding styles; design guidelines can be found on the Esperan SystemC Fundamentals class (more details at www.esperan.com)

Introduction

SystemC is a C++ library used for supporting system level modeling. It supports various abstraction levels and can be used for fast, efficient designs and verification. The SystemC library is provided by the Open SystemC Initiative, a non-profit independent organisation. The OSCI is composed of numerous companies, universities and individuals. The ultimate aim of the organisation is to achieve IEEE standardisation for SystemC.

The SystemC reference simulator is freely available at `www.systemc.org` . Numerous EDA vendors provide commercial implementations of the SystemC language and support for mixed languages simulations. Along with the SystemC language, a number of additional application specific libraries are freely available either at `www.systemc.org` or `www.testbuilder.net`

Why SystemC instead of C++

The C++ language is based on sequential programming. Consequently it is not suited for the modeling of concurrent activities. Furthermore most system and hardware models require a notion of delays, clocks or time. features which are not present in C++ as a software programming language. As a result, complex and detailed systems cannot be easily described in C++ alone. Additionally communication mechanisms used in hardware models, such as signals and ports, are very different from those used in software programming. Lastly, the data types found in C++ are too remote from the actual hardware implementation. Ultimately, new dedicated data types and communication mechanisms have to be provided.

SystemC Implementation

The SystemC core language is based on C++. As a C++ library, layered on top of C++, SystemC defines data types dedicated to hardware modeling such as bit and vector types as well as fixed point types. Core language elements such as modules, processes, events, channels, event driven simulation kernel are also present. Finally, elementary channels such as signals or FIFOs are provided to implement communication mechanisms between concurrent objects.

Modules

A module is a C++ class - it encapsulates a hardware or software description. SystemC defines that any module has to be derived from the existing class sc_module. SystemC modules are analogous to Verilog modules or VHDL entity/architecture pairs, as they represent the basic building block of a hierarchical system. By definition, modules communicate with other modules through channels and via ports. Typically a module will contain numerous concurrent processes used to implement their required behaviour. An example of a module is as follows:

```
class counter: public sc_module {
  int value;
public:
  sc_in<bool> clk;
  sc_in<bool> count;
  sc_in<bool> reset;
  sc_out<int> q;

  SC_HAS_PROCESS(counter);

  counter(sc_module_name nm): sc_module(nm), value(0) {
  SC_METHOD(do_count);
    sensitive<< clk.pos() << reset ;
  }
protected:
  void do_count() {
    if (reset) { value = 0; }
    else if (count) {
      value++;
      q.write(value);
    }
  }
};
```

This example illustrates the creation of a counter module. The first part consists in the declaration of input and output ports. Ports are created form existing SystemC template classes: sc_in<> and sc_out<>. A type passed as a template argument defines the type of the data exchanged on the ports. The macro SC_HAS_PROCESS is used to indicate to the simulator's kernel that this specific counter class will contain concurrent processes.

The constructor function is used to register any of the access functions as concurrent processes. In this case, the do_count function is registered an SC_METHOD which is similar to a Verilog always or a VHDL process. It is worth observing that the constructor function defines a parameter of type sc_module_name. This parameter is required by the sc_module parent class. For the sake of simplicity, this parameter can be seen as a string type.

The last part of this module defines the do_count function. At this point, it can be observed that the do_count function is being made sensitive to the reset port and the positive edge of the clk port. Consequently, it will only be activated on a change of either the reset or the clk ports.

Ports and Interfaces

As can be observed in the previous example, ports are defined as objects inside a module. For obvious reasons they will have to be made publicly available to the outside world through the use of the `public` keyword.

Although SystemC provides numerous predefined ports such as: `sc_in<>`, `sc_out<>`, `sc_inout<>`, `sc_fifo_in<>`, `sc_fifo_out<>`, and more, the language also allows the definition of user defined ports and how should they be accessed. This is done through the use of the `sc_port<>` class. An example of the use of the sc_port is as follows:

```
sc_port< sc_signal_in_if<bool>, 1 > clk;
```

This declaration is, in effect, identical to:

```
sc_in<bool> clk ;
```

As it can be observed, ports are defined by a name `clk,` a type `bool` and an interface `sc_signal_in_if<>`.

The interface `sc_signal_in_if<>` defines the restricted set of available messages that can be send to the port `clk`. For instance the functions: `read()` or `default_event()` are defined by the `sc_signal_in_if<>` class. At this stage it worth pointing out that the implementation of the functions defined inside an interface, is provided by a third party element: the channel. Channels will be discussed in greater detail at the end of this section.

Processes

Two kinds of process exist in SystemC: `SC_METHOD` and `SC_THREAD`. To some extent, the two processes are similar since they will both will contain sequential statements and have their own thread of execution; hence, they will operate concurrently. Additionally, both SystemC processes allow for both static sensitivity and dynamic sensitivity.

By definition, `SC_METHOD` cannot be suspended during its execution. Once the execution of the `SC_METHOD` as been performed, it halts and waits for new activities on its static or dynamic sensitivity list before executing again. This is similar to a VHDL `process` or Verilog `procedure` with a sensitivity list.

In contrast with `SC_METHOD`, `SC_THREAD` can be suspended during execution and resumed at a later stage. Furthermore, by definition, `SC_THREAD` only executes once during simulation and then suspends. This mechanism is akin to a Verilog `initial` block. For this reason, designers will commonly use an infinite loop inside an `SC_THREAD` to prevent it from ever exiting before the end of the simulation. An example of an `SC_THREAD` is as follows:

```
void do_count() {
  while(true) {
    if (reset) { value = 0; }
    else if (count) {
      value++;
      q.write(value); }
    wait();
  }
}
```

Because of performance overheads, `SC_METHOD` implementations are usually preferred to `SC_THREAD`.

Channels

Channels are SystemC's communication medium. They can be seen as a more generalized form of signals. SystemC provides a exhaustive range of predefined channels for generic uses such as: `sc_signal`, `sc_fifo`, `sc_semaphore` etc. Additionally SystemC permits the creation of user defined channels. This feature of the language significantly differentiates it form other Hardware Description Languages such as Verilog or VHDL. Providing abstraction at the interface level of components achieves greater design and modeling flexibilities. By definition modules are interconnected via channels and ports. In turn, ports and channels communicate via a common interface.

Note: For the purpose of this tutorial we will only consider predefined SystemC channels.

Composition

The principle of composition is analogous to hierarchical design. SystemC encourages composition for the creation of complex systems. To illustrate composition, an additional component (`testbench`) is created. A simple example of a test bench is as follows:

```
class testbench: public sc_module {
public:
  sc_out<bool> clk;
  sc_out<bool> reset;
  sc_out<bool> count;
  SC_HAS_PROCESS(testbench);

  testbench(sc_module_name nm): sc_module(nm) {
    SC_THREAD(clk_gen);
    SC_THREAD(stimuli);
  }

  void clk_gen() {
    while(true) {
      clk.write(true);
      wait(10, SC_NS);
      clk.write(false);
      wait(10, SC_NS);
    }
  }

  void stimuli() {
    while(true) {
      reset.write(true);
      count.write(false);
      wait(10, SC_NS);
      reset.write(false);
      wait(50, SC_NS);
      count.write(true);
      wait(200, SC_NS);
    }
  }
};
```

This module defines two processes `clk_gen` and `stimuli`. The `clk_gen` process is used to generate the clock signal used by the counter. The `stimuli` process is used to generate the `enable` signal used by the counter. Both processes are registered as `SC_THREAD` since they contain `wait()` statements.

The next step consists of instantiating a `counter` and a `testbench` object inside a top level module as follows:-

```
class top: public sc_module {
public:
  sc_signal<bool> clk_sig, count_sig, reset_sig;
  sc_signal<int> q_sig;

  counter uut ;   // Counter instance
  testbench tb ;  // Testbench instance

  top(sc_module_name nm): sc_module(nm), uut("uut"), tb("tb") {
    // Testbench’s ports connection
    tb.clk(clk_sig);
    tb.reset(reset_sig);
    tb.count(count_sig);

    // Counter’s ports connection
    uut.clk(clk_sig);
    uut.reset(reset_sig);
    uut.count(count_sig);
    uut.q(q_sig);
  }
};
```

This example illustrates composition as two objects are declared inside the class `top`. SystemC defines that port/channel connections are performed inside the constructor function. The semantics used for port/channel connection is:

```
instance_name.port_name(channel_name);
```

Simulation

Simulation instructions are usually located inside a function called: `sc_main`. The `sc_main` functions are equivalent to the more conventional `main` function in C++.

This function will execute simulation specific commands such as setting the simulator’s resolution, channels to be traced, top level instance, simulation running time and more. An example of a `sc_main` function is as follows:-

```
int sc_main(int argc, char* argv[])
{
  sc_set_time_resolution(1, SC_NS);
  top verif_env("verif_env");  // Top level instance

  // Creating a trace file
  sc_trace_file *tf;
  tf = sc_create_vcd_trace_file("trace");
  sc_trace(tf, verif_env.clk_sig, "clk_sig");
  sc_trace(tf, verif_env.reset_sig, "reset_sig");
  sc_trace(tf, verif_env.count_sig, "count_sig");
  sc_trace(tf, verif_env.q_sig, "q_sig");

  // Running the simulation
  sc_start(1000, SC_NS);
  sc_close_vcd_trace_file(tf);
  return (0);
}
```

In this example the top level component of class `top` is called: `verify_env`. This declaration along with the `sc_start()` function call are sufficient to run a SystemC simulation. However a number of trace commands have been added to allow visualization of the results of the simulation. These traces files are stored using the VCD format inside a file called: `trace.vcd`.

Conclusions

This section presented the creation of SystemC modules and their associated concurrent processes. We examined the differences between `SC_METHOD` and `SC_THREAD` and illustrated the composition mechanism in SystemC. Lastly we saw how the `sc_main` function is used to provide a simulation “layer”.

The next section of the tutorial considers the simulation of a simple worked example.

III.SystemC Worked Example

SystemC Setup

These instructions assume that you are operating under a Linux environment. Alternatively you can install cygwin (`www.cygwin.com`) on an existing Windows™ environment. For versions or platform specific information, please refer to the `INSTALL` file inside the SystemC tar file.

1. Download the SystemC core
   a) Go to `www.systemc.org` and create an account.
   b) Go to download page and download the version 2.0.1 version of SystemC core, `systemc-2.0.1.tgz`.

Note if you are using a `cygwin` based environment you may want to download SystemC version 2.1 instead.

2. Uncompress the SystemC core
   a) `gunzip –d systemc-2.0.1.tgz`
   b) `tar –xvf systemc-2.0.1.tar`

Note: By default the files will be extracted to directory `systemc-2.0.1`.

3. Install SystemC core
   a) Change to the directory `systemc-2.0.1`.
   ```
   cd systemc-2.0.1
   ```
   b) Create a temporary directory and change into it
   ```
   mkdir objdir
   cd objdir
   ```
   c) Set the environment
   ```
   setenv CXX g++
   ```
   d) Configure the package for your system
   ```
   ../configure
   ```
   e) Compile the package
   ```
   gmake (or make)
   ```
   f) Install the package
   ```
   gmake install or make install
   ```
   g) Remove the temporary directory
   ```
   cd ..
   rm –rf objdir
   ```

Waveform Viewer Setup

A waveform viewer will allow you to graphically display a VCD trace file from SystemC simulation. Two free-ware VCD viewers are recommended. Other tools are available.

1. Download `Dinotrace` from `www.veripool.com/dinotrace` or `GtkWave` from `www.cs.man.ac.uk/apt/tools/gtkwave`
2. Install the waveform viewer according to its instructions

Workshop Files Setup

1. Download Esperan's SystemC workshop file from: `www.esperan.com/tutorial/sc_tutorial.tgz`
2. Uncompress the archive

   ```
   gunzip –d sc_tutorial.tgz
   tar –xvf sc_tutorial.tar
   ```

Note: By default the files will be extracted to directory sc_tutorial

1. Move to the `Makefile` directory

   ```
   cd sc_tutorial
   ```

2. Configure the `Makefile` for your environment
   a) Edit the file: `Makefile.defs`

      ```
      vi Makefile.defs
      ```

   b) Modify the SystemC installation path

      ```
      SYSTEMC = put_your_installation_path_here
      ```

      e.g. `SYSTEMC = /usr/local/SystemC/systemc-2.0.1`

Running the SystemC Counter Example

1. Move to the workshop directory

   ```
   cd simpleCounterSC
   ```

2. Compile the counter example

   ```
   make –f  Makefile.osci
   ```

3. Execute the simulation

   ```
   ./run.exe
   ```

4. Examine the results of the simulation

   `gtkwave trace.vcd &` or alternatively `dinotrace trace.vcd &`