1. Introduction

As system complexity increases and design time shrinks, it becomes extremely important that system specification be written down in a form that leads to unambiguous interpretation by the system implementers. The most common form of system specification, 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: first, they provide the control and data abstractions necessary to develop compact and efficient system descriptions, second, most systems contain both hardware and software and for the software, one of these languages is the natural choice, and third, designers are familiar with these languages and there exists a 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 to be modeled. However, creating a functional model in a programming language like C or C++ is problematic because these languages are intended for software development and do not provide the constructs necessary to model timing, concurrency, and reactive behavior, which are all 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 is superior to a proprietary new language because it allows designers to use the language and tools they are familiar with.

---

1. This work was partially supported by the DFG Priority Research Program „Rapid Prototyping of Embedded Systems With Hard Time Constraints“ and Synopsys Inc., Mountain View, CA.
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 every company is free to join and participate. Backed by a growing community of well over 50 charter member companies, the Open SystemC Initiative includes representatives from the systems, semiconductor, IP, embedded software and EDA industries. The steering group consists of a large number of market leading companies, including ARM, Cadence, CoWare, Ericsson, Fujitsu Microelectronics, Infineon Technologies, Lucent Technologies, Motorola, NEC, Sony Corporation, STMicro-electronics, Synopsys, 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 provide an environment of structured innovation ensuring that interoperability is retained.

Because of the excellent background of the Open SystemC Initiative, SystemC is on the best way to become a de-facto-standard for system-level modeling and design. 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 1.0 which is currently available for free web download at www.systemc.org.

The fundamental building blocks in a SystemC description are processes. A process is similar to 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 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 uses the full C++ language. A user only 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. The use of C/C++ and the SystemC classes is not limited to the development of the system, but can also be used for the implementation of testbenches. The functionality of the SystemC classes together with the
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 variety of software development tools like debuggers and integrated development environments can be utilized.

4. Features of the SystemC Class Library

In the following, some features of SystemC version 1.0 are presented. More details can be found in the SystemC Release 1.0 Reference Manual which is included in the open source distribution of SystemC.

- Modules: In SystemC, the fundamental building block is a module. Processes are contained inside modules and modules support multiple processes inside them. Modules can also be used for describing hierarchy: a module can contain submodules, which allows to break complex systems into smaller more manageable pieces. 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. SystemC provides three different process abstractions to be used by hardware and software designers: methods (asynchronous blocks), threads (asynchronous processes) and clocked threads (synchronous processes).

- Ports: Ports of a module are the external interface passing information to and from a module, and triggering actions within the module. Ports can be single-direction or bidirectional.

- Signals: Signals create connections between module ports allowing modules to communicate. SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus) while unresolved signals can only have a single driver.
Signal types: To support different levels of abstraction, ranging from the functional level to the register-transfer level, as well as to support software, SystemC supports a rich set of signal types. This is different to languages like Verilog that only support bit and bit-vectors as signal types. SystemC supports both two-valued and four-valued signal types.

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. For example, SystemC version 1.0 provides arbitrary precision fixed-point data types, together with a rich set of overloaded operators, quantization and overflow modes, and type conversion mechanisms.

Clocks: SystemC has the notion of clocks as special signals. Clocks are the timekeepers of the system during simulation. SystemC supports multiple clocks with arbitrary phase relationships.

Reactivity: For modeling reactive behavior, SystemC provides mechanisms for waiting on events like clock edges 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.

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 standard waveform viewers.

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 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 synchronization is done via boolean signals (see figure 2).

Figure 3 shows the header and the implementation code file of process_1. The process is encapsulated in a SystemC module given by a C++ class. This can be done using the SystemC macro SC_MODULE (SC_MODULE(process_1) {...} is equivalent to struct process_1: sc_module {...}, where the base class sc_module is provided by the SystemC class library). Module ports are described by data members of the C++ class. Input ports are declared by
sc_in<T>, output ports are declared by sc_out<T> (where T is an arbitrary data type). In addition, a special input port of type sc_in_clk has to be specified for the clock signal. The process functionality is encapsulated in a function member void do_process_1() of the C++ class. process_1 is implemented to be a synchronous process, which is done in the class constructor. Again, the class constructor doesn’t have to be coded directly, but can be specified using the SystemC macro SC_CTOR. In the body of the class constructor, process_1 is made to be a synchronous process by using SC_CTHREAD. Arguments of SC_CTHREAD are the name of the process function member and the clock edge to which the process is sensitive. In the body of the function member, process_1 waits until the boolean input signal ready_a becomes 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 output port b. Next, the boolean input signal ready_b is set to true for one clock cycle.

process_2 is specified in a very similar way as shown in figure 4.
The next step is to create an instance of each process and tie them together with signals in a top-level routine. By convention, this routine is called \texttt{sc\_main}. Figure 5 shows the corresponding source code file. In the top-level routine, all process header files and the file \texttt{systemc.h} are included, because these files contain the declaration for all the process classes and SystemC library functions. Inside \texttt{sc\_main}, signals used for process communication are 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 \texttt{Clock}, it has a period of 20 time units and a 50% duty cycle.
cycle. Next, processes process_1 and process_2 are declared and the ports are connected by signals. sc_main also contains the declaration of an output file for waveform tracing and the specification of a set of signals to be traced during simulation. In our example, waveform tracing is optional and can be activated with the \texttt{w} argument during the call of the executable. Furthermore, an initialization of the signals is done. Once all processes are instantiated and connected to signals, the clock is generated to simulate the system. This is done by the SystemC function \texttt{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 (\texttt{process_1.cc}, \texttt{process_2.cc}, \texttt{main.cc}) and two header files (\texttt{process_1.h}, \texttt{process_2.h}). Figure 6 shows the file structure of the source code. The implementation files can be compiled individually and finally linked with the SystemC library. In addition, compilation of each file requires header files from SystemC. Compilation can be done using any standard ANSI C++ compiler (for example, gnu \texttt{gcc}). Executing the resulting binary is equivalent to running a simulation of the system description for the specified number of time units.

In our example, system behavior can be verified by observing the process’ outputs. Figure 7 shows the first view output lines produced when executing the binary.

```
SystemC (TM) Version 1.0 — Apr 4 2000 10:12:32
ALL RIGHTS RESERVED
Copyright (c) 1988 2000 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 = 33
P1: v = 37
P2: v = 40
P1: v = 45
P2: v = 48
P1: v = 53
P2: v = 56
P1: v = 61
P2: v = 64
P1: v = 69
P2: v = 72
```

Figure 6. Source code file structure.

Figure 7. Process outputs.
Figure 8 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 0.08 seconds on a Sun Ultra Sparc 5 with 384 MByte main memory (for runtime measurement, the process outputs and writing of waveform files were skipped). Using version 0.9 of the SystemC library, the simulation of the system description for 100000 time units takes 0.31 seconds on the same machine.

![Waveform view of a SystemC simulation.](image)

6. Conclusion

This paper gives a brief overview of the SystemC modeling platform. SystemC provides innovative mechanisms for C++-based system-level description and is freely available 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.