Modeling and Simulation with SystemC

1 Introduction

This document is a basic tutorial in SystemC [3]. It is supposed to be studied as a preparation for the lab assignments in the courses TDTS07 and TDDI08. The assignments are found in Section 3. Parts of this document are based on the books by Black and Donovan [1] and Grötker et al. [2].

The SystemC language has in the recent years become an important alternative to the traditional hardware and system description languages VHDL and Verilog. In an industrial context, SystemC has been used extensively, and nowadays synthesis tools and other design automation tools support SystemC models. In this lab, we shall use SystemC to model and describe timed systems on a high level of abstraction (transactions-level modeling). We shall further use simulations to validate our models.

2 SystemC

SystemC is a language that allows designers to develop both the hardware and software components of their system together. This is all possible to do at a high level of abstraction. Strictly speaking, SystemC is not a language, but rather a library for C++, containing structures for modeling hardware components and their interactions. Consequently, SystemC can be compared to the hardware description languages VHDL and Verilog. An important aspect that these languages have in common is that they all come with a simulation kernel, which allows the designer to evaluate the system behavior through simulations. Nowadays, there exist tools for high-level synthesis of SystemC models; this has pushed the industry to adopt this unified hardware–software design language in large scale.

The rest of this section will introduce the basic data types and structures in the SystemC library together with a set of illustrative examples. Note that we cover only the part of the SystemC language that is needed to finish the assignments in this lab successfully. The interested reader is encouraged to search for additional literature in the library as well as on the Internet. It can also be useful to look into the SystemC language reference manual (LRM), which can be downloaded from the SystemC Web page [3]. We use SystemC version 2.3.1, which is installed at `/courses/TDTS07/sw/systemc`.

2.1 Model of Time

One of the most important components in the SystemC library is the model of time. The underlying model is based on 64-bit unsigned integer values. However, this is hidden to the programmer through the data type (class) `sc_time`. Due to the limits of the underlying implementation of time, we cannot represent continuous time, but only discrete time. Therefore, in SystemC there is a minimum representable time quantum, called the *time resolution*. This can be set by the user, as we shall demonstrate in later parts of this document. Note that this time resolution limits the maximum representable time, because the underlying data type representing the time is a 64-bit integer value. Thus, any time value smaller than the time resolution will be rounded to zero. The default time resolution is set to one picosecond, that is, to 10^{-12} seconds.

The class `sc_time` has a constructor of the following form:

```
sc_time(double, sc_time_unit);
```

where `sc_time_unit` is an enumerated type (`enum`) with the following variants:

- `SC_FS` (femtosecond),
- `SC_PS` (picosecond),
- `SC_NS` (nanosecond),
- `SC_US` (microsecond),
- `SC_MS` (millisecond), and
- `SC_SEC` (second).

The following C++ statement (using the constructor of `sc_time`) creates an instance `t1` of the type `sc_time`, representing 45 microseconds:

```
sc_time t1(45, SC_US);
```

or alternatively (using the copy constructor of `sc_time`)

```
sc_time t1 = sc_time(45, SC_US);
```

When dealing with `sc_time` objects, it is sometimes useful to convert them to a floating-point number. This can be achieved by the following statement:

```
double t1_magnitude = t1.to_default_time_units();
```

In this way we obtain the time in the default time unit. The default time unit can be set by the user; for example,

```
sc_set_default_time_unit(2, SC_MS);
```

sets the default time-unit to two milliseconds. This function may be called only once, and it has to be done in the `sc_main` function before starting the simulation with `sc_start`. This will be explained in Section 2.4. There is also a function for setting the time resolution, which also should be called before simulation start; for example,

```
sc_set_time_resolution(1, SC_US);
```

sets the time resolution to 10^{-6} seconds.

One special constant is worth mentioning, namely, `SC_ZERO_TIME`, which is simply `sc_time(0, SC_SEC)`. Finally, we mention that a lot of arithmetic and boolean operations are defined in the class `sc_time`, such as, $+$, $-$, $>$, $<$, etc. For more detailed information, refer to the SystemC language reference manual (LRM), which you can download from the SystemC web page [3].

2.2 Modules

Modules are the basic building blocks in SystemC. A module comprises ports, concurrent processes, and some internal data structures and channels that represent the model state and the communication between processes, respectively. A module can also use another module in a hierarchy. A module is described with the macro `SC_MODULE`. Actually, the macro `SC_MODULE(`*Module*`)` expands to

`class` *Module* : `public sc_module`

where `sc_module` is the base class for all SystemC modules. Its constructor has a mandatory argument *name,* which is the name of the created instance; therefore, all modules have a name that is needed mainly for debugging and information purposes.

A module must have a constructor. The constructor serves the same purpose as it does for a normal C++ class. Moreover, the constructor declares which functions are to be treated as concurrent processes (Section 2.3). The declaration of the constructor is done with the macro `SC_CTOR`, whose argument is the module name. Thus, a template for a simple module could look like the following piece of code:

```
SC_MODULE(TemplateModule) {
  // Ports, processes, internal data, etc.
  SC_CTOR(TemplateModule) {
    // Body of the constructor.
    // Process declaration, sensitivities, etc.
  }
};
// IMPORTANT! Remember that a class/module declaration
// should end with a semicolon (;).
```

An instance of the module can then be created as follows:

```
TemplateModule instance1("Instance_1");
```

In this example the constructor only has one argument, namely, the module name. Note that it is possible to obtain the name of any instance of the module by calling `name()` anywhere in the module. This is a function in the base class `sc_module`. In some cases, you may need more arguments in the constructor. Then the constructor must be declared together with the `SC_HAS_PROCESS` macro. The following piece of code illustrates the use of the `SC_HAS_PROCESS` macro:

```
SC_MODULE(TemplateModule) {
  // Ports, processes, internal data, etc.
  SC_HAS_PROCESS(TemplateModule);
  TemplateModule(sc_module_name name, int other_parameter)
    : sc_module(name)
  {
    // Body of the constructor.
    // Process declaration, sensitivities, etc.
  }
};
```

2.3 Processes and Events

The functionality in SystemC is achieved with *processes*. As opposed to C++ functions, which are used to model sequential system behavior, processes provide a mechanism for simulating concurrency. A process is a C++ member function of the SystemC module. The function is declared to be a process in the constructor. There are two macros (`SC_METHOD` and `SC_THREAD`) which can be used in the constructor for such process registration with the simulation kernel.

The two types of processes differ in a number of ways. `SC_THREAD`s are run at the start of simulation and are run only once. They can suspend themselves with the `wait` statement, and thus let the simulation resume with another process. Typically, a process of type `SC_THREAD` contains an infinite loop (`for(;;)`) and at least one `wait` statement. For example, if you want to suspend a thread for two seconds you may write

```
wait(2, SC_SEC);
```

A process of the type `SC_METHOD` cannot contain any `wait` statements. Thus, when such a process is started by the simulation kernel, it is run from the beginning till the end without any interrupts and in zero simulated time. An `SC_METHOD` can be sensitive to a set of *events*. This sensitivity must be declared in the constructor. This declaration makes an event a trigger for a set of processes. Whenever a certain event occurs, it makes the corresponding sensitive processes to become ready for execution. In Section 2.5, we shall illustrate how to declare processes and sensitivities to events.

An *event* `e` is declared as

```
sc_event e;
```

without any arguments. The only possible action on an event is to trigger it. This is done with the `notify` function. A SystemC event is the result of such a notification and has no value and no duration, but it only happens at a single point in time and possibly triggers one or several processes. An `sc_event` can be notified using *immediate* or *timed* notification. Immediate notification of `e` is achieved with

```
e.notify();
```

The statement

```
e.notify(10,SC_MS);
```

triggers the event `e` after 10 milliseconds of simulated time (it overrides any previous timed notification that did not yet occur). Sometimes it is useful to cancel a scheduled notification of an event. For example, the most recent scheduled notification of `e` can be canceled with the following statement:

```
e.cancel();
```

Instead of suspending a thread for a given amount of time, it is also possible to use the `wait` statement with an `sc_event` as an argument. The statement

```
wait(e);
```

suspends the thread until event `e` is triggered. It is important to note that only `wait` statements and event notifications can advance simulated time.

2.4 Writing a Test Bench

Having a set of SystemC modules, it is possible to test them by writing a test bench. A test bench is an implementation of the function

```
int sc_main(int argc, char **argv);
```

which is the starting point of the execution. The two parameters `argc` and `argv` are the same as for a C++ `main` function. These contain information about command-line arguments to the program. The `sc_main` function contains instantiations of a set of modules, as well as an optional part consisting of function calls for setting up the simulation. Thereafter, the function `sc_start` is called with the simulation time as argument. This starts the simulation phase, and the function returns when the simulation is finished. Note that the function `sc_start` can be called at most once. The simulation phase consists of executions of the concurrent processes (`SC_METHOD`s and `SC_THREAD`s) in the instantiated modules. The following piece of code shows a simple template:

```
#include <systemc.h>

int sc_main(int argc, char **argv)
{
  // Create instances of the modules
  // contained in the simulation model.
  // ...
  // Setup the simulation.
  sc_set_time_resolution(1, MS);

  // Invoke the simulation kernel.
  sc_start(20,SC_SEC);
  // or sc_start(sc_time(20, SC_SEC));
  // or sc_start(); for infinite simulation

  // Clean up if needed.

  return 0;
}
```

Note that it is possible to set the time resolution. However, this has to be done before the invocation of the SystemC simulation kernel.

2.5 Example 1

We shall now study an example with a SystemC module that represents a counter. The counter is incremented periodically every second. Further, we illustrate how to write a test bench. In the directory

```
/courses/TDTS07/tutorials/systemc/counter
```

you may find the files `counter.h` and `counter.cc` that defines a counter module, as well as the test bench `counter_testbench.cc`. Copy the whole directory to your account. You also find `Makefile`, which together with the `make` tool compiles and links the two `.cc`-files into an executable called `counter.x`. Run `make` in the terminal to build the example. This creates an executable called `counter.x` by compiling and linking the `.cc`-files together with the SystemC library. Study the `sc_main` function to find out how to run the program from the command line. Figures 1 and 2 list the files `counter.h` and `counter.cc` respectively whereas Figure 3 lists `counter_testbench.cc`.

The module `Counter` has two variables: an integer variable holding the value of the counter and an `sc_event` variable, which is the event that is supposed to be triggered by the part of the module responsible for incrementing the counter. The constructor takes two arguments, the name of the instance and the initial value of the counter. Note that the second argument has the default value 0. The constructor registers two SystemC processes, namely, `count_method` as an `SC_METHOD` sensitive to `count_event`, and `event_trigger_thread` as an `SC_THREAD` that generates periodic notifications of `count_event`. This makes the simulation kernel call `count_method` every second. This function increments the counter value and prints the current value of the counter to the standard output. Note how events are registered to the sensitivity list of `SC_METHOD`s by using the `sensitive` statement. For example, to register a process called `print_method` to the events `e1` and `e2` you should write the following in the constructor:

```
#ifndef COUNTER_H
#define COUNTER_H

#include <systemc.h>

SC_MODULE(Counter) {
  int value;
  sc_event count_event;

  SC_HAS_PROCESS(Counter);
  Counter(sc_module_name name, int start = 0);
  void count_method();
  void event_trigger_thread();
};

#endif // COUNTER_H
```

Figure 1: `counter.h`

```
#include <iostream>
#include "counter.h"

using std::cout;
using std::endl;

Counter::Counter(sc_module_name name, int start)
  : sc_module(name), value(start)
{
  SC_THREAD(event_trigger_thread);

  SC_METHOD(count_method);
  dont_initialize();
  sensitive << count_event;
}

void Counter::count_method()
{
  value++;
  cout << sc_time_stamp() << ": Counter has value "
       << value << "." << endl;
}

void Counter::event_trigger_thread()
{
  for (;;) {               // Loop infinitely.
    wait(1, SC_SEC);       // Wait one second.
    count_event.notify();  // Trigger count_method.
  }
}
```

Figure 2: `counter.cc`

```
#include <cassert>
#include <systemc.h>
#include "counter.h"

int sc_main(int argc, char **argv)
{
  // The command-line arguments are as follows:
  // 1. the initial value of the counter and
  // 2. the simulation time (in seconds).
  assert(argc == 3);

  int init_value = atoi(argv[1]);
  sc_time sim_time(atof(argv[2]), SC_SEC);

  // Instantiate the module.
  Counter c1("Counter_1", init_value);

  // Start the simulation.
  sc_start(sim_time);

  return 0;
}
```

Figure 3: `counter_testbench.cc`

```
SC_METHOD(print_method);
sensitive << e1 << e2;
```

Note the use of the function `sc_time_stamp()` to obtain the current time as an `sc_time` object. This function can be called anytime during simulation. Also note that `event_trigger_thread` is an infinite loop that uses the `wait` function to suspend itself and let the other process execute.

Finally, when you run the example, note that the runtime of your program is much less than the simulated time. The runtime (or simulation time) depends on the size of the model, the number of processes, and the complexity of each process itself.

2.6 SystemC Simulator Kernel

Having studied a simple example, let us now study the details of the SystemC simulation semantics. The execution of the simulator kernel can be viewed according to the following list of steps:

1. *Initialize.* In the initialize phase, each process is executed once (i.e., `SC_METHOD` or `SC_THREAD`). Processes of the type `SC_THREAD`s are executed until the first synchronization point, that is, the first `wait` statement, if there is one. As we have discussed before, it is possible to disable a process for being executed in this phase. This is done by calling `dont_initialize()` after the corresponding process declaration inside the constructor of the module. Observe that this is only done for processes of type `SC_METHOD`.

2. *Evaluate.* Select a ready to run process and execute/resume it. This may cause immediate notifications to occur (`e.notify()`), which may cause more processes to be ready to run in the same phase. The order in which the processes are executed is unspecified but deterministic, which means that two simulation runs will yield identical results.

3. Repeat Step 2 until there are no more processes to run.

4. *Update phase.* In this phase, the kernel updates the values assigned to channels in the previous evaluate cycle. We shall describe channels and their behavior in the next section.

5. Steps 2–4 is referred to as a *delta-cycle*, just as in VHDL and Verilog. If Step 2 or 3 resulted in delta event notifications (`wait(0)` or `e.notify(0)`), go back to Step 2 without advancing simulation time.

6. If there are no more timed event notifications, the simulation is finished and the program terminates.

7. Advance to the next simulation time that has pending events.

8. Determine which processes are ready to run due to the events that have pending notifications at the current simulation time. Go back to Step 2.

We now make some very important remarks about immediate and delayed notification. With *immediate* notification, that is, `e.notify()`, the event is triggered during the current delta-cycle, which means that any process sensitive to the event will be executed immediately without any update phase. With *delayed* notification, that is `e.notify(SC_ZERO_TIME)` or `wait(SC_ZERO_TIME)`, the triggered processes execute during a new delta-cycle, after the update phase.

2.7 Channels and Ports

We have now studied how concurrency is handled by using processes and events. This section introduces the basics of channels and ports, which is used to communicate between modules.

There are several types of channels in SystemC, and it is also possible to define customized channels. We shall study one of the basic channels, namely, `sc_signal`. This is called an *evaluate–update* channel, meaning that if some process writes a new value to the channel, the channel's value is changed in the update phase of the simulation kernel, that is, at the end of a delta-cycle. This is done in order to achieve concurrency. For example, if several processes read and write to the same channel in the same delta-cycle, the order of execution of the processes should not have an effect on the functionality. Thus, if a process writes to a channel with a new value, a reader in the same delta-cycle (same time) reads the "old" value. This can be compared to signals in VHDL.

The syntax for declaring a SystemC signal is as follows:

```
sc_signal<T> signame;
```

where `T` is the data type for the signal. Thus, it is possible to create signals with integer values, floating point values, boolean values, etc. For example, the following piece of code shows how to create three different signals:

```
sc_signal<int> sig_int;
sc_signal<double> sig_double;
sc_signal<bool> sig_bool;
```

Reading and writing to signals can be done using the `read` and `write` functions. For example, consider the following piece of code in an `SC_THREAD`, assuming that the current value of `sig_int` is 0:

```
sig_int.write(1);
int value = sig_int.read();
cout << value << endl;
wait(SC_ZERO_TIME);
value = sig_int.read();
cout << value << endl;
sig_bool.write(false);
```

The first print statement prints 0 on the screen, even though there is a preceding `write` statement. However, after the wait statement, there has been an update phase, and thus the second print statement prints 1 on the screen. An `sc_signal` also has an event that is triggered when the signal value changes. Thus, it is possible to have the following statement in an `SC_THREAD`:

```
wait(sig_int.default_event());
```

We shall now describe how to communicate between modules on a channel. For this purpose we shall use *ports*, which can be viewed as the inputs and outputs of a module. We focus on the two most important basic ports, `sc_in<T>` and `sc_out<T>`, which are used to model input and output ports of any data type. We also mention that there is another port type `sc_inout<T>`.

Basically, a port can be viewed as a pointer to a channel. Thus, a port is connected to a channel through an interface that enables reading and writing. In this way, modules can communicate changes to other modules. The following piece of code illustrates how to declare a module with two input ports and one output port:

```
SC_MODULE(Divider) {
  sc_in<int> numerator;
  sc_in<int> denominator;
  sc_out<double> quotient;

  // Constructor, processes, etc.
  // ...
};
```

The ports will then be connected to channels at module instantiation. This is demonstrated with an example in the next section. Further, it is possible to initialize a value to output ports in the constructor of the corresponding module. Thus, we could write

```
quotient.initialize(1.5);
```

to initialize the channel connected to the output port to the value 1.5. Reading from and writing to the channels is achieved through the ports by using the `read` and `write` methods. Remember that we view a port as a pointer to a channel. Thus, reading from the port `numerator` is done by using the expression `numerator->read()`. Similarly, writing a value to the output port is done with `quotient->write(4.2)`. We also mention that we can achieve the same thing by viewing ports as normal variables. Thus, the following statements have the same effect:

```
int sum = numerator + denominator;
int sum = numerator->read() + denominator->read();
```

Similarly, for the write operation, the following statements achieve the same result:

```
quotient = 1.1;
quotient->write(1.1);
```

Note that both of the latter updates will have effect *only* after the evaluate-update cycle, that is, the delta-cycle, as we discussed in the beginning of this section. Finally, the port of type `sc_inout<T>` is used exactly as the input and output ports. For this type of port it is possible to both read and write to the channel connected to the port.

2.8 Example 2

In this section, we discuss a divider module, which is implemented with channels and ports. We shall also study a test bench and show how to connect channels to ports. Copy the entire contents of the directory

```
/courses/TDTS07/tutorials/systemc/divider
```

to your home directory. You will find the following files:

- `divider.h` and `divider.cc`, describing a divider module that takes two inputs and computes the quotient whenever any of the two inputs changes.
- `input_gen.h` and `input_gen.cc`, describing an input generator for the divider module. The inputs are read from a text file, given by the user. The file consists of pairs of integers representing the two inputs. Each row in the file represents one pair of inputs. The file `input.txt` shows what such a file can look like.
- `monitor.h` and `monitor.cc`, describing a monitor module that takes one input and writes the value to a file as soon as the value is updated. This is used to monitor the output value of the divider module, as will be seen in the test bench.

```
#ifndef DIVIDER_H
#define DIVIDER_H

#include <systemc.h>

SC_MODULE(Divider) {
  sc_in<int> numerator;
  sc_in<int> denominator;
  sc_out<double> quotient;

  SC_HAS_PROCESS(Divider);
  Divider(sc_module_name name);

  void divide_method();
};

#endif // DIVIDER_H
```

Figure 4: `divider.h`

- `divider_testbench.cc` implements a test bench for a divider module connected to three signals. These signals are also connected to an input generator and a monitor. Study the code to see how to run the simulation.
- `Makefile` is the file that helps you build the program using the `make` tool. The resulting executable is called `divider.x`.

The source code for the whole example is listed in Figures 4–10. Study the source code carefully and revisit the discussions in this document to understand the example. You may also make modifications to the example to further try things out. Before we end this section, let us make a number of remarks regarding some parts of the source code listed in the figures.

First, in `divider.cc`, note that `divide_method` is sensitive to both ports. That is, if any of the input ports is updated with a new value, the divide method will be triggered.

Next, in the implementation of the input generator and the output monitor, we use the C++ classes `ifstream` and `ofstream` for achieving file I/O.

Finally, there are many interesting aspects to note in `divider_testbench.cc`. First of all, we create three signals that represent the two inputs and one output in the divider module. After the modules are instantiated, we must connect the signals to the ports of the modules. This is done in *positional form* for all instances. For example, for the instance `divider`, this is done with the following statement:

```
divider(numerator_sig, denominator_sig, quotient_sig);
```

This connects the signals, in the order they are listed, to the ports, in the order they are declared in the module declaration. Thus, the order of the signals in the previous statement is important in order to connect the signals to the ports correctly. It is also possible to use *named form* to connect the signals to the ports. For example, the following statements have the same effect as the positional form used for the divider module:

```
#include "divider.h"

Divider::Divider(sc_module_name name)
  : sc_module(name)
{
  quotient.initialize(0);

  SC_METHOD(divide_method);
  dont_initialize();
  sensitive << numerator << denominator;
}

void Divider::divide_method()
{
  int num = numerator->read();
  int denom = denominator->read();
  double q = double(num) / denom;
  quotient->write(q);
}
```

Figure 5: `divider.cc`

```
#ifndef INPUT_GEN_H
#define INPUT_GEN_H

#include <systemc.h>
#include <fstream>

using std::ifstream;

SC_MODULE(Generator) {
  sc_out<int> numerator;
  sc_out<int> denominator;

  SC_HAS_PROCESS(Generator);
  Generator(sc_module_name name, char *datafile);
  ~Generator();

  void generate_thread();

  ifstream *in;
};

#endif // INPUT_GEN_H
```

Figure 6: `input_gen.h`

```
#include "input_gen.h"
#include <cassert>

Generator::Generator(sc_module_name name, char *datafile)
  : sc_module(name)
{
  assert(datafile != 0);       // An input file should be given.

  in = new ifstream(datafile); // Open the input file.
  assert(*in);                 // Check that everything is OK.

  SC_THREAD(generate_thread);

  numerator.initialize(0);
  denominator.initialize(0);
}

Generator::~Generator()
{
  delete in;
}

void Generator::generate_thread()
{
  int num, denom;
  for (;;) {
    wait(1, SC_SEC);     // Generate new inputs every second.
    *in >> num >> denom; // Read from the input file.
    numerator->write(num);
    denominator->write(denom);
  }
}
```

Figure 7: `input_gen.cc`

```
#ifndef MONITOR_H
#define MONITOR_H

#include <systemc.h>
#include <fstream>

using std::ofstream;

SC_MODULE(Monitor) {
  sc_in<double> quotient;
  sc_in<int> denominator;

  SC_HAS_PROCESS(Monitor);
  Monitor(sc_module_name name, char *outfile);
  ~Monitor();

  void monitor_method();
  void check_constraints_method();

  ofstream *out;
};

#endif // MONITOR_H
```

Figure 8: `monitor.h`

```
divider.numerator(numerator_sig);
divider.quotient(quotient_sig);
divider.denominator(denominator_sig);
```

Here, the order does not matter anymore. The first form is more compact whereas the second form is clearer.

2.9 Optional Exercise

Change the divider example in Section 2.8 such that the inputs are generated by the `Counter` module (Section 2.5) instead of by the `Generator` module. Further, instead of waiting a second before each increment step in the counter, you should wait for a random amount of time between 1 and 10 seconds. You are free to make appropriate changes to the source code for both the examples.

3 Assignment

3.1 Description

In this assignment, your task is to model a traffic light controller and to implement it in SystemC. Further, you should simulate your system to check some properties.

```
#include <cassert>
#include "monitor.h"

Monitor::Monitor(sc_module_name name, char *outfile)
  : sc_module(name)
{
  assert(outfile != 0);
  out = new ofstream(outfile);
  assert(*out);

  SC_METHOD(monitor_method);
  dont_initialize();
  sensitive << quotient;

  SC_METHOD(check_constraints_method);
  dont_initialize();
  sensitive << denominator;
}

Monitor::~Monitor()
{
  delete out;
}

void Monitor::monitor_method()
{
  double q = quotient->read();
  *out << "quotient(" << sc_time_stamp() << ") = " << q << endl;
}

void Monitor::check_constraints_method()
{
  assert(denominator != 0);
}
```

Figure 9: `monitor.cc`

```
#include <systemc.h>
#include "divider.h"
#include "input_gen.h"
#include "monitor.h"

int sc_main(int argc, char **argv)
{
  // The command-line arguments are as follows:
  // 1. the simulation time (in seconds),
  // 2. the file with input data (see input.txt), and
  // 3. the file to write output data.
  assert(argc == 4);

  sc_time sim_time(atof(argv[1]), SC_SEC);
  char *infile = argv[2];
  char *outfile = argv[3];

  // Create channels.
  sc_signal<int> numerator_sig;
  sc_signal<int> denominator_sig;
  sc_signal<double> quotient_sig;

  // Instantiate modules.
  Divider divider("Divider");
  Generator gen("Generator", infile);
  Monitor monitor("Monitor", outfile);

  // Connect the channels to the ports.
  divider(numerator_sig,denominator_sig,quotient_sig);
  gen(numerator_sig,denominator_sig);
  monitor(quotient_sig,denominator_sig);

  // Start the simulation.
  sc_start(sim_time);

  return 0;
}
```

Figure 10: `divider_testbench.cc`

The traffic light controller is specified as follows. The system must control the lights in a road crossing. There are four lights: one for vehicles traveling in direction North–South (NS), one for vehicles traveling in direction SN, one for the direction West–East (WE), and one for the direction EW. There are also four sensors detecting vehicles in the corresponding directions. The system must satisfy the following properties:

1. The lights shall work *independently*. For example, if the light NS is green, the light SN is red if there are no cars coming in the direction SN.

2. Of course, safety constraints apply. For instance, SN must not be green at the same time as WE is green.

3. If a vehicle arrives at the crossing (as detected by the respective sensor), it will eventually be granted the green light.

You must describe the controller in SystemC, and further—using print statements, file I/O, or monitors—implement functionality to demonstrate that all the mentioned properties hold during simulation. To check the properties and to study the execution of your system, you have to generate traffic at the traffic lights. To demonstrate that your implementation satisfies the required properties, you must run the controller with traffic that is generated randomly and with traffic generated in a way that specifically targets to check some of the properties.

The changing of the lights (red to green, green to red) must be represented by channels (one channel for each light). The arrival of cars must be modeled with events or channels (boolean or otherwise); for example, you may use `sc_event` or the change of the value of a boolean channel. For each of the four directions (NS, SN, WE and EW) there should be a corresponding event/channel declared.

3.2 Examination

You must write and hand in a report in which you present and explain your solution to the problem described above. In the report, you must describe the simulations you have conducted to validate your solution. You must present the simulation results and explain them in sufficient detail.

References

[1] D. C. Black and J. Donovan. *SystemC: From The Ground Up*. Eclectic Ally, 2004.

[2] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[3] SystemC. `http://accellera.org/downloads/standards/systemc`.