

An Introduction to System Level Modeling in SystemC 2.0

By Stuart Swan, Cadence Design Systems, Inc.
May 2001

Abstract

SystemC is a new modeling language based on C++ that is intended to enable system level design and IP exchange. This tutorial paper briefly reviews the hardware modeling features available in SystemC 1.0 and then introduces the new system level modeling features in SystemC 2.0. A small design example is presented that demonstrates how the new features facilitate system level design tasks such as communication refinement and mapping of design specifications to hardware and software implementations. Also discussed is how the new modeling features enable a wide variety of models of computation to be cleanly expressed within SystemC.

Introduction

The emergence of the system-on-chip (SoC) era is creating many new challenges at all stages of the design process. At the systems level, engineers are reconsidering how designs are specified, partitioned and verified. Today, with systems and software engineers programming in C/C++ and their hardware counterparts working in hardware description languages such as VHDL and Verilog, problems arising from the use of different design languages, incompatible tools and fragmented tool flows are becoming common.

Momentum is building behind the SystemC language and modeling platform, based on C++, as the solution for representing functionality, communication, software and hardware at various system levels of abstraction. The reason is clear: increasing design complexity demands very fast executable specifications to validate system concepts, and only C/C++ delivers adequate levels of abstraction, hardware/software integration and performance. System design today also demands a single common language and modeling foundation in order to make a market for interoperable system-level design tools, services and IP a reality.

In response to these needs, SystemC has been developed as a standardized modeling language intended to enable system level design and IP exchange at multiple abstraction levels, for systems containing both hardware and software components.

SystemC is entirely based on C++, and the source code for the SystemC reference simulator can be freely downloaded from www.systemc.org under an Open Community Licensing agreement.

The SystemC standard is controlled by a steering group composed of thirteen major companies in the EDA and electronics industries. For the past year the technical members of this organization have been focused on developing system level modeling extensions for SystemC. A direct result of this work is the SystemC 2.0 release introduced here.

This paper first reviews the primary modeling constructs in SystemC 1.0. 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 of the new SystemC 2.0 features.

This paper assumes that the reader has some familiarity with C++ and with an HDL such as Verilog or VHDL. SystemC 2.0 is a superset of SystemC 1.0: all SystemC 1.0 designs are still valid in SystemC 2.0.

A 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 numeric type. Fixed-point numbers are frequently used in DSP applications that target both hardware and software implementations since floating point operations usually consume too many hardware resources. An example fixed point operation might be to add two signed numbers that have three bits of integer precision and four bits of fractional precision and assign the result to a similar fixed point number. Often users wish to specify rounding and overflow modes (e.g. saturate or wrap on overflow) when using fixed point numbers. 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 that 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 that complicate the task of modeling them in software. First, 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. In SystemC 1.0, this feature is provided within the `sc_logic` and `sc_lv` data types.

Second, hardware signals sometimes have multiple drivers. In this case, a function is needed to compute a resolved value based on each of the driving values. This function must automatically be called when any of the driving values changes. For example, when a signal is driven with a 1 and a Z, the resolved value should be 1, but when driven with a 1 and a 0, the resolved value should be X. In SystemC 1.0, resolved logic signals are provided to handle this modeling.

Third, hardware signals do not immediately change their output value when they are assigned a new value, either in simulation or in the real world. There is always some delay (perhaps very small) until the new value assigned to a signal is made available to other processes in the design. This delay is crucial to proper modeling of hardware, since it allows, for example, two registers to swap values on a clock edge. In comparison, two software variables cannot swap values without the introduction of a third temporary variable.

Like VHDL and Verilog, SystemC 1.0 supports the concept of delayed signal assignments using delta cycles in order to properly model hardware signals. A delta cycle can be thought of as a very small step of time within the simulation that does not increase the user-visible time. Multiple delta steps can occur at a given time point. When a signal assignment occurs, other processes do not see the newly assigned value until the next delta step. Processes that are sensitive to the signal then resume execution if the signal value is changed with respect to its previous value.

Objectives of SystemC 2.0

One of the primary goals of the SystemC 2.0 release is to enable system-level modeling—that is, modeling of systems above the RTL level of abstraction, including systems which might be implemented in software or hardware or some combination of the two. One of the challenges in providing a system-level design language is that there are a wide range of design models of computation, design abstraction levels and design methodologies used in system design. To address this challenge in SystemC 2.0, a small but very general purpose modeling foundation has been added to the language. On top of this language foundation we can then add the more specific models of computation, design libraries, modeling guidelines and design methodologies which are required for system design.

The small, general purpose modeling foundation in SystemC 2.0 is termed the *core language* and is the central component of the SystemC 2.0 standard.

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. These elementary models are called *elementary channels* in SystemC 2.0.

It is recognized that many different models of computation and design methodologies may be used in conjunction with SystemC. For this reason, the design libraries and models needed to support these specific design methodologies are considered to be separate from the SystemC 2.0 core language standard.

Communication and Synchronization in SystemC 2.0

Many of the language features already within SystemC 1.0 are also very useful for system-level modeling. The structural description features available in SystemC 1.0 (modules and ports) are also useful for system design, as are the extensive set of data types and the ability to express concurrency using processes. However, the SystemC 1.0 mechanism for communication and synchronization—the hardware signal—is not sufficiently general for system-level modeling. For example, in a system level design, a designer might want to specify that several modules communicate using queues, or that several processes execute concurrently and manage access to shared global data using mutexes.

SystemC 2.0 introduces a new set of features for generalized modeling of communication and synchronization. These are *channels*, *interfaces* and *events*. A channel is an object that serves as a container for communication and synchronization. Channels implement one or more interfaces. An interface specifies a set of access methods to be implemented within a channel, but the interface itself does not provide the implementation. An event is a flexible, low-level synchronization primitive that is used to construct other forms of synchronization.¹

Channels, interfaces and events enable designers to model the wide range of communication and synchronization found in system designs. Examples include HW signals, queues (FIFO, LIFO, message queues, etc.), semaphores, memories and busses (both as RTL and transaction-based models).

¹ The channels, interfaces, and events constructs within SystemC 2.0 were inspired by similar constructs in Gajski's SpecC language [2].

SystemC 2.0 Language Architecture

Figure 1 below summarizes the SystemC 2.0 language architecture. There are several important concepts to understand from this diagram.

- All of SystemC builds on C++.
- Upper layers within the diagram are cleanly built on top of the lower layers.
- The SystemC core language provides only a minimal set of modeling constructs for structural description, concurrency, communication, and synchronization.
- Data types are separate from the core language and user-defined data types are fully supported.
- Commonly used communication mechanisms such as signals and fifos can be built on top of the core language. Commonly used models of computation (MOCs) can also be built on top of the core language.
- If desired, lower layers within the diagram can be used without needing the upper layers.

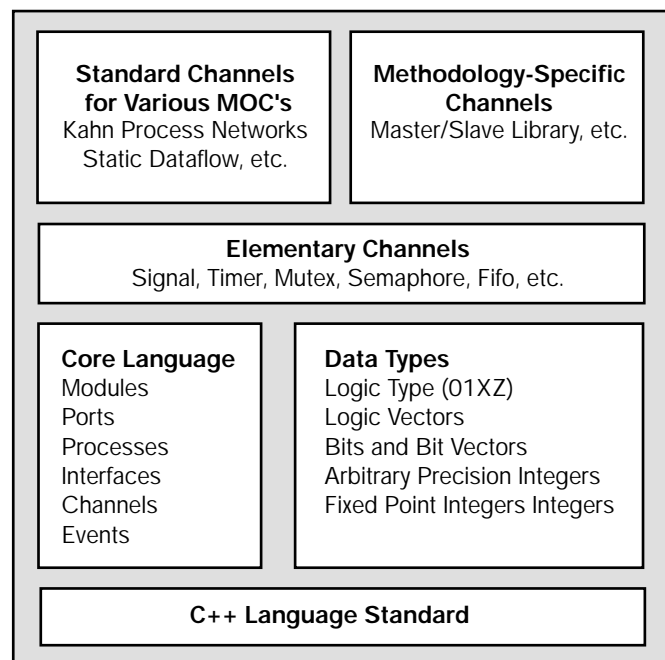


Figure 1 – SystemC 2.0 Language Architecture

A Communication Modeling Example

Let's look at a simple communication modeling example: a FIFO which stores ten characters. The FIFO will have blocking read and write interfaces such that characters are always reliably delivered. (A working version of this example is included in the "examples/simple_fifo" directory within the SystemC 2.0 distribution.)

First, we specify the separate read and write interfaces to the FIFO. To illustrate, let's say that the write interface allows the FIFO to be reset, while the read interface allows a non-blocking query of the number of characters currently in the FIFO.

```

class write_if : virtual public sc_interface
{
public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};

class read_if : virtual public sc_interface
{
public:
    virtual void read(char &) = 0;
    virtual int num_available() = 0;
};

```

As illustrated above, an interface is an abstract base class in C++ that inherits from `sc_interface`. Interfaces specify a set of access methods for a channel but provide no implementation of those methods.

Next, let's specify a high-level model for the FIFO. The FIFO uses C++ multiple inheritance to inherit both the read and write interfaces, and also the properties of channels from `sc_channel`. (Multiple inheritance in C++ is sometimes discouraged, but the form used within SystemC for channels and interfaces is "safe" since interfaces never implement methods or contain data members.)

```

class fifo : public sc_channel,
            public write_if,
            public read_if
{
public:
    SC_CTOR(fifo) { // channel constructor
        num_elements = first = 0;
    }

    void write(char c) {
        if (num_elements == max)
            wait(read_event);
        data[(first + num_elements) % max] = c;
        ++ num_elements;
        write_event.notify();
    }

    void read(char& c) {
        if (num_elements == 0)
            wait(write_event);
        c = data[first];
        -- num_elements;
        first = (first + 1) % max;
        read_event.notify();
    }

    void reset() { num_elements = first = 0; }

    int num_available() {return num_elements;}

private:
    enum e { max = 10 };
    //max is just a constant in class scope
    char data[max];
    int num_elements, first;
    sc_event write_event, read_event;
}

```

```
};
```

The channel provides the implementation for the access methods specified in the read and write interfaces. First, it has a constructor that sets the number of available characters in the FIFO to zero. Next, it has an implementation of the write method which waits, if necessary, until there is space available in the FIFO, then adds the new character into the proper place within the data array. (The FIFO functionality is implemented using a circular buffer within the data array). Finally the write method notifies the `write_event`, which will then resume the execution of any read requests which are waiting for new input.

The read method implementation looks similar. It waits, if necessary, until there is data within the FIFO, then reads the first item and removes it from the data array by incrementing the `first` index. Finally it notifies the `read_event`, which will then resume the execution of any write requests which are waiting for space to be made available.

Recall that processes in SystemC have their own independent threads of execution. When a producer process invokes the write method of the FIFO above, the producer process may need to have its execution suspended until there is space available in the FIFO. The `wait()` call within the write method achieves this suspension of the producer process. It is important to note that this suspension is handled completely by the channel and is not visible to the producer process. If the producer process is suspended, it will be resumed the next time the consumer process reads a character from the FIFO, which will result in a notification of the `read_event`, signaling that space is now available within the FIFO.

SystemC 1.0 also provided a `wait()` call, but the call could not take any arguments because a process was always sensitive to a fixed set of signals. In the example above, we see a significant extension beyond SystemC 1.0: the `wait()` call can now have arguments specified (possibly many), and processes can be sensitive to both events and signals. The ability to provide arguments to the `wait()` call is termed *dynamic sensitivity*. This term is used because as processes execute they can dynamically select the set of events which will cause the process to be resumed using the `wait()` call. Not only can processes wait on specific events in SystemC 2.0, but they can also wait for a specific amount of time. For example, a process can call `wait(200, SC_NS)`, which would cause the process to suspend execution for exactly 200 nanoseconds, or it could call `wait(200, SC_NS, e)` which would cause it to wait until event `e` is notified or a 200 nanosecond timeout occurs.

Events are the fundamental synchronization primitives within SystemC 2.0 and they have several important differences from signals, which were the only synchronization mechanism available in SystemC 1.0. Unlike signals, an event does not have a type and does not transmit a value—events only transfer control from one thread of execution to another. Also, an event notification always causes sensitive processes to be resumed, while an assignment to a signal only causes sensitive processes to be resumed if the new signal value is different from the previous value. Finally, event notifications can be specified to occur immediately, in one delta step in the future, or at a specific time in the future. Signal assignments always occur exactly one delta step in the future.

Completing the Communication Modeling Example

To complete the communication modeling example, let's now show how the FIFO can be used as a communication channel between a producer and a consumer module:


```

// the producer module
class producer : public sc_module
{
public:
    sc_port<write_if> out; // producer output port

    SC_CTOR(producer)      // module constructor
    {
        SC_THREAD(main);   // start the process
    }
    void main()             // the producer process
    {
        char c;
        while (true) {
            ...
            out->write(c);   // write c into fifo
            if (...)
                out->reset(); // reset the fifo
        }
    }
};

// the consumer module
class consumer : public sc_module
{
public:
    sc_port<read_if> in; // consumer input port

    SC_CTOR(consumer)     // module constructor
    {
        SC_THREAD(main);  // start the process
    }

    void main()            // the consumer process
    {
        char c;
        while (true) {
            in->read(c);     // read c from the fifo
            if (in->num_available() > 5)
                ...;        // perhaps speed up processing
        }
    }
};

// the top module
class top : public sc_module
{
public:
    fifo *fifo_inst;       // a fifo instance
    producer *prod_inst;   // a producer instance
    consumer *cons_inst;   // a consumer instance

    SC_CTOR(top)           // the module constructor
    {
        fifo_inst = new fifo ("Fifo1");
        prod_inst = new producer("Producer1");
        // bind the fifo to the producer's port
        prod_inst->out(fifo_inst);

        cons_inst = new consumer("Consumer1");
        // bind the fifo to the consumer's port
        cons_inst->in(fifo_inst);
    }
};

```

It is important to note that the output port of the producer has a template type argument which is specified in this case to be the write interface (`write_if`) of the FIFO. Similarly, the consumer input port has a template type argument which is the read interface of the FIFO. Because of this, only those interface methods specified within the write interface are available for use within the producer, while only those interface methods specified in the read interface are available in the consumer. Thus, the consumer could not call the `fifo.reset()` method as the producer does, for example.

In this example, the FIFO instance is bound to the producer's output port and the consumer's input port. However, the producer's output port is not aware that it is bound to a FIFO. It actually only knows that it is bound to an object which implements the write interface `write_if`. Similarly, the consumer's input is not aware that it is bound to a FIFO, it only knows that it is bound to an object which implements the read interface `read_if`. This hiding of the channel implementation is intentional and is a key ingredient in providing flexible communication modeling and communication refinement within SystemC 2.0.

Here are several examples of how the hiding of the channel implementation mentioned above aids communication modeling and communication refinement:

- Imagine that the designer wished to experiment with a slightly different functional specification of this design which used a FIFO that discarded characters when it was full, rather than suspending execution of the producer process. All the designer would need to do is to write the channel implementation for this new FIFO which reuses the existing read and write interfaces. Then the only change needed to the original design would be the substitution of the new FIFO for the original one within the top module.
- Imagine that the designer wished to refine this design to an implementation running on an RTOS. The producer and consumer processes would be implemented using independent threads running on the RTOS, while the original FIFO instance would be replaced with a new one in which the read and write methods directly access the RTOS' built-in queuing services. Again, the only change needed in the original design is the substitution of the original FIFO with the new one in the top module.
- Imagine that the designer wished to refine this design to a custom hardware implementation that uses a hardware FIFO. Again, a new FIFO channel is written to replace the existing one in the top module. Channels in SystemC can contain other channels and modules, just as modules can contain multiple child modules. In this case we write a new FIFO channel which instantiates the hardware FIFO within it, and which contains the hardware signals necessary to interface with the hardware FIFO. We then implement the read and write interface methods within this channel to drive the hardware signals with the protocol required to cause data to be properly loaded into and unloaded from the hardware FIFO. This then allows us to simulate the design while using a model of the actual hardware FIFO. As a final step in the communication refinement process, the code which implements the write and read protocols to access the hardware FIFO within the channel can be inlined into the producer and consumer, respectively, enabling this code to be synthesized and optimized with the producer and consumer code. (The SystemC language does not automate this inlining step, but it does provide the constructs needed so tools can perform this task).

One final note on this example. For simplicity, the FIFO channel that is presented above is only able to store characters. In practice, channels such as this would be written using C++ templates to allow the data type to be specified at the time that the channel is instantiated. Using this technique, a single FIFO channel could store any C++ data type, including user-defined datatypes. SystemC 2.0 fully supports this template-based design technique.

Models of Computation within SystemC

There has been a lot of discussion about models of computation in recent years. This is perhaps because there are many different models of computation and it is not always obvious which one is best suited for a particular system design task. In the broadest sense, a model of computation is defined by the following:

- The model of time employed (real-valued, integer-valued, untimed) and the event ordering constraints within the system (globally ordered, partially ordered, etc.).
- The supported method(s) of communication between concurrent processes.
- The rules for process activation.

In SystemC 2.0 the simple and flexible synchronization capabilities provided by events and the `wait()` call 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 that can be quite naturally modeled in SystemC 2.0, include:

- Kahn Process Networks
- Static Multi-rate Dataflow
- Dynamic Multi-rate Dataflow
- Communicating Sequential Processes
- Discrete Event as used for
 - RTL hardware modeling
 - Network modeling (e.g. stochastic or “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.

Future Evolution of SystemC

Because the SystemC standard is controlled by a number of companies and its development is dependent on open contributions, its future evolution can be difficult to predict. Nevertheless, a consensus does exist concerning some likely extensions to SystemC. These extensions can be separated into two categories. The first set of extensions include new features within the core language. These are likely to include:

- Ability to fork and join threads, and dynamic thread creation.
- Ability to interrupt or abort a thread and its children.
- Specification and checking of timing constraints.
- Support for abstract RTOS modeling and scheduler modeling.
- Possible support for analog mixed signal modeling.

The second set of extensions are features that can be developed as libraries on top of the core language. These may include:

- Standardized channels for various models of computation such as static dataflow and Kahn process networks.
- Libraries to facilitate development of testbenches, such as data structures that aid stimulus generation and response checking, functions that help generate randomized stimulus, etc.
- System level modeling guidelines and library code that helps users create models following the guidelines.
- Standard APIs for interfacing SystemC with other simulators, emulators, etc.

Summary

The new modeling constructs in SystemC 2.0 enable users to cleanly model a wide range of communication and synchronization methods in system designs and even allow users to implement new models of computation within SystemC. In addition, these modeling constructs make it easy to explore different communication schemes in systems designs and to perform communication refinement to software or hardware implementations.

SystemC 2.0 allows the following tasks to be performed within a single language:

- Complex system specifications can be developed and simulated.
- System specifications can be refined to mixed software and hardware implementations.
- Hardware implementations can be accurately modeled all the way to the RT-level.
- Complex data types can be easily modeled, and a flexible fixed-point numeric type is supported.
- The extensive knowledge, infrastructure and code base built around C and C++ can be leveraged.

The ability to cleanly support such a wide range of system design tasks within a single language is unique to SystemC.

References

- [1] Functional Specification for SystemC 2.0. Available at www.systemc.org.
- [2] Gajski, D., SpecC: Specification Language and Methodology. Kluwer Academic Publishers, Norwell MA, 2000. Available at www.wkap.nl.



Open SystemC Initiative (OSCI)

The Open SystemC Initiative grants permission to copy and distribute this document in its entirety. Each copy shall include all copyrights, trademarks, and service marks, if any. All products or service names mentioned herein are trademarks of their respective holders and should be treated as such. All rights reserved.

©2001 Open SystemC Initiative (OSCI) 5/01.TM.WO