SystemC - A modeling platform supporting multiple design abstractions

Preeti Ranjan Panda
Synopsys Inc.
700 E. Middlefield Rd.
Mountain View, CA 94043, USA
panda@synopsys.com

ABSTRACT

SystemC is a C++ based modeling platform supporting design abstractions at the register-transfer, behavioral, and system levels. Consisting of a class library and a simulation kernel, the language is an attempt at standardization of a C/C++ design methodology, and is supported by the Open SystemC Initiative (OSCI), a consortium of a wide range of system houses, semiconductor companies, Intellectual property (IP) providers, embedded software developers, and design automation tool vendors. The advantages of SystemC include the establishment of a common design environment consisting of C++ libraries, models and tools, thereby setting up a foundation for hardware-software co-design; the ability to exchange IP easily and efficiently; and the ability to reuse test benches across different levels of modeling abstraction. We outline the features of SystemC that make it an attractive language for design specification, verification, and synthesis at different levels of abstraction, with particular emphasis on the new features included in SystemC 2.0 that support system-level design.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic Synthesis, Hardware description languages, Optimization, Simulation, Verification*; B.6.3 [**Logic Design**]: Design Aids—*Hardware description languages*; B.7.2 [**Integrated Circuits**]: Design Aids—*Simulation, Verification*; C.0 [**General**]: *Hardware/software interfaces, System specification methodology*; C.3 [**Special-purpose and application based systems**]: *Signal processing systems, Real time and embedded systems*; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms

Design, Languages, Standardization, Verification, Performance

Keywords

SystemC, C/C++ based design, System level design, Hardware description language

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

1. INTRODUCTION

The level of abstraction at which hardware is designed has increased significantly with the widespread adoption of Hardware Description Languages (HDLs) as the specification format, or the design entry point, which has led to an enormous increase in productivity over the earlier schematic entry based design methodology. The leap in productivity came about because HDLs such as VHDL and Verilog allowed designers to specify complex functionality at the behavioral and register transfer level (RTL) in a relatively succinct manner compared to the earlier structural-only view. However, after a decade of successful deployment, it appears that the current generation of HDLs are insufficiently equipped to handle the ever-increasing complexity hardware design and system-level design. It is no longer productive for designers to model at the level of individual bits imposed by HDL; more sophisticated data abstraction capabilities are needed. Further, hardware is no longer designed as an independent entity. Hardware modules frequently co-exist on the same chip with processor cores, embedded software, and other complex IP blocks, which forces designers to perform slow and inefficient co-simulations of hardware and software parts when attempting to simulate the entire system together. A cleaner mechanism for handling software and hardware components in the same environment is badly needed.

Several modeling platforms have been proposed in the past years for increasing the level of abstraction and enabling hardware-software co-design. Specification at higher levels of abstraction is possible in environments such as SpecC [4]. A unified and integrated approach to hardware-software co-design is possible if the hardware modeling description is based on the C/C++ languages that are popular in the software community. Hardware-C [6] is an example of such a proposal.

SystemC [7, 5, 8, 3] is an emerging standard modeling platform based on C++ that addresses the issues discussed above, and supports design abstraction at the RTL, behavioral, and system levels. Consisting of a class library and a simulation kernel, the language is an attempt at standardization of a C/C++ design methodology, and is supported by the Open SystemC Initiative (OSCI), a consortium of a wide range of system houses, semiconductor companies, IP providers, embedded software developers, and design automation tool vendors. Apart from the modeling benefits available in C++ such as data abstraction, modularity, and object orientation, the advantages of SystemC include the establishment of a common design environment consisting of C++ libraries, models and tools, thereby setting up a foundation for hardware-software co-design; the ability to exchange IP easily and efficiently; and the ability to reuse test benches across different levels of modeling abstraction.

We discuss the overall SystemC design flow in Section 2; the

SystemC features that enable hardware and system level modeling in Sections 3 and 4; the language architecture in Section 5; and our conclusions in Section 6.

2. THE SYSTEMC DESIGN FLOW

We outline here a typical design flow using the SystemC environment. The flow highlights the most important steps involved in the verification and synthesis/implementation tasks and is not exhaustive. Since the technology is new, several tools and methodologies aimed at providing a more complete design environment are currently under development.

2.1 Simulation with SystemC

Figure 1 illustrates a typical simulation methodology in the SystemC environment. The designer writes the SystemC models at the system level, behavioral level, or RTL level using C/C++ augmented by the SystemC class library. The class library serves two important purposes. First, it provides the implementation of many types of objects that are hardware-specific, such as concurrent and hierarchical modules, ports, and clocks. Second, it contains a lightweight kernel for scheduling the processes. The user's SystemC code can now be compiled and linked together with the class library with any standard C++ compiler (such as GNU's `gcc`), and the resulting executable serves as the simulator of the user's design. The testbench for verifying the correctness of the design is also written in SystemC and compiled along with the design. The executable can be debugged in any familiar C++ debugging environment (such as GNU's `gdb`). Additionally, trace files can also be generated to view the history of selected signals using a standard waveform display tool.

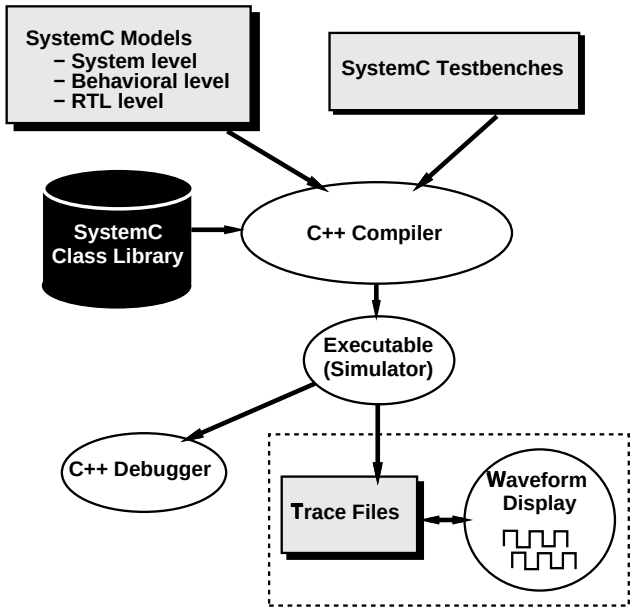

Figure 1: Simulation methodology. SystemC models and test benches are processed by a standard C++ compiler. The generated executable serves as the simulator, and can also be debugged using a standard C++ debugger. History of selected signals can be dumped into a trace file for waveform display. (The figure is adapted from an illustration in [5])

The import of a traditional software development environment into the hardware design and system design scenario entails some powerful advantages. The sophisticated program development infrastructure already in place for C/C++ can be directly utilized for the SystemC verification and debugging tasks. For hardware designers traditionally used to viewing simulation data in the form of waveform displays, the trace file generation facility provides a familiar interface. Conceptually, the most powerful feature is that the hardware, software, and testbench parts of the design can be simulated in one simple and unified simulation environment without the need for clumsy co-simulations of disparate modeling paradigms.

The source code for the latest SystemC class library (currently version 2.0 beta) can be freely downloaded from [1].

2.2 Implementation and Synthesis

The most important feature of the SystemC implementation flow is that the specification is in a common language for both hardware and software parts. In fact, in the system level SystemC model, the two are indistinguishable as the assignment of modules to hardware or software has not yet been made. Thus, trace-offs in the implementation of different parts of the design in hardware or software, can be explored in a seamless fashion, eliminating the need for re-implementing each module in both C and HDL. This also eliminates the need for the system design tools to understand and analyze the syntax and semantics of two disparate modeling environments.

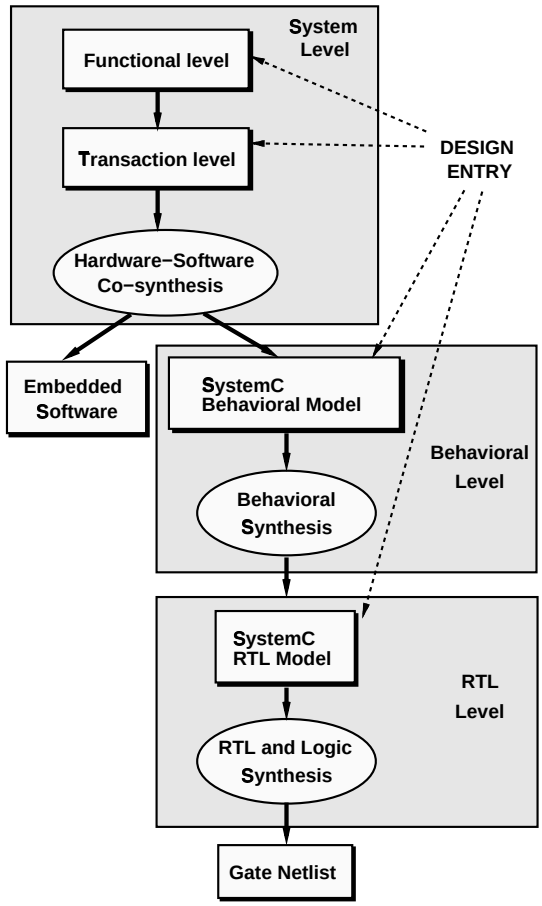

Figure 2: Synthesis/Implementation flow. Design entry could be at the System, Behavioral, or RTL levels. The same test bench could be used to validate models at different levels of abstraction

A typical top-down synthesis/implementation flow is illustrated in Figure 2. The design entry could be at any level of abstraction: system level (which could be an untimed functional model, or a transaction-level model), behavioral level, or RTL level. The transition from a higher level to a lower level of abstraction could be achieved either through automatic synthesis and compilation tools (such as hardware/software partitioning and co-synthesis tools for determining which portion of the design is synthesized into gates and which portion is compiled into embedded software; and behavioral synthesis tools), or through a manual refinement process. Finally, the RTL-level design, whether generated by hand or by previous synthesis steps, is the input to RTL- and logic-synthesis tools familiar to hardware designers; the output is a gate-level netlist.

The specification language remains the same across all levels of synthesis, and the changes in abstraction level involves a refinement into greater detail within the same language and design environment. This allows, for example, the same test bench to be used to verify the design at multiple levels, if carefully designed, resulting in a design environment that is very tightly integrated. Of course, since C++ has many constructs that are unrelated to hardware, appropriate subsets (e.g., a synthesizable RTL style [9]) will have to be used for synthesis.

Model refinement, where we proceed from an abstract specification into a more detailed one, could be in terms of either data or communication. Data refinement involves the fixing of the exact number of bits for each data item, and is typically performed late in the design phase, i.e., in the behavioral and RTL design phases. In contrast, communication refinement typically occurs early in the system design phase. Early models of system designs may employ abstract transactions for communication, which, after verification, need to be replaced by an actual implementation. SystemC provides a powerful communication refinement mechanism where such a refinement can be performed easily by first fixing the interface of the communication channels and then replacing abstract protocols with concrete implementations. Such facilities, which are absent in current HDLs, make SystemC an attractive design language at the system level.

3. HARDWARE MODELING

In order for a language to be acceptable for designing at multiple levels of abstraction, it is important that its expressive power should at least match that of the current hardware description languages familiar to designers. SystemC provides mechanisms to model the typical hardware functionality by means of constructs analogous to HDLs.

3.1 Structure and Hierarchy

Modules

Structural decomposition is one of the fundamental hardware modeling concepts because it helps partition a complex design into smaller entities. In SystemC, structural decomposition is specified with modules, which are the basic building blocks. A SystemC description consists of a set of connected modules, each encapsulating some behavior or functionality. Modules can be hierarchical, containing instances of other modules. The nesting of hierarchy can be arbitrarily deep, which is an important requirement for structural design representation.

Signals and Ports

The simplest means of connecting together different SystemC modules is by using ports and signals. Actually, the interface of modules to the external world can be much more general and sophisticated, and is described in Section 4.2, but the interface at the lowest and most primitive levels matches the typical facilities available in current HDLs. A port has an associated direction which can be input, output, or bidirectional.

Figure 3 illustrates a simple structural design consisting of a module C with hierarchical instantiations of two modules A and B within it (the instantiations being named $A1$ and $B1$ respectively) with the following characteristics:
Module A: input ports $a1$ and $a2$ and output port $a3$
Module B: input ports $b1$ and $b2$ and output port $b3$
Module C: input ports $c1$ and $c2$ and output port $c3$

The ports are connected as shown in Figure 3. The SystemC description of the above structure looks as follows.

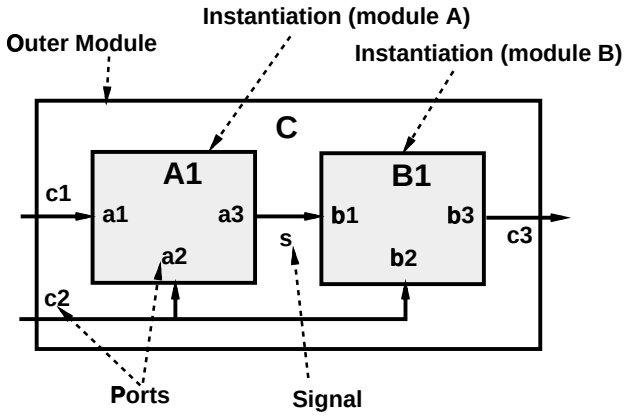

Figure 3: Illustration of modules, ports, signals, and hierarchy. Modules A and B are instantiated within module C. Signal s is used to connect ports of A and B.

```
SC_MODULE (A) { // Module declaration
sc_in<bool> a1; // Port declarations
sc_in<bool> a2;
sc_out<bool> a3;
// rest omitted
};
SC_MODULE (B) {
sc_in<bool> b1;
sc_in<bool> b2;
sc_out<bool> b3;
// rest omitted
};
SC_MODULE (C) {
sc_in<bool> c1;
sc_in<bool> c2;
sc_out<bool> c3;
A *A1;
B *B1;
sc_signal<bool> s; // signal declaration
 SC_CTOR (C) {
   A1 = new A ("A1"); // Module instantiation
   (*A1) (c1, c2, s); // Port mapping
   B1 = new B ("B1");
   (*B1) (s, c2, c3);
 }
};
```

A module is declared with the keyword `SC_MODULE`, and ports are specified with `sc_in, sc_out`, and `sc_inout` keywords,

with the template parameter <bool> indicating that the type is boolean (single bit). Other data types, including user defined ones, could also be used as port types. The structural hierarchy is specified inside the *constructor* for the module, specified with the keyword SC_CTOR. The pointer declaration and invocation of new for $A1$ and $B1$ establish a module instantiation. The port mapping parameters connect ports $c1$, $c2$, and signal s to three ports of $A1$. Thus, signal s serves as the wire connecting the output port $a3$ of $A1$ to the input port $b1$ of $B1$.

3.2 Functionality and Concurrency: Processes

The functionality of a system is described in processes in SystemC. Analogous to VHDL processes, the SystemC processes are used to represent concurrent behavior – multiple processes within a module represent hardware or software blocks executing in parallel. Processes have an associated sensitivity list – a list of signals that trigger the execution of the process. There are two important types of processes.

Methods

A method process behaves like a function call and can be used to model simple combinational behavior. It does not have its own thread of execution, and hence, cannot be suspended. This characteristic allows for high simulation efficiency.

Threads

A thread process can be used to model sequential behavior. It is associated with its own thread of execution, and can be suspended and re-activated.

A simple example involving method and thread processes is shown below. Functions p and q (whose definitions are omitted) are registered as a method process and a thread process respectively. The sensitivity list is specified using the sensitive keyword and the << operator.

```
SC_MODULE (X) {
sc_in<bool> a, b;
void p(); // Function definition omitted
void q();
 SC_CTOR (X) {
    SC_METHOD (p); sensitive << a;
    SC_THREAD (q); sensitive << a << b;
 }
};
```

3.3 Time and Clocks

Since the concepts of time and clocks are very important in modeling hardware, SystemC provides a mechanism to specify them. A clock with a period of 10 ns can be specified as:

```
sc_clock clk ("clk", 10, SC_NS);
```

The sensitive, sensitive_pos, and sensitive_neg keywords can be used to specify synchronization of a process to a clock.

```
SC_THREAD (x);
sensitive_pos << clk;
```

ensures that process x is activated on the positive edge of clock signal clk.

3.4 Test Benches

Test bench design is an important part of hardware modeling and consumes a significant amount of time. In SystemC, a test bench is specified with an SC_THREAD, just like any other process, and is easily integrated into the overall design. Sophisticated test benches can be built using all the constructs available in C++, in contrast to the relatively primitive capabilities of VHDL and Verilog with respect to, for example, file I/O, data abstraction, and text processing. Since the test bench does not need to be synthesized, there is no need to conform to any synthesizable subset of C++ while writing them.

3.5 Data Types

In addition to the standard C++ data types such as int, bool, char, etc., SystemC provides a rich set of data types which can be used to model hardware-specific concepts. We outline some useful data types here. The complete list of data types is given in [2].

4-state Logic

In addition to the standard bit values '0' and '1', it is useful to provide a mechanism to indicate that the value of a bit is *unknown.* This helps identify initialization or conflict (multiple driver) problems during simulations. Further, there is the need to specify the *high impedance* (or *tristate*) state on signals. With this in mind, SystemC provides sc_logic, a four state logic data type, the states being '0' (low or false), '1' (high or true), 'X' (unknown), and 'Z' (high impedance or tristate).

A logic vector data type, sc_lv, is used to specify data items more than one bit wide that need to be modeled with 4-state logic, e.g., a tristatable data bus. A bus can be tristated as follows:

```
sc_lv<8> data; // 8 bits wide
data = "ZZZZZZZZ"; // set to high impedance
```

SystemC also provides data types to represent resolved logic signals which is useful in modeling wires and buses with multiple drivers.

Bit and Bit Vector

The sc_bit and sc_bv types can be used to model bits and bit vectors for which only two states, '0' and '1' are sufficient, and on which logical operations such as *logical AND, logical OR*, etc are performed. Useful operations for these types include the reduction (and_reduce, or_reduce, and xor_reduce) and part-select (range). For example,

```
sc_bv<100> x,y; // 100-bit vectors
x = x | y; // logical OR
sc_logic r = x.and_reduce(); // AND reduction
sc_bv<50> z = x.range (49,0); // part select
```

Fixed and Arbitrary Precision

The integer data types provided in C++, such as int and unsigned have an implementation dependent bit width. However, the designer may wish to fix the precision of a data item if the range of values it takes is known in advance. SystemC provides two data type families for achieving this: fixed precision and arbitrary precision.

The fixed precision types sc_int and sc_uint can be used to model data that is up to 64 bits wide. These data types are implemented with a 64 bit integer. The usual operations associated with C++ integers can be applied to the fixed precision types, one useful addition being the bit-select operation. For example:

```
sc_int<48> x; // signed, 48 bits wide
sc_uint<40> y; // unsigned, 40 bits wide
x = x * y; // result truncated to 48 bits
sc_logic p = x[3]; // bit select
```

There may be cases where a bit-width larger than 64 bits is needed to model some data, for example, a wide data bus. In such cases, the arbitrary precision types, `sc_bigint` and `sc_biguint` provided by SystemC can be utilized. For example,

```
sc_bigint<128> x; // signed, 128 bit
```

The regular arithmetic operations can be performed on these types.

Fixed Point Representation

While the `float` data type can be used to model real numbers in the early stages of simulation, the hardware designer may have in mind an exact representation of such data in terms of the precision used for integral and fractional parts. The `sc_fixed` and `sc_ufixed` data types, which are used to represent such fixed point numbers in SystemC, are accompanied by the standard characteristics of fixed point arithmetic, such as quantization mode, overflow mode, and saturation bits.

```
sc_fixed<8, 5, SC_RND, SC_WRAP, 2> x;
```

defines a variable `x` with total word length: 8 bits; integer word length: 5 bits (left of decimal point); quantization mode: round to plus infinity; overflow mode: wrap around; and 2 saturation bits. The specified characteristics are used in all arithmetic for the fixed point variables. The fixed point data type is an important modeling feature of SystemC that is not found in HDLs.

The choice of data types has a significant impact on the simulation speed, and care must be taken to use the correct data types during modeling. For example, `sc_lv` should be used only in specific instances where either high impedance behavior is involved, or reset behavior in simulation is important; otherwise, the faster `sc_bv` type should be used. Similarly, if extensive arithmetic is performed, the `sc_int` and `sc_bigint` types should be preferred to `sc_bv` to prevent unnecessary type conversions. The fixed precision types (`sc_int`) should be used wherever possible (i.e., required bit width less than 64) instead of arbitrary precision (`sc_bigint`) for simulation efficiency. Finally, the native C++ types (`int`) are the most efficient.

The reader is referred to [2] for an exhaustive list of all the SystemC data types and the relevant operations.

4. SYSTEM LEVEL MODELING

The features of SystemC reviewed in the previous section make it a suitable hardware description language. However, the above mentioned capabilities would make SystemC only a minor improvement over the existing HDLs. The true advantage of SystemC as a specification language lies in the fact that it encompasses all the important hardware modeling features, as well as provides powerful modeling constructs for system level design. This ensures that the transition to a SystemC-based methodology entails no compromise in terms of expressive power at the lower levels of abstraction, and yet provides a useful framework for modeling at the higher levels. Current HDLs sorely lack the latter ability.

The system level modeling features introduced in SystemC 2.0 [3] mainly include the support of a much more general and abstract means of communication between processes and a more general mechanism for event synchronization.

4.1 Events and Sensitivity

SystemC 2.0 introduces a general mechanism for specification and notification of events. Events are no longer equated with the toggling of a single bit, but are abstract types that can be used for more general and complex interactions. The `sc_event` type can be used to declare events that can be created using the `notify` keyword and be synchronized with in `wait` statements, as shown below.

```
sc_event e1, e2; // declare events
sc_time t (5, SC_NS);
e1.notify (t); // notify event e1 after 5 ns
wait (e1); // suspend execution until
           // event e1 occurs
wait (); // wait until an event occurs
         // on the process sensitivity list
wait (10, SC_NS, e1 | e2);
    // wait for events e1 or e2 to occur
    // but for a maximum of 10 ns
```

4.2 Interfaces and Channels

In Section 3.1, we illustrated a simple example of modules connected using signals. This picture of communication, where interaction between modules is restricted to the passing of values on individual wires, is, however, at the lowest level of abstraction. At the system level, we need the ability to model more abstract, sophisticated, and intelligent communication paradigms. SystemC 2.0 introduces the notion of channels and interfaces which provide this modeling ability.

A system level SystemC design consists of a set of modules and channels at the top level. Loosely speaking, modules cover the functionality-related aspects of the system, and channels carry the communication-related aspects. Channels can be very general and implement complex algorithms within themselves, e.g., a complex bus protocol with arbitration; in fact channels can have hierarchical structure.

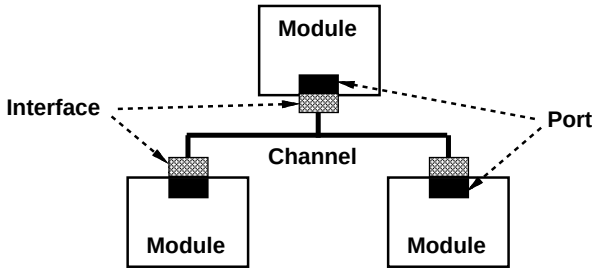

Figure 4: Interfaces and Channels. Ports of modules are connected to channels through interfaces.

Figure 4 shows a simple design with three modules connected to a channel. An interface consists of a set of method declarations (not related to SC_METHODs) implemented by the channel. These methods are visible to a port that is connected to the channel through the interface. Thus, the port (and consequently, the module) is insulated from the implementation details (such as local data) of the channel itself. This architecture helps keep the functionality-related parts of a design separate from communication, as far as possible, and allows the modification of one without affecting the other as long as the interface is unchanged.

The interface and channel constructs were inspired by similar concepts in the SpecC language [4].

4.3 Primitive and Hierarchical Channels

Channels in SystemC can be either *primitive* or *hierarchical*. Primitive channels are relatively simple; SystemC 2.0 provides a set of primitive channels which have wide applicability, such as `sc_signal` (classical signals discussed in Section 3.1), `sc_mutex`

(used to model mutual exclusion) and `sc_fifo` (used for modeling queues). Hierarchical channels can exhibit structure; they are modules themselves, which in turn, can contain processes, and other channels and modules. Complex bus protocols which have several sub-tasks can be effectively modeled using hierarchical channels.

A simple example of two modules connected by a channel of type `sc_fifo` is shown below. A FIFO connection is established between the output port of module M1 and the input port of M2.

```
SC_MODULE (A) {
sc_fifo_in<int> in;
sc_fifo_out<int> out;
// rest omitted
};
...
A *M1, *M2;  // instances of module A
...
sc_fifo<int> q (5); // create FIFO channel
                    // with buffer size 5
M1->out (q); // connect port 'out' of M1 to q
M2->in (q); // connect port 'in' of M2 to q
```

4.4 Methodology-Specific Libraries

The modeling mechanism presented above is general enough for modeling many different models of communication and computation. Different communication methodologies could be built on this basic modeling infrastructure. A future release of SystemC will contain one such example – the master-slave communication library. The elements provided here can be used to easily model communication interfaces based on master-slave bus protocols.

5. SYSTEMC ARCHITECTURE

5.1 Language Design

The overall architecture of the SystemC class library is summarized in Figure 5. The simulation kernel, i.e., the lightweight scheduler that is responsible for activating and suspending the SystemC processes is at the heart of the implementation, and forms the base layer. The generalized event mechanism discussed in Section 4.1, which forms the basis of synchronization, is introduced in the next layer. With these layers as the foundation, the communication elements – interfaces, channels, and ports are defined in the next layer. The design is based on the *interface-method-call* (IMC) scheme: essentially, the ports access the channels only through the interfaces. The example primitive channels supplied by SystemC is built on this layer. Finally, the hierarchical and other user-defined channels are built on the top layer. The most important observation is that the upper layers are built cleanly on the lower ones, and the designer can use the modeling mechanisms at any of the levels.

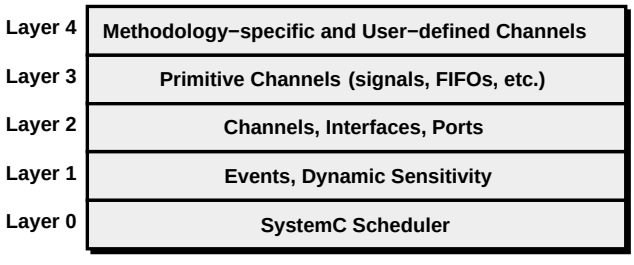

Figure 5: SystemC Language Architecture. Upper layers are built cleanly on top of lower layers.

5.2 Simulation Kernel

The simulation kernel for SystemC follows the evaluate-update paradigm that is common in HDLs. The concept of *delta cycles*, where multiple evaluate-update phases can occur at the same simulation time, is supported. A simplified version of the simulation algorithm is as follows:

1. *Initialization*: Execute all processes to initialize the system.
2. *Evaluate*: Execute a process that is ready to run. Iterate until all ready processes are executed. Events occurring during the execution could add new processes to the ready list.
3. *Update*: Execute any `update` calls made during step 2.
4. If delayed notifications are pending, determine list of ready processes and proceed to Evaluate phase (step 2).
5. Advance the simulation time to the earliest pending timed notification. If no such event exists, simulation is finished, else determine ready processes and proceed to step 2.

6. CONCLUSION

SystemC is a new modeling environment that is rapidly emerging as an industry standard for describing designs at the RTL, behavioral, and system levels. While including all the important hardware modeling capabilities offered by current generation HDLs, SystemC also provides a powerful mechanism for system level design, with the facility to specify abstract communication protocols that help model most known models of computation. The key advantage of a SystemC based design methodology is that the same language infrastructure is used for specifying the design at various levels of abstraction; thus, design refinement from abstract to detailed levels include only a change in data types and communication, and do not involve a switch of design language. All SystemC constructs have been implemented with standard C++, and no extensions to the C++ language were needed. The use of a standard C++ based framework for specifying both hardware and software parts of a design promises to smooth hardware-software co-design by bridging the long-standing specification language gap.

7. REFERENCES

[1] The Open SystemC Initiative. *http://www.systemc.org*.

[2] SystemC 1.0 user's guide. *http://www.systemc.org*, 2000.

[3] Functional specification for SystemC 2.0. *http://www.systemc.org*, Jan. 2001.

[4] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Norwell, U.S.A, 2000.

[5] J. Gerlach and W. Rosenstiel. System level design using the SystemC modeling platform. In *Workshop on System Design Automation*, pages 185–189, Rathen, Germany, Mar. 2000.

[6] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, Boston, U.S.A, 1995.

[7] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *Design Automation Conference*, pages 70–75, Anaheim, CA, June 1997.

[8] S. Swan. An introduction to system level modeling in SystemC 2.0. *http://www.systemc.org*, May 2001.

[9] Synopsys Inc., Mountain View, CA. *Describing Synthesizable RTL in SystemC*, May 2001.