FUNCTIONAL SPECIFICATION FOR SYSTEMC 2.0

Update for SystemC 2.0.1

Version 2.0-Q
April 5, 2002

Bugs and Suggestions

Please report bugs and suggestions about this document to

http://www.systemc.org

CONTRIBUTORS

The contributors to this functional specification are the members of the SystemC Language Working Group.

- *Stuart Swan, Cadence Design Systems, Inc.*
- *Dirk Vermeersch, CoWare, Inc.*
- *Dündar Dumlugöl, CoWare, Inc.*
- *Peter Hardee, CoWare, Inc.*
- *Takashi Hasegawa, Fujitsu Microelectronics, Inc.*
- *Adam Rose, Motorola*
- *Marcello Coppola, ST Microelectronics*
- *Martin Janssen, Synopsys, Inc.*
- *Thorsten Grötker, Synopsys, Inc.*
- *Abhijit Ghosh, Synopsys, Inc.*
- *Kevin Kranen, Synopsys, Inc.*

TABLE OF CONTENTS

1. INTRODUCTION

This functional specification describes the system level modeling features made available in SystemC 2.0. First, the primary modeling constructs in SystemC 1.0 are briefly reviewed. Next, the new system level modeling constructs in SystemC 2.0 are introduced. We show how these new modeling constructs enable users to cleanly model communication and synchronization in systems and even allow users to implement new models of computation within SystemC. The new features for modeling communication and synchronization are sufficiently general that all of the existing mechanisms in SystemC 1.0 for communication and synchronization can now be constructed on top the new SystemC 2.0 features.

This specification assumes that the reader has some familiarity with C++ and with an HDL such as Verilog or VHDL. Furthermore, a full understanding of SystemC 2.0 requires an understanding of SystemC 1.0 as outlined in the SystemC 1.0 User's Guide. The SystemC language itself is entirely based on C++ and the modeling constructs within SystemC are provided as a C++ class library.

Note: SystemC 2.0 is a superset of SystemC 1.0, i.e., all SystemC 1.0 designs are still valid in SystemC 2.0.

1.1 BRIEF REVIEW OF SYSTEMC 1.0

SystemC 1.0 provides a set of modeling constructs that are similar to those used for RTL and behavioral modeling within an HDL such as Verilog or VHDL.

Similar to HDLs, users can construct structural designs in SystemC 1.0 using modules, ports, and signals. Modules can be instantiated within other modules, enabling structural design hierarchies to be built. Ports and signals enable communication of data between modules, and all ports and signals are declared by the user to have a specific data type. Commonly used data types include single bits, bit vectors, characters, integers, floating point numbers, vectors of integers, etc. SystemC 1.0 also includes support for four state logic signals (i.e. signals that model 0, 1, X, and Z).

An important data type that is found in SystemC 1.0 but not in HDLs is the fixed-point type. This type is used to model fixed-point numbers in digital signal processing applications. It is easy and natural to model fixed-point numbers in SystemC, but this is very difficult to do in HDLs.

In VHDL, concurrent behaviors are modeled using processes. In Verilog, concurrent behaviors are modeled using "always" blocks and continuous assignments. In SystemC 1.0, concurrent behaviors are also modeled using processes. A process can be thought of as an independent thread of control, which resumes execution when some set of events occur or some signals change, and then suspends execution after performing some action. In SystemC 1.0, there is a limited ability for specifying the condition under which a process resumes execution: the process can only be sensitive to changes of values of particular signals, and the set of signals to which the process is sensitive must be pre-specified before simulation starts.

Since processes execute concurrently and may suspend and resume execution at user-specified points, SystemC process instances generally require their own independent execution

stack. (An equivalent situation in the software world arises in multi-threaded applications— each thread requires its own execution stack.) Certain processes in SystemC which suspend at restricted points in their execution do not actually require an independent execution stack— these processes types are termed "SC_METHODs". Optimizing SystemC designs to take advantage of SC_METHODs provides dramatic simulation performance improvements when the number of process instances in a design is large.

Hardware signals have several properties, which make modeling them in software non-trivial. Firstly, users often want to simulate hardware signals and registers as being initialized to "X" when simulation starts. This is useful for detecting reset problems in designs via X propagation techniques in simulation. Ins 0.8(Sy612.3(stem718. 0 ,2.70.9featu0.7iw[s prov612.8dents)10i.70(when)10.9(td '

Other components of the SystemC 2.0 standard include elementary library models which build on the core language (e.g. timers, FIFOs, signals, etc.) and which are widely applicable.

In SystemC 2.0, the simple and flexible synchronization capabilities provided by events and the `wait()` method allow a broad range of different channel types to be implemented without having to change the underlying simulation engine. All the required functionality is already present in the simulation kernel. Thus, SystemC 2.0 supports a very powerful generic model of computation. While the global model of time is fixed to an integer model, designers can construct specific channels to achieve their precise rules for communication between processes, process activation, and system wide event ordering.

Although continuous time models as used for example in analog modeling cannot yet be constructed in SystemC, virtually any discrete time system can be modeled in SystemC. Some well-known models of computation, which can be quite naturally modeled in SystemC 2.0, include:

- Static Multi-rate Data-flow
- Dynamic Multi-rate Data-flow
- Kahn Process Networks
- Communicating Sequential Processes
- Discrete Event as used for
  - RTL hardware modeling
  - network modeling (e.g. waiting-room models)
  - transaction-based SoC platform modeling

One example of how it is possible to achieve this layering of specific models of computation on top of the core language features within SystemC 2.0 is the hardware signal. In SystemC 1.0, the hardware signal was the only mechanism available for communication and synchronization between processes. In SystemC 2.0, the hardware signal is now implemented completely on top of channels, interfaces, and events. The SystemC 2.0 simulation kernel has no special support for hardware signals and is not aware if any are being used in a particular design. Note that this layering was introduced into SystemC in an unobtrusive way that enables existing SystemC 1.0 designs to continue to work unchanged in SystemC 2.0.

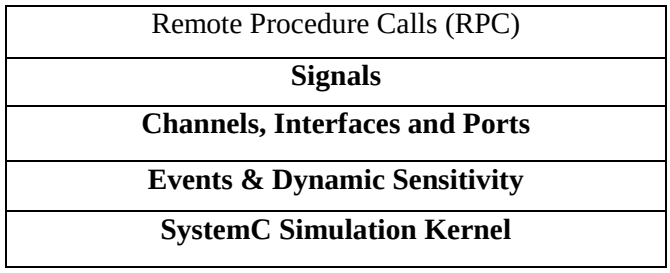
1.5 LAYERED APPROACH

We are pursuing a layered approach (see Figure 1-1). The SystemC simulation kernel forms the base layer. In order to facilitate both the modeling at higher levels of abstraction as well as the creation of refined communication channels, we introduce *dynamic sensitivity* and extend the notion of *events* at the next layer. This allows defining *channel types* and *interfaces*[1] and, where needed, *ports* offering dedicated access methods. This layer implements the so-called *interface-method-call* (IMC) scheme, which supports e.g. dynamic master/slave relationships between

[1] Interfaces are prerequisite to implement channels.

processes. Parts of SystemC 1.0, i.e., *signals*, will be implemented on top of the channels, interfaces and ports layer. The IMC scheme is a generalization of the RPC scheme as introduced in SystemC 1.1beta. Hence, the RPC scheme can be implemented on top of the other four layers.

Figure 1-1: Layered Approach.



1.6 SCOPE OF THIS SPECIFICATION

For SystemC to be successful as a system level modeling and IP delivery language, the following standards need to be in place.

1) The core language features which support system level modeling need to be defined and standardized.

2) Modeling guidelines which allow for interoperability of models from multiple vendors, and within multiple design flows, need to be established.

3) Library code which is not part of the core SystemC language definition, but which provides useful commonly used "utility" functionality needs to be developed and standardized. This library code will facilitate model interoperability and will make it easier for users to adopt SystemC.

4) Common design methodologies need to be established and proliferated.

A key point is that core language features need to be kept separate from modeling guidelines, library code, and design methodologies. In other words, the core language features should be general and support a wide variety of modeling styles and design methodologies.

By keeping the core language small and general purpose, we will make SystemC easier to use and implement.

1.7 DOCUMENT OVERVIEW

This functional specification is organized in three parts. The first and biggest part describes the core language features. The second part describes the elementary channels that are part of SystemC, and the third part describes methodology-specific channels, such as the `sc_link_mp` channel, which is part of the master-slave communication library. An appendix contains the pseudo-code of the SystemC 2.0 scheduler.

1.8 ACKNOWLEDGEMENTS

Many companies and individuals have contributed time and resources in the development of both SystemC 1.0 and SystemC 2.0. Some of these contributors are listed in the contributors section of this specification and in the SystemC 1.0 User's Guide.

It should be noted that the fundamental mechanisms used to model communication and synchronization in SystemC 2.0 – interfaces, channels, and events – were inspired by similar constructs in Professor Daniel Gajski's SpecC language. (For further information, see "SpecC: Specification Language and Methodology" at www.wkap.nl).

PART I. CORE LANGUAGE

2. GLOSSARY OF TERMS

The most common terms used in this document are explained in Table 2-1. Terms related to processes are explained in the following chapter.

Table 2-1: General terminology.

Method	A C++ method, i.e., a member function of a class.
Module	A structural entity, which can contain processes, ports, channels, and other modules. Modules allow expressing structural hierarchy.
Interface	An interface provides a set of method declarations, but provides no method implementations and no data fields.
Channel	A channel implements one or more interfaces, and serves as a container for communication functionality.
Port	A port is an object through which a module can access a channel's interface. But modules can also access a channel's interface directly.
Primitive Channel	A primitive channel is atomic, that is, it doesn't contain processes or modules, and it cannot directly access other channels.
Hierarchical Channel	A hierarchical channel is a module, that is, it can contain processes and other modules, and it can directly access other channels.
Event	A process can suspend on, or be sensitive to, one or more events. Events allow for resuming and activating processes.
Sensitivity	The sensitivity of a process defines when this process will be resumed or activated. A process can be sensitive to a set of events. Whenever one of the corresponding events is triggered, the process is resumed or activated.
Static Sensitivity	The sensitivity of the process is declared statically; i.e., it is declared during elaboration and cannot be changed once simulation has started. A so-called sensitivity list is used to define the static set of events.
Dynamic Sensitivity	The sensitivity of a process can be altered during simulation.
IMC	Interface Method Call
RPC	Remote Procedure Call

3. PROCESSES

Note: In the SystemC 1.0 Users' Guide, the term "process" is used to denote an `SC_THREAD` ("thread process"), `SC_METHOD` ("method process"), or `SC_CTHREAD` ("clocked thread process"). We will continue to use these terms, but we will also make the distinction between process (method) and its thread-of-execution more clear.

Processes play a central role in SystemC. They describe the functionality of the system, and allow expressing concurrency in the system. Processes are contained in modules, and they access external channel interfaces through the ports of a module. There are different types of processes, and different ways to activate processes. Before diving into the details, the terms related to processes are explained.

3.1 PROCESS TERMINOLOGY

To avoid any confusion in terminology, we first define the terms related to processes in Table 3-1.

Table 3-1: Process terminology.

Thread	A SystemC thread has its own thread of execution, but is not preemptive.
Automatically activated	Certain module methods (processes) are activated automatically when events occur that the processes are sensitive to.
Explicitly activated	Certain module methods must be called explicitly by other code in order to be activated.
`wait()`	A method that suspends execution of a thread. The arguments passed to `wait()` determine when execution of the thread is resumed.
Ok to call `wait()`	`SC_METHOD`s and the code that they call cannot call `wait()`, because they don't have their own thread of execution. `SC_THREAD`s and the code that they call can call `wait()`.
`SC_THREAD`	A module method which has its own thread of execution, and which can call code that calls `wait()`. `SC_THREAD`s are automatically activated. Also known as thread process.
`SC_METHOD`	A module method which does not have its own thread of execution, and which cannot call code that calls `wait()`. `SC_METHOD`s are automatically activated. Also known as method process.
`SC_CTHREAD`	A module method which has its own thread of execution, and which only has a positive or negative clock edge event in its sensitivity list. It can call code that calls `wait()` with a restricted argument list. `SC_CTHREAD`s are

	automatically activated. Also known as clocked thread process.

Note: All forms of `wait()` need to dynamically check that the current process (i.e., the top level caller) is not an `SC_METHOD`.

3.2 PROCESS INITIALIZATION

In SystemC 1.0, thread processes are not executed during the initialization phase of the simulation. Method processes are executed during the initialization phase of the simulation if they are made sensitive to input signals/ports. In VHDL, the initialization behavior is to execute them all during the initialization phase of the simulation.

The SystemC 2.0 scheduler will execute all thread processes and all method processes during the initialization phase of the simulation. If the behavior of a thread process is different between SystemC 1.0 and SystemC 2.0, insert one `wait()` statement before the infinite loop of the thread process.

To prevent the scheduler from executing a thread process or method process during the initialization phase of the simulation, you can use the `dont_initialize()` function. The function applies to the last declared process. For example,

```
SC_MODULE( my_module )
{
    // port(s)
    sc_in_clk clk;

    // process(es)
    void proc_a();
    void proc_b();

    // constructor
    SC_CTOR( my_module )
    {
        SC_THREAD( proc_a );
        sensitive << clk.pos();
        dont_initialize();  // don't initialize proc_a

        SC_METHOD( proc_b );
        sensitive << clk.neg();
        dont_initialize();  // don't initialize proc_b
    }
};
```

4. MODEL OF TIME

In SystemC 1.0, a real-valued and relative model of time is employed, i.e., the global clock is of type double and its time units do not have any relationship to absolute time, such as seconds or nanoseconds. For IP exchange, is should be possible to specify absolute time. Furthermore, an integer-valued model of time has certain advantages over a real-valued model of time. SystemC 2.0 uses an absolute and integer-valued model of time.

4.1 ABSOLUTE TIME VS. RELATIVE TIME

In SystemC 1.0, we can create a clock as follows:

```
sc_clock clk( “clk”, 20 );
```

This creates a clock with a period of 20 time units. The user can *interpret* this as 20 nanoseconds, or 20 seconds, etc. If there are other clocks in the system, the interpretation of the time unit should be the same. But SystemC does not enforce or check this, since the time unit is relative. It is not possible to deliver an IP block that has a clock with a period of, for example, exactly 20 nanoseconds, independent of the system it is integrated into.

What we want to be able to write is, for example

```
sc_clock clk( “clk”, 20, SC_NS );
```

, or

```
sc_time t1( 20, SC_NS );
sc_clock clk( “clk”, t1 );
```

, which in both cases creates a clock with a period of 20 nanoseconds.

4.2 INTEGER-VALUED VS. REAL-VALUED

A real-valued model of time has the advantage that the (dynamic) range of time values is much larger than in an integer-valued model of time. But when we look at problems such as underflow, overflow, and rounding of time values when additions or conversions occur, an integer-valued model of time has clear advantages.

Underflow can occur, for example, when a very small time value is added to a very large time value. With real-valued time values, this results in the very large time value (i.e., the very small time value is “lost”). With integer-valued time values, underflow cannot occur.

Overflow occurs when the resulting time value cannot be represented by the underlying data type. With integer-valued time values this is more of a problem then with real-valued time values.

The last problem has to do with rounding. If time is represented as an integer, then models may try to wait for a period of time which does not exactly match integer time units. This will be particularly common if delays are computed by estimation or delay calculation tools.

The biggest problem is underflow, because it's difficult to detect and prevent. The overflow and rounding problems are easy to detect and easy to report to the user in a sensible way. Hence, the integer-valued model of time is preferred over the real-valued model of time, provided that the underlying integer data type can represent a sufficiently large range of time values.

4.3 NEW MODEL OF TIME

In SystemC 2.0, the underlying data type for time is a 64 bit unsigned integer[2]. The default *time resolution*[3] is 1 picosecond. The user can change the time resolution with function **`sc_set_time_resolution()`**. For example,

```
sc_set_time_resolution( 10, SC_PS );
```

The following restrictions apply to setting the time resolution.

- The time resolution must be a power of ten.
- The time resolution can only be specified before the start of simulation.
- The time resolution can only be specified once.
- The time resolution can only be specified before any non-zero `sc_time` declaration.

For example,

```
sc_set_time_resolution( 10, SC_PS );
...
wait( 3.456, SC_NS );
```

rounds the time to wait to 3460 ps.

It is possible to get the current time resolution with function **`sc_get_time_resolution()`**, which does not take any arguments and returns a value of type `sc_time`. This function can be called anywhere in the code (i.e., before or after the start of simulation).

SystemC provides a type **`sc_time`** to specify time values. A variable of type `sc_time` can be created with two arguments: an argument of type double and an argument of type `sc_time_unit`. Type `sc_time_unit` is an enumerated type with the following values.

- `SC_FS` – femtoseconds
- `SC_PS` – picoseconds

[2] As in VHDL and Verilog.

[3] Also known as minimum resolvable time unit.

- `SC_NS` – nanoseconds
- `SC_US` – microseconds
- `SC_MS` – milliseconds
- `SC_SEC` – seconds

Type `sc_time` defines the copy constructor, the assignment operator, and the relational and equality operators.

For backward compatibility with SystemC 1.0, the *default time unit* is assumed to be 1 nanosecond. The default time unit can be changed by the user with function **`sc_set_default_time_unit()`**. To change the default time unit to 1 picosecond, for example, specify the following.

```
sc_set_default_time_unit( 1, SC_PS );
```

The following restrictions apply to setting the default time unit.

- The default time unit must be a power of ten.
- The default time unit must be larger than or equal to the time resolution.
- The default time unit can only be specified before the start of simulation.
- The default time unit can only be specified once.

For example,

```
sc_set_default_time_unit( 100, SC_PS );
...
sc_clock clk1( “clk1”, 10 );
```

sets the clock period for `clk1` to 1000 ps.

It is possible to get the current default time unit with function **`sc_get_default_time_unit()`**, which does not take any arguments and returns a value of type `sc_time`. This function can be called anywhere in the code (i.e., before or after the start of simulation).

Although still legal, the following old forms of `sc_clock` and `sc_start` are discouraged.

```
sc_clock clk1( “clk1”, 15 );
sc_start( 1000 );
```

Instead, the following new forms of `sc_clock` and `sc_start` should be used.

```
sc_clock clk1( “clk1”, 15, SC_NS );
sc_start( 1000, SC_NS );
```

5. EVENTS AND DYNAMIC SENSITIVITY

With SystemC 1.0, processes can be sensitive to events on input signals/ports only. Process sensitivity is statically defined, i.e., it cannot change during simulation. In SystemC 2.0, dynamic sensitivity allows for making processes sensitive to events on e.g. both input and output ports. The dynamic sensitivity of a process can be altered during simulation.

To be able to implement dynamic sensitivity, we use the SystemC 1.0 scheduling paradigm with an extended notion of events. The latter is needed because events are supported to a limited extend in SystemC 1.0. This extension requires only minor changes in the SystemC 1.0 simulation kernel.

5.1 WAIT() METHOD

In SystemC 1.0, static sensitivity is supported with a sensitivity list for each process in a module. These sensitivity lists are defined in the constructor of a module. Consider the following example.

Example 5-1: Static sensitivity.

```
SC_MODULE( my_module )
{
    // ports
    sc_in<int> input;
    sc_in_clk  clock;

    // processes
    void proc_a();
    void proc_b();

    // constructor
    SC_CTOR( my_module )
    {
        SC_THREAD( proc_a );
        sensitive_pos << clock;

        SC_THREAD( proc_b );
        sensitive << input;
        sensitive_neg << clock;
    }
};
```

In the above example, there are two thread processes in the module. Process `proc_a` is sensitive to the positive edge of the clock, whereas process `proc_b` is sensitive to a change in value on `input` and to a negative edge of the clock[4]. These sensitivity lists are static, i.e., they cannot change during simulation.

[4] In SystemC 2.0, "`sensitive <<`" takes events as arguments. The ports that can be used as arguments in SystemC 1.0 (and 2.0) are a special case, for which the default event from the signal (or channel) attached to the port is actually used.

In some cases, we want a process to be sensitive to a specific event or a specific collection of events, and this may change during simulation. This dynamic sensitivity is possible by using the **`wait()`** method. This method has been extended to allow specifying one or more events or a collection of events to wait for. For example:

Example 5-2: Dynamic sensitivity with the wait() method.

```
...
// wait until event e1 has been notified
wait( e1 );
...
// wait until event e1 or event e2 has been notified
wait( e1 | e2 );
...
```

The `wait()` method can be called anywhere in the thread of execution of a thread process. When it is called, the specified events will temporarily overrule the sensitivity list, and the calling thread process will suspend. When one (or all) of the specified events is notified, the waiting thread process is resumed. The calling process is again sensitive to the sensitivity list.

When the `wait()` method is called without arguments, the calling thread process will suspend. When one of the events in the sensitivity list is notified, the waiting thread process is resumed. The static sensitivity of the calling thread process doesn't change. In SystemC 1.0, the `wait()` method without arguments is already supported.

In addition to events, it is also possible to wait for time. This can be used, for example, as a timeout when waiting for one or more events.

The following forms of `wait()` are supported.

```
// wait on events in sensitivity list (SystemC 1.0).
wait();

// wait on event e1.
wait( e1 );

// wait on events e1, e2, or e3.
wait( e1 | e2 | e3 );

// wait on events e1, e2, and e3.
wait( e1 & e2 & e3 );

// wait for 200 ns.
wait( 200, SC_NS );

// wait on event e1, timeout after 200 ns.
wait( 200, SC_NS, e1 );

// wait on events e1, e2, or e3, timeout after 200 ns.
wait( 200, SC_NS, e1 | e2 | e3 );

// wait on events e1, e2, and e3, timeout after 200 ns.
wait( 200, SC_NS, e1 & e2 & e3 );

sc_time t( 200, SC_NS );
```

```
// wait for 200 ns.
wait( t );

// wait on event e1, timeout after 200 ns.
wait( t, e1 );

// wait on events e1, e2, or e3, timeout after 200 ns.
wait( t, e1 | e2 | e3 );

// wait on events e1, e2, and e3, timeout after 200 ns.
wait( t, e1 & e2 & e3 );

// wait for 200 clock cycles, SC_CTHREAD only (SystemC 1.0).
wait( 200 );

// wait one delta cycle.
wait( 0, SC_NS );

// wait one delta cycle.
wait( SC_ZERO_TIME );
```

The semantics of the `wait()` method with one or more event arguments, is that the method returns (i.e., thread of execution is resumed) either when at least one of the events is notified, or when all events are notified. The former is specified with the `|`-operator between event arguments, the latter is specified with the `&`-operator between event arguments. A mixture of `|`-operators and `&`-operators is not supported in SystemC 2.0.

5.2 NEXT_TRIGGER() METHOD

The `wait()` method can only be used with `SC_THREAD` processes. The equivalent function for `SC_METHOD` processes is the `next_trigger()` method. The `next_trigger()` method takes the same arguments as the `wait()` method. The differences between `next_trigger()` and `wait()` are as follows.

- The `next_trigger()` method returns immediately, without passing control to another process.
- Multiple `next_trigger()` calls are allowed in one activation of an `SC_METHOD` process. The last `next_trigger()` call determines the (dynamic) sensitivity for the next activation.

After this high level view on dynamic and static sensitivity, the next sections will go into some more details regarding the event type that is needed for supporting dynamic sensitivity, and the SystemC scheduler.

5.3 EVENT TYPE

SystemC 1.0 provides a fixed set of channels and corresponding events. To support user-defined channel types, i.e., an extendable set of channel types, the set of events must also be extendable. To that purpose, we introduce the event type **`sc_event`**. The implementation and the behavior of this type in SystemC 2.0 are described next.

The event type `sc_event` provides the following functionality.

- Constructor – An event object can be created by calling the constructor without any arguments. For example,

```
sc_event my_event;
```

- Notify – An event can be notified by calling the (non-const) `notify()` method of the event object. For example,

```
my_event.notify();                // notify immediately
my_event.notify( SC_ZERO_TIME );  // notify next delta cycle
my_event.notify( 10, SC_NS );     // notify in 10 ns
sc_time t( 10, SC_NS );
my_event.notify( t );             // same
```

  In addition, functions are provided allowing a functional notation for notifying events. For example,

```
notify( my_event );                // notify immediately
notify( SC_ZERO_TIME, my_event );  // notify next delta cycle
notify( 10, SC_NS, my_event );     // notify in 10 ns
sc_time t( 10, SC_NS );
notify( t, my_event );             // same
```

- Cancel – An event notification can be cancelled by calling the (non-const) `cancel()` method of the event object. For example,

```
my_event.cancel();          // cancel a delayed notification
```

- The copy constructor and assignment operator of the event type are disabled.

A channel can construct any number of event objects – one for each type of event it can generate. A channel can notify an event by calling one of the notify methods of the event object. However, creation and notification of events is not restricted to channels.

For a pseudo code description of the event type, see Appendix A.

5.4 SYSTEMC SCHEDULER

The scheduler's task is to determine the order of execution of processes within the design based on the event sensitivity of the processes and the event notifications which occur.

The SystemC scheduler has support for both software and hardware-oriented modeling.

Similar to VHDL and Verilog, the SystemC scheduler supports delta cycles. A delta cycle is comprised of separate evaluate and update phases, and multiple delta cycles may occur at a particular simulated time. Delta cycles are useful for modeling fully-distributed, time-synchronized computation as found for example in RTL hardware. In SystemC, using `notify()` with a zero time argument causes the event to be notified in the evaluate phase of the next delta cycle, while a call to `request_update()` causes the `update()` method to be called in the update phase of the current delta cycle. Using these facilities, channels which model the behavior of hardware signals can be constructed.

SystemC also supports timed event notifications. Timed event notifications are specified using `notify()` with a time argument. A timed notification causes the specified event to be notified at a specified time in the future. Timed notifications are found in both VHDL and Verilog and are also useful in modeling software systems.

Lastly, SystemC supports immediate event notifications, which are specified using `notify()` with no arguments. An immediate event notification causes processes which are sensitive to the event to be made immediately ready to run (i.e. ready to run in the current evaluate phase.) Immediate event notifications are useful for modeling software systems and operating systems, which lack the concept of delta cycles.

The following steps outline the execution of the SystemC scheduler. More detailed pseudo-code for the scheduler is included in Appendix A of this document.

1) *Initialization Phase* – Execute all processes (except `SC_CTHREAD`s) in an unspecified order.

2) *Evaluate Phase* – 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) If there are still processes ready to run, go to step 2.

4) *Update Phase* – Execute any pending calls to `update()` resulting from `request_update()` calls made in step 2.

5) If there are pending delayed notifications, determine which processes are ready to run due to the delayed notifications and go to step 2.

6) If there are no more timed notifications, simulation is finished.

7) Advance the current simulation time to the earliest pending timed notification.

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

5.5 SYSTEMC EXECUTION MODEL

The SystemC simulation kernel relies on a co-routine execution model. Processes have independent threads of execution which have their own execution stack. Transfer of control from one thread of execution to another always happens at precisely identified points: threads can only suspend and resume execution when they call `wait()` (or, equivalently, when `SC_METHOD` processes return control to the simulator). The SystemC simulation kernel will never pre-empt execution of a thread as an RTOS might – instead, an executing thread must always yield execution by calling `wait()`.

Because transfer of control between threads only happens when a thread calls `wait()`, SystemC models can be written without concern that a thread may be pre-empted involuntarily. Specifically, the code within a thread delimited by two `wait()` statements can safely assume that

no other threads have modified any variables which are also accessible to other threads. (This makes it much easier to write high level models for channels than would be the case if pre-emption were an issue.)

Like Verilog and VHDL, SystemC models the execution of code within a thread between two `wait()` statements as happening instantaneously. Simulated time can only advance once a `wait()` statement has been called.

The execution model described above does not prevent SystemC from being used to model pre-emptive software systems. In order to model a pre-emptive software system, the expected execution delay of the target architecture must be incorporated into the SystemC model. For example, if the inner body of a loop takes 1 millisecond to execute in a target architecture, then a SystemC model might be written as:

```
while( ... ) {
    ...  // code for inner body of loop
    wait( 1, SC_MS );
}
```

In this case, the execution delay of the loop is accounted for, and if other threads are ready to run in the design at the same time, it's possible that execution of the loop body might be pre-empted at the point where it calls `wait()`.

5.6 NON-DETERMINISM IN SYSTEMC

Non-determinism is always a factor to consider when specifying and implementing concurrent systems. Often concurrent systems may have locally non-deterministic properties that are intentional, while still exhibiting deterministic global behavior. For example, at a very low level, individual packets transmitted over the internet may take unpredictable routes to reach their destination or may be lost altogether, but at a higher level reliable connections can be established over the internet. As another example, within a dataflow or process network model of computation, the order of execution of particular processes may be non-deterministic, but the overall result of the execution of such systems will always be deterministic.

Of course, it is also possible that local non-deterministic behavior may result in undesirable global non-deterministic behavior. A good example of this is when your PC boots fine one day but crashes the next day. A more specific example might be two threads which are scheduled to run at the same time and which both attempt to assign a global variable to a different value. In this case it is unpredictable what value the variable will have after the threads execute.

Just as in real software systems, non-determinism can be present in SystemC designs. In some cases this may be intentional because it is a property of the system that is being specified and modeled, while in other cases it may be undesirable because it represents a flaw in the design.

At a fundamental level, non-determinism in SystemC is introduced because the order of thread execution within a particular simulation phase (or part of a simulation delta cycle) is unspecified (or is "non-deterministic"). In a properly designed system, this aspect of SystemC will not affect the overall behavior of the system. But in an improperly designed system, it is possible that it might result in undesirable effects: the system might behave unpredictably from

one simulation run to the next (assuming the design or the stimulus has changed), or, even worse, an implementation of the system derived from the specification might not work properly.

There are several things designers can do to control non-determinism in SystemC designs.

- First, they can use channels such as hardware signals (`sc_signal`, etc) and fifos, which always result in globally deterministic behavior.

- Second, they should be aware that the order of thread execution within a particular simulation phase is unspecified.

- Third, they can use command line options to the SystemC simulator to randomize the order of execution of threads within each simulation phase. This feature is useful for detecting design flaws resulting from inadequate synchronization within design specifications.

As noted above, the order in which the scheduler selects threads to run within each simulation phase is unspecified and implementation-dependent. However, when the same design is simulated multiple times using the same stimulus and the same version of the simulator, the thread ordering between different runs will not vary.

6. INTERFACES, PORTS, AND CHANNELS

The necessary building blocks for process synchronization and communication refinement are (user-defined) interfaces, ports, and channels. An interface defines a set of methods, but does not implement these methods. It is a pure functional object without any data in order not to anticipate implementation details. A channel implements one or more interfaces. A port enables a module, and hence its processes, accessing a channel's interface. A port is defined in terms of an interface type, which means that the port can be used only with channels that implement that interface type.

With channels, we make a distinction between so-called *primitive channels* and *hierarchical channels*. Primitive channels do not exhibit any visible structure, do not contain processes, and cannot (directly) access other primitive channels. Hierarchical channels, on the other hand, are modules, which means they can have structure, they can contain other modules and processes, and they can (directly) access other channels.

The use of interfaces enables a very powerful scheme called *interface-method-call* (IMC). IMC refers to a process calling an interface method of a channel. The interface method is implemented in the channel, but it is executed in the context of the caller (the process). An example of an interface method is a blocking read method of a FIFO. When calling this interface method, the caller (process) can be suspended if there is not enough data available.

In the following chapters, the individual building blocks are addressed.

7. INTERFACES

An interface provides a set of method declarations, but provides no method implementations and no data fields. Interfaces are used to define sets of methods that channels must implement. Ports are connected to channels through interfaces. A port that is connected to a channel through an interface sees only those channel methods that are defined by the interface. A port is not able to access any other method or data field in the channel. SystemC 2.0 allows users to define their own interfaces.

7.1 INTERFACE EXAMPLES

Below, the implementations of three example interfaces are given. The first example interface defines a read interface. The second example interface defines a write interface. Both the read interface and the write interface derive from interface base class `sc_interface`, which is described in the next section. The third example interface defines a read/write interface by deriving from the read interface and write interface. Note that all interface methods are pure virtual methods, i.e., they don't have an implementation.

Example 7-1: A read, write, and read/write interface.

```
// ----------------------------------------
//  An example read interface: sc_read_if
//  this interface provides a 'read' method
// ----------------------------------------

template <class T>
class sc_read_if
: virtual public sc_interface
{
public:

    // interface methods
    virtual const T& read() const = 0;
};


// ----------------------------------------
//  An example write interface: sc_write_if
//  this interface provides a 'write' method
// ----------------------------------------

template <class T>
class sc_write_if
: virtual public sc_interface
{
public:

    // interface methods
    virtual void write( const T& ) = 0;
};



// ------------------------------------------------
```

```
//  An example read/write interface: sc_read_write_if
// ------------------------------------------------

template <class T>
class sc_read_write_if
: public sc_read_if<T>,
  public sc_write_if<T>
{};
```

The read interface provides a `read()` method. The write interface provides a `write()` method. The read/write interface is derived from the read interface and the write interface. This allows, for example, connecting a port with a read interface to a channel with a read/write interface, but not vice versa.

7.2 INTERFACE BASE CLASS

All interfaces are (directly or indirectly) derived from base class `sc_interface`. This class defines a method `register_port()`, which can be used by channels to do static design rule checking when binding ports to channels (see Section 8.2 and Section 9.3). The arguments to this method are (i) a reference to the port, and (ii) the type name of the interface that the port expects. The default behavior of this method is to do nothing. The interface base class also defines a method `default_event()`, which can be used by channels to return the default event for static sensitivity. The default behavior of this method is to return a reference to an event that is never notified. The implementation of the `sc_interface` base class is depicted below (without details).

Figure 7-1: Pseudo code implementation of the interface base class.

```
class sc_interface
{
public:

    // register a port with this interface (does nothing by default)
    virtual void register_port( sc_port_base&, const char* ) {}

    // get the default event
    virtual const sc_event& default_event() const;

    // destructor (does nothing)
    virtual ~sc_interface() {}
};
```

7.3 LAYERED COMMUNICATION EXAMPLE

The concept of interfaces is very useful when we want to model layered communication. At present, this kind of communication is implemented, e.g., in the On Chip Bus (OCB) defined by VSIA. In such a communication model, there are several layers of abstraction. For instance, in the OCB backbone there are three levels: transaction, packet, and cell. Using interfaces, we can define three types as shown in Example 7-2. With this approach, we can define in an optimum way the information that must be considered at each level of abstraction. For instance, the

information carried by "EOP[5]" is meaningful at packet level, but is not visible at transaction level. Hence, the user cannot specify this information when he is using the `OCB_trans_if` interface.

Example 7-2: Layered interfaces for the OCB backbone.

```
// pseudo code; some parts are not shown

// --------------------------------
//  INTERFACE : OCB_cell_if
// --------------------------------

template <class T>
class OCB_cell_if
: virtual public sc_interface
{
public:

    // interface methods
    ...
    virtual void set_eom( bool ) = 0;
    virtual bool get_eom() = 0;
    virtual void set_eop( bool ) = 0;
    virtual bool get_eop() = 0;
};


// ---------------------------------
//  INTERFACE : OCB_packet_if
// ---------------------------------

template <class T>
class OCB_packet_if
: public OCB_cell_if
{
public:

    // interface methods
    ...
};


// --------------------------------
//  INTERFACE : OCB_trans_if
// --------------------------------

template <class T>
class OCB_trans_if
: public OCB_packet_if
{
public:

    // interface methods
    ...
    virtual void set_eom( bool ) = 0;
```

[5] Denotes the end of the packet that has been transmitted.

```
    virtual bool get_eom() = 0;

private:

    // not to be defined
    virtual void set_eop( bool );
    virtual bool get_eop();
};
```

Looking at the implementation, all necessary methods are defined pure virtual inside the base class, `OCB_cell_if`. These methods are accessible both by `OCB_packet_if` and `OCB_trans_if`. Since some of the methods are meaningless at certain levels of abstraction, it is necessary to declare them private. For instance, the methods `get_eop()` and `send_eop()` are declared private in `OCB_trans_if`.

The layered approach, modeled in SystemC by using interfaces, allows also interconnecting a module at different levels of abstraction. In this way, we can attach either modules that need a very close control around the clock (i.e., at cell level), or modules that work at transaction level. Hence, the concept of interfaces provides all the necessary freedom to the user to model its intellectual property (IP) at different levels of abstraction.

8. PORTS

A port is an object through which a module, and hence its processes, can access a channel's interface. But modules can also access a channel's interface directly.

In SystemC 1.0, we have three basic port types: `sc_in<T>`, `sc_out<T>`, and `sc_inout<T>`. They are all derived from the base class `sc_port`. Each of these port types provides a set of interface methods, such as `read()` and `write()`. What these methods basically do is to call the corresponding interface method of the attached channel.

With other channel types, this simple scheme has to be extended, because the interfaces assumed by `sc_in<T>`, `sc_out<T>`, and `sc_inout<T>`, are not sufficient for all channel types. Some channel types may require additional or altogether different interface methods. In general, different channels implement different interfaces. Hence, we need different types of ports in order to access these interfaces. Different interfaces can be created by refining predefined interface types, or by inheriting directly from `sc_interface`. See Chapter 7 for details. Specialized ports can be created by refining port base class `sc_port` or one of the predefined port types.

It is advisable to separate function and communication as much as possible in order to increase the reusability of components. Therefore, a good design style is to always select the "minimal" port type[6] that offers the required interface methods. Selecting the "minimal" port type translates directly into selecting the "minimal" interface type for a port. Using highly specialized ports and interfaces will limit the number of different channel types that a module can be connected to.

Of course, this separation is not always feasible. The more tightly coupled functionality and communication are, the more likely it is that specialized port types are required. Cases in which the use of specialized ports is advisable or mandatory are those where the access methods of the basic port classes are not sufficient. A few examples are:

- Addresses are used in addition to data, e.g., bus interface.
- One needs additional information on the channel's status, e.g., the number of samples available in a FIFO/LIFO.
- One needs higher forms of sensitivity such as `wait_for_request()` (see Section 9.2).

The concept of interfaces, ports, and channels bears strong similarity to the notion of *interfaces* and *channels* used in SpecC. In fact, a port type assumes a certain interface. A channel cannot be connected to a port if it doesn't implement the port's interface.

[6] A "minimal" port type is a port type that requires the most general possible interface.

The question arises whether port objects are needed at all – one could argue that it would be sufficient to only have the notion of interfaces being implemented by channels. Basically, port objects serve a dual purpose. Firstly, they allow for the implementation and enforcement of (static) design rule checks. Secondly, they provide objects that can be attached attributes such as names or priorities.

8.1 ATTACHING MULTIPLE INTERFACES

A SystemC 1.0 port can be connected to one and only one channel. SystemC 2.0 allows for connecting a port to multiple channels *implementing the same interface*. We will refer to this as the *multi-port* capability. This port behaves much the same as the SystemC 1.0 port, except that more than one channel can be connected to it, and the subscript operator must be used to access a channel (for the first channel bound to this port the subscript operator does not have to be used).

Consider the following example, where multiple memory modules (channels) are attached to a bus module (channel).

Example 8-1: Attaching multiple memories to a bus using one port.

```
// pseudo code; some parts are not shown

// ----------------------------
//  INTERFACE : simple_bus_if
// ----------------------------

class simple_bus_if
: virtual public sc_interface
{
public:

    // methods return false if address out of range
    virtual bool read_data( unsigned address, int& data ) = 0;
    virtual bool write_data( unsigned address, int data ) = 0;
};


// ----------------------------
//  INTERFACE : simple_mem_if
// ----------------------------

class simple_mem_if
: public simple_bus_if
{
public:

    // methods to determine address range
    virtual unsigned start_address() const = 0;
    virtual unsigned end_address() const = 0;
};


// ----------------------
//  CHANNEL : simple_mem
// ----------------------

class simple_mem
```

```
: public sc_module,
  public simple_mem_if
{
public:

    ...

    // interface methods

    virtual bool read_data( unsigned address, int& data );
    virtual bool write_data( unsigned address, int data );

    virtual unsigned start_address() const;
    virtual unsigned end_address() const;

    ...
};


// --------------------------
//  CHANNEL : simple_bus
// --------------------------

class simple_bus
: public sc_module,
  public simple_bus_if
{
public:

    // a port to connect memories to (maximum 10 in this case)
    sc_port<simple_mem_if,10> mem_port;

    // interface methods

    virtual bool read_data( unsigned address, int& data )
    {
        // inefficient, but illustrates the use model
        for( int i = 0; i < mem_port.size(); i ++ )
            if( ( address >= mem_port[i]->start_address() ) &&
                ( address <= mem_port[i]->end_address() ) )
                return mem_port[i]->read_data( address, data );
        return false;
    }

    virtual bool write_data( unsigned address, int data );

    ...
};
```

The clear advantage in the above example is that multiple memories can be attached to the bus, without having to instantiate a port for each memory. Hence, we don’t have to change the bus code when we use it in designs where the number of memories is not known up front, or where the number of memories can differ.

8.2 PORT BASE CLASS

A pseudo-code definition of the port base class `sc_port<IF,N>` is given below. Note that this port type is a template class that is parameterized over an interface type (`IF`) and the maximum number of interfaces that can be connected to it (`N`). The default value for `N` is one.

Figure 8-1: Pseudo code definition of the port base class.

```
// ------------------------------
//  The port base class: sc_port
// ------------------------------

template <class IF, int N = 1>
class sc_port
: public sc_port_b<IF>
{
public:

    // default constructor
    sc_port();

    // bind an interface to this port
    void operator () ( IF& interface );

    // bind a (parent module's) port to this port
    void operator () ( sc_port_b<IF>& port );

    // access to FIRST interface

    // allow to call methods provided by the first interface
    IF* operator -> ();
    const IF* operator -> () const;

    // access to ALL interfaces

    // number of connected interfaces
    int size() const;

    // allow to call methods provided by the specified interface
    IF* operator [] ( int index );
    const IF* operator [] ( int index ) const;

    ... // rest not shown

protected:

    vector<IF*> m_interface_vec; // all interfaces

    // the first interface this port is connected to
    // (maintain extra pointer for more efficient access)
    IF* m_interface;

    ... // rest not shown
};
```

Class `sc_port_b<IF>` is derived from class `sc_port_base`.

A port of a module can be connected to

- zero or more channels at the same level of hierarchy,
- zero or more ports of its parent module,
- but at least one interface or port.

Because of this, an `sc_port` keeps track of all channels, parent ports, and child ports it is connected to. When a port is connected to a port of its parent module, the port stores the parent port, and then it will register itself with the parent port. This enables the parent port to store the registered port as a child port. When a port is connected to a channel, the port will pass the channel down to its child ports, such that during simulation these ports can access the channel directly. This is a recursive process. After elaboration (i.e. just before the simulation starts), each port must check that it doesn't have parent ports that are not connected to any channel.

In the most common case, where the maximum number of interfaces is one, an `sc_port` is connected to one channel or one parent port only. The port can still have zero or more child ports.

In any case, it will be very difficult for static design rule checks to determine the actual number of drivers of a channel. Dynamic design rule checks are needed to deal with this problem. Static and dynamic design rule checking are explained in Section 9.3.

The port base class `sc_port` allows accessing a channel's interface methods by using `operator ->` or `operator []`. With `input` an input port of a process, and `read()` an interface method of the attached channel, one can write, for example:

```
a = input->read();    // read from the first (or only) channel of input
b = input[2]->read(); // read from the third channel of input
```

Specialized ports can be defined, which provide shortcuts to some of the channel's interface methods. In the next section, some examples are given.

8.3 PORT EXAMPLES

Below, the implementations of three example ports are given. These ports use the example interfaces described in Section 7.1.

Example 8-2: A read, write, and read/write port.

```
// ------------------------------------
//  An example read port: sc_read_port
// ------------------------------------

template <class T>
class sc_read_port
: public sc_port<sc_read_if<T> >
{
public:

    // a convenience function for reading a value
    operator const T&() const
        { return (*this)->read(); }
};


// -------------------------------------
//  An example write port: sc_write_port
// -------------------------------------

template <class T>
class sc_write_port
: public sc_port<sc_write_if<T> >
{
public:

    // a convenience function for writing a value
    sc_write_port<T>& operator = ( const T& elem )
        { (*this)->write( elem ); return *this; }
};


// ------------------------------------------------
//  An example read/write port: sc_read_write_port
// ------------------------------------------------

template <class T>
class sc_read_write_port
: public sc_port<sc_read_write_if<T> >
{
public:

    // a convenience function for reading a value
    operator const T&() const
        { return (*this)->read(); }

    // a convenience function for writing a value
    sc_read_write_port<T>& operator = ( const T& elem )
        { (*this)->write( elem ); return *this; }
};
```

With the ports in Example 8-2, one could write channel access as follows.

```
SC_MODULE( my_module )
{
    sc_read_port<int>       in;
    sc_write_port<int>      out;
    sc_read_write_port<int> inout;

    void main_action() {
        int a({)]TJ 2.4 t ia({)]TJ 2.4 t i->_rean(a({)]TJ 20.16 Tc[(o)-600u{
```

Example 8-3: Port-less access for intra-module level communication.

```
// pseudo code

SC_MODULE( mod1 )
{
    // port(s)
    sc_in<int>  in;
    sc_out<int> out;

    sc_mutex mutex; // channel; used to protect common resource ‘in’

    int f( int );
    int g( int );

    // process(es)

    void thread1()
    {
        while( true ) {
            wait( 10, SC_NS );
            mutex.lock();               // direct access to channel
            wait( in.value_changed_event() );
            out = f( in );
            mutex.unlock();             // direct access to channel
        }
    }

    void thread2()
    {
        while( true ) {
            wait( 10, SC_NS );
            mutex.lock();               // direct access to channel
            wait( in.value_changed_event() );
            out = g( in );
            mutex.unlock();             // direct access to channel
        }
    }

    // constructor
    SC_CTOR( mod1 )
    {
        SC_THREAD( thread1 );
        SC_THREAD( thread2 );
    }

    ...
};
```

Note that direct access is not restricted to channel interfaces, i.e., other channel methods can also be called directly, e.g. for setting parameters and attributes.

9. CHANNELS

A channel implements one or more interfaces, and serves as a container for communication functionality. A channel is not necessarily a point-to-point connection; a channel may be connected to more than two modules. SystemC 2.0 allows users to create their own channel types enabling them, for instance, to develop efficient abstract bus models or to model proprietary communication protocols.

With channels, we make a distinction between so-called *primitive channels* and *hierarchical channels*. Primitive channels do not exhibit any visible structure, do not contain processes, and cannot (directly) access other primitive channels. Hierarchical channels, on the other hand, are modules, which means they can have structure, they can contain other modules and processes, and they can (directly) access other channels.

In this chapter, the common characteristics of primitive and hierarchical channels are described. The following two chapters focus on primitive channels and hierarchical channels, respectively.

The following example shows how ports are connected to a channel.

Example 9-1: Connecting ports to a channel.

```
// create a channel called ‘fifo’ with a buffer size of 10
sc_fifo<int> fifo( 10 );

// connect port ‘out’ from module ‘mod_a’ to the channel;
// this will register port ‘out’ with channel ‘fifo’
mod_a.out( fifo );

// connect port ‘in’ from module ‘mod_b’ to the channel;
// this will register port ‘in’ with channel ‘fifo’
mod_b.in( fifo );

// channel ‘fifo’ cannot have more than two ports
mod_c.in( fifo ); // Error
```

9.1 CHANNEL REFINEMENT

Refinement of channels is facilitated through the use of inheritance. All primitive channel types (directly or indirectly) inherit from base class `sc_prim_channel`; its API will be presented in Chapter 10. All hierarchical channel types (directly or indirectly) inherit from base class `sc_module`. Inheritance can be used to put common functionality into base classes and specialized functionality in derived classes. It can also be applied to refine or modify a communication protocol.

Different channel types may have different interfaces. While `sc_signal<T>` has a `read()` interface method without any function arguments, this will not be the case for a bus-like channel were we can expect an address to be required. For that reason, channels inherit (and implement) also one or more interfaces. Thus, inheritance is used to both modify the behavior of existing access methods as well as to add new, more specialized interfaces.

9.2 SENSITIVITY

Processes can be made sensitive to events on channels, which can be different for different channels. For example, if a process is sensitive to an `sc_signal<T>`, then it will be triggered if the value of the signal changes. One can envision a different channel type `sc_buffer<T>` that behaves just like `sc_signal<T>`, with the exception that a process gets triggered whenever a value is assigned to that channel – even if the new value is identical to the current one.

Additionally, channel types like `sc_fifo<T>` can internally use events to implement blocking interface methods. Dynamic sensitivity also allows for creating more specific forms of sensitivity. To illustrate this, think of a channel type that supports addresses (e.g. `generic_bus<Addr,Data,Arbiter>`). This channel type may offer interface methods such as:

```
void write( Data data, Addr address );
void read( Data& data, Addr address );
void wait_for_request( Addr start_address, Addr end_address );
```

Using `wait_for_request()`, a process might tell the channel to trigger it when there is a (read or write) request in the address range from `start_address` to `end_address`.

It is important to note that dynamic sensitivity allows for implementing such types of channels *without* having to change the underlying simulation engine. All the functionality that is required goes into the derived channel class.

9.3 DESIGN RULES

Different channel types bring along different *design rules* (cf. Table 9-1). For instance, a FIFO channel may require that at most two ports are connected to it, one of which must be an input port, the other one an output port. One must not attach a bi-directional port to a FIFO.[8] Other channel types can have different design rules. A channel of type `sc_signal<T>` for instance has a *"not more than one driver"* rule (it is a 1-to-many connection).

Table 9-1: Design rules for some channel types.

`sc_signal<T>`	• No more than one driver, i.e., at most one output (`sc_out<T>`) or bi-directional port (`sc_inout<T>`) connected. • Arbitrary number of input ports (`sc_in<T>`) can be connected.
`sc_signal_rv<N>`	• Arbitrary number of input, output, and bi-directional ports can be connected.

[8] Note that in general channels are many-to-many connections supporting bi-directional ports.

sc_fifo<T>	• At most one input port can be connected. • At most one output port can be connected. • No bi-directional ports.

SystemC 2.0 provides a mechanism that allows channels to enforce their respective design rules. When a port is attached to a channel, the (non-pure) virtual method `register_port()` of the corresponding channel's interface gets called. Overriding this method enables the channel to check whether this connection is legal. The default implementation is empty, that is, it does not carry out any checks. Figure 9-1 shows a pseudo-code implementation for the case of a FIFO channel.

Figure 9-1: Pseudo code implementation of design rule check (FIFO channel).

```
template <class T>
void sc_fifo<T>::register_port( sc_port_base& port,
                                const char* if_typename )
{
    sc_string nm( if_typename );
    if( nm == typeid( inout_interface<T> ).name() ) {
        error( "cannot connect to a bi-directional port" );
    } else if( nm == typeid( in_interface<T> ).name() ) {
        if( no input port registered so far )
            input_port = port;
        else
            error( "cannot connect to multiple inputs" );
    } else { // nm == typeid( out_interface<T> ).name()
        if( no output port registered so far )
            output_port = port;
        else
            error( "cannot connect to multiple outputs" );
    }
}
```

The `register_port()` method supports *static* design rule checking, i.e., it can check how many ports are connected and what the interface types are that these ports require. These checks are done before the simulation starts. What the `register_port()` method cannot check is, for example, that there are two processes using the same port, hence, even with one port attached, a channel can have more than one driver. The same is true when a port is connected to one or more child ports. Is that port only used by the child ports (which are registered to the attached channel), or are there processes that make use of the port as well?

When static design rule checking does not suffice, we need *dynamic* design rule checking. Dynamic design rule checking happens after the simulation has started. An example of dynamic design rule checking is a channel that only allows one driver (as e.g. `sc_signal<T>`), which checks the process that is accessing its interface methods. It stores the process the first time, and afterwards checks whether the accessing process is the same as the stored process. If not, there is more than one driver, which produces a run-time error.

See Section 10.3 for an example of dynamic design rule checking in primitive channel `sc_signal<T>`.

9.4 CHANNEL ATTRIBUTES

Channel attributes can be used for a per-port configuration of the communication. An example of such a channel attribute is the priority of ports that are connected to a certain channel. When we use channel attributes for a per-port configuration, the port itself does not have to be aware of the attribute mechanism, i.e., it allows to keep function and communication separated.

Channel attributes are especially helpful when modules are connected to a bus. Attributes that can be used in this context include:

- Addresses (in case the module doesn't use specialized ports, where addresses are specified as arguments of the access methods)
- Addressing schemes (e.g. constant address vs. auto-increment)
- Connect module as master or slave or master/slave
- Priorities
- Buffer sizes

Consider the following example.

Let `mod` be an instance of a module and let `port` be a port of this module. We will find the following or similar code in the enclosing module:

```
// create a local channel
message_queue mq;
...
// connect the module port to the channel
mod.port( mq );
...
```

A channel attribute can now be specified, for example, as:

```
// specify a channel attribute
mq.priority( mod.port, 2 );
...
```

which sets the priority attribute for `mod.port` to 2.

For this to work, channels have to provide dedicated methods for reading and writing channel attributes. These methods may or may not be part of some interface. If so, modules that are connected to such a channel's interface through a port can also read and write such attributes.

In general, channel attributes are favorable compared to using specialized ports (where the per-port configuration information is stored in the specialized ports), because they better help to keep function and communication separated. But storing per-port configuration information with the ports seems more natural, and simplifies the channel's internal data structures. Case by case, a careful trade-off has to be made between using channel attributes and using specialized ports. See Section 10.6 for an example implementation of a message queue channel, which illustrates both ways of supporting per-port configuration of the communication.

10. PRIMITIVE CHANNELS

10.1 SYNCHRONIZATION

Just as simultaneous actions require a special treatment in the case of simple channels – such as signals – they also require special attention in the case of more specialized primitive channel types. Examples include simultaneous read and write operations in the case of a queue or simultaneous bus requests issued by several bus masters. They are handled in the same way: channel accesses can be decomposed into the following two steps.

1) The *evaluate* step – executed when an interface method of the channel is invoked (accessed via a port). This interface method is executed in the context of the calling process during the evaluate phase of a delta cycle.

2) The *update* step – if requested, executed in a separate context during the update phase of a delta cycle.

The update step is required in cases where:

- simultaneous actions[9] have to be serialized, or
- any form of arbitration or resolution[10] is required in order to make the channel access deterministic.

For performance reasons the update step should only be carried out if necessary. The approach pursued here is to give the channel the ability to request an update during the evaluate part (i.e. when an interface method of the channel is executing). Only if an update was requested the channel's update code will be executed. For example, when reading a signal or querying the number of samples available in a FIFO channel, the update step is not required.

10.2 PRIMITIVE CHANNEL BASE CLASS

The implementation of a new primitive channel class uses the set of methods defined in the abstract base class **`sc_prim_channel`**, and defined in one or more interface classes. Primitive channel base class `sc_prim_channel` provides support for the two-step evaluate-update scheme with the following methods.

Figure 10-1: Pseudo code implementation of the primitive channel base class.

```
class sc_prim_channel
{
protected:
```

[9] E.g. simultaneous reads and writes in case of a signal; we get a read-before-write sequence in this case.

[10] E.g. simultaneous bus requests requiring arbitration.

```
    // constructor with name
    sc_prim_channel( const char* name = 0 );

    // get the name
    const char* name() const;

    // request the update() method
    // to be executed during the update step
    void request_update();

    // the update method (does nothing by default)
    virtual void update() {}

    // destructor (does nothing by default)
    virtual ~sc_prim_channel() {}
};
```

Note that for reasons of deterministic behavior there are two restrictions:

- we cannot give a process, which is executed during the evaluate phase, direct access to the `request_update()` and `update()` methods provided by `sc_prim_channel`, and
- we cannot allow the `update()` method to access other channels or call `wait()`.

The evaluate-update scheme for primitive channels is a powerful tool enabling the implementation of a broad range[11] of communication and synchronization elements without requiring subsequent changes of the simulation kernel. It allows for an efficient implementation keeping the number of context switches and signal updates small. Even very specific channels can be implemented in a deterministic way.

However, even if an `update()` method is provided, a channel class can still show non-deterministic behavior. The creator of a new channel type has to follow the guidelines described above in order to ensure determinism.

It should also be noted that channels themselves can be complex IP blocks. Designing a new channel type in an efficient way is by no means trivial. We can certainly expect that the majority of SystemC users will not design their own channel types but rather create and integrate modules using channel types provided by either the standard SystemC distribution or by third parties.

In addition to methods supporting the evaluate-update scheme, the primitive channel base class provides a constructor with a name argument and a `name()` method. The former allows specifying the name of a primitive channel instance, much like module instances and channel instances such as `sc_signal<T>` in SystemC 1.0. The name of an instance can be obtained by calling the `name()` method.

[11] In fact, we cannot think of a communication scheme that one could not implement this way.

The following sections show pseudo-code implementations[12] of different primitive channels. Some of these channel types will be part of SystemC 2.0 (with potentially different implementations), while other channel types are shown only to illustrate certain principles.

10.3 SC_SIGNAL<T>

The `sc_signal<T>` primitive channel is part of SystemC 1.0, and is part of SystemC 2.0.

The "classical" signal type uses the evaluate-update scheme to ensure deterministic behavior in the case of simultaneous read and write actions. We maintain a current and new value as explained earlier. The `write()` method will submit an update request if the new value is different from the current value.

The `sc_signal<T>` channel demonstrates how to do dynamic design rule checking in addition to static design rule checking. During static design rule checking, the channel makes sure that only one writer port is attached. During dynamic design rule checking, the channel checks that there is only one driver process writing to the channel.

The `sc_signal<T>` channel implements the `sc_signal_inout_if<T>` interface. A simple example is given of how one can use the `sc_signal<T>` channel.

```
// pseudo code; some parts are not shown

// --------------------------------
//  INTERFACE : sc_signal_in_if<T>
// --------------------------------

template <class T>
class sc_signal_in_if
: virtual public sc_interface
{
public:

    // read the current value
    virtual const T& read() const = 0;
};


// -----------------------------------
//  INTERFACE : sc_signal_inout_if<T>
// -----------------------------------

template <class T>
class sc_signal_inout_if
: public sc_signal_in_if<T>
{
public:

    // write the new value
    virtual void write( const T& ) = 0;
};
```

[12] These "implementations" are by no means complete; they just focus on the relevant parts.

```
// ---------------------------------
//  PRIMITIVE CHANNEL : sc_signal<T>
// ---------------------------------

template <class T>
class sc_signal
: public sc_prim_channel,
  public sc_signal_inout_if<T>
{
public:

    // default constructor
    sc_signal() : m_output( 0 ), m_driver( 0 ) {}

    // register port: static design rule checking
    virtual void register_port( sc_port_base& port,
                                const char* if_typename )
    {
        sc_string nm( if_typename );
        if( nm == typeid( sc_signal_inout_if<T> ).name() ) {
            // a write or read/write port; only one can be connected
            if( m_output != 0 )
                error( “more than one write port\n” );
            m_output = &port;
        }
        // any number of read ports can be connected: no check needed
    }

    // interface methods

    virtual const T& read() const
        { return m_cur_val; }

    virtual const sc_event& default_event() const
        { return m_value_changed_event; }

    virtual void write( const T& value )
    {
        // dynamic design rule checking
        if( current_process() != m_driver ) {
            if( m_driver == 0 )
                m_driver = current_process();
            else
                error( “more than one driver\n” );
        }
        // do the write
        m_new_val = value;
        if( m_new_val != m_cur_val )
            request_update();
    }

protected:

    virtual void update()
    {
        if( !( m_new_val == m_cur_val ) ) {
            m_cur_val = m_new_val;
            m_value_changed_event.notify( SC_ZERO_TIME );
        }
```

```
    }

    sc_port_base* m_output; // for static design rule checking
    sc_process_b* m_driver; // for dynamic design rule checking

    T m_cur_val;
    T m_new_val;

    sc_event m_value_changed_event;
};


// ---------
//  EXAMPLE
// ---------

SC_MODULE( writer )
{
    // port(s)
    sc_out<int> out;
    sc_in_clk   clk;

    // process(es)
    void main_action()
    {
        int val = 0;
        while( true ) {
            wait();       // wait for a positive edge of the clock
            out = val ++; // shortcut for: out->write( val ++ );
        }
    }

    SC_CTOR( writer )
    {
        SC_THREAD( main_action );
        sensitive_pos << clk;
    }
};

SC_MODULE( reader )
{
    // port(s)
    sc_in<int> in;

    // process(es)
    void main_action()
    {
        int val;
        while( true ) {
            wait();   // wait for a change of value on in
            val = in; // shortcut for: val = in->read();
            cout << val << endl;
        }
    }

    SC_CTOR( reader )
    {
        SC_THREAD( main_action );
        sensitive << in; // shortcut for: sensitive << in.default_event()
    }
};
```

```
int sc_main( int, char*[] )
{
    // declare channel(s)
    sc_signal<int> sig;

    // create clock(s)
    sc_clock clock( “clock”, 10, SC_NS );

    // instantiate block(s) and connect to channel(s) and clock(s)
    writer w( “writer” );
    reader r( “reader” );

    w.out( sig );
    r.in( sig );

    w.clk( clock );

    // run the simulation for a while
    sc_start( 1000, SC_NS );
    return 0;
}
```

10.4 SC_FIFO<T>

The sc_fifo<T> primitive channel is not part of SystemC 1.0, but is part of SystemC 2.0.

This FIFO channel comes with a number of methods. Basically, we find both blocking and non-blocking I/O as well as some functions to query the state of the FIFO. We use the evaluate-update scheme in order to ensure deterministic behavior. Here we also find an example of dynamic sensitivity; using the blocking I/O interface methods, the user is not required to explicitly declare the calling process to be sensitive to the FIFO.

The sc_fifo<T> primitive channel implements the sc_fifo_in_if<T> interface and the sc_fifo_out_if<T> interface, which are given first. A simple example is given of how one can use the sc_fifo<T> channel.

```
// pseudo code; some parts are not shown

// ------------------------------
//  INTERFACE : sc_fifo_in_if<T>
// ------------------------------

template <class T>
class sc_fifo_in_if
: virtual public sc_interface
{
public:

    // blocking read
    virtual void read( T& ) = 0;

    // non-blocking read
    virtual bool nb_read( T& ) = 0;

    // get the number of available samples
    virtual int num_available() const = 0;
```

```
};


// -------------------------------
//  INTERFACE : sc_fifo_out_if<T>
// -------------------------------

template <class T>
class sc_fifo_out_if
: virtual public sc_interface
{
public:

    // blocking write
    virtual void write( const T& ) = 0;

    // non-blocking write
    virtual bool nb_write( const T& ) = 0;

    // get the number of free spaces
    virtual int num_free() const = 0;
};


// --------------------------------
//  PRIMITIVE CHANNEL : sc_fifo<T>
// --------------------------------

template <class T>
class sc_fifo
: public sc_prim_channel,
  public sc_fifo_in_if<T>,
  public sc_fifo_out_if<T>
{
public:

    // constructor with size
    explicit sc_fifo( int size )
    : m_mem( size ), m_size( size ),
      m_num_readable( 0 ), m_num_read( 0 ), m_num_written( 0 ),
      m_reader( 0 ), m_writer( 0 )
        { assert( size > 0 ); }

    // register port
    virtual void register_port( sc_port_base& port,
                                const char* if_typename )
    {
        sc_string nm( if_typename );
        if( nm == typeid( sc_fifo_in_if<T> ).name() ) {
            // only one reader can be connected
            if( m_reader != 0 )
                error( “more than one read port\n” );
            m_reader = &port;
        } else { // nm == typeid( sc_fifo_out_if<T> ).name()
            // only one writer can be connected
            if( m_writer != 0 )
                error( “more than one write port\n” );
            m_writer = &port;
        }
    }
```

```
    // blocking read and write access

    virtual void read( T& data )
    {
        while( num_available() == 0 )
            wait( m_data_written_event );
        m_num_read ++;
        data = m_mem.pop_front();
        request_update();
    }

    virtual void write( const T& data )
    {
        while( num_free() == 0 )
            wait( m_data_read_event );
        m_num_written ++;
        m_mem.push_back( data );
        request_update();
    }

    // non-blocking read and write access
    // return 'true' on success

    virtual bool nb_read( T& data )
    {
        if( num_available() == 0 )
            return false;
        m_num_read ++;
        data = m_mem.pop_front();
        request_update();
        return true;
    }

    virtual bool nb_write( const T& data )
    {
        if( num_free() == 0 )
            return false;
        m_num_written ++;
        m_mem.push_back( data );
        request_update();
        return true;
    }

    // get the number of available samples and free spaces

    virtual int num_available() const
        { return m_num_readable - m_num_read; }

    virtual int num_free() const
        { return m_size - m_num_readable - m_num_written; }

protected:

    virtual void update()
    {
        if( m_num_written > 0 )
            notify( SC_ZERO_TIME, m_data_written_event );
        if( m_num_read > 0 )
            notify( SC_ZERO_TIME, m_data_read_event );

        m_num_readable = m_mem.length();
```

```
        m_num_read = m_num_written = 0;
    }

    fifo<T>  m_mem;          // the fifo memory
    unsigned m_size;         // size of the fifo

    unsigned m_num_readable; // #samples readable
    unsigned m_num_read;     // #samples read during this delta cycle
    unsigned m_num_written;  // #samples written during this delta cycle

    sc_event m_data_read_event;
    sc_event m_data_written_event;

    sc_port_base* m_reader;
    sc_port_base* m_writer;
};


// ---------
//  EXAMPLE
// ---------

SC_MODULE( writer )
{
    // port(s)
    sc_port<sc_fifo_out_if<int> > out;

    // process(es)
    void main_action()
    {
        int val = 0;
        while( true ) {
            wait( 10, SC_NS ); // wait for 10 ns
            for( int i = 0; i < 20; i ++ )
                out->write( val ++ ); // blocking write
        }
    }

    SC_CTOR( writer )
    {
        SC_THREAD( main_action );
    }
};

SC_MODULE( reader )
{
    // port(s)
    sc_port<sc_fifo_in_if<int> > in;

    // process(es)
    void main_action()
    {
        int val;
        while( true ) {
            wait( 10, SC_NS ); // wait for 10 ns
            for( int i = 0; i < 15; i ++ ) {
                in->read( val ); // blocking read
                cout << val << endl;
            }
            cout << “Available: “ << in->num_available() << endl;
        }
```

```
    }

    SC_CTOR( reader )
    {
        SC_THREAD( main_action );
    }
};

int sc_main( int, char*[] )
{
    // declare channel(s)
    sc_fifo<int> fifo( 10 );

    // instantiate block(s) and connect to channel(s)
    writer w( “writer” );
    reader r( “reader” );

    w.out( fifo );
    r.in( fifo );

    // run the simulation
    sc_start( -1 );
    return 0;
}
```

10.5 SC_MUTEX

The sc_mutex primitive channel is not part of SystemC 1.0, but is part of SystemC 2.0. In addition, another mutex primitive channel will be included in the SystemC 2.0 User’s Guide, one which

- does FIFO queuing of pending requests and that
- issues a warning if multiple requests are issued during the same delta cycle.

In this section, a simple mutex channel is given. If the mutex is not locked, it is given to the first process that issues a request. Only the process that locked the mutex is allowed to unlock it. Dynamic sensitivity is used to suspend (and later resume) processes that request locking the mutex when it is already locked.

The sc_mutex primitive channel can be used to model shared variables, either through inheritance (deriving a shared variable channel from the mutex channel) or by convention (agreeing that access to a certain variable is protected by a mutex).

```
// pseudo code; some parts are not shown

// -------------------------
//  INTERFACE : sc_mutex_if
// -------------------------

class sc_mutex_if
: virtual public sc_interface
{
public:

    // the classical operations: lock(), trylock() and unlock().
```

```
    // blocks until mutex could be locked
    virtual int lock() = 0;

    // returns -1 if mutex could not be locked
    virtual int trylock() = 0;

    // returns -1 if mutex was not locked by caller
    virtual int unlock() = 0;
};


// ------------------------------
//  PRIMITIVE CHANNEL : sc_mutex
// ------------------------------

class sc_mutex
: public sc_prim_channel,
  public sc_mutex_if
{
public:

    sc_mutex() : m_owner( 0 ) {}

    bool in_use() const
        { return ( m_owner != 0 ); }

    virtual int lock()
    {
        while( in_use() )
            wait( m_free );
        m_owner = current_process();
        return 0;
    }

    virtual int trylock()
    {
        if( in_use() )
            return -1;
        m_owner = current_process();
        return 0;
    }

    virtual int unlock()
    {
        if( m_owner != current_process() )
            return -1;
        m_owner = 0;
        notify( m_free );
        return 0;
    }

protected:

    sc_process_b* m_owner;
    sc_event      m_free;
};
```

10.6 SC_MQ

In this section, an example implementation of a message queue primitive channel is given. This example channel should illustrate the following.

- A channel which cannot be used without ports.
- Two ways to support per-port configuration information: (i) as channel attribute and (ii) in a specialized port.

The message queue channel uses two port attributes, i.e., a priority and a non-blocking flag. The priority attribute is stored in the specialized port, while the non-blocking flag attribute is stored in the channel. Both attributes can be changed dynamically, i.e., methods are provided in the interface and the specialized port to read and write the attributes.

The basic functionality of the message queue channel are the `send()` and `receive()` methods. In both methods, the priority port attribute is used as request priority, when multiple processes are waiting to (de)queue a message. In the `send()` method, the priority attribute is also used as message priority (for the underlying priority queue that stores the messages). The non-blocking flag port attribute is used to make the `send()` and `receive()` methods either blocking (default) or non-blocking.

The `sc_mq` primitive channel implements a single interface, `sc_mq_if`, which can be used with specialized port, `sc_mq_port`. Note that this channel can only be used with thread processes.

```
// pseudo code; some parts are not shown

// ----------------------
//  INTERFACE : sc_mq_if
// ----------------------

class sc_mq_if
: virtual public sc_interface
{
public:

    // send message
    virtual bool send( const sc_port_base& port,
                       const char*         msg,
                       unsigned            prio ) = 0;

    // receive message
    virtual bool receive( const sc_port_base& port,
                          char*&              msg,
                          unsigned            prio ) = 0;

    // set non-blocking flag
    virtual void non_blocking( const sc_port_base& port,
                               bool                nb_flag ) = 0;

    // get non-blocking flag
    virtual bool non_blocking( const sc_port_base& port ) const = 0;

    // get maximum number of messages
    virtual unsigned max_num_msgs() const = 0;
```

```
    // get maximum priority
    virtual unsigned prio_max() const = 0;

    // get number of messages currently queued
    virtual unsigned cur_num_msgs() const = 0;
};


// --------------------
//  PORT : sc_mq_port
// --------------------

class sc_mq_port
: public sc_port<sc_mq_if>
{
public:

    // default constructor
    sc_mq_port() : m_prio( 1U ) {}

    // send message
    bool send( const char* msg )
        { return (*this)->send( *this, msg, m_prio ); }

    // receive message
    bool receive( char*& msg )
        { return (*this)->receive( *this, msg, m_prio ); }

    // set non-blocking flag
    void non_blocking( bool nb_flag )
        { (*this)->non_blocking( *this, nb_flag ); }

    // get non-blocking flag
    bool non_blocking() const
        { return (*this)->non_blocking( *this ); }

    // set priority
    void priority( unsigned prio )
    {
        if( prio > 0U && prio <= (*this)->prio_max() )
            m_prio = prio;
    }

    // get priority
    unsigned priority() const
        { return m_prio; }

protected:

    unsigned m_prio;
};


// ----------------------------
//  PRIMITIVE CHANNEL : sc_mq
// ----------------------------

template <unsigned N, // maximum number of messages
          unsigned P> // maximum priority
class sc_mq
```

```
: public sc_prim_channel,
  public sc_mq_if
{
public:

    // default constructor
    sc_mq() {}

    // register port
    virtual void register_port( sc_port_base& port, const char* )
    {
        // any number of ports can be connected; no check needed

        // set non_blocking flag default to false
        m_non_blocking.insert( &port, false );
    }

    // interface methods

    // send message
    virtual bool send( const sc_port_base& port,
                       const char*         msg,
                       unsigned            prio )
    {
        if( ! space_available() || send_requests() )
            if( non_blocking( port ) )
                return false;
            else {
                sc_event send_ack;
                register_send_request( port, prio, send_ack );
                wait( send_ack );
            }
        m_msgs.insert( strdup( msg ), prio );
        request_update();
        return true;
    }

    // receive message
    virtual bool receive( const sc_port_base& port,
                          char*&              msg,
                          unsigned            prio )
    {
        if( ! data_available() || receive_requests() )
            if( non_blocking( port ) )
                return false;
            else {
                sc_event receive_ack;
                register_receive_request( port, prio, receive_ack );
                wait( receive_ack );
            }
        // extract the (oldest) message with the highest priority
        m_msgs.extract_max( msg );
        request_update();
        return true;
    }

    // set non-blocking flag
    virtual void non_blocking( const sc_port_base& port,
                               bool                nb_flag )
        { m_non_blocking[&port] = nb_flag; }
```

```
    // get non-blocking flag
    virtual bool non_blocking( const sc_port_base& port ) const
        { return m_non_blocking[&port]; }

    // get maximum number of messages
    virtual unsigned max_num_msgs() const
        { return N; }

    // get maximum priority
    virtual unsigned prio_max() const
        { return P; }

    // get number of messages currently queued
    virtual unsigned cur_num_msgs() const
        { return m_msgs.num_elements(); }

    // support methods

    bool space_available() const
        { return ( m_msgs.num_elements() < N ); }

    bool data_available() const
        { return ( m_msgs.num_elements() > 0U ); }

    void register_send_request( const sc_port_base& port,
                                unsigned prio,
                                sc_event& send_ack )
    {
        m_send_requests.insert( &port, prio );
        m_send_acks.insert( &port, &send_ack );
    }

    void register_receive_request( const sc_port_base& port,
                                   unsigned prio,
                                   sc_event& receive_ack )
    {
        m_receive_requests.insert( &port, prio );
        m_receive_acks.insert( &port, &receive_ack );
    }

    // are there any send requests?
    bool send_requests() const
        { return ( m_send_requests.num_elements() != 0U ); }

    // are there any receive requests?
    bool receive_requests() const
        { return ( m_receive_requests.num_elements() != 0U ); }

protected:

    virtual void update()
    {
        if( space_available() && send_requests() ) {
            const sc_port_base* port;
            // extract the (oldest) highest priority port
            m_send_requests.extract_max( port );
            // trigger corresponding event
            notify( SC_ZERO_TIME, *(m_send_acks.remove( port )) );
        }

        if( data_available() && receive_requests() ) {
```

```
            const sc_port_base* port;
            // extract the (oldest) highest priority port
            m_receive_requests.extract_max( port );
            // trigger corresponding event
            notify( SC_ZERO_TIME, *(m_receive_acks.remove( port )) );
        }
    }

    priority_queue<char*> m_msgs; // the message queue

    priority_queue<const sc_port_base*> m_send_requests;
    priority_queue<const sc_port_base*> m_receive_requests;

    map<const sc_port_base*, sc_event*> m_send_acks;
    map<const sc_port_base*, sc_event*> m_receive_acks;

    map<const sc_port_base*, bool> m_non_blocking;
};
```

Note: The `map` data type in the above code is an STL map, which implements an associative array. The two template arguments are the key type and value type, respectively.

10.7 RGPROTOCOL

Below we find an example of an RG-protocol (request, grant) channel. The `RGprotocol` channel is derived from channel base class `sc_prim_channel` and interface classes `RGprotocol_master_if` and `RGprotocol_slave_if`. The interface classes define both blocking and non-blocking I/O interface methods. This channel is also using the evaluate-update scheme to ensure deterministic behavior.

```
// pseudo code; some parts are not shown

struct RGtype
{
    bool REQUEST;
    bool GRANT;
    unsigned LEN;
    unsigned RLEN;
};


// ---------------------------------
//  INTERFACE : RGprotocol_master_if
// ---------------------------------

template <class T>
class RGprotocol_master_if
: virtual public sc_interface
{
public:

    // blocking write and read access
    // out port side
    virtual void write_master( const T&, const RGtype& ) = 0;
    virtual void read_master( T&, RGtype& ) = 0;

    // non-blocking read and write access
```

```
    // out port side
    virtual bool nb_write_master( const T&, const RGtype& ) = 0;
    virtual bool nb_read_master( T&, RGtype& ) = 0;
};


// ---------------------------------
//  INTERFACE : RGprotocol_slave_if
// ---------------------------------

template <class T>
class RGprotocol_slave_if
: virtual public sc_interface
{
public:

    // blocking write and read access
    // in port side
    virtual void read_slave( T&, RGtype& ) = 0;
    virtual void write_slave( const T&, const RGtype& ) = 0;

    // non-blocking write and read access
    // in port side
    virtual bool nb_read_slave( T&, RGtype& ) = 0;
    virtual bool nb_write_slave( const T&, const RGtype& ) = 0;
};


// ----------------------
//  CHANNEL : RGprotocol
// ----------------------

template <class T>
RGprotocol
: public sc_prim_channel,
  public RGprotocol_master_if<T>,
  public RGprotocol_slave_if<T>
{
public:

    // constructor with size
    RGprotocol( unsigned data_size )
    : m_size( data_size )
    {
        m_projected_signals.reset();
        m_current_signals.reset();
    }

    // register port
    virtual void register_port( sc_port_base& port,
                                const char* if_typename )
    {
        sc_string nm( if_typename );
        if( nm == typeid( RGprotocol_master_if<T> ).name() ) {
            // only one master can be connected
            assert( m_master == 0 );
            m_master = &port;
        } else { // nm == typeid( RGprotocol_slave_if<T> ).name()
            // only one slave can be connected
            assert( m_slave == 0 );
            m_slave = &port;
```

```
    }
}

// blocking write and read access
// out port side

virtual void write_master( const T& data,
                           const RGtype& signals )
{
    if( m_current_signals.REQUEST )
        wait( m_no_request_event );
    m_projected_signals.REQUEST = true;
    m_projected_signals.LEN = signals.LEN;
    m_projected_value = data;
    request_update();
}

virtual void read_master( T& data, RGtype& signals )
{
    if( ! m_current_signals.GRANT )
        wait( m_grant_event );
    data = m_current_value;
    signals.LEN = m_current_signals.RLEN;
    m_projected_signals.REQUEST = false;
    m_projected_signals.GRANT = false;
    request_update();
}

// non-blocking read and write access
// out port side
// returns 'true' on success

virtual bool nb_write_master( const T& data,
                              const RGtype& signals )
{
    if( m_current_signals.REQUEST )
        return false;
    m_projected_signals.REQUEST = true;
    m_projected_signals.LEN = signals.LEN;
    m_projected_value = data;
    request_update();
    return true;
}

virtual bool nb_read_master( T& data, RGtype& signals )
{
    if( ! m_current_signals.GRANT )
        return false;
    data = m_current_value;
    signals.LEN = m_current_signals.RLEN;
    m_projected_signals.REQUEST = false;
    m_projected_signals.GRANT = false;
    request_update();
    return true;
}

// blocking write and read access
// in port side

virtual void read_slave( T& data, RGtype& signals )
{
```

```
        if( ! m_current_signals.REQUEST )
            wait( m_request_event );
        signals.LEN = m_current_signals.LEN;
        data = m_current_value;
    }

    virtual void write_slave( const T& data,
                              const RGtype& signals )
    {
        if( ! m_current_signals.REQUEST )
            wait( m_request_event );
        m_projected_signals.GRANT = true;
        m_projected_signals.LEN = signals.LEN;
        m_projected_value = data;
        request_update();
    }

    // non-blocking write and read access
    // in port side

    virtual bool nb_read_slave( T& data, RGtype& signals )
    {
        if( ! m_current_signals.REQUEST )
            return false;
        signals.LEN = m_current_signals.LEN;
        data = m_current_value;
        return true;
    }

    virtual bool nb_write_slave( const T& data,
                                 const RGtype& signals )
    {
        if( ! m_current_signals.REQUEST )
            return false;
        m_projected_signals.GRANT = true;
        m_projected_signals.LEN = signals.LEN;
        m_projected_value = data;
        request_update();
        return true;
    }

protected:

    virtual void update()
    {
        m_current_signals = m_projected_signals;
        m_current_value = m_projected_value;

        if( m_current_signals.REQUEST )
            m_request_event.notify( SC_ZERO_TIME );
        else
            m_no_request_event.notify( SC_ZERO_TIME );

        if( m_current_signals.GRANT )
            m_grant_event.notify( SC_ZERO_TIME );
    }

    unsigned m_size; // size of the fifo

    sc_event m_no_request_event;
    sc_event m_request_event;
```

```
    sc_event m_grant_event;

    T      m_projected_value;
    T      m_current_value;
    RGtype m_projected_signals;
    RGtype m_current_signals;

    sc_port_base* m_master;
    sc_port_base* m_slave;
};
```

11. HIERARCHICAL CHANNELS

Until now, we have been mainly discussing *primitive channels*, i.e., channels without visible structure or processes, and no direct access to other channels. But for refining a communication protocol into an implementation, we can use so-called *hierarchical channels*. A hierarchical channel is a module, i.e., it can have structure, it can contain processes, and it can directly access other channels. As a channel, the module must also be derived from one or more interface classes.

Hierarchical channels are very useful to model the new generation of SoC communication infrastructures. For instance, OCB (On Chip Bus) is the present standard backbone from VSIA. The OCB backbone consists of several intelligent units, such as an Arbiter unit, a Control Programming unit, and a Decode unit. This ensures full scalability, IP reuse, and a rapid time to market for SoC. For modeling complex channels such as the OCB backbone, primitive channels are not well suited, because of the lack of processes and structure. For modeling this type of channels, hierarchical channels should be used.

11.1 GUIDELINES

Here are some guidelines on when to use primitive channels and when to use hierarchical channels.

Use primitive channels:

- When you need to use the request-update scheme.
- When channels are atomic and cannot reasonably by chopped into smaller pieces.
- When speed is absolutely crucial (using primitive channels we can often reduce the number of delta cycles).
- When it doesn't make any sense trying to build up a channel (such as a semaphore or a mutex) out of processes and other channels.

Use hierarchical channels:

- When channels are truly hierarchical and users would want to be able to explore the underlying structure.
- When channels contain processes.
- When channels contain other channels.

Both primitive and hierarchical channels can be refined. Refinement has to do with interfaces, not with whether a channel is primitive or hierarchical.

11.2 HIERARCHICAL CHANNEL EXAMPLE

In Figure 11-1, an example of a hierarchical channel (FIFO) is given. This hierarchical FIFO channel (`sc_fifo_clocked`) uses the same interface as the primitive `sc_fifo<T>` channel (see Section 10.4). The channel contains a module (`clocked_handshake_fifo`) that is clocked and uses a handshake protocol for reading and writing. At a positive clock edge, it will look for a read and write request. If there is such a request, it will start the corresponding handshake.

Figure 11-1: Refined FIFO as hierarchical channel.

```
// pseudo code

// ------------------------------------------------------------------------
//  MODULE : clocked_handshake_fifo
//  (this is not yet the channel but a clocked module used by the channel)
// ------------------------------------------------------------------------

template <class T>
class clocked_handshake_fifo
: public sc_module
{
public:

    // port(s)

    sc_in_clk    clk;

    sc_in<T>     write_data;
    sc_in<bool>  write_req;
    sc_out<bool> write_ack;
    sc_out<bool> write_enable;

    sc_out<T>    read_data;
    sc_in<bool>  read_req;
    sc_out<bool> read_ack;
    sc_out<bool> read_enable;

    // process(es)

    void write_fifo()
    {
        write_ack = false;
        write_enable = true;

        while( true ) {
            wait();
            if( write_req ) {
                assert( write_enable );
                do_write( write_data );
                write_ack = true;
                wait( write_req.default_event() );
                write_ack = false;
            }
        }
    }

    void read_fifo()
    {
        read_ack = false;
```

```
        read_enable = true;

        while( true ) {
            wait();
            if( read_req ) {
                assert( read_enable );
                do_read( read_data );
                read_ack = true;
                wait( read_req.default_event() );
                read_ack = false;
            }
        }
    }

    // constructor

    clocked_handshake_fifo( const char* name, unsigned size )
    : sc_module( name ),
      m_mem( size ), m_size( size )
    {
        assert( size > 0 );
        SC_THREAD( write_fifo );
        sensitive_pos << clk;
        SC_THREAD( read_fifo );
        sensitive_pos << clk;
    }

    // support method(s)

    void do_write( const T& data )
    {
        m_mem.push_back( data );
        if( m_mem.length() == m_size )
            write_enable = false;
        if( ! read_enable )
            read_enable = true;
    }

    void do_read( T& data )
    {
        data = m_mem.pop_front();
        if( m_mem.length() == 0 )
            read_enable = false;
        if( ! write_enable )
            write_enable = true;
    }

protected:

    fifo<T>  m_mem;  // the fifo memory
    unsigned m_size; // size of the fifo
};


// ----------------------------------------
//  HIERARCHICAL CHANNEL : sc_fifo_clocked
// ----------------------------------------

template <class T>
sc_fifo_clocked
: public sc_module,
```

```
  public sc_fifo_in_if<T>,
  public sc_fifo_out_if<T>
{
public:

    // port(s)

    sc_in_clk clk;

    // channel(s)

    sc_signal<T>    write_data;
    sc_signal<bool> write_req;
    sc_signal<bool> write_ack;
    sc_signal<bool> write_enable;

    sc_signal<T>    read_data;
    sc_signal<bool> read_req;
    sc_signal<bool> read_ack;
    sc_signal<bool> read_enable;

    // module(s)

    clocked_handshake_fifo<T>* chfifo;

    // constructor

    sc_fifo_clocked( const char* name, unsigned size )
    : sc_module( name ),
      m_size( size ), m_num_readable( 0 ),
      m_reader( 0 ), m_writer( 0 )
    {
        // create the clocked handshake fifo
        chfifo = new clocked_handshake_fifo<T>( size );

        // connect the clock
        chfifo->clk( clk );

        // connect the write & read channels
        chfifo->write_data( write_data );
        chfifo->write_req( write_req );
        chfifo->write_ack( write_ack );
        chfifo->write_enable( write_enable );

        chfifo->read_data( read_data );
        chfifo->read_req( read_req );
        chfifo->read_ack( read_ack );
        chfifo->read_enable( read_enable );
    }

    // interface methods

    // register port
    virtual void register_port( sc_port_base& port,
                                const char* if_typename )
    {
        sc_string nm( if_typename );
        if( nm == typeid( sc_fifo_read_if<T> ).name() ) {
            // only one reader can be connected
            if( m_reader != 0 )
                error( “more than one read port\n” );
```

```
            m_reader = &port;
        } else { // nm == typeid( sc_fifo_write_if<T> ).name()
            // only one writer can be connected
            if( m_writer != 0 )
                error( "more than one write port\n" );
            m_writer = &port;
        }
    }

    // blocking read and write access

    virtual void write( const T& data )
    {
        if( ! write_enable )
            wait( write_enable.default_event() );
        write_data = data;
        write_req = true;
        wait( write_ack.default_event() );
        write_req = false;
        m_num_readable ++;
    }

    virtual const T& read()
    {
        if( ! read_enable )
            wait( read_enable.default_event() );
        read_req = true;
        wait( read_ack.default_event() );
        data = read_data;
        read_req = false;
        m_num_readable --;
    }

    // non-blocking read and write access,
    // return 'true' on success

    virtual bool nb_write( const T& data );

    virtual bool nb_read( T& data );

    // request #samples available and #spaces free

    virtual unsigned num_available() const
        { return m_num_readable; }

    virtual unsigned num_free() const
        { return m_size - m_num_readable; }

protected:

    unsigned m_size;         // size of the fifo
    unsigned m_num_readable; // #samples readable

    sc_port_base* m_reader;
    sc_port_base* m_writer;
};
```

11.3 COMPOSITE CHANNELS

Interfaces and ports allow grouping channels into other channels. Consider the following example.

Example 11-1: Grouping channels.

```
class producer_if
: virtual public sc_interface
{
public:
    sc_read_write_if<int>& data() = 0;
    sc_read_write_if<bool>& data_valid() = 0;
};


class consumer_if
: virtual public sc_interface
{
public:
    sc_read_if<int>& data_out() = 0;
    sc_read_if<bool>& data_valid_out() = 0;
};


// group data and data_valid signals

class simple_data_channel
: public sc_module,
  public producer_if,
  public consumer_if
{
public:
    sc_signal<int> data_:
    sc_signal<bool> data_valid_;
    sc_read_write_if<int>& data() { return data_; }
    ...
};
```

Modules that are connected to a `simple_data_channel` via a `sc_port<producer_if>` can do

```
port->data().write( 42 );
port->data_valid().write( true );
```

Example 11-1 illustrates two important aspects. The first aspect is that grouping of channels results in so called *composite channels*. Composite channels have the advantage that one binding (of composite channel to port) can replace many bindings. The second aspect is that it is possible to *export lower-level interfaces*, i.e., the interfaces of the embedded channels.

The disadvantage of the latter aspect is that static design rule checking doesn't work anymore, because access to the lower-level interfaces is done port-less. In these cases, dynamic design rule checking should be used. Another disadvantage is that convenience short-cuts, which are normally defined by specialized ports (see Section 8.3), are not available.

11.4 TYPEDEFS

The distinction between hierarchical channels and modules is somewhat fuzzy, since both hierarchical channels and modules (directly or indirectly) derive from module base class `sc_module`, both can implement interfaces, and both can have ports. SystemC 2.0 does not enforce strict rules to separate hierarchical channels from behaviors (modules). This interpretation is left to the user. To allow the user showing his/her design intent, the following two typedefs will be provided.

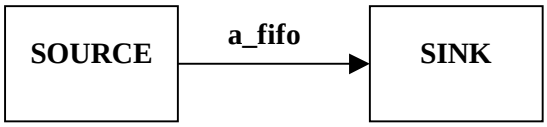
- `typedef sc_module sc_channel;`
- `typedef sc_module sc_behavior;`

As a consequence, the SystemC 0.9 `sc_channel<T>` type will become obsolete. This type is replaced by the `sc_fifo<T>` primitive channel (see Section 10.4).

12. COMMUNICATION REFINEMENT

The ability to design user-defined channels with arbitrary interfaces supports an *interface-based design style*. This design style allows for a simple and flexible communication refinement process, i.e., it enables the refinement of abstract communication channels into an actual implementation. The following example shall clarify this by presenting one possible way to achieve this.

Figure 12-1: Thread processes communicating via an sc_fifo channel.



In Figure 12-1, we see a simple example of two functional models communicating over a FIFO link (sc_fifo, cf. Section 10.4). The relevant portions of the source code are listed in Example 12-1. One can see that `read()` and `write(.)` methods provided by the interface of `sc_fifo` are used to initiate read and write transactions.

Example 12-1: Thread processes communicating via an sc_fifo channel.

```
SC_MODULE( source )
{
    // output port
    sc_port<sc_fifo_write_if<int> > output;

    // thread process
    void main_action()
    {
        while( true ) {
            ... // not shown
            output->write( data );
            ... // not shown
        }
    }

    // constructor
    SC_CTOR( source )
    {
        SC_THREAD( main_action );
    }
};

SC_MODULE( sink )
{
    // input port
    sc_port<sc_fifo_read_if<int> > input;

    // thread process
    void main_action()
    {
        while( true ) {
            ... // not shown
```

```
            data = input->read();
            ... // not shown
        }
    }

    // constructor
    SC_CTOR( sink )
    {
        SC_THREAD( main_action );
    }
};

int sc_main( int, char*[] )
{
    // instantiate channel
    sc_fifo<int> a_fifo( 10 ); // size is 10

    // instantiate SOURCE and SINK
    source SOURCE;
    sink SINK;

    // connect modules
    SOURCE.output( a_fifo );
    SINK.input( a_fifo );

    ... // rest not shown
}
```

The task at hand is now to refine the communication so that a HW implementation of a FIFO buffer is used. For the sake of simplicity we use a component with a simple handshaking interface (hw_fifo, see Example 12-2).

Example 12-2: FIFO with simple handshaking interface.

```
SC_MODULE( hw_fifo )
{
    // input: can accept at most one sample per clock cycle
    sc_in<int>   data_in;       // data
    sc_in<bool>  data_in_valid; // sender: data is valid
    sc_out<bool> data_in_ready; // FIFO: ready to accept input

    // output: can provide at most one sample per clock cycle
    sc_out<int>  data_out;          // data
    sc_out<bool> data_out_valid;    // FIFO: data is available
    sc_in<bool>  data_out_was_read; // receiver: data was read

    // clock
    sc_in<bool>  clock; // the, uh, clock

    ... // rest not shown
};
```

Now, we refine the communication in a stepwise process.

1) *Channel refinement*
   We replace `sc_fifo` with `hw_fifo`.

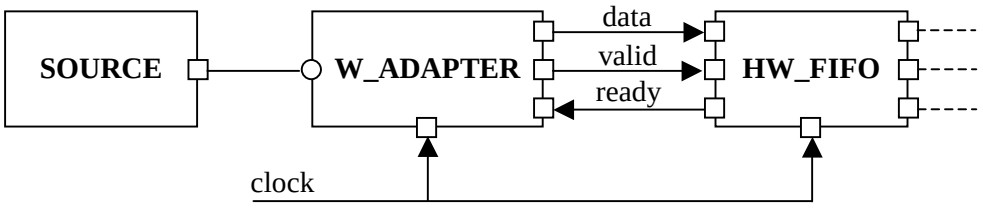
2) *Adapter insertion*
We insert "adapters" to convert the interfaces expected (used) by the `SOURCE` and `SINK` module instances into the HW interface provided by `hw_fifo`. Using an adapter to connect `SOURCE` to a `hw_fifo` instance is depicted in Figure 12-2. There a circle is used to indicate the interface provided by the adapter while the little square boxes represent ports.

3) *Validation*
Simulate the design. Timing – and thus the overall behavior – may have changed due to the use of `hw_fifo` instead of `sc_fifo` (at most one read/write transaction per clock cycle). Validate that the system still behaves satisfactory.

4) *Interface refinement*
Merge adapter and `SOURCE` module in order to make the abstract interface (still present between the `SOURCE` and the adapter) disappear. This effectively refines the interface of the `SOURCE` model from using an abstract transaction-based interface (`read()` and `write(.)` methods) into a pin-based HW interface. The same approach is pursued for `SINK`. For the sake of conciseness we focus on `SOURCE` for the rest of this chapter.

5) *Re-validation*
Re-simulate the design. The behavior should be the same as in step 3.

Figure 12-2: SOURCE connected via an adapter to HW_FIFO.



The two key steps are *adapter insertion* and *interface refinement*. The use of adapters is a flexible and intuitive approach (everyone who traveled abroad has used this approach at some point in time in order to get electric devices working). In an IP-centric environment it is to be expected that creators of refined communication channels will also provide adapters to standard interfaces such as the read/write transaction interface used by `sc_fifo`.

The use of adapters is made possible by SystemC's interface-based approach to communication. Modules do not specify that they require a specific type of *channel*. Instead, they specify what type of *interface* the channel must provide. This allows for replacing a channel with a different one that implements the same interface. For instance, the source module in Example 12-1 has a port of type `sc_port<sc_fifo_write_if<int> >`. This means, that it can be connected to any channel that implements `sc_fifo_write_if<int>`. This is exactly what `W_ADAPTER` in Figure 12-2 does.

Interface method calls (IMC) establish a caller-callee relationship between a channel (callee) and a module. The interface methods provided by the channel are executed in the thread of

execution of the calling process. This makes merging an adapter into the calling module simple – interface methods just have to be converted (e.g. simply via *copy&paste*) into methods of the module.

In the following we briefly carry out the steps 1, 2, and 4 as described above (we skip the validation steps).

Channel refinement

Here we simply replace the `sc_fifo` instance with an instance of `hw_fifo`. The code shown in Example 12-1 needs to be changed in the following way (only relevant part are shown).

Example 12-3: Channel refinement.

```
... // rest not shown
int sc_main( int, char*[] )
{
    // instantiate channel
    hw_fifo HW_FIFO;

    ... // rest not shown
}
```

Adapter insertion

The relevant portions of the adapter code are shown in Example 12-4. The basic idea is that the adapter is a *user-defined channel* that implements the interface required by the `source` module. It translates the transaction-oriented interface consisting of methods such as `write(.)` into an RTL pin interface that can then be connected to the refined FIFO implementation.

In order to instantiate and connect the adapter the `sc_main(.)`function needs to be changed too. This is shown in Example 12-5.

Example 12-4: Relevant portions of the adapter code.

```
class w_adapter
: public sc_module,
  public sc_fifo_write_if<int>
{
public:

    // ports
    sc_out<int>  data_port;
    sc_out<bool> valid_port;
    sc_in<bool>  ready_port;
    sc_in_clk    clock;

    // implement write(.) method (part of sc_fifo_write_if)
    virtual void write( const int& data )
    {
        // wait until fifo is ready
        while( ready_port == false )
            wait( clock.pos_edge() );
```

```
        // write data and assert valid signal
        data_port = data;
        valid_port = true;
        // wait for next clock cycle and reset valid signal
        wait( clock.pos_edge() );
        valid_port = false;
    }

    ... // rest not shown
};
```

Example 12-5: Instantiation of the adapter.

```
int sc_main( int, char*[] )
{
    // instantiate channel
    hw_fifo HW_FIFO;

    // instantiate SOURCE and SINK
    source SOURCE;
    sink SINK;

    // instantiate adapters
    w_adapter W_ADAPTER;
    ... // need a second adapter for SINK

    // connect SOURCE to W_ADAPTER
    SOURCE.output( W_ADAPTER );

    // connect W_ADAPTER to HW_FIFO

    sc_signal<int>  data_in;
    sc_signal<bool> data_in_valid;
    sc_signal<bool> data_in_ready;

    W_ADAPTER.data_port( data_in );
    W_ADAPTER.valid_port( data_in_valid );
    W_ADAPTER.ready_port( data_in_ready );

    HW_FIFO.data_in( data_in );
    HW_FIFO.data_in_valid( data_in_valid );
    HW_FIFO.data_in_ready( data_in_ready );

    ... // rest not shown
}
```

Interface refinement

During this step we merge the `W_ADAPTER` and `SOURCE`. We create a new module called `refined_source` for that purpose. The relevant portions of its code are shown in Example 12-6. The changed version of `sc_main(.)` can be seen in Example 12-7.

Example 12-6: Version of source with refined interface.

```
SC_MODULE( refined source )
{
    // output port
```

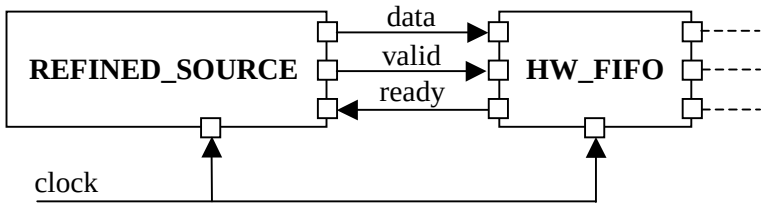
```
    // sc_port<sc_fifo_write_if<int> > output; // REMOVED

    // NEW: add ports of adapter (COPY&PASTE)
    sc_out<int>  data_port;
    sc_out<bool> valid_port;
    sc_in<bool>  ready_port;
    sc_in_clk    clock;

    // thread process
    void main_action()
    {
        while( true ) {
            ... // not shown
            // output->write( data );  // REPLACED BY NEXT LINE
            write( data );
            ... // not shown
        }
    }

    // NEW: copy&paste write(.) method from adapter (doesn't need
    //      to be virtual any longer)

    // implement write(.) method (part of sc_fifo_write_if)
    /* virtual */ void write( const int& data )
    {
        // wait until fifo is ready
        while( ready_port == false )
            wait( clock.pos_edge() );
        // write data and assert valid signal
        data_port = data;
        valid_port = true;
        // wait for next clock cycle and reset valid signal
        wait( clock.pos_edge() );
        valid_port = false;
    }

    // constructor
    SC_CTOR( refined_source )
    {
        SC_THREAD( main_action );
    }
};
```

Example 12-7: sc_main after interface refinement.

```
int sc_main( int, char*[] )
{
    // instantiate channel
    hw_fifo HW_FIFO;

    // instantiate refined SOURCE and SINK
    refined_source SOURCE;
    ...

    // connect SOURCE to HW_FIFO

    sc_signal<int>  data_in;
    sc_signal<bool> data_in_valid;
    sc_signal<bool> data_in_ready;
```

```
    SOURCE.data_port( data_in );
    SOURCE.valid_port( data_in_valid );
    SOURCE.ready_port( data_in_ready );

    HW_FIFO.data_in( data_in );
    HW_FIFO.data_in_valid( data_in_valid );
    HW_FIFO.data_in_ready( data_in_ready );

    ... // rest not shown
}
```

Figure 12-3: Design after communication refinement.



The outcome of the communication refinement process is depicted in Figure 12-3. The refined version of the `source` module is connected directly to HW version of the FIFO.

Of course, the approach to communication refinement presented in this chapter is more generally applicable than to just refining point-to-point FIFO links. Adapters can be used wherever interfaces have to be converted. The interface-based design style supported by SystemC allows for replacing channels with adapters where needed[13]. If required, modules and adapters can be merged[14] later in the design process in order to create new modules with refined interfaces.

In more general terms the 5 steps presented above can be described as follows.

1) *Channel/infrastructure refinement*
   Refinement of the communication infrastructure. Abstract channels are merged and/or replaced with more detailed ones. Interfaces are likely to change in this process leaving the system model in a crippled state.

2) *Adapter insertion*
   Adapters are inserted to convert interfaces where required. The system model becomes executable again.

3) *Validation / analysis*
   Timing and behavior may have changed during steps 1 and 2. Simulation is used to validate that the system's behavior and performance is satisfactory.

[13] Ports specify which *interface* they need to be connected to. Different channels – including adapters – can implement the same interface.

[14] This is easily feasible due to the use of interface method calls that are executed in the thread of execution of the calling module. Still, adapters may contain multiple processes or additional data fields. These will be copied together with the adapter's interface methods and ports into the target module's code.

4) *Interface refinement / protocol in-lining*
   Modules with refined interfaces are created by merging adapters into the module itself. This step is also referred to as *protocol inlining*. This step is carried out if

   - step 3 proved that the refinement process so far was successful, and
   - the use of refined modules is desirable, for instance in order to get synthesizable code.

5) *Re-validation / verification*
   Simulation is used to verify that no design errors were introduced during the protocol in-lining phase.

13. MISCELLANEOUS

13.1 MODULE INHERITANCE AND SC_CTOR

SystemC 1.0 defines several useful macros, e.g., SC_MODULE, SC_CTOR, SC_METHOD, SC_THREAD, SC_CTHREAD. Unfortunately, macros are not very flexible, e.g., it is not possible to overload a macro.

In SystemC 2.0, there are two features that are hampered by the SC_CTOR macro.

1) Passing constructor arguments to the constructor of a module.

2) Inheritance of modules, which also requires access to the constructor argument list of a module.

To enable these important features, do the following. If your module has processes, use the new **SC_HAS_PROCESS** macro. This macro takes one argument, which is the name of the module type (as with SC_CTOR). The constructor can now be defined with any constructor arguments you want. There is one restriction: one argument must be of type sc_module_name, and must be passed to the base class. This argument contains the name of the module instance.

Example 13-1: Module inheritance.

```
SC_MODULE( base_module )
{
    ...

    // constructor
    SC_CTOR( base_module )
    { ... }
};

class derived_module
: public base_module
{
    ...

    // process(es)
    void proc_a();

    SC_HAS_PROCESS( derived_module );

    // parameter(s)
    int some_parameter;

    // constructor
    derived_module( sc_module_name name_, int some_value )
    : base_module( name_ ), some_parameter( some_value )
    {
        SC_THREAD( proc_a );
    }
};
```

The use of macro `SC_MODULE` is fully optional. In Example 13-1, one can write:

```
class base_module
: public sc_module
{
    ...
};
```

13.2 END_OF_ELABORATION() METHOD

During elaboration, static design rule checking is supported by the `register_port()` method of interface base class `sc_interface`. But some static checks cannot be performed during elaboration, such as checking whether an `sc_signal<T>` has a writer at all. For these kind of static checks, the **`end_of_elaboration()`** method is provided to modules, channels, and ports. This virtual method is empty by default, but can be redefined to perform the required static check. After elaboration, i.e., just before the simulation starts, this method will be called for all modules, channels, and ports.

13.3 CHANGES WRT SYSTEMC 1.0

This section lists the most prominent changes wrt SystemC 1.0.

- *Model of time* – this has changed from real-valued and relative to integer-valued and absolute. See Chapter 4.

- *Extended notion of events* – to support user-defined channels and dynamic sensitivity, an `sc_event` type is introduced and the `wait()` method has been extended significantly. See Chapter 5.

- *Interfaces, ports, and channels* – to support communication refinement at higher levels of abstraction, user-defined channels can be defined, implementing one or more (user-defined) interfaces. Modules are connected to channels through (user-defined) ports and interfaces. Channels can be primitive or hierarchical. This enables the so-called IMC (interface method call) scheme, which is a generalization of the RPC (remote procedure call) scheme as described in Chapter 15. See Chapter 6 and further.

- *RPC library* – the RPC scheme is implemented as a library on top of the above layers. See Chapter 15 and Figure 1-1.

13.4 ROADMAP

The SystemC 2.0 reference implementation is released in five stages: 1.1 Beta, 1.2 Beta, 2.0 Beta-1, 2.0 Beta-2, and 2.0 Production.

- *1.1 Beta* – Released in April 2000; implements most of Chapter 15.
- *1.2 Beta* – Released in January 2001; implements Chapter 4 (completely), Chapter 5 (no immediate event notification), and Chapter 15 (completely).
- *2.0 Beta-1* – Released mid July 2001; implements the core language and the elementary channels that are part of the SystemC standard, with limited platform support.
- *2.0 Beta-2* – Released early September 2001; implements the core language and the elementary channels that are part of the SystemC standard, with extended platform support and bugfixes of the Beta-1 release.
- *2.0 Production* – Released early October 2001; implements the core language and the elementary channels that are part of the SystemC standard, with full platform support and bug fixes of the beta releases.

PART II. ELEMENTARY CHANNELS

14. ELEMENTARY CHANNELS

This chapter lists the elementary channels that are part of the SystemC 2.0 standard.

- `sc_signal<T>`, `sc_signal_rv<N>`, and `sc_signal_resolved` as in SystemC 1.0.1
- `sc_fifo<T>`, primitive, deterministic version, see Section 10.4
- `sc_mutex`, primitive, non-deterministic version, see Section 10.5
- `sc_semaphore`, primitive, non-deterministic version
- `sc_buffer<T>`, primitive, same as `sc_signal<T>`, but with new-value events instead of value-changed events.

The following is a list of nice to have channels. These channels cannot be guaranteed to become part of the SystemC 2.0 standard.

- `sc_timer` – a general purpose timer, which supports reset and can be programmed to fire once or as many times as desired.
- `sc_event_queue` – a resettable queue of future event notifications.

PART III. METHODOLOGY-SPECIFIC LIBRARIES

OSCI welcomes the development of different methodologies based on the infrastructure of the core language. The following chapter outlines the master-slave communication library. OSCI expects that other methodologies and libraries would be added later.

15. MASTER-SLAVE COMMUNICATION LIBRARY

The master-slave communication library is targeted at systems that utilize master-slave bus communication protocols. Systems that consist of one or more processor cores, DSPs, peripheral devices and custom ASICs communicating over a set of buses are particularly well suited for this library. This library introduces a sequential execution and communication semantics between processes, which is well suited for modeling sequential SW-SW communication, HW-SW interfaces and HW-HW interfaces that are based on master-slave bus protocols. At the functional level, an abstract master-slave communication protocol is defined which when refined to master-slave bus protocol communication preserves the sequential communication/execution order of the functional level[15].

Master-slave library is based on the following key paradigms:

- A sequential channel sc_link_mp<T> that at the functional level defines a sequential (in-lined) execution and communication semantics between processes. Processes that execute sequentially connect through specialized master-slave ports to a sequential channel. Master-slave ports[16] encode the fundamental (orthogonal) components of a sequential communication explicitly with the port.

    - Direction of a communication (in, out, inout)
    - Initiator of a communication (master or slave)
    - Data type
    - Optionally a transaction index (= abstract address)
    - Bus protocol at the bus cycle accurate (BCA) level

    This explicit encoding expresses communication intent and can be utilized by interface synthesis tools.

- A concurrent communication and execution paradigm[17] based on concurrent communication channels between concurrent processes. Concurrent processes synchronize over events/signals and communicate over shared variables, which can be local in a module or external in a communication channel.

[15] This may seem odd since at the cycle accurate level clocked processes run concurrently. Nonetheless the state progression in their FSM's preserves the sequential inter-process behavior of the functional level in a conform channel refinement (=order invariant transformation). Communication resource sharing and other optimizations may however alter this behavior.

[16] Master and slave ports as a group are called abstract ports at the functional level and bus ports at the Bus Cycle Accurate (BCA) level.

[17] Concurrent communication has been discussed extensively in previous chapters.

- A combined sequential-concurrent execution and communication semantics through combined use of master-slave and event/signal[18] ports in the same process.

- A model for re-use based on structural encapsulation of concurrent and sequential behavior in modules, which must communicate with the outside world exclusively over module ports.

In the following, we introduce first the sequential execution semantics of sc_link_mp<T> together with master-slave port classes at the functional level. Then concurrent communication is discussed in the context of master-slave communication. In the next section, refinement of functional level sequential communication to bus protocol communication is presented.

15.1 FUNCTIONAL LEVEL

15.1.1 IN-LINED EXECUTION SEMANTICS[19],[20] OF THE SEQUENTIAL CHANNEL SC_LINK_MP<T>

In previous chapters we have seen how SC_METHOD, SC_THREAD and SC_CTHREAD define a concurrent execution semantics. In this chapter, we're introducing a new process type, called `SC_SLAVE` whose member method, which implements its behavior, can be invoked like a function from another process.

A slave process has a single slave port through which its behavior is invoked. The slave process executes when its method is invoked from a caller process, which is connected to the slave port over a sequential channel sc_link_mp<T>. The slave method executes in-line with the caller process and returns to the caller after execution. The caller process is connected through a master port to a sequential channel sc_link_mp<T> that connects to the slave port of the slave process. The caller invokes the slave method by writing to (reading from) its master port a data value of type <T> if it's an output (input) port. The read or write in the caller blocks until the slave returns. A caller can read from (write to) the same input (output) master port more than once in a given execution.

The caller process can be of any type including slave and can have multiple master ports through which it can invoke slaves. This can lead to in-lined execution chains with complex data dependent behavior. In our experience, complex systems can be described with only a few concurrent threads of execution with each thread having its own complex in-lined execution chains.

---

[18] Ports (sc_in, sc_out, sc_inout) that connect to signals.

[19] Interface Method Call (IMC) discussed in earlier chapters provides for in-lined execution of interface methods of communication channels. However, IMC does not define a sequential channel semantics.

[20] In-lined execution is also called Remote Procedure Call (RPC), which is a term borrowed from Unix with similar semantics. Each concurrent thread defines a thread of execution. In-lined execution is equivalent to executing in the same thread of execution.

A slave process reads the data value if it has an input slave port and writes a data value if it has an output slave port, only once per invocation. Multiple reads or writes in a given slave invocation are illegal and will result in run time warnings. Read and write methods in a slave are non-blocking. The data value on the channel is immediately available to the other process, i.e. sc_link_mp<T> does not follow the evaluate-update semantics discussed in previous chapters.

The last value written to an sc_link_mp<T> is stored until it's overwritten.

Processes in a module may share abstract ports. A process can access abstract and signal ports.

The connection over sc_link_mp<T> allows the caller to invoke the slave method without knowing its method pointer. This is essential for module re-use since a module behavior should not depend on its external connectivity.

We defined above the sequential execution semantics of point-to-point communication. In a multi-point channel, multiple master and slave ports can be inter-connected. Its semantics are described later

15.1.2 MODULE RE-USE AND STRUCTURAL DECOMPOSITION

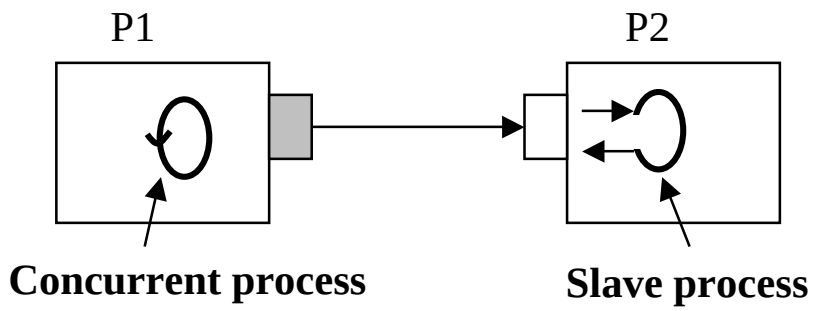
We'll illustrate the relationship between in-lined execution and module re-use with the following example of C code in which a function generate_data() produces a set of integer numbers and for each number calls the accumulate() function that accumulates the numbers and prints them out.

```
void generate_data (int &out)
  {
    for (int i = 0;  i < 10;  i++)
      {
        accumulate(out);
      }
  }

void accumulate (int &in1)
  {
    sum += in1;
    cout << "Sum =   "  << sum << endl;
  }
```

For re-usability purposes, we'd like to partition these two functions into separate processes and encapsulate each in a different module as shown in Figure 15-1. The intent is that this structural decomposition does not change the behavior of the original sequential C code. The SystemC code after this structural decomposition is shown below. We have created a producer and a consumer module which encapsulate the generate_data() and accumulate() processes respectively.

In Figure 15-1, by convention a master port is represented as a gray box and a slave port as a white box. A concurrent process is shown as an oval with an arrow while a slave process is an open oval with an ingoing and an outgoing arrow indicating that it is invoked by another process and returns after execution.

Figure 15-1: In-lined execution.



generate_data() process does not invoke accumulate() directly since the latter is in a different module. Instead it writes to its output port which through the sequential channel invokes accumulate(). Through this structural encapsulation the two modules can now be re-used independently.

Structural decomposition of sequential behavior has the following advantages:

- Modular re-use of sequential behavior
- HW-SW partitioning at module boundaries
- Refinement of sequential communication channels with bus protocols

```
SC_MODULE(producer)
{
  sc_outmaster<int> out1;
  sc_in<bool>       start;  // to kick-start the producer

  void generate_data ()
    {
      for (int i = 0;  i < 10;  i++)
```

```
        {
          out1 = i ;   // this will invoke the slave;
        }
    }

  SC_CTOR(producer)
    {
      SC_METHOD(generate_data);
      sensitive << start;
    }
};

SC_MODULE(consumer)
{
  sc_inslave<int> in1;
  int sum;   // declare as a module state variable

  void accumulate ()
    {
      sum += in1;
      cout << "Sum =   "  << sum << endl;
    }

  SC_CTOR(consumer)
    {
      SC_SLAVE(accumulate, in1);
      sum = 0;   // initialize the accumulator
    }
};

SC_MODULE(top) // structural module
{
  producer *A1;
  consumer *B1;
  sc_link_mp<int> link1;

  SC_CTOR(top)
    {
      A1 = new producer("A1");
      A1.out1(link1);
      B1 = new consumer("B1");
      B1.in1(link1);
    }
};
```

15.1.3 MASTER-SLAVE PORT SYNTAX

Master and slave ports are specialized port classes. At the functional level, they are called *abstract* ports since the communication is abstract at this level.

There are eight abstract port types. `<T>` is the data type that is communicated over the port.

- `sc_master<>`, `sc_inmaster<T>`, `sc_outmaster<T>`, `sc_inoutmaster<T>`
- `sc_slave<>`, `sc_inslave<T>`, `sc_outslave<T>`, `sc_inoutslave<T>`

where `T` can be any of the C++ built-in types (`long`, `int`, `char`, `short`, `float`, `double`, and so forth) or any of the SystemC types (`sc_int<N>`, `sc_uint<N>`, `sc_bigint<N>`, `sc_biguint<N>`, `sc_bit`, `sc_logic`, or fixed-point types), or a user-defined data type.

Examples of abstract port declarations:

```
sc_inmaster<int>              inp1;

sc_outmaster<sc_bigint<128> > out1;

sc_inoutslave<sc_lv<8> >      inout1;
```

15.1.3.1 Examples of read and write methods of abstract ports

Read/write methods are supported in implicit and explicit form as illustrated below.

```
sc_inmaster<int>              inp1;
sc_outmaster<sc_bigint<128> > out1;

{ // a caller process
 int   ival;
ival = inp1;           // blocking read from input port & invoke slave
ival = inp1.read();    // blocking read (explicit) & invokes slave
sc_bigint<128> bigival;
bigival = 1000;
out1 = bigival;        // blocking write to output port & invoke slave
out1.write(bigival);   // blocking write (explicit) & invoke slave

}
```

15.1.3.2 Control flow abstract ports

Two specialized port classes, `sc_master` and `sc_slave`, only convey control flow without data flow direction or data type. An example would be to model an interrupt.

Examples:

```
sc_master<> mport;

sc_slave<>  sport;
```

In the calling process, a transaction is invoked by calling the function method on the master port, which will invoke the slave process connected to the master port. For the above example this would be:

```
mport();
```

It is illegal to write to or read from ‘directionless’ ports. Data type can be specified optionally but will have no effect.

15.1.4 SC_LINK_MP SYNTAX

Syntax for links is:

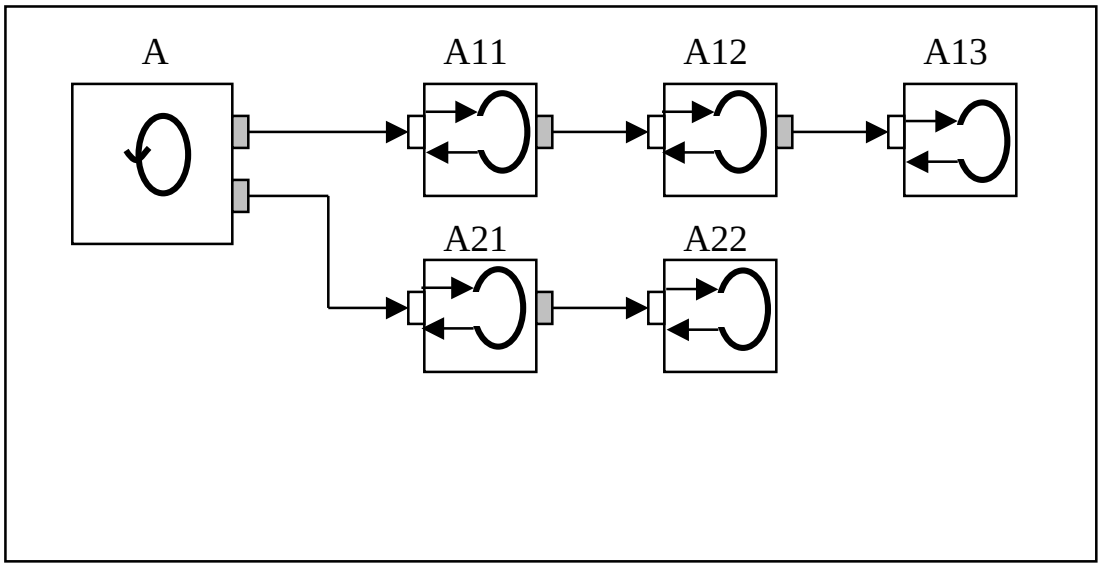
```
sc_link_mp<T> link1;
```

where <T> is the data type communicated over the link. Ports connected to a link must have the same data type as the link. sc_link_mp<T> can only bind to abstract ports.

15.1.5 EXAMPLE: IN-LINED EXECUTION (RPC) CHAINS

Two RPC-chains each attached to a master port of a concurrent process are illustrated in Figure 15-2.

Figure 15-2: RPC chain example.



The execution order in this example is A, A11, A12, A13, A21, and A22 in the assumption that the top most master port is accessed first.

RPC chains cannot form a loop since this would require a slave process to have more than one slave port.

15.1.6 WAITING IN SLAVE PROCESSES

A slave process can synchronize with events and signals through embedded `wait()` statements (with static and dynamic sensitivity) provided that the slave process runs in-lined in a thread process. Otherwise you will get a run time warning. Statically sensitive wait()’s in a slave are sensitive to the sensitivity list of the thread in which it executes in-lined.

15.1.7 SLAVE PROCESS SYNTAX

The slave process takes two arguments: *slave_method*, which implements the behavior of slave process and the unique *slave_port*, which must be an abstract port. *slave_method* must be a member method of the module class.

```
SC_CTOR(my_mod)
{
  // module constructor
  ...
  SC_SLAVE(slave_method, slave_port);
  // defines slave process with its unique slave port;
  ...
}
```

15.1.8 SEQUENTIAL EXECUTION SEMANTICS IN A MULTI-POINT COMMUNICATION (SC_LINK_MP)

In a multi-point (*mp)* communication, a master process communicates with multiple slaves as follows: a master performs an access to its master port, which will invoke all compatible slaves. Compatible slaves are explained below. Slaves are invoked sequentially but in an undefined order. Each slave executes and returns, then the next slave is invoked in a depth-first order. If a slave blocks at a `wait()`, then the remaining slaves in the list block as well because of the depth-first ordering.

Two concurrent processes accessing the same link over master ports, access the link in a sequential but undefined order. This allows abstraction of any relationship of simultaneity between the two concurrent processes.

The sc_link_mp<T> object does not provide built-in arbitration or prioritization functions. It is a primitive (atomic) object that allows abstraction of such refinements. Complex bus models with bus arbitration and prioritization schemes can be built up by composition of primitive channels.

On a write operation to a `sc_outmaster or sc_inoutmaster` port, all slaves connected to the link with `sc_inslave` or `sc_inoutslave` ports will be invoked and may (optionally) read the data value on the link. This constitutes a broadcast communication.

On a read from an `sc_inmaster` or a sc_inoutmaster port, all slaves connected to the link with `sc_outslave` or `sc_inoutslave` ports will be invoked but one and only one must respond with a write operation to its slave port. Otherwise a run-time warning will be generated and the last value written to the link will be retained (undefined condition). The caller avoids clashes by providing an index (=address) with the transaction (see indexed ports).

A write to an input (master or slave) or a read from an output (master or slave) port is illegal which will result in a run-time warning.

The new value assigned to an abstract link is visible immediately[21] to all processes connected to the link.

15.1.9 CONCURRENT COMMUNICATION COMBINED WITH MASTER-SLAVE COMMUNICATION

As stated earlier, a process can access abstract and signal ports. This forms the basis for combining concurrent and sequential master-slave communication.

Concurrent processes synchronize over signals or events using static and dynamic wait() statements discussed in previous chapters. Concurrent processes communicate (data) over shared variables/memory when they are inside the same module. Shared variables are data members of a module. If the communication is across modules, then the communication takes place over a concurrent channel[22], which contains the shared variables/memory. A concurrent channel is not a new type; it is a module. The processes that communicate over the concurrent channel can connect to it over primitive sequential channels (sc_link_mp) as illustrated in the FIFO example below. Access to shared variables can be protected by mutexes[23]. Mutex models are described in previous chapters.

In-lined (RPC) communication between two threads of execution is per definition not possible.

[21] It is possible to write non-deterministic behavior due to this immediate update semantics of `sc_link_mp`. In our opinion this does not take away from the usefulness of this sequential paradigm. Non-determinism is also a fact of life in multi-threaded programming, which did not prevent its widespread adoption. Non-determinism is avoided through careful design.

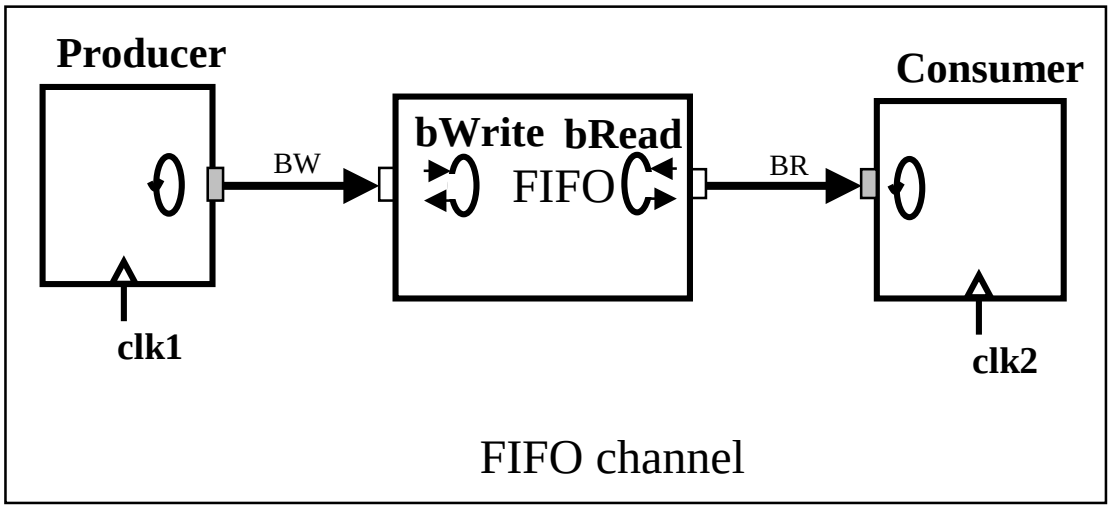
[22] Concurrent channels are also called hierarchical channels. Signals are a special form of primitive concurrent channels with an evaluate-update semantics which models delta-delay propagation in cycle accurate HW.

[23] SystemC is not pre-emptive. Therefore data corruption with unprotected access to share memory cannot be modeled.

15.1.9.1 Example of a concurrent communication channel

An example of a concurrent channel is depicted in Figure 15-3 where a producer and a consumer are concurrent processes, which communicate over a concurrent FIFO channel. The FIFO implements a blocking write (bWrite) and a blocking read (bRead) slave process to write to and read from the FIFO buffer. The buffer inside the FIFO is a shared memory object that the two slave processes access. BW and BR channels are sequential for in-lined execution. If the producer tries to write into the FIFO buffer while it's full, then the bWrite process will block the producer process. Similarly when the consumer attempts to read from an empty FIFO buffer.

Figure 15-3: Concurrent FIFO channel.



15.1.10 CONNECTIVITY RULES AND EXAMPLES[24]

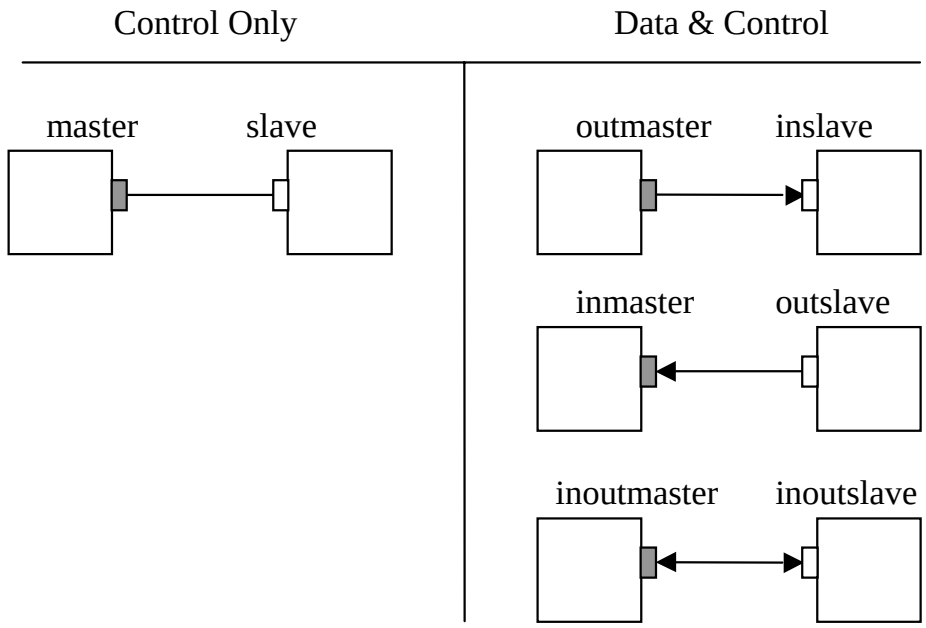
15.1.10.1 Point-to-point (p2p) communication connectivity rules

Abstract ports can be used in the following combinations in a p2p link:

```
sc_master          --- sc_link_mp ---     sc_slave
sc_outmaster       --- sc_link_mp ---     sc_inslave
sc_inmaster        --- sc_link_mp ---     sc_outslave
sc_inoutmaster     --- sc_link_mp ---     sc_inslave
sc_inoutmaster     --- sc_link_mp ---     sc_outslave
sc_outmaster       --- sc_link_mp ---     sc_inoutslave
sc_inmaster        --- sc_link_mp ---     sc_inoutslave
sc_inoutmaster     --- sc_link_mp ---     sc_inoutslave
```

A master and a slave port must always be used as a pair at both ends of a p2p communication link. You need to match an output port with an input port. A pair of two masters, two slaves, two outputs, or two inputs at the opposite ends of a p2p link is illegal. Figure 15-4 illustrates some legal p2p connections.

Figure 15-4: Some legal p2p connections.



[24] This section can be skipped without loss of continuity.

15.1.10.2 Multi-point link connectivity rules

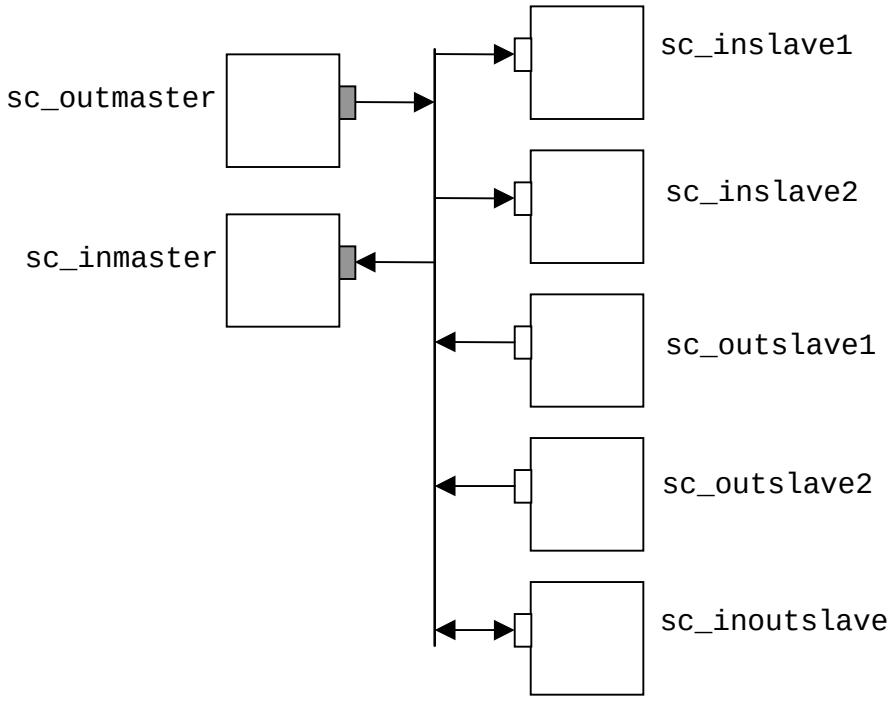
The following connectivity rule is enforced for multi-point connections: with each master port connection, there should at least be one slave connection that can respond to the master. This rule is statically checked at the start of a simulation.

In practice, this means the following:

- With an sc_master port in a link there must at least be one sc_slave connection.
- With an `sc_inmaster` port in a link there must at least be one `sc_outslave` or one `sc_inoutslave` connection.
- With an `sc_outmaster` port in a link there must at least be one `sc_inslave` or one `sc_inoutslave` connection.
- With an `sc_inoutmaster` port in a link, there must at least be one `sc_inslave` and one `sc_outslave` or one `sc_inoutslave` connection.

15.1.10.3 Multi-point link example

Figure 15-5: Multi-point link example.

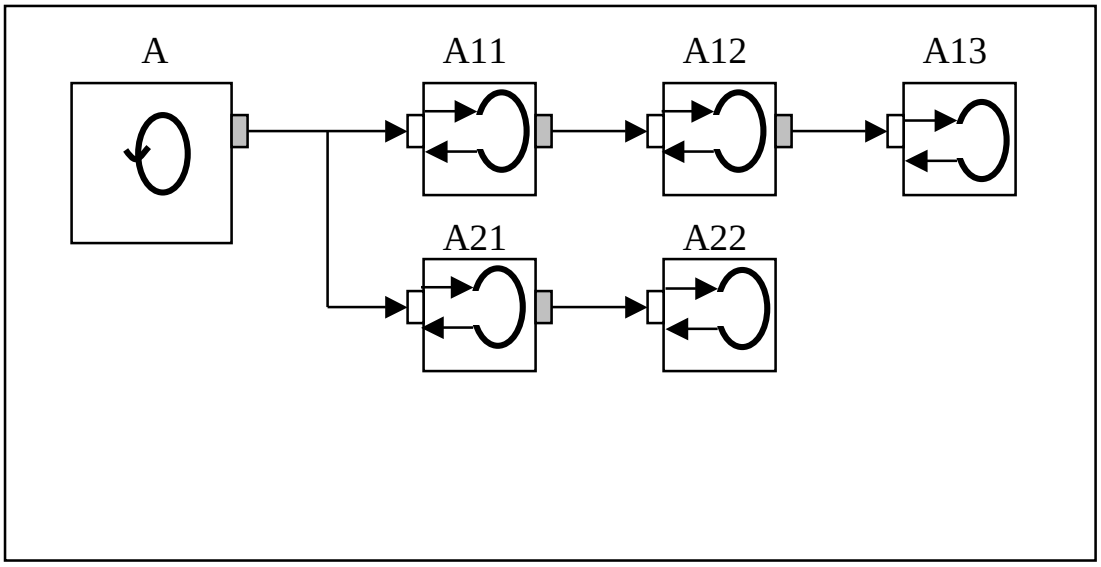


A multi-point link is illustrated in Figure 15-5, where two concurrent processes are connected to multiple slaves.

When the `sc_outmaster` writes to its `sc_outmaster` port, all `sc_inslave` and `sc_inoutslave` processes will be invoked. In this case, `sc_inslave1`, `sc_inslave2` and `sc_inoutslave` will be invoked. Similarly a read operation from an `sc_inmaster` port will invoke all `sc_outslave` and `sc_inoutslave` processes. In the example below, `sc_outlsave1`, `sc_outslave2` and `sc_inoutslave` will be invoked. However, only one slave should write a value to its output port.

15.1.10.4 Example: execution order in a multi-point link

Figure 15-6: Execution order in a multi-point link.



In the example of Figure 15-6, we show two RPC chains A, A11, A12, A13 (chain1) and A, A21, A22 (chain 2) in a multi-point link. Valid execution orders are (depth-first): A, A11, A12, A13, A21, A22 or A, A21, A22, A11, A12, A13 .

15.1.11 ABSTRACT PORT CLASSES DETAILED[25]

15.1.11.1 Port arrays

Arrays of ports are defined using the C array syntax. For example, the following is an array of `sc_inmaster` ports of type `int`, with 10 elements.

```
sc_inmaster<int> P1[10];
```

15.1.11.2 Indexed ports[26]

Master and slave processes can perform data transactions that have an address, also called an index. Using this mechanism, a master process can write to or read from an address in a memory block in a slave process. Indexed ports are declared with an integer valued range parameter that specifies the upper limit of the address range, which always starts from 0. Master and slave ports in a link must have matching address ranges. The master specifies the index value of a transaction as a master port index, while the slave reads this value using the `get_address()` method of its slave port. You can not specify the index of a transaction at a slave port. It assumes the index of the master transaction. Access to an index outside the range is illegal and will result in a run time error.

Syntax:

```
sc_outmaster<T, sc_indexed<range> >  port;

sc_inslave<T, sc_indexed<range> >   port;
```

Ports of type `sc_master` and `sc_slave` do not take a range parameter.

For example, you define an `sc_outmaster` port `P2` of type `int` with an address range of (0,1023) as follows:

```
sc_outmaster<int, sc_indexed<1024> > P2;
```

To illustrate a master write of an `int` value with address 68 on `sc_outmaster` `P2`, we have included the following code:

```
SC_MODULE(my_master)
{
  sc_outmaster<int, sc_indexed<1024> > P2;
```

[25] This section can be skipped without loss of continuity.

[26] Indexed ports are an abstraction at the functional level. They do not exist at the BCA/CA levels.

```
  int data;
  ...
  { // body of master process tied to port P2
    ...
    data = 10;
    P2[68] = data;
    ...
  }
};
```

In the following code for the slave, assume port `pslave` of `my_slave` is connected to port `P2` above. The result of this transaction is that the value 10 is written into memory location 68.

```
SC_MODULE(my_slave)
{
  sc_inslave<int, sc_indexed<1024> > pslave;
  int memory[1024]; // memory block
  ...
  { // body of slave process triggered by pslave
    int index;
    index = pslave.get_address();
    // index gets value 68 of the master transaction
    memory[index] = pslave;
    // memory[68] is assigned the value 10
    // don't specify an index with pslave
    ...
  }
};
```

In the example above, a master process attached to port `P2` writes out a value with index 68. The slave process reads the index value of the transaction by invoking the member function `get_address()` of the `pslave` port.

15.1.11.3 Indexed port arrays

You can define an array of indexed ports. An indexed port array `P2` with 10 members and an address range of (0,1023) for each member is specified in the following example:

```
sc_inmaster<int, 1024>  P2[10];
```

To read the value of port member 5 at address 68, you use the C++ syntax for two-dimensional arrays as shown in the following example. The first index indicates the member of the port array, and the second index indicates the address of the data.

```
int data;
data = P2[5][68];
```

15.1.11.4 Inout ports

When a slave process is triggered by an `sc_inoutslave` port in response to a transaction initiated by a master process, the slave process needs to determine the direction of the data transfer requested by the master. This is done by calling the `input()` method of the slave port. This method returns true for an input transaction into the slave port and false for an output. The

master always determines the direction of a transaction by writing to or reading from its master port. A slave response that is incompatible with the master request will result in a run time error.

In the following example, the direction of the data transfer on port `P1` is obtained by the `P1.input()` method.

```
sc_inoutslave<int> P1;
// the rest of the code is not shown
if ( P1.input() ) {
    val = P1;  // read from P1
} else {
    P1 = val;  // write to P1
}
```

15.1.11.5 Connectivity rules for abstract ports in hierarchical designs

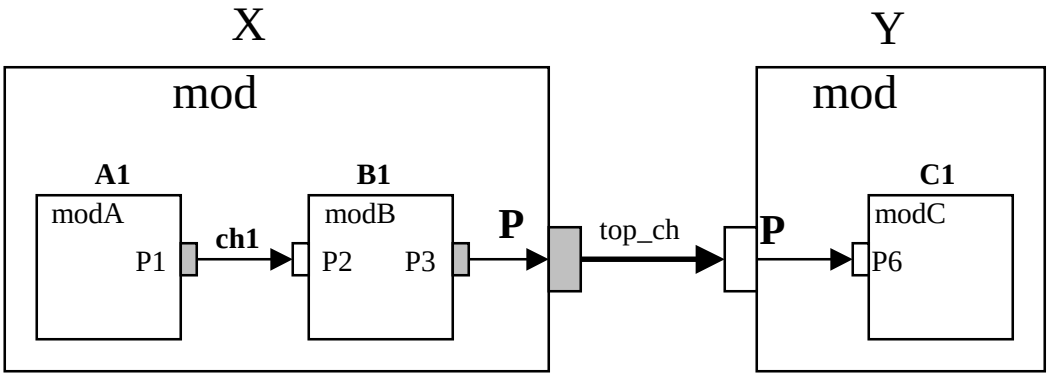
When modules are instantiated inside other modules, a child module port that connects to a parent module port must be of compatible type as follows:

- a master port must connect to a master port, a slave port to a slave port
- a child input port can connect to a parent input or inout port
- a child output port can connect to a parent output or inout port
- a child inout port can connect to a parent inout port
- data type of child and parent module must be the same

These rules are checked at initialization by the SystemC class library.

Figure 15-7 illustrates how abstract ports and links are used in a hierarchical system. In this system, P4 is a representation of P3 at another level of hierarchy, so it retains the same abstract protocol specification as P3. In module modY, P5 has the same relationship to P6. The code is also shown below.

Figure 15-7: Hierarchical example.



```
SC_MODULE(modA) {
  sc_outmaster<int> P1;
  ...
};

SC_MODULE(modB) {
  sc_inslave<int> P2;
  sc_outmaster<int> P3;
  ...
};

SC_MODULE(modC) {
  sc_inslave<int> P6;
  ...
};

SC_MODULE(modX) {
  sc_outmaster<int> P4;
  sc_link_mp<int> ch1;
  modA *A1;
  modB   *B1;
  ...
  SC_CTOR(modX) {
    A1 = new modA(“A1”);
    A1(ch1);
    B1 = new modB(“B1”);
    B1(ch1, P4);
  }
};

SC_MODULE(modY) {
  sc_inslave<int> P5;
  modC *C1;
  ...
  SC_CTOR(modY) {
    C1 = new modC(“C1”);
    C1(P5);
  }
};

SC_MODULE(top) {
  sc_link_mp<int> top_ch;
  modX *X1;
  modY *Y1;
  ...
  SC_CTOR(top) {
    X1 = new modX(“X1);
    X1(top_ch);
    Y1 = new modY(“Y1”);
    Y1(top_ch);
  }
};
```

15.2 BUS CYCLE ACCURATE LEVEL

15.2.1 REFINING COMMUNICATION WITH BUS PROTOCOLS

When you refine a design from the functional abstraction to the Bus Cycle Accurate (BCA) or Cycle Accurate (CA) HW abstraction levels, you do the following transformations:

- Sequential communication channels of the functional level are refined to concurrent communication channels using a bus protocol.
- Functional processes are refined to synchronous concurrent processes by introducing clocks and resets.

Channel refinement is not as straightforward as simply replacing a functional channel with a bus protocol channel. This may work for simple point-to-point communication channels but does not for complex multi-point channels. Complex channels require design work, either manually or aided by an interface synthesis tool involving design of address decoders, multiplexers, bridges, protocol adapters, etc.

In this discussion, we will not describe the refinement process itself, which is methodology and tool dependent and falls outside the scope of this discussion. Instead we present the BCA[27] level language constructs for design description after this refinement has taken place.

At the BCA level, master-slave ports are specialized with a bus protocol template parameter. Channels are sc_link_mp<T> objects, which do not have a bus protocol parameter. A channel gets its bus protocol type from the ports that are connected to it. All ports connected to a channel must have the same protocol type.

A bus protocol class specifies terminals for communicating data, control and address information. These terminals become members of the bus port class after specialization. A protocol terminal is a signal port that will connect to a matching signal in a bus protocol channel. Terminals have read / write methods that follow the update-evaluate semantics of signals. A bus port does not have read/write methods. All communication takes place through its terminals[28].

sc_link_mp<T> object for a given bus protocol is a bundle of signals derived from the bus protocol type. A protocol port is bound as a whole to an sc_link_mp<T> object[29], not its terminals. sc_link_mp<T> will automatically bind the terminals of the port to the corresponding signals in the channel.

---

[27] In this discussion we use BCA to denote both BCA and CA level descriptions.

[28] This restriction will be removed in the future. A read/write method on a bus port will then read/write the port terminals according to the semantics of the bus protocol.

[29] This restriction will be removed in the future.

sc_link_mp<T> is an overloaded object. As a bus protocol channel, it does not have an execution semantics, which it had at the functional level.

SystemC comes with 3 pre-defined bus protocols: no-handshake, enable-handshake and full-handshake protocols. How you can define your own bus protocols is explained later.

Port specialization with bus protocols allows tools to synthesize the channel from the different port protocols that are connected to it. This is more flexible than the channel specialization approach, which would require pre-defined channels for different channel configurations.

15.2.2 EXAMPLE BUS PROTOCOL PORT

The example below shows how a functional level abstract port is refined into a bus port using the 'enable-handshake' protocol.

```
sc_outmaster<T> myPort; // abstract port

sc_outmaster<T, sc_enableHandshake<T> > myPort; // bus port
```

The data type <T> must be specified twice[30], once as a template parameter of the port and once as a template parameter of the protocol. The `sc_enableHandshake` protocol has a 'data' terminal and an 'enable' control terminal.

You can read from and write to the terminals of a bus port with the member '.' operator.

```
sc_inoutslave<int, sc_enableHandshake<int> > myPort;
value = myPort.d; // read from data terminal of port
myPort.d = value; // write to data terminal of port
```

Other terminal accesses on the same port are shown below:

```
if (myPort.en) {...}// checks the value of the enable terminal
if (myPort.input) {...} // checks the direction of the requested transaction
```

15.2.3 POINT-TO-POINT COMMUNICATION REFINEMENT EXAMPLE

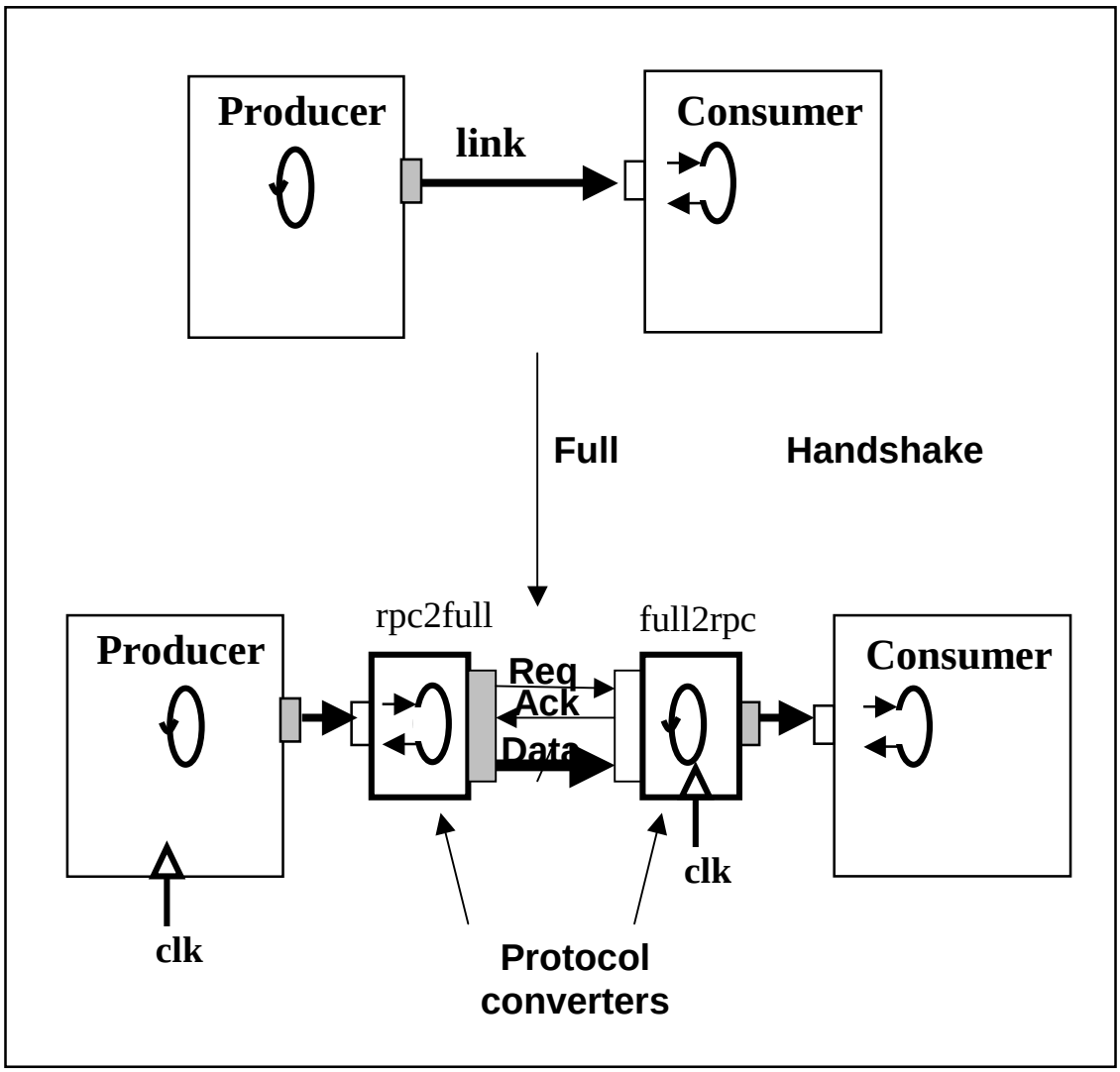
In the example below an abstract communication channel at the functional level is refined into a full handshake protocol channel. Refinement is entirely done in the channel by inserting protocol conversion modules in the channel as shown in the Figure 15-8. This approach allows separation of communication and behavior. Alternatively, bus protocols could have been implemented directly in the producer or consumer modules, which would not have allowed such a

[30] This double specification of <T> is an unfortunate artifact of the implementation. Both data types must be the same.

separation. The protocol converters in this example convert a communication from an abstract into a bus protocol (rpc2full) communication or vice-versa (full2rpc).

rpc2full converter is a slave process which does not have a clock input of its own. It “gets” its clock from the calling process (producer). The full2rpc converter has a clock input which is passed on to the consumer module.

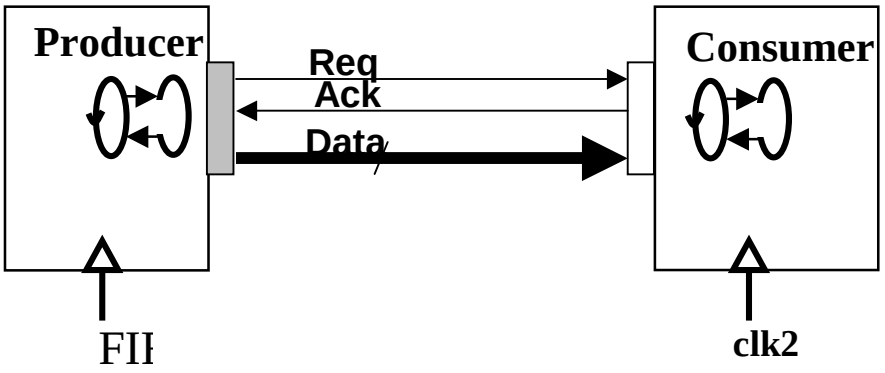
Figure 15-8: Communication refinement example.



In the final step, protocol conversion modules are in-lined inside the producer and consumer modules as shown in Figure 15-9. In-lining merges the slave process function into the calling

process such that abstract ports are removed as a result of this transformation. In the future, this in-lining will be done automatically inside the read/write methods of the bus protocol[31].

Figure 15-9: After in-lining of protocol converters.



We're using the same producer/consumer example from the beginning of this chapter. It is copied below for convenience.

```
SC_MODULE(producer)
{
  sc_outmaster<int> out1;
  sc_in<bool>       start;  // to kick-start the producer

  void generate_data ()
    {
      for (int i = 0;  i < 10;  i++)
        {
          out1 = i ;   // this will invoke the slave;
        }
    }

  SC_CTOR(producer)
    {
      SC_METHOD(generate_data);
      sensitive << start;
    }
};

SC_MODULE(consumer)
{
```

[31] This functionality will be provided in the future. Read/write methods of a protocol will be accessed from the port using the -> operator. For example: port->read() will access the read() method of the port's bus protocol which will implement its semantics.

```
  sc_inslave<int> in1;
  int sum;   // declare as a module state variable

  void accumulate ()
    {
      sum += in1;
      cout << “Sum =   “  << sum << endl;
    }

  SC_CTOR(consumer)
    {
      SC_SLAVE(accumulate, in1);
      sum = 0;   // initialize the accumulator
    }
};

SC_MODULE(top) // structural module
{
  producer *A1;
  consumer *B1;
  sc_link_mp<int> link1;

  SC_CTOR(top)
    {
      A1 = new producer(“A1”);
      A1.out1(link1);
      B1 = new consumer(“B1”);
      B1.in1(link1);
    }
};
```

The code for the above example with protocol converters but before in-lining.

```
SC_MODULE(rpc2Full) {
   sc_inslave<int> in;
   sc_outmaster<int, sc_fullHndshk <int> > out;
    void doit() { // executes in the thread of the producer
      wait(); // wait for the clock in the producer
     out.req = true;
      out.data = in;
      while (!out.ack) {
        wait();
      out.req = false;
      }
    }
    SC_CTOR(rpc2Full) {
      SC_SLAVE(doit,in);
    }

};
SC_MODULE(full2RPC) {
   sc_outmaster<int> out;
   sc_in<bool> clk;
   sc_inslave<int, sc_fullHndshk <int> > in;
    void doit() {
      while (1) {
        wait();
        if (in.req) {
          out = in.data; // RPC call
          wait();
```

```
            in.ack = true;
            wait();
            in.ack = false;
           }
         }
      }
      SC_CTOR(full2RPC) {
        SC_THREAD(doit);
        sensitive_pos << clk;
      }

  }


  SC_MODULE(top) {
    producer *A1;
    consumer *B1;
    rpc2Full *adapter1;
    full2RPC *adapter2;
    sc_link_mp<int> link1;
    sc_link_mp<int> link2;
    sc_link_mp<int> buslink;
    sc_clock clk("clk",1000);
    SC_CTOR(top)
      {
        A1 = new producer("A1");
        A1.out1(link1);
        A1.clk(clk);
        B1 = new consumer("B1");
        B1.in1(link2);
        adapter1 = new rpc2Full("adapter1");
        adapter2 = new full2RPC("adapter2");
        adapter1.in(link1);
        adapter1.out(buslink);
        adapter2.in(buslink);
        adapter2.clk(clk);
        adapter2.out(link2);
     }
   };
```

The code after in-lining (Figure 15-9).

```
  SC_MODULE(producer)
  {
    sc_outmaster<int,sc_fullHndshk<int> > out1;
    sc_in<bool>       clk;

    void generate_data ()
      {
        for (int i = 0;  i < 10;  i++)
          {
             wait();
             out1.req = true;
             out1.data = i;
             while (!out1.ack) {
               wait(); // wait for the clock in the producer
             out1.req = false;
             }
          }
      }
```

```
  SC_CTOR(producer)
    {
      SC_THREAD(generate_data);
      sensitive_pos << clk;
    }
};

SC_MODULE(consumer)
{
  sc_inslave<int, sc_fullHndshk<int> > in1;
  int sum;   // state variable

  void accumulate ()
    {
      while (1) {
        wait();
        if (in1.req) {
          sum += in1.data;
          cout << “Sum =   “  << sum << endl;
          wait();
          in1.ack = true;
          wait();
          in1.ack = false;
         }
       }
    }

  SC_CTOR(consumer)
    {
      SC_THREAD(accumulate);
      sensitive_pos << clk;
      sum = 0;   // initialize the accumulator
    }
};

SC_MODULE(top) // structural module
{
  producer *A1;
  consumer *B1;
  sc_clock clk(“clk”,1000);
  sc_link_mp<int> link1;

  SC_CTOR(top)
    {
       A1 = new producer(“A1”);
       A1.out1(link1);
       A1.clk(clk);
       B1 = new consumer(“B1”);
       B1.in1(link1);
       B1.clk(clk);

    }
};
```

15.2.4 PRE-DEFINED BUS PROTOCOLS

The following bus protocols are provided as a part of the SystemC class library:

- `sc_noHandshake<T>`
- `sc_enableHandshake<T>`
- `sc_fullHandshake<T>`

Figure 15-10 shows the terminals for pre-defined protocols.

Figure 15-10: Terminals of pre-defined bus protocols.

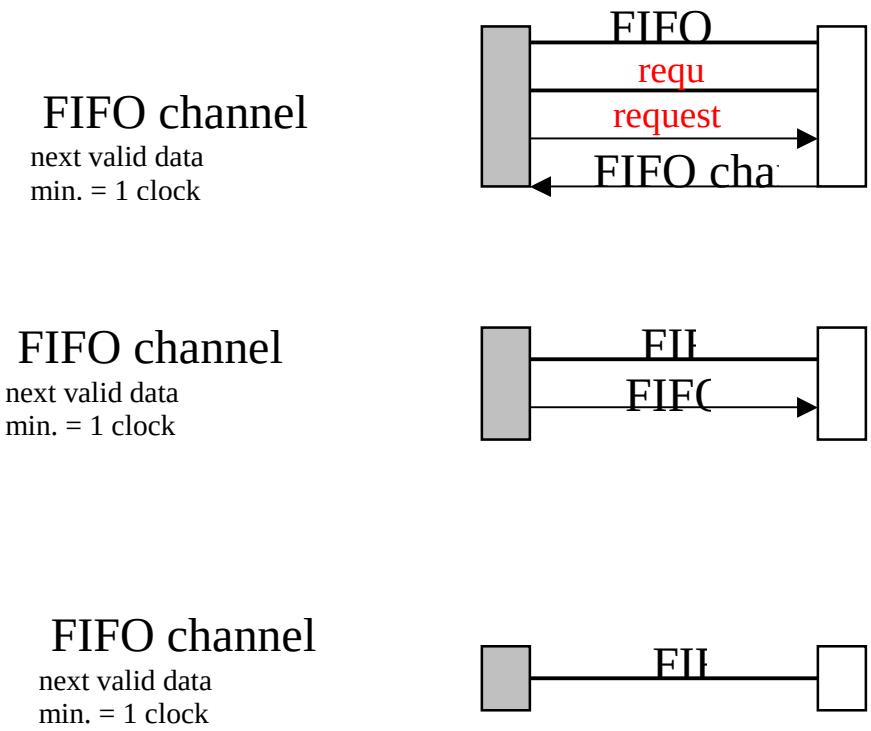
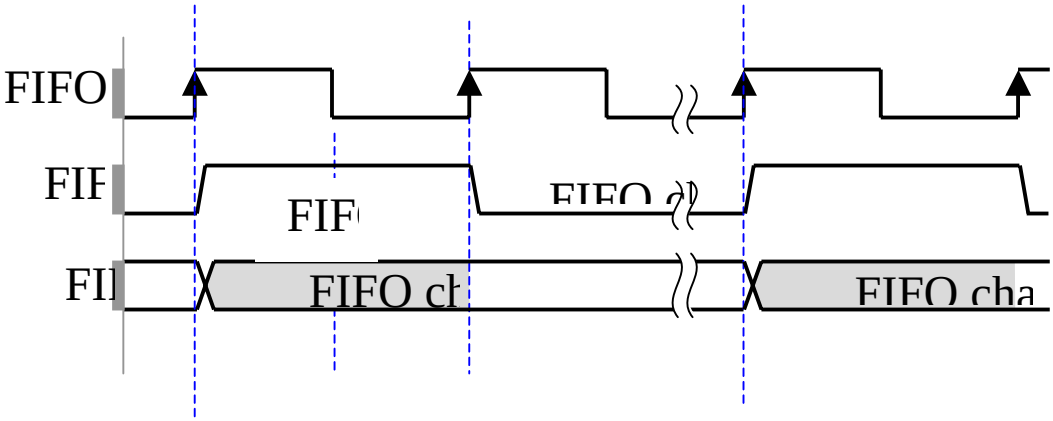


Figure 15-11 shows the timing diagram for the `sc_enableHandshake` protocol.

Figure 15-11: Enable-handshake timing diagram.



15.2.5 USER-DEFINED BUS PROTOCOLS[32]

New bus protocols can be defined using the `SC_PROTOCOL` construct. A `SC_PROTOCOL` defines a structure that contains the terminal definitions of a bus protocol. A terminal defines a signal port together with its type as data (`TT_DATA`), control (`TT_CONTROL`) or address (`TT_ADDRESS`). A bus protocol defines terminals for each abstract port type since terminals need to be configured per port type. For example, an enable terminal is configured as an output type for a master port and as an input for a slave port. When a port is specialized with a bus protocol, port terminals are created that match the port type. Port terminals are data members of the port.

Type information of a terminal is used by sc_link_mp<T> to link terminal's signal port to the matching signal in the bus channel.

For example the `SC_INMASTER_P` construct is used to define the terminals that can be accessed on an `sc_inmaster` port.

Terminals are declared using the `sc_terminal_in`, `sc_terminal_out` and `sc_terminal_inout` classes.

The terminals of bus ports are connected automatically by sc_link_mp to the bus signals in the channel as follows:

- If the terminals have an id template argument, then terminals of a type (`TT_DATA`, `TT_CONTROL`, `TT_ADDRESS`) will connect to the terminals with the same type and terminal id.
- If no id is specified then the terminals are connected by position: e.g. the first `TT_DATA` of a bus port will connect to the first `TT_DATA` of other bus ports

[32] This section can be skipped without loss of continuity.

connected to the same sc_link_mp (similar for TT_CONTROL and TT_ADDRESS)

EXAMPLE

In this example, a bus protocol for an “enable-handshake” protocol is defined.

```
template <class datatype>
SC_PROTOCOL(sc_MyEnableHandshake)
{
  SC_OUTMASTER_P {
    sc_terminal_out<datatype,TT_DATA> d;
    sc_terminal_out<bool,TT_CONTROL>  en;
  };

  SC_INMASTER_P {
    sc_terminal_in<datatype,TT_DATA>  d;
    sc_terminal_out<bool,TT_CONTROL>  en;
  };

  SC_INOUTMASTER_P {
    sc_terminal_in<datatype,TT_DATA>  din;
    sc_terminal_out<datatype,TT_DATA> dout;
    sc_terminal_out<bool,TT_CONTROL>  en;
    sc_terminal_out<bool,TT_CONTROL>  nRW;
  };

  SC_OUTSLAVE_P {
    sc_terminal_out<datatype,TT_DATA> d;
    sc_terminal_in<bool,TT_CONTROL>   en;
  };

  SC_INSLAVE_P {
    sc_terminal_in<datatype,TT_DATA>  d;
    sc_terminal_in<bool,TT_CONTROL>   en;
  };

  SC_INOUTSLAVE_P {
    sc_terminal_out<datatype,TT_DATA> din;
    sc_terminal_in<datatype,TT_DATA>  dout;
    sc_terminal_out<bool,TT_CONTROL>  en;
    sc_terminal_out<bool,TT_CONTROL>  nRW;
  };
};


SC_MODULE(Producer)
{
  sc_outmaster<int,sc_MyEnableHandshake> mstr;
  sc_in<bool> clk;
  int data;

  void send_data()
    {
      while (true)
        {
          mstr.d = data;  // in 1.1beta1 this would have been mstr->d
          mstr.en = true; // in 1.1beta1 this would have been mstr->en
          wait(); // wait for clock
        }
```

```
    }

  SC_CTOR...
};
```

15.2.5.1 BNF for bus protocol definitions

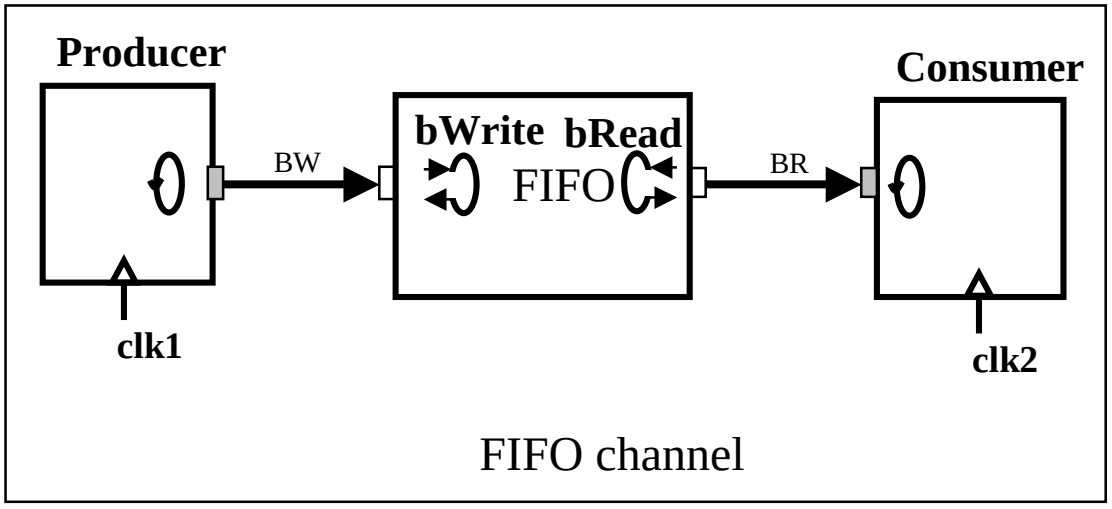
```
protocol_definition:
   SC_PROTOCOL ( identifier  ) {  port_protocols };

port_protocols :
   port_protocol
   port_protocols port_protocol

port_protocol :
   port_role { port_protocol_members };

port_role :
   SC_SLAVE_P
   SC_INSLAVE_P
   SC_OUTSLAVE_P
   SC_INOUTSLAVE_P
   SC_MASTER_P
   SC_INMASTER_P
   SC_OUTMASTER_P
   SC_INOUTMASTER_P

port_protocol_members :
    /*empty*/
    port_protocol_member port_protocol_members

port_protocol_member :
   terminal_member


terminal_member :
   terminal_type < datatype > identifier ;
   terminal_type < datatype ,terminal_kind > identifier ;
   terminal_type < datatype, terminal_kind, terminal_id > identifier ;

terminal_kind :
   TT_DATA
   TT_CONTROL
   TT_ADDRESS

terminal_id :
   number

terminal_type :
   sc_in_terminal
   sc_out_terminal
   sc_inout_terminal
   sc_in_terminal_rv
   sc_out_terminal_rv
   sc_inout_terminal_rv
```

15.3 EXAMPLES

15.3.1 FIFO MODEL AT THE FUNCTIONAL LEVEL

In this example, we illustrate how blocking RPC is used to model blocking write and read in a FIFO communication link. A producer process produces data items (integers in this example) and sends it through the FIFO link to the consumer. The producer runs at a higher speed than the consumer process such that the FIFO buffer will get full after some time. This will block the producer process since it will not be able to write into the FIFO buffer. As soon as the consumer removes an item from the FIFO buffer, the producer will be unblocked and write another item into the buffer.

Figure 15-12: FIFO model at the functional level.



In the example code below, producer and consumer are implemented as clocked `SC_THREAD` processes to allow the two processes to run at different ‘speeds’. Producer is sensitive to the clock. Producer has a `sc_outmaster` port through which it invokes the `blockingWrite()` slave process in the FIFO. The `blockingWrite()` blocks when the buffer is full. The state of the buffer is tested on every sensitive clock edge of the producer clock . Consumer is sensitive to a different clock. Consumer has an `sc_inmaster` port through which it invokes the `blockingRead()` slave process in the FIFO. The `blockingRead()` blocks when the buffer is empty. The state of the buffer is tested on every sensitive clock edge of the consumer clock. For the sake of brevity, the buffer class is not shown in the code below. The buffer has internally a ‘full’ and an ‘empty’ signal to keep the buffer’s state such that the buffer has a deterministic behavior, i.e. its behavior does not depend on which of `blockingWrite()` and `blockingRead()` processes checks first the state of the buffer in a given delta cycle. Simulation results below show a correct operation of the FIFO.

```
// file buffer.h
#define BUFFER_SIZE 10
```

```
template <class itemT>
class buffer
{
public:
  buffer(); // constructor
  virtual ~buffer() {}
  itemT get();      // get item from the top of buffer
  bool put(itemT);  // put item at the bottom of buffer
  bool full();      // buffer full
  bool empty();     // buffer empty

private:
  // private section not shown
};

// File Name : fifo.h
// A module with inslave and outslave abstract ports,
// and a fifo store

#ifndef SYSTEMC_H
#include "systemc.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

#ifndef BUFFER_H
#include "buffer.h"
#endif

SC_MODULE(fifo)
{
  // ports
  sc_out<int> full;
  sc_out<int> empty;
  sc_inslave<int> Pwrite; // slave port
  sc_outslave<int> Pread; // slave port

  buffer<int> buf; // FIFO buffer
  int item;        // buffer item

  // slave methods

  void blockingWrite()
    {
      if (buf.full() )
        {
          do {wait();} // wait for sensitive edge of the producer clock
          while( buf.full());
        }

      // buffer is not full
      // write into buffer
      item = Pwrite; // read from slave port
      cout << "Writing into buffer: item = " << item << endl;
      buf.put(item);
    }

  void blockingRead()
```

```
    {
      cout << "\nfifo:blockingRead called" << endl;
      if (buf.empty())
        {
          do {wait();}  // wait for sensitive edge of the consumer clock
          while (buf.empty() );
        }

      // buffer is not empty
      // read from buffer
      item = buf.get();
      cout << "  Item read = " << item << endl;
      Pread = item; // write to slave port
    }

  SC_CTOR(fifo)
    {
      SC_SLAVE( blockingWrite, Pwrite);
      SC_SLAVE( blockingRead, Pread);
    }
};


//
// producer.h  ::  producer
//

#ifndef SYSTEMC_H
#include "systemc.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

SC_MODULE(producer)
{
  // port declaration
  sc_in<int> full;
  sc_outmaster<int> Pout; // refinable port
  sc_in_clk clk;

  // Internal variable
  int val;

  // outmaster process
  void producer_thread()
    {
      while (true)
        {
          val += 2;
          Pout = val;
          wait();  // wait for pos_edge clock event
        }
    }

  SC_CTOR(producer)
    {
      SC_THREAD(producer_thread);
      sensitive_pos << clk;
      val = 0;
```

```
    }
};


//
// consumer.h  ::  consumer module
//

#ifndef SYSTEMC_H
#include "systemc.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

SC_MODULE(consumer)
{
  // declare ports
  sc_in<int>       empty;
  sc_inmaster<int> Cin;
  sc_in_clk        clk;

  // Internal variable
  int x;

  // inmaster process
  void consumerFunc()
    {
      while (true)
        {
          x = Cin;
          wait();  // wait for pos_edge clk
        }
    }

  SC_CTOR(consumer)
    {
      SC_THREAD(consumerFunc);
      sensitive_pos << clk;
    }
};


//
// top.cc : contains sc_main ; Instantiates FIFO design
//

#ifndef SYSTEMC_H
#include "systemc.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

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

#ifndef BUFFER_H
```

```
#include "buffer.h"
#endif

int sc_main(int ac, char *av[] )
{

  // declare channels/signals

  sc_link_mp<int> BW;
  sc_link_mp<int> BR;

  // create clocks with diff frequencies
  sc_clock clock1 ("Clock1", 5, 0.5, 0.3,true);
  sc_clock clock2 ("Clock2", 40, 0.5, 0.1,true);

  // instantiate all blocks and connect to channels, signals

  producer      p1("Master");
  fifo          f1("fifo");
  consumer      c1("Slave");

  p1.Pout(BW);
  p1.clk(clock1);
  f1.Pwrite(BW);
  f1.Pread(BR);
  c1.Cin(BR);
  c1.clk(clock2);

  sc_start(100);
  // return zero if no error
  return 0;
}
```

Output results from simulation of the FIFO example are shown below. Note that the FIFO buffer gets filled up by blockingWrite's starting from the left and gets emptied by blockingRead from the right. When the buffer is full, blockingWrite is blocked and unblocks when an item is removed from the buffer.

SIMULATION OUTPUT

```
fifo:blockingRead called
Writing into buffer: item = 2
  Array after shifting = 2 0 0 0 0 0 0 0 0 0
Writing into buffer: item = 4
  Array after shifting = 4 2 0 0 0 0 0 0 0 0
Writing into buffer: item = 6
  Array after shifting = 6 4 2 0 0 0 0 0 0 0
Writing into buffer: item = 8
  Array after shifting = 8 6 4 2 0 0 0 0 0 0
  Item read = 2
Writing into buffer: item = 10
  Array after shifting = 10 8 6 4 0 0 0 0 0 0
Writing into buffer: item = 12
  Array after shifting = 12 10 8 6 4 0 0 0 0 0
Writing into buffer: item = 14
  Array after shifting = 14 12 10 8 6 4 0 0 0 0
Writing into buffer: item = 16
  Array after shifting = 16 14 12 10 8 6 4 0 0 0

fifo:blockingRead called
```

```
  Item read = 4
Writing into buffer: item = 18
  Array after shifting = 18 16 14 12 10 8 6 0 0 0
Writing into buffer: item = 20
  Array after shifting = 20 18 16 14 12 10 8 6 0 0
Writing into buffer: item = 22
  Array after shifting = 22 20 18 16 14 12 10 8 6 0
Writing into buffer: item = 24
  Array after shifting = 24 22 20 18 16 14 12 10 8 6

fifo:blockingRead called
  Item read = 6
Writing into buffer: item = 26
  Array after shifting = 26 24 22 20 18 16 14 12 10 8
```

15.3.2 FIFO EXAMPLE AT THE BCA LEVEL

The same FIFO at the BCA level is shown below. Full-handshake protocol is used between the producer/ consumer threads and the FIFO. Enable-handshake protocol would not work here since the FIFO is blocking. The processes are shown after the protocol modules have been in-lined inside the respective modules, i.e. the producer, consumer and the FIFO modules. For the sake of brevity, we did not show the consumer process, which is very analogous to the producer.

```
//
// producer_bp.h  ::  producer with fullHandshake protocol
//

#include "systemc.h"

SC_MODULE(producer)
{
  // internal variable
  int val;
  int state;  // state variable of FSM

  // port declaration
  sc_outmaster<int, sc_fullHandshake<int> > Pout; // bus port
  sc_in_clk clk;

  // outmaster process
  void producer_thread()
    {
      state = 0;
      Pout.req = 0;
      while (true)
        {
          wait();
          switch(state)
            {
            case 0:
              Pout.req = 0;
              if (Pout.ack == 0)
                {
                  state = 1;
                }
              break;
            case 1:
```

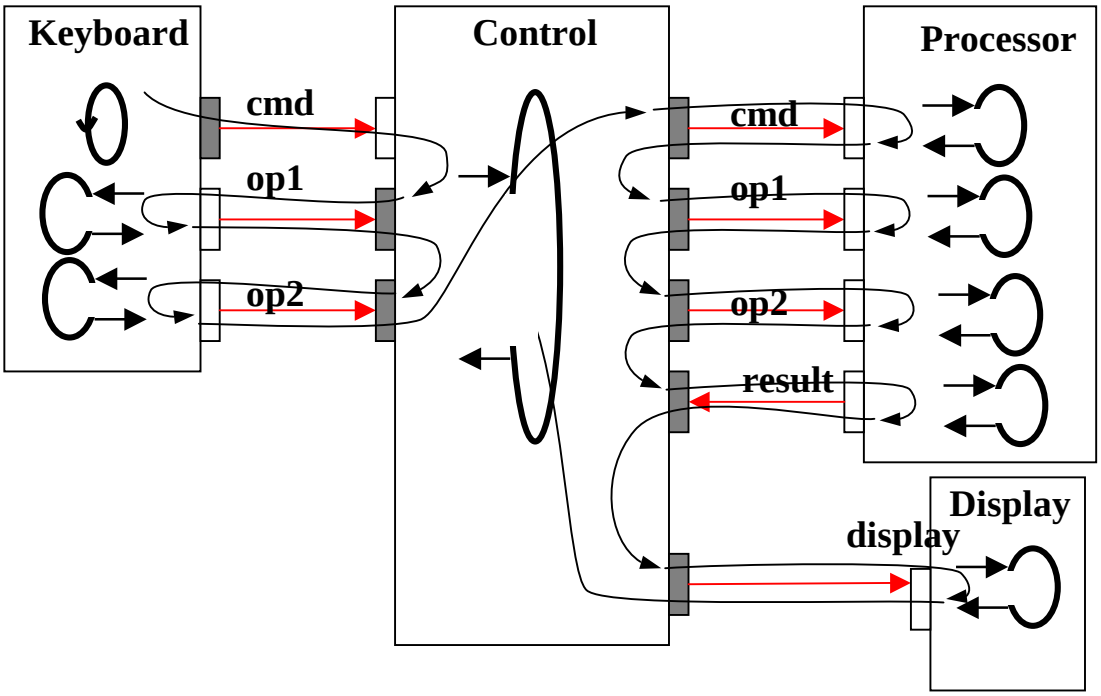
```
              Pout.req = 1;
              val += 2;
              Pout.d = val;
              state = 2;
              break;
            case 2:
              if (Pout.ack==1)
                {
                  state = 0;
                  Pout.req = 0;
                }
              break;
            }
        }
    }

  SC_CTOR(producer)
    {
      SC_THREAD(producer_thread);
      sensitive_pos << clk;
      val = 0;
    }
};


// File Name : fifo_bp.h
// A module with inslave and outslave fullHandshake bus ports,
// and a fifo store

#include "systemc.h"

#ifndef BUFFER_H
#include "buffer.h"
#endif

SC_MODULE(fifo)
{
  // ports
  sc_in_clk wclk;
  sc_in_clk rclk;
  sc_inslave<int, sc_fullHandshake<int> > Pwrite; // slave port
  sc_outslave<int, sc_fullHandshake<int> > Pread; // slave port

  buffer<int> buf;
  int item;

  // slave methods
  void blockingWrite()
    {
      Pwrite.ack = 0;
      while(1)
        {
          wait();  // wait for wclk
          if (Pwrite.req)
            {
              if (buf.full() )
                {
                  // wait for buffer not full;
                  do {wait(); }  // wait for wclk
                  while( buf.full());
                }
```

```
            item = Pwrite.d; // read from slave port
            cout << "Writing into buffer: item = " << item << endl;

            buf.put(item);
            Pwrite.ack = 1;
            while(Pwrite.req == 1)
              wait();  // wait for wclk
            Pwrite.ack = 0;
          }
      }
  }

  void blockingRead()
  {
    Pread.ack = 0;
    while(1)
      {
        wait(); // wait for clock
        if (Pread.req)
          {
            cout << "\nfifo:blockingRead called" << endl;
            if (buf.empty())
              {  // wait for not empty buffer
                do {wait();} // wait for rclk
                while (buf.empty() );
              }

            item = buf.get();
            // cout << "  Item read = " << item << endl;
            Pread.d = item; // write to slave port
            Pread.ack = 1;
            while(Pread.req == 1)
            wait(); //wait for rclk
            Pread.ack = 0;
          }
      }
  }

  SC_CTOR(fifo)
  {
    SC_THREAD(blockingWrite);
    sensitive_pos << wclk;
    SC_THREAD(blockingRead);
    sensitive_pos << rclk;
  }
};
```

15.3.3 EXAMPLE: SIMPLE ARITHMETIC PROCESSOR

In this example, the key module (keyboard) serves as a test bench to generate inputs for a simple processor that takes a command for an arithmetic operation together with two operands and calculates the results. The control module (ctrl) dispatches the commands coming from the keyboard to the processor, then gets the results from the processor and sends it to the display module for display. There is a single concurrent thread (key_input) in the key module. All other processes are slaves executing in this single concurrent thread. The single thread of execution is shown in Figure 15-13.

Figure 15-13: Simple arithmetic processor.




```
#include "systemc.h"

# define ADD 1
# define SUBTRACT 2
# define MULTIPLY 3
# define MYMAX 5

SC_MODULE(key)
{
  // ports
  sc_outmaster<int > cmd;
  sc_outslave<int > op1;
  sc_outslave<int > op2;
```

```
  static int array1[MYMAX];
  static int array2[MYMAX];
  int i, state;
  int scratch;

  void key_input ()
    { // implements concurrent thread
      // runs forever
      while (true)
        {
          switch (state)
            {
            case 0:
              scratch = ADD;
              cmd = scratch;
              if (i == MYMAX-1)
                state = 1;
              break;
            case 1:
              cmd = SUBTRACT;
              if (i == MYMAX-1)
                state = 2;
              break;
            case 2:
              cmd = MULTIPLY;
              if (i == MYMAX-1)
                state = 0;
              break;
            }
          if (i == MYMAX-1)
            i = 0;
          else i++;
        }
    }

  void slave_to_op1 ()
    {
      op1 = array1[i];
    }

  void slave_to_op2 ()
    {
      op2 = array2[i];
    }

  // constructor
  SC_CTOR(key)
    {
      SC_THREAD(key_input); // concurrent thread
      SC_SLAVE(slave_to_op1, op1);
      SC_SLAVE(slave_to_op2, op2);
      state = 0;
      i = 0;
    }
};


#include "systemc.h"

SC_MODULE(ctrl)
{
```

```
  // ports
  sc_inslave<int > cmd_in;
  sc_inmaster<int > op1_in;
  sc_inmaster<int > op2_in;

  sc_outmaster<int > op1_out;
  sc_outmaster<int > op2_out;
  sc_outmaster<int > cmd_out;
  sc_inmaster<int > result;

  sc_outmaster<int > display;

  // internal variables
  int operand1;
  int operand2;
  int cmd_2;

  // slave "methods"
  void slave_to_cmd_in ()
    {
      operand1 = op1_in;
      operand2 = op2_in;

      op1_out = operand1;
      op2_out = operand2;
      cmd_out = cmd_in;
      display = result;

      cmd_2 = cmd_in;

      cout << "\n operand1 = " << operand1 << "  operand2 = " << operand2
           << "  cmd = " << cmd_in;
    }

  SC_CTOR(ctrl)
    {
      SC_SLAVE(slave_to_cmd_in, cmd_in);
    }
};


#include "systemc.h"

SC_MODULE(fun)
{
  // ports
  sc_inslave<int > op1;
  sc_inslave<int > op2;
  sc_inslave<int > cmd;
  sc_outslave<int > result;

  // internal variables
  int operand1;
  int operand2;
  int command;

  void slave_to_op1 ()
    {
      operand1 = op1;           // read operand 1
    }
```

```
  void slave_to_op2 ()
    {
      operand2 = op2;           // read operand 2
    }

  void slave_to_cmd ()
    {
      command = cmd;            // read command
    }

  void slave_to_result ()
    {
      switch (command)
        {
        case 1 :
          result = operand1 + operand2; // Add
          break;
        case 2 :
          result = operand1 - operand2; // Subtract
          break;
        case 3 :
          result = operand1 * operand2; // Multiply
          break;
        default :
          printf ("\nfun: Undefined operation\n");
          break;
        }
    }

  // constructor
  SC_CTOR(fun)
    {
      SC_SLAVE(fun,slave_to_op1, op1);
      SC_SLAVE(slave_to_op2, op2);
      SC_SLAVE(slave_to_result, result);
      SC_SLAVE(slave_to_cmd, cmd);
    }
};


#include "systemc.h"

SC_MODULE(disp)
{
  // ports
  sc_inslave<int > din;

  // internal variables
  int first_time;

  // slave "method"
  void slave_to_din ()
    {
      if (first_time) first_time = 0;
      else cout << "     Result = " << din << endl;
    }

  // constructor
  SC_CTOR(disp)
    {
      SC_SLAVE(slave_to_din, din);
```

```
      first_time = 1;
    }
};


#include "systemc.h"

#include "key.h"
#include "ctrl.h"
#include "fun.h"
#include "disp.h"

int key::array1[MYMAX] = { 1,2,3,4,5 };
int key::array2[MYMAX] = {1,3,5,7,9 };
int sc_main(int argc, char* argv[])
{
  sc_link_mp<int > cmd;
  sc_link_mp<int > in1;
  sc_link_mp<int > in2;
  sc_link_mp<int > op1out;
  sc_link_mp<int > op2out;
  sc_link_mp<int > cmd_out;
  sc_link_mp<int > result;
  sc_link_mp<int > display;

  key k1("key");
  k1.cmd(cmd);
  k1.op1(in1);
  k1.op2(in2);
  k1.clk(clock);

  ctrl c1("ctrl");
  c1.cmd_in(cmd);
  c1.op1_in(in1);
  c1.op2_in(in2);
  c1.op1_out(op1out);
  c1.op2_out(op2out);
  c1.cmd_out(cmd_out);
  c1.result(result);
  c1.display(display);

  fun f1("fun");
  f1.op1(op1out);
  f1.op2(op2out);
  f1.cmd(cmd_out);
  f1.result(result);

  disp d1("display");
  d1.din(display);

  sc_start(50);

  return (0);
};
```

SIMULATION RESULTS

```
operand1 = 0  operand2 = 0  cmd = 1     Result = 0

operand1 = 1  operand2 = 1  cmd = 1     Result = 2
```

```
operand1 = 2  operand2 = 3  cmd = 1     Result = 5

operand1 = 3  operand2 = 5  cmd = 1     Result = 8

operand1 = 4  operand2 = 7  cmd = 1     Result = 11

operand1 = 5  operand2 = 9  cmd = 2     Result = -4

operand1 = 1  operand2 = 1  cmd = 2     Result = 0

operand1 = 2  operand2 = 3  cmd = 2     Result = -1

operand1 = 3  operand2 = 5  cmd = 2     Result = -2

operand1 = 4  operand2 = 7  cmd = 2     Result = -3

operand1 = 5  operand2 = 9  cmd = 3     Result = 45
```

15.3.4 SIMPLE PROCESSOR AT THE BCA LEVEL

For the sake of brevity, we're showing only two modules of this example at the BCA level.

```
#include "systemc.h"

# define ADD 1
# define SUBTRACT 2
# define MULTIPLY 3
# define MYMAX 5

SC_MODULE(key)
{
  // ports
  sc_outmaster<int, sc_fullHandshake<int> > cmd;
  sc_outslave<int, sc_enableHandshake<int> > op1;
  sc_outslave<int, sc_enableHandshake<int> > op2;

  sc_in_clk    clk;
  sc_in<int>   rstp;

  static int array1[MYMAX];
  static int array2[MYMAX];

  int i, state;

  void key_f ()
    {
      if (rstp == 0)
        {
          state = i = 0;
          cmd.d = 1;
          cmd.req = false;
        }
      else
        {
          switch (state)
            {
            case 0:
              cmd.req = false;
```

```
              if (cmd.ack == false)
                state = 1;
              else
                state = 0;
              break;
            case 1:
              cmd.req = true;
              cmd.d = ADD;
              state = 2;
              break;
            case 2:
              cmd.req = true;
              cmd.d = SUBTRACT;
              state = 3;
              break;
            case 3:
              cmd.req = true;
              cmd.d = MULTIPLY;
              state = 0;
              break;
            }

          if (op1.en == true)
            {
              if (i == MYMAX-1)
                i = 0;
              else
                i++ ;
              op1.d = array1[i];
            }
          if (op2.en == true)
            {
              op2.d = array2[i];
            }
        }
    }

  // constructor
  SC_CTOR(key)
    {
      SC_METHOD(key_f)
      sensitive << clk;
    }
};


#include "systemc.h"

# define idle      0
# define transmit  1

SC_MODULE(ctrl)
{
  // ports
  sc_inslave<int, sc_fullHandshake<int> > cmd_in;
  sc_inmaster<int, sc_enableHandshake<int> > op1_in;
  sc_inmaster<int, sc_enableHandshake<int> > op2_in;

  sc_outmaster<int, sc_enableHandshake<int> > op1_out;
  sc_outmaster<int, sc_enableHandshake<int> > op2_out;
  sc_outmaster<int, sc_enableHandshake<int> > cmd_out;
```

```
sc_inmaster<int, sc_enableHandshake<int> > result;

sc_outmaster<int, sc_enableHandshake<int> > display;

// sc_inslave<bool> rstp;
// sc_inslave<bool> clk;
//
sc_in_clk clk;
sc_in<int> rstp;

static int state;

void ctrl_f ()
  {
    cmd_in.ack = false;
    if (rstp == 0)
      {
        state = idle;
        cmd_in.ack = false;
      }
    else
      {
        if (cmd_in.req == true)
          {
            //state = transmit;
            cmd_out.en = true;

            op1_in.en = true;
            op2_in.en = true;
            op1_out.en = true;
            op2_out.en = true;
            result.en = true;
            display.en = true;

            int x = cmd_in.d;
            cmd_out.d = x;
            //  cout << "cmd =   "<< x ;

            x = op1_in.d;
            op1_out.d = x;
            // cout << " op1 = "<< x;

            x = op2_in.d;
            op2_out.d = x;
            // cout << " op2 = "<< x << endl;

            x = result.d;
            display.d = x;
            // cout << "    Result = " << x << endl;
            // cmd_in.ack = true;
          }
        else
          {
            int x = result.d;
            display.d = x;
            //  cout << "    Result2 = " << x << endl;
          }
      }
  }

// constructor of the module
```

```
  SC_CTOR(ctrl)
    {
      SC_METHOD(ctrl_f);
      sensitive << clk;
    }
};
```

```
    // which is true iff a signal value changed in the previous delta cycle,
    // the scheduler maintains a delta cycle counter. This counter is never
    // reset.

    unsigned long long delta_count;
};


class sc_event
{
public:

    sc_event()
        : notify_type( NONE ), when_to_notify( 0 ) {}

    notify() // an immediate notification
        { trigger(); }

    notify( sc_time t ) // a timed notification
    {
        if( notify_type == DELAYED )
            return;

        if( t == SC_ZERO_TIME ) {
            delayed_notify();
            return;
        }

        if( notify_type == TIMED ) {
            if( when_to_notify <= sc_now() + t )
                return;

            remove this event from its current notification set ;
        }

        notify_type = TIMED;
        when_to_notify = sc_now() + t;
        add this to timed_notifications at time when_to_notify ;
    }

    ~sc_event()
    {
        for t in waiting_dynamic_threads
            t.remove_dynamic_event( *this );

        for t in waiting_static_threads
            t.remove_static_event(*this);

        if( notify_type != NONE )
            remove this event from its current notification set ;
    }

private:

    friend class sc_thread;

    waiting_dynamic_threads: A set of references to
            threads dynamically waiting on this event ;
    waiting_static_threads: A set of references to
            threads statically waiting on this event ;
```

```
    enum notify_t { NONE, DELAYED, TIMED };
    notify_t notify_type;
    sc_time  when_to_notify;

    delayed_notify() // a delayed notification (one delta cycle)
    {
        if( notify_type == DELAYED )
            return;

        if( notify_type == TIMED )
            remove this event from its current notification set ;

        notify_type = DELAYED;
        add this event to the delayed_notifications set ;
    }

    trigger()
    {
        if( notify_type != NONE )
            remove this event from its current notification set ;

        for t in waiting_static_threads
            t.trigger( this, false );

        for t in waiting_dynamic_threads
            t.trigger( this, true );

        reset();
    }

    reset()
    {
        notify_type = NONE;
        when_to_notify = 0;
        clear waiting_dynamic_threads set ;
    }

    add_dynamic_thread( sc_thread& t )
        { add t to waiting_dynamic_threads ; }

    remove_dynamic_thread( sc_thread& t )
        { remove t from waiting_dynamic threads ; }

    add_static_thread( sc_thread& t )
        { add t to waiting_static_threads ; }

    remove_static_thread( sc_thread& t )
        { remove t from waiting_static_threads ; }
};


class sc_thread  // base class for a thread or process
{
public:

    wait( e_col: A reference to a collection of event references )
    {
        if( is_sc_method ) {
            cerr << "Can't call wait() from an SC_METHOD\n";
            return;
        }
```

```
    add each event in e_col into dynamic_events ;

    for e in dynamic_events
        e.add_dynamic_thread( this );

    set "and_sensitive" to "true" if collection e_col is an
        AND collection ;

    waiting_dynamically = true;

    save the context (ie. stack) of this thread ;

    suspend execution in the current context and resume execution
        in the previous context (the scheduler's context) ;
}

wait( sc_time t )
{
    sc_event e;
    e.notify( t );
    wait( e );  // calls wait( e_col ) via a conversion
}

wait( sc_time t, e_col: A reference to a collection of event references)
{
    sc_event e;
    e.notify( t );

    if( e_col is not an "AND" collection ) {
        add e to e_col ;
        wait( e_col );
        remove e from e_col ;
    }
    else {
        and_timeout_event = &e;
        wait( e_col );
    }
}

wait()  // ie. wait on static sensitivity
{
    if( is_sc_method ) {
        cerr << "Can't call wait() from an SC_METHOD\n";
        return;
    }

    and_sensitive = false
    waiting_dynamically = false;

    save the context (ie. stack) of this thread ;

    suspend execution in the current context and resume execution
        in the previous context (the scheduler's context) ;
}

sc_thread( bool sc_method_flag )
{
    is_sc_method = sc_method_flag;
    waiting_dynamically = false;
    and_sensitive = false;
```

```
        if( ! is_sc_method )
            create a new context (ie execution stack) for this thread ;
    }

    ~sc_thread()
    {
        unwait();

        for e in static_events
            remove_from_static_sensitivity( e );

        remove "this" from runnable or not_runnable ;

        if( ! is_sc_method )
            delete the context (ie execution stack) for this thread ;
    }

    add_to_static_sensitivity( sc_event& e )
    {
        if( e already in static_events )
            return;

        add e to static_events ;
        e.add_static_thread( this );
    }

    remove_from_static_sensitivity( sc_event& e )
    {
        remove e from static_events ;
        e.remove_static_thread( this );
    }

private:

    friend class sc_event;
    friend class scheduler;

    dynamic_events: A set of references to events that
            this thread is dynamically sensitive to ;
    static_events: A set of references to events that
            this thread is statically sensitive to ;

    bool and_sensitive;       // true iff currently waiting
                              // on "AND" event collection
    sc_event* and_timeout_event;
    bool is_sc_method;        // true iff this thread is an SC_METHOD
                              // rather than SC_THREAD
    bool waiting_dynamically; // true iff this thread is currently
                              // waiting on dynamic_events

    unwait()
    {
        for e in dynamic_events
            e.remove_dynamic_thread( this );

        clear dynamic_events ;
        and_sensitive = false;
        and_timeout_event = 0;
    }
```

```
    resume()
    {
        if( is_sc_method ) {
            call the SC_METHOD "start method" ;
        }
        else {
            suspend execution of the current thread (i.e. the scheduler's
                context) and resume execution of this thread in its
                previously saved context ;
            // (We automatically return to this point when this thread next
            //  waits, and we will be executing in the scheduler's context
            //  again.)
        }
    }

    trigger( sc_event& e, bool dynamic )
    {
        if( dynamic != waiting_dynamically )
            return;

        if( is_sc_method ) {
            move "this" from not_runnable to runnable ;
            return;
        }

        if( ! and_sensitive ) {
            move "this" from not_runnable to runnable ;
           unwait();
        }
        else {
            if( &e == and_timeout_event ) {
                move "this" from not_runnable to runnable ;
                unwait();
                return;
            }

            remove e from dynamic_events ;

            if( dynamic_events is empty ) {
                move "this" from not_runnable to runnable ;
                unwait();
            }
        }
    }
};


void scheduler::execute()
{
    // Initialization Phase:
    // Note that all threads except SC_CTHREADS are run once at
    // initialization in an unspecified order.

    for t in ( all threads except SC_CTHREADS )
        t.resume();

    while( true ) {

        // Evaluate Phase:

        while( runnable is not empty ) {
```

```
            select a thread t in runnable and move it to not_runnable ;
            t.resume();
        }

        // Update Phase:

        execute any pending calls to update() resulting from
            request_update() calls ;

        delta_count ++ ;

        // Process delayed notifications

        if( delayed_notifications is not empty ) {
            for e in delayed_notifications
                e.trigger();

            continue;
        }

        // Process timed notifications

        if( timed_notifications is empty )
            break;

        _sc_now = the earliest time in timed_notifications ;

        for e in ( set of events to be notified at earliest time in
                   timed_notifications )
            e.trigger();

        delete the earliest time in timed_notifications and the set
            of events to be notified at that time ;
    }
}
```

Scheduler Notes:

The order in which the scheduler selects threads to run within the evaluate and update phases is unspecified and implementation-dependent. However, when the same design is simulated multiple times using the same stimulus and the same version of the simulator, the thread ordering between different runs will not vary.

SystemC will provide simulator command line options that allow the user to randomize the thread ordering within the evaluate phase. This feature is useful for detecting design flaws resulting from inadequate synchronization within design specifications.