System Level Design Using the SystemC Modeling Platform [1]

Joachim Gerlach, University of Tübingen, Germany
gerlach@informatik.uni-tuebingen.de

Wolfgang Rosenstiel, University of Tübingen, Germany
rosenstiel@informatik.uni-tuebingen.de

Abstract

This paper gives an overview of the SystemC modeling platform and outlines the features supported by the SystemC class library. The use of the modeling platform is shown in terms of an example.

1. Introduction

As system complexity increases and design time shrinks, it becomes extremely important that system specification be captured in a form that leads to unambiguous interpretation by the system implementers. The most common form of specification capture, a written document, has several drawbacks: natural language is ambiguous and open to interpretation; the specification may be incomplete and inconsistent; and, finally, there is no way to verify the correctness of such a specification. These drawbacks have driven many system, hardware, and software designers to create *executable specifications* for their systems. For the most part, these are functional models written in a language like C or C++. These languages are chosen for three reasons: they provide the control and data abstractions necessary to develop compact and efficient system descriptions; most systems contain both hardware and software and for the software, one of these languages is the natural choice; designers are familiar with these languages and the large number of development tools associated with them. A functional model in C or C++ is essentially a program that when executed exhibits the same behavior as the system. However, creating a functional model in a programming language like C or C++ is problematic because these languages are intended for programming software and do not have the constructs necessary to model timing, concurrency, and reactive behavior, all of which are needed to create accurate models of systems containing both hardware and software. To model concurrency, timing, and reactivity, new constructs need to be added to C++. An object-oriented programming language like C++ provides the ability to extend a language through classes, without adding new syntactic constructs. A class-based approach to providing modeling constructs is superior to a proprietary new language because it allows designers to continue to use the language and tools they are familiar with.

2. The SystemC Approach

On September 1999, leading EDA, IP, semiconductor, systems and embedded software companies announced the "Open SystemC Initiative" (OSCI) and immediate availability of a C++ modeling platform called SystemC for free web download at the Embedded Systems Conference, San Jose, California. Achieving a break-through in industry cooperation, SystemC is the first result of the initiative, which enables, promotes and accelerates system-level intellectual property (IP) model exchange and co-design using a common C++ modeling platform. Through an Open Community Licensing model, designers can create, validate and share models with other companies using SystemC and a standard ANSI C++ compiler. In addition, electronic design automation (EDA) vendors have complete access to the SystemC modeling platform required to build interoperable tools. There are no licensing fees associated with the use of SystemC, and any company is free to join and participate. Backed by a growing community of well over 45 charter member companies, the Open SystemC Initiative includes representation from the systems, semiconductor, IP, embedded software and EDA industries. The steering group includes ARM, CoWare, Inc., Cygnus Solutions, Ericsson, Fujitsu, Infineon, Lucent Technologies, Sony Corporation STMicroelectronics, Synopsys, Inc. and Texas Instruments. The goal of the Open Community Licensing model is to provide a foundation to build a market upon, and the role of the steering group is to

1. This work was partially supported by Synopsys, Inc., Mountain View, CA.

provide an environment of structured innovation ensuring that interoperability is retained.

In regard of the excellent background of the Open SystemC Initiative, SystemC is on the best way to become a de-facto-standard for system-level specification. This paper gives an overview of the SystemC modeling platform and outlines the features supported by the SystemC class library.

3. Overview of the SystemC Design Flow

The following overview refers to SystemC version 0.9, which is currently available for free web download at *www.systemc.org*. Extensions supported by SystemC version 1.0 are summarized in a later section. SystemC 1.0 is announce on March 2000.

In a SystemC description, the fundamental building block is a *process*. A process is like a C or C++ function that implements behavior. A complete system description consists of multiple concurrent processes. Processes communicate with one another through *signals*, and explicit *clocks* can be used to order events and synchronize processes. All the building blocks are objects (classes) that are part of SystemC. Special data types required to model hardware efficiently are also provided as a part of the library. SystemC is written using the full C++ language. A user just needs to understand how to use the classes and functions provided by the library, but she/he does not need to know how they are implemented. Using the SystemC library, a system can be specified at various levels of abstraction. At the highest level, only the functionality of the system may be modeled. For hardware implementation, models can be written either in a functional style or in a register-transfer level style. The software part of a system can be naturally described in C or C++. Interfaces between software and hardware and between hardware blocks can be easily described either at the transaction-accurate level or at the cycle-accurate level. Moreover, different parts of the system can be modeled at different levels of abstraction and these models can co-exist during system simulation. C/C++ and the SystemC classes can be used not only for the development of the system, but also for the testbench. The functionality of the SystemC classes together with the object-oriented nature of C++ provides a powerful mechanism for developing compact, efficient, and reusable testbenches. SystemC consists of a set of header files describing the classes and a link library that contains the simulation kernel. The header file can be used by the designer in her/his program. Any ANSI C++-compliant compiler can compile SystemC, together with the program. During linking, the SystemC library, which contains the simulation kernel is used. The resulting executable serves as a simulator for the system described, as shown in figure 1. Choosing C/C++ as the modeling language, a host of software development tools like debuggers and integrated development environments can be leveraged.

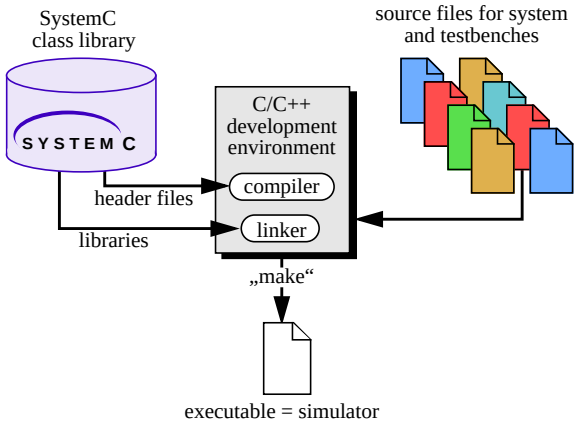

Figure 1. SystemC Design Flow.

4. Features of the SystemC Class Library

In the following, some features of SystemC version 0.9 are presented. More details can be found in the SystemC Release 0.9 Reference Manual, which is enclosed in the open source distribution of SystemC.

- *Modules*: SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it. Modules and processes can have a functional interface, which allows to hide implementation details and, for this, include blocks of IP.
- *Processes*: Processes are used to describe functionality. Processes can be stand alone entities or can be contained inside modules. SystemC provides three different process abstractions to be used by hardware and software designers: asynchronous blocks, synchronous and asynchronous processes.
- *Signals*: SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus) while unresolved signals can have only one driver.
- *Rich set of signal types*: To support modeling at different levels of abstraction, from the functional to the register-transfer level, as well as to support software, SystemC supports a rich set of signal types. This is different than languages like Verilog that only support bit and bit-vectors as signal types. SystemC supports both two-valued and four-valued signal types.

- *Rich set of data types*: SystemC has a rich set of data types to support multiple design domains and abstraction levels. The fixed precision types allow fast simulation. The arbitrary precision types can be used for computations with large numbers and to model large busses. SystemC supports both two-valued and four-valued data types. There is no size limitation for arbitrary precision SystemC data types. SystemC also includes a rich set of overloaded operators and type conversion mechanisms for those data types.
- *Clocks*: SystemC has the notion of clocks as special signals. Clocks are the timekeepers of the system during simulation. Multiple clocks, with arbitrary phase relationship, are supported.
- *Reactivity*: For modeling reactive behavior, SystemC provides mechanisms for waiting on clock edges, events, and signal transitions. SystemC also supports watching for a certain event, regardless of the execution stage of the process (the most common example is the watching of a reset signal).
- *Multiple abstraction levels*: SystemC supports modeling at different levels of abstraction, ranging from high level functional models to detailed register-transfer level models. It supports iterative refinement of high level models into lower levels of abstraction.
- *Creating functional models*: For creating abstract functional models, SystemC supports a communication primitive called channel. A channel is a special type of signal that synchronous and asynchronous processes may use to communicate with each another. The capabilities provided by a channel are different from those provided by a signal. Though writing to and reading from channels take some number of clock cycles, the intent of using channels is not to focus on the time required for communication. Channels are meant to be used in the initial stages of modeling when the primary interest is the system functionality. In such situations, channels provide mechanisms for guaranteed data delivery. Channels implicitly provide the necessary handshaking to ensure correct delivery of data from the sender to the receiver.
- *Cycle-based simulation*: SystemC includes a cycle-based simulation kernel that allows high speed simulation. SystemC also provides mechanisms for simulation control at any point of the input specification.
- *Debugging support and waveform tracing*: SystemC classes have run-time error checking that can be turned on during compilation. The SystemC kernel contains basic routines to dump waveforms to a file (VCD, WIF, and ISDB format), which can be viewed by typical waveform viewers.

SystemC version 1.0, which is announced on March 2000, will extend SystemC 0.9 for some important features. More details can be found in the SystemC Version 1.0 Draft Specification, which is also enclosed in the open source distribution of SystemC. The most important extensions of SystemC 1.0 are:

- *Modules*: In SystemC 1.0, the fundamental building block will be a module. A module, as in SystemC 0.9, is a pure container class. Processes will be contained inside modules and modules will support multiple processes inside them. Modules will have (single-direction and/or bidirectional) ports which they connect to other modules.
- *Fixed-point data types*: SystemC 1.0 will provide arbitrary precision fixed-point data types, together with a rich set of overloaded operators, quantization and overflow modes, and type conversion mechanisms.
- *Communication protocols*: In SystemC 1.0, channel functionality and implementation will be significantly enhanced to support complex communication protocols. Multi-level communication semantics will be provided which enables to describe SoC and system I/O protocols with different levels of communication abstraction. SystemC 1.0 will provide abstract ports that include communication semantics defined in the form of protocols, and multi-level communication semantics that allows to describe communication interfaces at different levels of abstraction (data transaction level, bus-cycle level, clock-cycle level).

5. Example

In the following, the use of the SystemC modeling platform is shown in terms of an example. Objective of this quite small example is to give an overview of how systems are modeled using SystemC, so the reader should not concentrate on system functionality or system complexity. The example consists of two synchronous processes, *process_1* and *process_2*, communicating with one another. *process_1* increments the value of an integer input port by 5 and

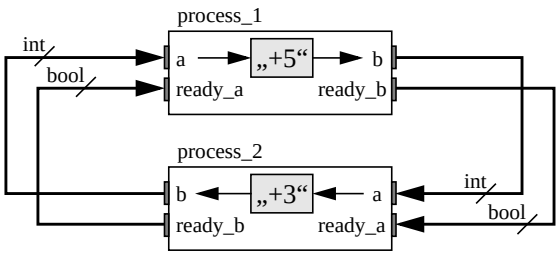

Figure 2. Example system.

assigns the result to an integer output port, *process_2* increments the value of an integer input port by 3 and assigns the result to an integer output port. Both processes are connected in a way that an integer value is alternately incremented by *process_1* and *process_2*. Process synchron-ization is done via boolean signals (see figure 2). Figure 3 shows the header and implementation code file of *process_1*. A process is encapsulated in a C++ class of type *struct*. *process_1* is implemented to be a synchronous process, which is done by deriving it from the SystemC *sc_sync* base class. Input and output ports of the process are described by data elements of the class. Input ports of the process are declared as *const*-qualified references to *sc_signal<T>* (where *T* corresponds to an arbitrary data type). Output ports are declared in the similar way, but are not *const*-qualified. The next major part of the

```
// header file: process_1.h

struct process_1: sc_sync
{
   // Ports
   const sc_signal<int>& a;
   const sc_signal<bool>& ready_a;
   sc_signal<int>& b;
   sc_signal<bool>& ready_b;

   // Constructor
   process_1(const char *NAME,sc_clock_edge& TICK,
      const sc_signal<int>& A,
      const sc_signal<bool>& READY_A,
      sc_signal<int>& B,
      sc_signal<bool>& READY_B)
         :  sc_sync(NAME,TICK),
            a(A),
            ready_a(READY_A),
            b(B),
            ready_b(READY_B)
   {
   }

   // Process functionality
   void entry();
};
```

```
// implementation file: process_1.cc

#include "systemc.h"
#include "process_1.h"

void process_1::entry()
{
   int v;

   while (true)
   {
      wait_until( ready_a.delayed()
                  == true);
      v = a.read();
      v += 5;
      cout << "P1: v = "
           << v << endl;
      b.write(v);

      ready_b.write(true);
      wait();
      ready_b.write(false);
   }
}
```

Figure 3. *process_1.h* and *process_1.cc*.

process declaration is the declaration of the constructor function. The first argument of the constructor function is a *const*-qualified pointer to a string. This will be the name of the process when the constructor is called during process instantiation. The second argument is a reference to the clock edge that the process is sensitive to. The subsequent arguments are the input and output ports of the process. Initializers are used to connect process ports to signals (for example, port *a* gets connected to signal *A* in the constructor function of *process_1*). The process functionality is specified in a member function *void entry()*. *process_1* waits until the boolean input signal *ready_a* changes to *true*. Then, input port *a* is read and the corresponding integer value is assigned to a local integer variable *v*. *v* is incremented by 5, displayed for evaluation purpose, and assigned to port *b*. After this, the boolean input signal *ready_b* is set to *true* for one clock cycle. *process_2* is specified in a very similar way, figure 4 shows the corresponding header and implemen-tation code file. The next step is to create an instance of each

```
// header file: process_2.h

struct process_2: sc_sync
{
   // Ports
   const sc_signal<int>& a;
   const sc_signal<bool>& ready_a;
   sc_signal<int>& b;
   sc_signal<bool>& ready_b;

   // Constructor
   process_2(const char *NAME,sc_clock_edge& TICK,
      const sc_signal<int>& A,
      const sc_signal<bool>& READY_A,
      sc_signal<int>& B,
      sc_signal<bool>& READY_B)
         :  sc_sync(NAME,TICK),
            a(A),
            ready_a(READY_A),
            b(B),
            ready_b(READY_B)
   {
   }

   // Process functionality
   void entry();
};
```

```
// implementation file: process_2.cc

#include "systemc.h"
#include "process_2.h"

void process_2::entry()
{
   int v;

   while (true)
   {
      wait_until(ready_a.delayed()
                 == true);
      v = a.read();
      v += 3;
      cout << "P2: v = "
           << v << endl;
      b.write(v);

      ready_b.write(true);
      wait();
      ready_b.write(false);
   }
}
```

Figure 4. *process_2.h* and *process_2.cc*.

process and tie them together with signals in a top-level routine. This routine must have the name *sc_main*, figure 5 shows the corresponding source code file. In the top-level routine, all process header files and the file *systemc.h* have to be included, because these files contain the declaration for all the process classes and SystemC library functions. Inside the *sc_main* routine, signals that the processes use for communication were declared. After the signals are instantiated, the clock object, which is a special signal, is instantiated. In our example, the name of the clock object is *Clock*, it has a period of 20 time units, and has a 50% duty cycle. Recall that in a SystemC description, processes are classes, and all classes have constructor functions. Therefore, instantiating a process involves providing the constructor of the corresponding class with the right set of arguments. This is equivalent to connecting the ports of the process to signals. The following parts in *sc_main* include the declaration of an output file for waveform tracing and the

specification of a set of signals to be traced during simulation. Furthermore, an initialization of the signals

```
// implementation file: main.cc

#include "systemc.h"
#include "process_1.h"
#include "process_2.h"

int sc_main(int ac,char *av[])
{
    sc_signal<int> s1 ("Signal-1");
    sc_signal<int> s2 ("Signal-2");
    sc_signal<bool> ready_s1 ("Ready-1");
    sc_signal<bool> ready_s2 ("Ready-2");

    sc_clock clock("Clock",20,0.5,0.0);

    process_1 p1 ("P1",clock.pos(),s1,ready_s1,s2,ready_s2);
    process_2 p2 ("P2",clock.pos(),s2,ready_s2,s1,ready_s1);

    sc_trace_file *tf = sc_create_wif_trace_file("trace_file");
    sc_trace(tf,s1,"Signal-1");
    sc_trace(tf,s2,"Signal-2");
    sc_trace(tf,ready_s1,"Ready-1");
    sc_trace(tf,ready_s2,"Ready-2");

    s1.write(0);
    s2.write(0);
    ready_s1.write(true);
    ready_s2.write(false);

    sc_start(100000);

    return 0;
}
```

Figure 5. Top-level routine *main.cc*.

is done. Once all processes are instantiated and connected with signals, the clock has to be generated in order to simulate the system. This is done by the SystemC function *sc_start(n)*, where *n* is the number of time units for which the simulation is intended to last. In our example, the entire system consists of three implementation files (*process_1.cc*, *process_2.cc*, *main.cc*) and two header files (*process_1.h*, *process_2.h*). The implementation files need to be compiled individually and finally linked together with SystemC. In addition, compilation of each file requires header files from SystemC. Compilation can be done using any standard ANSI C++ compiler (for example, *gnu gcc*). Executing the resulting binary is equivalent to run a simulation of the system description for the specified number of time units. In our example, system behavior can be checked by observing the process outputs. Figure 6 shows the first view output lines produced when executing the binary. Figure 7 shows parts of the waveform file produced during simulation. The execution of the binary (which means, the simulation of the system description for 100000 time units) takes 0.31 seconds on a Sun Ultra Sparc 5 with 384 MByte of main memory (for runtime measurement, the writing of a waveform file was skipped).

```
SystemC (TM) Version 0.9  --- Sep 28 1999 14:32:42
                    ALL RIGHTS RESERVED
          Copyright (c) 1988-1999 by Synopsys, Inc.
P1: v = 5
P2: v = 8
P1: v = 13
P2: v = 16
P1: v = 21
P2: v = 24
P1: v = 29
P2: v = 32
P1: v = 37
P2: v = 40
P1: v = 45
P2: v = 48
P1: v = 53
.....
```

Figure 6. Process outputs.

6. Conclusion

This paper gives a brief overview of the new SystemC modeling platform. SystemC provides innovative mechanisms for C++-based system-level description and is freely available for everybody through an open source licensing model. Because of those facts and the continuously growing number of leading EDA, IP, semiconductor, systems and embedded software companies joining the Open SystemC Initiative, SystemC is on the best way of becoming a de-facto-standard for system-level specification.

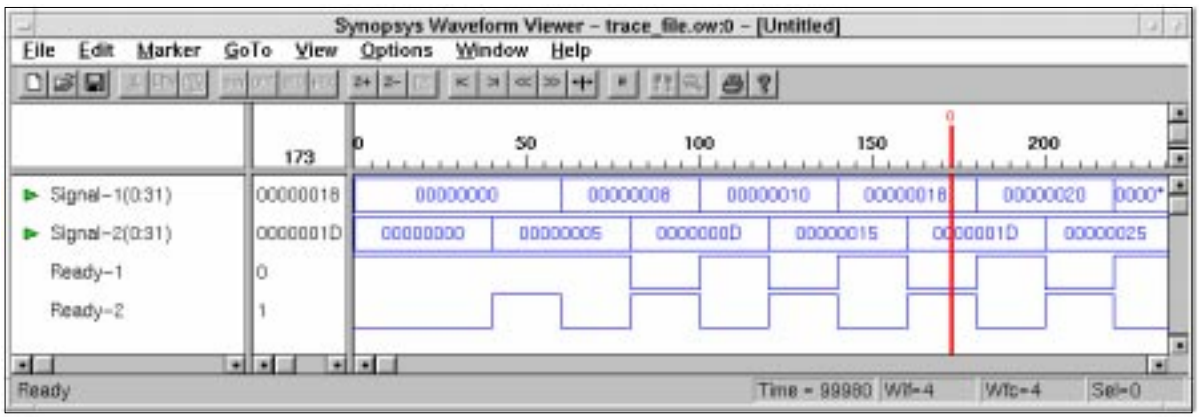

Figure 7. Waveform view of a SystemC simulation.