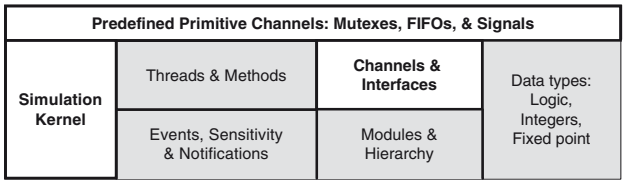
Chapter 8
Basic Channels



Thus far, we have communicated information between concurrent processes using events and using ordinary module member data. Within one instance of time (a delta cycle), the order of execution is not defined; therefore, we must be extremely careful when sharing data.

Events let us manage simulation concurrency, but they require careful coding. Because the code may miss capturing an event, it is important to update a handshake variable indicating when a request is made, and clear it when the request is acknowledged. This mechanism also allows safe data communication between concurrent simulation processes.

Let's consider the gas station example again (Fig. 8.1). The customer notices an empty tank and arrives at the pump ①. The attendant has to be watching when the customer requests a fill-up ②, and has to make note of it if in the middle of filling up another customer ③. In the case of two arriving customers, if the attendant waits on either customer's request (i.e., **`wait`**`(e_request1|e_request2)`), the **`sc_event`** semantics do not allow the attendant to know which customer made the request. In other words, the attendant does not know which request triggered the **`wait()`** to return. This is why the `gas_station` model uses the status of the gas tank as an indicator to choose whether to fill the tank. Similarly, the customer must watch to see if the tank was actually filled when the attendant yells done ④.

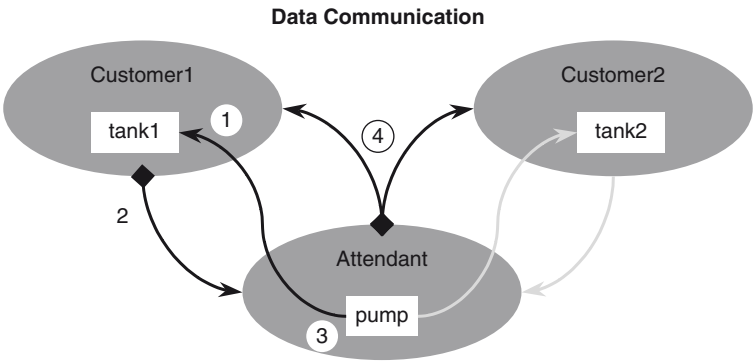


Fig. 8.1 Gas station processes and events

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_8, © Springer Science+Business Media, LLC 2010

SystemC has built-in mechanisms, known as channels, to reduce the tedium of these chores, to aid communications, and to encapsulate complex communications. SystemC has two types of channels: primitive and hierarchical. This chapter covers the topic of primitive channels. Hierarchical channels are the subject matter of the chapter on Custom Channels.

8.1 Primitive Channels

SystemC's primitive channels are known as primitive because they contain no hierarchy, no simulation processes, and are designed simple to be very fast. All primitive channels inherit from the base class **sc_prim_channel**. Because they are SystemC channels, they must also inherit from and implement one or more SystemC interface classes.

The SystemC library contains several built-in primitive channels. This chapter focuses on the simplest channels, **sc_mutex**, **sc_semaphore**, and **sc_fifo**<*T*>. Additional primitive channel topics will be discussed in later chapters.

8.2 sc_mutex

Mutex is short for mutually exclusive text. In computer programming, a mutex is a program object that lets multiple program threads share a common resource, such as file access, without colliding.

During elaboration, a mutex is created with a unique name. Subsequently, any process that needs the resource must lock the mutex to prevent other processes from using the shared resource. The process should unlock the mutex when the resource is no longer needed. If another process attempts to access a locked mutex, that process is prevented from doing so until the mutex becomes available (unlocked).

SystemC provides the mutex function via the **sc_mutex** channel. The **sc_mutex** class implements the **sc_mutex_if** interface class (Fig. 8.2). This class contains several access methods including both blocked and unblocked styles. Remember that blocking methods can only be used in **SC_THREAD** processes.

```
sc_mutex NAME;

NAME.lock();     // Lock the mutex,
                 // wait until unlocked if in use
int NAME.trylock() // Non-blocking, returns success

NAME.unlock();  // Free a previously locked mutex
```

Fig. 8.2 Syntax of **sc_mutex_if** and **sc_mutex**

The example of the gas station attendant is a good example of a resource that needs to be shared. Our gas station only has a single pump, so only one car at a time is filled.

Another example of a resource requiring a mutex is automobile controls. Only one driver at a time can sit in the driver's seat. In a simulation modeling the interaction of drivers across town with a variety of vehicles, having one driver per set of controls might be interesting to model (Fig. 8.3).

```
class car : public sc_module {
  sc_mutex drivers_seat;
public:
  void drive_thread(void);
  …
};

void car::drive_thread(void) {
  drivers_seat.lock(); // sim driver acquires seat
  start();
  … // operate vehicle
  stop();
  drivers_seat.unlock(); // sim driver leaves
                         // vehicle
   …
}
```

Fig. 8.3 Example of **sc_mutex**

An electronic design application of an **sc_mutex** is arbitration for a shared bus. Here the ability of multiple masters to access the bus must be controlled. In lieu of an arbiter design, the **sc_mutex** might be used to manage the bus resource quickly until an arbiter can be designed. In fact, the mutex might even be part of the class implementing the bus model as illustrated in the following example (Fig. 8.4):

```
class bus : public sc_module {
  sc_mutex bus_access;
  …
  void write(int addr, int data) {
    bus_access.lock();
    // perform write
    bus_access.unlock();
  }
  …
};
```

Fig. 8.4 Example of **sc_mutex** used in bus class

Used with an **SC_METHOD** process, access might look like this (Fig. 8.5):

```
void grab_bus_method() {
  if (bus_access.trylock() == 0) {
    // access bus
    …
    bus_access.unlock();
  }
}
```

Fig. 8.5 Example of **sc_mutex** with an **SC_METHOD**

One downside to the **sc_mutex** is the lack of an event that signals when an **sc_mutex** is freed. This drawback necessitates using **trylock()** repeatedly based on some other event or time-based delay to use the shared resource. Assuming one process has locked a resource using **sc_mutex**, the second process wanting the same resource must call **trylock()** in conjunction with a **wait()** or return. The second process must use **wait()** or return between calls to **trylock()** to allow the first process some simulation cycles to finish and unlock the resource. Otherwise, the simulation will hang with infinite calls to **trylock()**.

8.3 sc_semaphore

For some resources, you may want to model more than one copy or owner. A good example of this would be parking spaces in a parking lot.

To manage this type of resource, SystemC provides the **sc_semaphore** class (Fig. 8.6). The **sc_semaphore** class inherits from and implements the **sc_semaphore_if** class. When creating an **sc_semphore** object, it is necessary to specify how many are available. In a sense, a mutex is merely a semaphore with a count of one. An **sc_semaphore** access consists of waiting for an available resource and then posting notice when finished with the resource.

```
sc_semaphore NAME(COUNT);

NAME.wait();       // Lock one semaphore
                   // Wait until available if in use
int NAME.trywait()  // Non-blocking, return success

int NAME.get_value() // Returns available semaphores

NAME.post();       // Free one previously locked
                   // semaphore
```

Fig. 8.6 Syntax of **sc_semaphore_if** and **sc_semaphore**

It is important to realize that the **sc_semaphore::wait()** is a distinctly different method from the **wait()** method previously discussed in conjunction with **SC_THREAD**. In fact, under the hood, the **sc_semaphore**::**wait()** is implemented with the **wait**(event).

A modern gas station with self-service would be a good example (Fig. 8.7) for using semaphores. Gas pumps can be represented with a semaphore where the count is set to the number of available pumps.

```
SC_MODULE(gas_station) {
  sc_semaphore pump(12);
  void customer1_thread {
    for(;;) {
      // wait till tank empty
      …
      // find an available gas pump
      pump.wait();
      // fill tank & pay
  }
};
```

Fig. 8.7 Example of **sc_semaphore**—gas_station

A multiport memory model is a good example (Fig. 8.8) of an electronic system-level design application using an **sc_semaphore**. You might use the semaphore to indicate the number of concurrent read or write accesses allowed.

```
class multiport_RAM {
  sc_semaphore read_ports(3);
  sc_semaphore write_ports(2);
  …
  void read(int addr, int& data) {
    read_ports.wait();
    // perform read
    read_ports.post();
  }
  void write(int addr, const int& data) {
    write_ports.wait();
    // perform write
    write_ports.post();
  }
  …
};//endclass
```

Fig. 8.8 Example of **sc_semaphore**—multiport_RAM

Other examples might include allocation of timeslots in a TDM (time division multiplex) scheme used in telephony, controlling tokens in a token ring, or perhaps even switching information to obtain better power management.

8.4 sc_fifo

Probably the most popular channel for modeling at the architectural level is the **sc_fifo**<*T*> channel (Fig. 8.9). First-in first-out queues (i.e., FIFOs) are a common data structure used to manage data flow. FIFOs are some of the simplest structures to manage.

In the very early stages of architectural design, the unbounded[1] STL **list**<*T*> (singly linked list) provides an easy implementation of a FIFO. Further in the design process, when FIFO depths are determined and SystemC elements come into stronger play, the **sc_fifo**<*T*> may be used to model at a higher level of detail.

The **sc_fifo**<*T*> class inherits from and implements two interface classes: **sc_fifo_in_if**<*T*> and **sc_fifo_out_if**<*T*>. It may not be intuitive at first, but the "in" interface is used for reading from the FIFO, and the "out" interface is for writing to the FIFO.

By default, an **sc_fifo**<*T*> has a depth of 16. The data type (i.e., **typename**) of the FIFO elements also needs to be specified. An **sc_fifo**<*T*> may contain any data type including large and complex structures (e.g., a TCP/IP packet or a disk block).

For example, FIFOs may be used to buffer data between an image processor and a bus, or a communications system might use FIFOs to buffer information packets as they traverse a network.

```
sc_fifo<ELEMENT_TYPENAME> NAME(SIZE);

NAME.write(VALUE);
NAME.read(REFERENCE);
… = NAME.read() /* function style */
if (NAME.nb_read(REFERENCE)) { // Non-blocking
                               // true if success
  …
}
if (NAME.num_available() == 0)
  wait(NAME.data_written_event());
if (NAME.num_free() == 0)
  next_trigger(NAME.data_read_event());
```

Fig. 8.9 Syntax of **sc_fifo**<*T*>—abbreviated

[1] This queue is limited only by the resources of the simulation machine itself.

Some architectural models are based on Kahn process networks[2] for which unbounded FIFOs provide the interconnect fabric. Although **sc_fifo**<*T*> is not unbounded, because reads and writes are blocking, it is possible to use them for this purpose given an appropriate depth. The depth needs to be set such that consumers and producers don't end up in a deadlock. This is illustrated in the next very simple example (Fig. 8.10).

```
SC_MODULE(kahn_ex) {
  …
  sc_fifo<double> a, b, y;
  …
};
// Constructor
kahn_ex::kahn_ex() : a(24), b(24), y(48)
{
  …
}
void kahn_ex ::stim_thread() {
  for (int i=0; i!=1024; ++i) {
    a.write(double(rand()/1000));
    b.write(double(rand()/1000));
  }
}
void kahn_ex::addsub_thread() {
  while(true) {
    y.write(kA*a.read() + kB*b.read());
    y.write(kA*a.read() - kB*b.read());
  }//endforever
}
void kahn_ex::monitor_method() {
  cout << y.read() << endl;
}
```

Fig. 8.10 Example of **sc_fifo<double>** kahn_ex

Software uses for FIFOs are numerous and include such concepts as mailboxes and other types of queues.

Note that when considering efficiency, passing pointers to large objects is most efficient. Be sure to consider using a safe pointer object if using a pointer. The **shared_ptr**<*T*> of the GNU publicly licensed Boost library (http://www.boost.org) makes implementation of smart pointers very straightforward.

Generally speaking, the STL may be more suited to software FIFOs. The STL **list**<*T*> might be used to manage an unknown number of stimulus data from a testbench.

In theory, an **sc_fifo**<*T*> could be synthesized at a behavioral level. It currently remains for a synthesis tool vendor to provide the functionality.

[2]Kahn, G. (1974). The semantics of a simple language for parallel programming. In J.L. Rosenfeld (Ed.), *Proceedings of IFIP Congress 74 (*pp.471–475). Amsterdam: North-Holland.

8.5 Exercises

For the following exercises, use the samples provided in www.scftgu.com

Exercise 8.1: Examine, predict the output, compile, and run `mutex_ex`.

Exercise 8.2: Examine, compile, and run `semaphore_ex`. Add another family member. Explain discrepancies in behavior.

Exercise 8.3: Examine, compile, and run `fifo_fir`. Add a second filter stage to the network.

Exercise 8.4: Examine, compile, and run `fifo_of_ptr`. Discuss how one might compensate for the simulated transfer of a large packet.

Exercise 8.5: Examine, compile, and run `fifo_of_smart_ptr`. Notice the absence of **`delete`**.