

Chapter 2

Overview of SystemC

The previous chapters gave a brief context for the application of SystemC. This chapter presents an overview of the SystemC language elements. Details are discussed in-depth in subsequent chapters.

Despite our best efforts not to use any part of the language before it is fully explained, some chapters may occasionally violate this goal due to the interrelated nature of SystemC. This chapter briefly discusses the major components of SystemC and their general usage and interactions as a way of giving context for the subsequent chapters.

The following diagram, Fig. 2.1, illustrates the major components of SystemC. As a form of roadmap, we have included a duplicate of this diagram at the beginning of each new chapter. Bolded type indicates the topics discussed within that chapter.

For the rest of this chapter, we will discuss all of the components within the figure that are outlined in bold; but first, we will discuss the mechanics of the SystemC development environment.

SystemC addresses the modeling of both hardware and software using C++. Since C++ already addresses most software concerns, it should come as no surprise that SystemC focuses primarily on non-software issues. The primary application area for SystemC is the design of electronic systems. However, SystemC also provides generic modeling constructs that can be applied to non-electronic systems¹

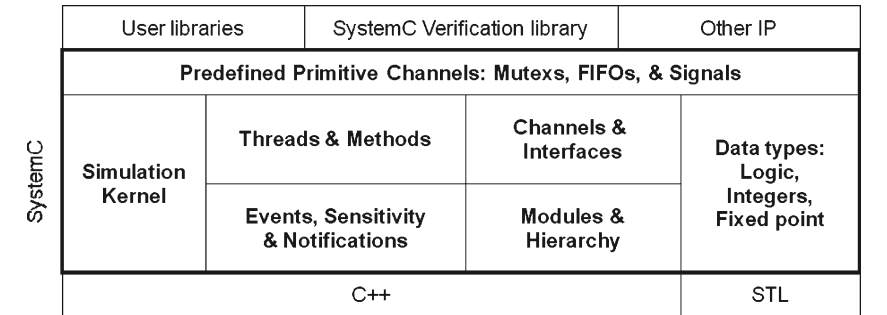


Fig. 2.1 SystemC language architecture

¹For example, the book, *Microelectrofluidic Systems: Modeling and Simulation* by Tianhao Zhang et al., CRC Press, ISBN: 0849312760, describes applying SystemC to a non-electronic system.

2.1 C++ Mechanics for SystemC

We would like to start with the obligatory *Hello_SystemC* program; but first we will look at the mechanics of compiling and executing a SystemC program or model.

As stated before, SystemC is a C++ class library. Therefore, to compile and run a *Hello_SystemC* program, one must have a working C++ and SystemC environment.

The components of a SystemC environment include a:

- SystemC-supported platform
- SystemC-supported C++ compiler
- SystemC library (downloaded and compiled for your C++ compiler)
- Compiler command sequence make file or equivalent

The latest Open SystemC Initiative (OSCI) SystemC release (2.2 at this writing) is available for free from www.systemc.org. The download contains scripts and make files for installation of the SystemC library as well as SystemC source code, examples, and documentation. The install scripts are compatible with the supported operating systems, and the scripts are relatively straightforward to execute by carefully following the documentation.

The latest OS requirements can be obtained from the download in a ReadMe file currently called *INSTALL*. SystemC is supported on various versions of Sun Solaris, Linux, HP/UX, Windows, and Mac OS X. At this time, the OS list is limited by the use of minor amounts of assembly code that is used for increased simulation performance in the SystemC simulation kernel. The current release is also supported for various C++ compilers including GNU C++, Sun C++, HP C++, and Visual C++. The currently supported compilers and compiler versions can also be obtained from the *INSTALL* ReadMe file in the SystemC download.

For beginners, this OS and compiler list should be considered exhaustive. Notably, some hardy souls have ported various SystemC versions to other unsupported operating systems and C++ compilers. In addition to one of these platforms and compilers, you will need GNU make installed on your system to compile and quickly install the SystemC library with the directions documented in the *INSTALL* file.

The flow for compiling a SystemC program or design is very traditional and is illustrated in Fig. 2.2 for GNU C++. Most other compilers will be similar. The C++ compiler reads each of the SystemC code file sets separately and creates an object file (the usual file extension is *.o*). Each file set usually consists of two files, typically with standard file extensions. We use *.h* and *.cpp* as file extensions, since these are the most commonly used in C++. The *.h* file is generally referred to as the header file and the *.cpp* file is often called the implementation file.

Note that the compiler and linker need to know two special pieces of information. First, the compiler needs to know where the SystemC header files are located. Second, the linker needs to know the location of the compiled SystemC libraries. This information is typically passed by providing an environment variable named

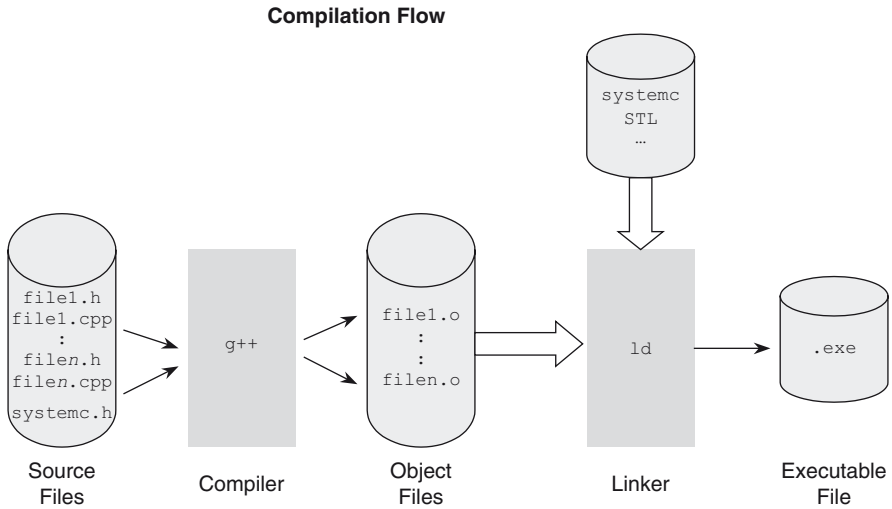


Fig. 2.2 SystemC compilation flow

SYSTEMC and by ensuring the makefile rules use the information.² If using *gcc*, the command probably looks something like Fig. 2.3.

The downloadable examples available from our web site include a makefile setup for Linux and *gcc*. Please refer to your C++ tool manuals for more information.

```

g++ -I$(SYSTEMC)/include \
    -L$(SYSTEMC)/lib-$(ARCH) -lsystemc \
    $(SRC)
    
```

Fig. 2.3 Partial *gcc* options to compile and link SystemC

After creating the object files, the compiler (actually the loader or linker) will link your object files and the appropriate object files from the SystemC library (and other libraries such as the standard template library or STL). The resulting file is usually referred to as an executable, and it contains the SystemC simulation kernel and your design functionality.

For the hardcore engineer types, you now have everything you need to compile and run a *Hello_SystemC* program; we have provided the obligatory program in Fig. 2.4 Keywords for both C++ and SystemC are in **bold**. The rest of you now have an overview of how to compile and run the code examples in this book as well as your own SystemC creations. Everyone is now ready to dive into the language itself.

²For some installations, dynamic libraries may also be referenced if using the SystemC Verification library.

```

#include <systemc>
SC_MODULE(Hello_SystemC) { // declare module class

    SC_CTOR(Hello_SystemC) { // create a constructor
        SC_THREAD(main_thread); // register the process
    } //end constructor

    void main_thread(void) {
        SC_REPORT_INFO(" Hello SystemC World!");
    }
};

int sc_main(int sc_argc, char* sc_argv[]) {

    //create an instance of the SystemC module
    Hello_SystemC HelloWorld_i("HelloWorld_i");

    sc_start(); // invoke the simulator

    return 0;
}

```

Fig. 2.4 Hello_SystemC program example

2.2 SystemC Class Concepts for Hardware

SystemC provides mechanisms crucial to modeling hardware while using a language environment compatible with software development. SystemC provides several hardware-oriented constructs that are not normally available in a software language; however, these constructs are required to model hardware. All of the constructs are implemented within the context of the C++ language. This section looks at SystemC from the viewpoint of the hardware-oriented features. The major hardware-oriented features implemented within SystemC include:

- Time model
- Hardware data types
- Module hierarchy to manage structure and connectivity
- Communications management between concurrent units of execution
- Concurrency model

The following sections briefly discuss the implementation of these concepts within SystemC.

2.2.1 Time Model

SystemC tracks time with 64 bits of resolution using a class known as `sc_time`. Global time is advanced within the kernel. SystemC provides mechanisms to obtain the current time and implement specific time delays. To support ease of

use, an enumerated type defines several natural time units from seconds down to femtoseconds.

For those models that require a clock, a class called `sc_clock` is provided. Since many applications in SystemC do not require a clock (but do require a notion of time), the clock discussion is deferred to later chapters of the book. Additionally, clocks do not add to the fundamental understanding of the language. By the later chapters, you should be able to implement the clock class yourself with the fundamentals learned throughout the book.

2.2.2 Hardware Data Types

The wide variety of data types required by digital hardware are not provided inside the natural boundaries of C++ native data types, which are typically 8-, 16-, 32-, and 64-bit entities.

SystemC provides hardware-compatible data types that support explicit bit widths for both integral and fixed-point quantities. Furthermore, digital hardware requires non-binary representation such as tri-state and unknowns, which are provided by SystemC.

Finally, hardware is not always digital. SystemC does not currently directly support analog hardware; however, a working group has been formed to investigate the issues associated with modeling analog hardware in SystemC. For those with immediate analog issues, it is reasonable to model analog values using floating-point representations and provide the appropriate behavior.

2.2.3 Hierarchy and Structure

Large designs are almost always broken down hierarchically to manage complexity, easing understanding of the design for the engineering team. SystemC provides several constructs for implementing hardware hierarchy. Hardware designs traditionally use blocks interconnected with wires or signals for this purpose. For modeling hardware hierarchy, SystemC uses the module entity interconnected to other modules using channels. The hierarchy comes from the instantiation of module classes within other modules.

2.2.4 Communications Management

The SystemC channel provides a powerful mechanism for modeling communications. Conceptually, a channel is more than a simple signal or wire. Channels can represent complex communications schemes that eventually map to significant

hardware such as the AMBA bus³. At the same time, channels may also represent very simple communications such as a wire or a FIFO (first-in first-out queue).

The ability to have several quite different channel implementations used interchangeably to connect modules is a very powerful feature. This feature enables an implementation of a simple bus replaced with a more detailed hardware implementation, which is eventually implemented with gates.

SystemC provides several built-in channels common to software and hardware design. These built-in channels include locking mechanisms like mutex and semaphores, as well as hardware concepts like FIFOs, signals and others.

Finally, modules connect to channels and other modules via port classes.

2.2.5 Concurrency

Concurrency in a simulator is always an illusion. Simulators execute the code on a single physical processor. Even if you did have multiple processors performing the simulation, the number of units of concurrency in real hardware design will always outnumber the processors used to do the simulation by several orders of magnitude. Consider the problem of simulating the processors on which the simulator runs.

Simulation of concurrent execution is accomplished by simulating each concurrent unit. Each unit is allowed to execute until simulation of the other units is required to keep behaviors aligned in time. In fact, the simulation code itself determines when the simulator makes these switches by the use of events. This simulation of concurrency is the same for SystemC, Verilog, VHDL, or any other hardware description languages (HDLs). In other words, the simulator uses a cooperative multitasking model. The simulator merely provides a kernel to orchestrate the swapping of the various concurrent elements, called simulation processes. SystemC provides a simulation kernel that will be discussed lightly in the last section of this chapter and more thoroughly in the rest of the book.

2.2.6 Summary of SystemC Features for Hardware Modeling

SystemC implements the structures necessary for hardware modeling by providing constructs that enable concepts of time, hardware data types, hierarchy and structure, communications, and concurrency. This section has presented an overview of SystemC relative to a generic set of requirements for hardware design. We will now give a brief overview of the constructs used to implement these requirements in SystemC.

³See AMBA AHB Cycle-Level Interface Specification at www.arm.com.

2.3 Overview of SystemC Components

In this section, we briefly discuss all the components of SystemC that are highlighted in Fig. 2.1 from the beginning of this chapter, that we will see at the beginning of each chapter throughout the book.

2.3.1 Modules and Hierarchy

Hardware designs typically contain hierarchy to reduce complexity. Each level of hierarchy represents a block. VHDL refers to blocks as entity/architecture pairs, which separate an interface specification from the body of code for each block. In Verilog, blocks are called modules and contain both interface and implementation in the same code.

SystemC separates the interface and implementation similar to VHDL. The C++ notion of header (.h file) is used for the entity and the notion of implementation (.cpp file) is used for the architecture.

Design components are encapsulated as “modules”. Modules are classes that inherit from the `sc_module` base class. As a simplification, the `SC_MODULE` macro is provided.

Modules may contain other modules, processes, and channels and ports for connectivity.

2.3.2 SystemC Threads and Methods

Before getting started, it is necessary to have a firm understanding of simulation processes in SystemC. As indicated earlier, the SystemC simulation kernel schedules the execution of all simulation processes. Simulation processes are simply member functions of `sc_module` classes that are “registered” with the simulation kernel.

Because the simulation kernel is the only caller of these member functions, they need no arguments and they return no value. They are simply C++ functions that are declared as returning a void and having an empty argument list.

An `sc_module` class can also have processes that are not executed by the simulation kernel. These processes are invoked as function calls within the simulation processes of the `sc_module` class. These are normal C++ member functions or class methods.

From a software perspective, processes are simply threads of execution. From a hardware perspective, processes provide necessary modeling of independently timed circuits. Simulation processes are member functions of an `sc_module` that are registered with the simulation kernel. Generally, registration occurs during the elaboration phase (during the execution of the constructor for the

`sc_module` class) using an `SC_METHOD`, `SC_THREAD`, or `SC_CTHREAD`⁴ SystemC macro.

The most basic type of simulation process is known as the `SC_METHOD`. An `SC_METHOD` is a member function of an `sc_module` class where time does not pass between the invocation and return of the function. In other words, an `SC_METHOD` is a normal C++ function that happens to have no arguments, returns no value, and is repeatedly and only called by the simulation kernel.

The other basic type of simulation process is known as the `SC_THREAD`. This process differs from the `SC_METHOD` in two ways. First, an `SC_METHOD` is invoked (or started) multiple times and the `SC_THREAD` is invoked only *once*. Second, an `SC_THREAD` has the option to *suspend* itself and potentially allow time to pass before continuing. In this sense, an `SC_THREAD` is similar to a traditional software thread of execution.

The `SC_METHOD` and `SC_THREAD` are the basic units of concurrent execution. The simulation kernel invokes each of these processes. Therefore, they are never invoked directly by the user. The user indirectly controls execution of the simulation processes by the kernel as a result of events, sensitivity, and notification.

2.3.3 Events, Sensitivity, and Notification

Events, sensitivity, and notification are very important concepts for understanding the implementation of concurrency by the SystemC simulator.

Events are implemented with the SystemC `sc_event` and `sc_event_queue` classes. Events are caused or fired through the event class member function, `notify`. The notification can occur within a simulation process or as a result of activity in a channel. The simulation kernel invokes `SC_METHOD` and `SC_THREAD` when they are sensitive to an event and the event occurs.

SystemC has two types of sensitivity: static and dynamic. Static sensitivity is implemented by applying the SystemC `sensitive` command to an `SC_METHOD` or `SC_THREAD` at elaboration time (within the constructor). Dynamic sensitivity lets a simulation process change its sensitivity on the fly. The `SC_METHOD` implements dynamic sensitivity with a `next_trigger(arg)` command. The `SC_THREAD` implements dynamic sensitivity with a `wait(arg)` command. Both `SC_METHOD` and `SC_THREAD` can switch between dynamic and static sensitivity during simulation.

⁴`SC_CTHREAD` is a special case of `SC_THREAD`. This process type is a thread process that has the requirement of being sensitive to a clock. `SC_CTHREAD` is under consideration for deprecation; however, several synthesis tools depend on it at the time of writing.

2.3.4 *SystemC Data Types*

Several hardware data types are provided in SystemC. Since the SystemC language is built on C++, all of the C++ data types are available. Also, SystemC lets you define new data types for new hardware technology (i.e., multi-valued logic) or for applications other than electronic system design.

These data types are implemented using templated classes and generous operator overloading, so that they can be manipulated and used almost as easily as native C++ data types. Hardware data types for mathematical calculations like `sc_fixed<T>` and `sc_int<T>` allow modeling of complex calculations like DSP functions. These data types evaluate the performance of an algorithm when implemented in custom hardware or in processors without full floating-point capability. SystemC provides all the necessary methods for using hardware data types, including conversion between the hardware data types and conversion from hardware to software data types.

Non-binary hardware types are supported with four-state logic (0,1,X,Z) data types (e.g., `sc_logic`). Familiar data types like `sc_logic` and `sc_lv<T>` are provided for RTL hardware designers who need a data type to represent basic logic values or vectors of logic values.

2.3.5 *Ports, Interfaces, and Channels*

Processes need to communicate with other processes both locally and in other modules. In traditional HDLs, processes communicate via ports/pins and signals or wires. In SystemC, processes communicate using channels or events. Processes may also communicate across module boundaries. Modules may interconnect using channels, and connect via ports. The powerful ability to have interchangeable channels is implemented through a component called an interface. SystemC uses the constructs `sc_port<T>`, `sc_export<T>`, and the base classes `sc_interface`, and `sc_channel` to implement connectivity.

SystemC provides some standard channels and interfaces that are derived from these base types. The provided channels include the synchronization primitives `sc_mutex` and `sc_semaphore`, and the communication channels `sc_fifo<T>`, `sc_signal<T>`, and others. These channels implement the SystemC-provided interfaces `sc_mutex_if`, `sc_semaphore_if`, `sc_fifo_in_if<T>`, `sc_fifo_out_if<T>`, `sc_signal_in_if<T>`, and `sc_signal_inout_if<T>`.

Interestingly, module interconnection happens programmatically in SystemC during the elaboration phase. This interconnection lets designers build regular structures using loops and conditional statements. From a software perspective, elaboration is simply the period of time when modules invoke their constructor methods.

2.3.6 Summary of SystemC Components

Now, it is time to tie together all of the basic concepts that we have just discussed into one illustration, Fig. 2.5 This illustration is used many times throughout the book when referring to the different SystemC components. It can appear rather intimidating since it shows almost all of the concepts within one diagram. In practice, a SystemC module typically will not contain all of the illustrated components.

The figure shows the concept of an `sc_module` that can contain instances of another `sc_module`. An `SC_METHOD` or `SC_THREAD` can also be defined within an `sc_module`.

Communication among modules and simulation processes (`SC_METHOD` and `SC_THREAD`) is accomplished through various combinations of ports, interfaces, and channels. Coordination among simulation processes is also accomplished through events.

We will now give a brief initial overview of the SystemC simulation kernel that coordinates and schedules the communications among all of the components illustrated in Fig. 2.5

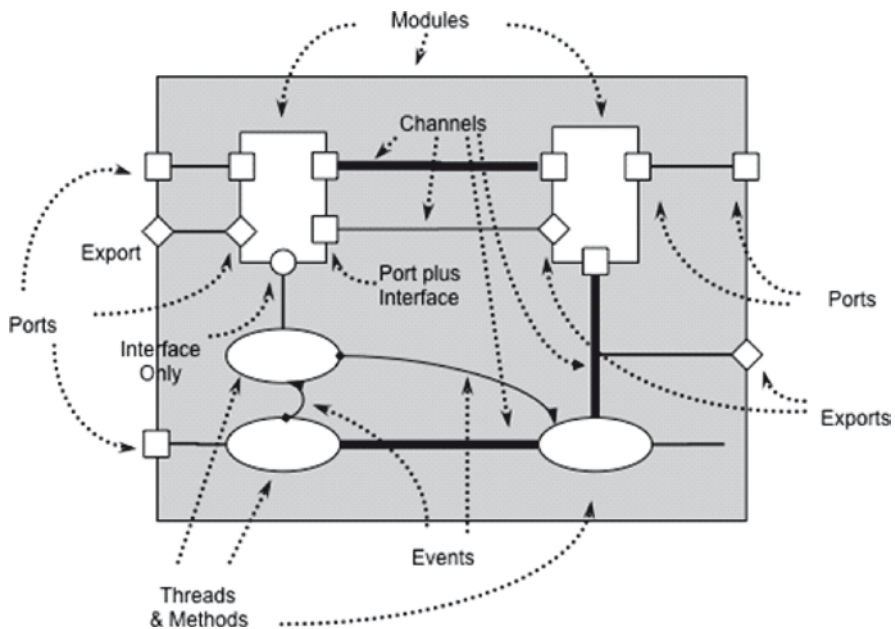


Fig. 2.5 SystemC components

2.4 SystemC Simulation Kernel

The SystemC simulator has two major phases of operation: *elaboration* and *execution*. A third, often minor, phase occurs at the end of execution; this phase could be characterized as post-processing or *cleanup*.

Execution of statements prior to the `sc_start()` function call are known as the elaboration phase. This phase is characterized by the initialization of data structures, the establishment of connectivity, and the preparation for the second phase, execution.

The execution phase hands control to the SystemC simulation kernel, which orchestrates the execution of processes to create an illusion of concurrency.

The illustration in Fig. 2-6 should look very familiar to those who have studied Verilog and VHDL simulation kernels. Very briefly, after `sc_start()`, all simulation processes (minus a few exceptions) are invoked in unspecified deterministic order⁵ during initialization.

After initialization, simulation processes are run when events occur to which they are sensitive. The SystemC simulator implements a cooperative multitasking environment. Once started, a running process continues to run until it yields control. Several simulation processes may begin at the same instant in simulator time. In this case, all of the simulation processes are evaluated and then their outputs are updated. An evaluation followed by an update is referred to as a *delta cycle*.

If no additional simulation processes need to be evaluated at that instant (as a result of the update), then simulation time is advanced. When no additional simulation processes need to run, the simulation ends.

This brief overview of the simulation kernel is meant to give you an overview for the rest of the book. This diagram will be used again to explain important

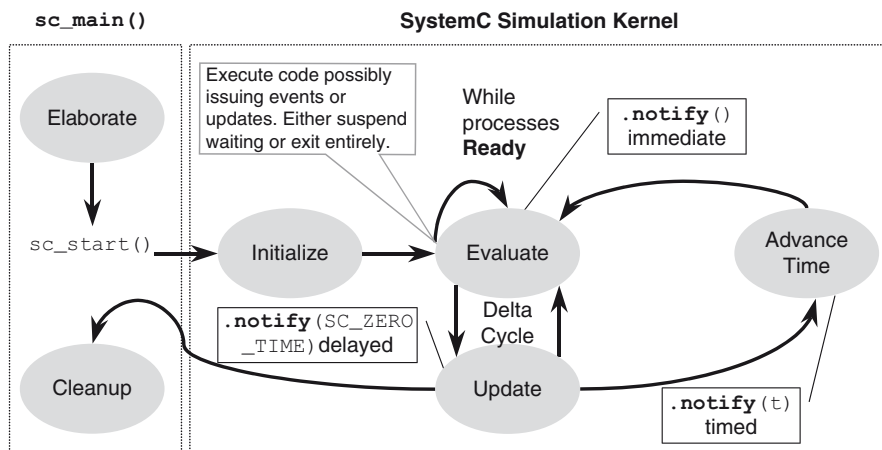


Fig. 2.6 SystemC simulation kernel

⁵Discussed later.

intricacies later. It is very important to understand how the kernel functions to fully understand the SystemC language.

We have provided an animated version of this diagram walking through a small code example at our web site, www.scftgu.com. The IEEE Standard 1666-2005 SystemC LRM (Language Reference Manual) specifies the behavior of the SystemC simulation kernel. This manual is the definitive source about SystemC. We encourage the reader to use any or all of these resources during their study of SystemC to fully understand the simulation kernel.