Towards a New Standard for System-Level Design

Stan Y. Liao
Advanced Technology Group
Synopsys, Inc.

Abstract—Huge new design challenges for system-on-chip (SoC) are the result of decreasing time-to-market coupled with rapidly increasing gate counts and embedded software representing 50-90 percent of the functionality. The exchange of system-level intellectual property (IP) models for creating executable specifications has become a key strategic element for efficient system-to-silicon design flows. Because C and C++ are the dominant languages used by chip architects, systems engineers and software engineers today, we believe that a C-based approach to hardware modeling is necessary. This will enable co-design, providing a more natural solution to partitioning functionality between hardware and software. In this paper we present the design of SystemC, a C++ class library that provides the necessary features for modeling design hierarchy, concurrency, and reactivity in hardware. We will also describe experiences of using SystemC 1) for the co-verification of 8051 processor with a bus-functional model and 2) for the modeling and simulation of an MPEG-2 video decoder.

I. Introduction

The high level of integration provided by advances in processing technology has brought new challenges in the design of digital systems. Higher integration has given rise to a trend to integrate entire complex systems, consisting of a heterogeneous mixture of hardware and software components, onto *system-on-chip (SoC)* designs [4] [5] [6] [14]. This trend challenges EDA tool developers to provide tools that can support the design and verification of such hardware–software systems. In a traditional design methodology, hardware and software designs take place in isolation, such that the final integration takes place after the hardware is fabricated. Design errors that cannot be corrected in software often lead to costly re-fabrication and can adversely affect time-to-market. Therefore, we must seek new design methodologies in which hardware and software components are integrated and verified earlier in the design cycle.

One of the most pressing problems in hardware–software co-design and co-verification is the use of multiple languages (e.g., HDLs for hardware and C/C++ for software) and heterogeneous programming environments. In such environments the communication between hardware and software are typically accomplished by programming-language interfaces (PLIs) or some form of interprocess communication (remote procedure calls (RPCs), sockets, etc.) [8] [12]. Minimizing this communication overhead is clearly desirable. Another problem with the design flows in use today is that system architects usually start with C/C++ models, and then hands these models to the ASIC designers for translation into synthesizable HDLs—an error-prone task.

The goal of SystemC is hence twofold. First, for design verification, SystemC facilitates the co-verification of hardware–software systems by supplying a single language framework—based on standard C++—with which the designer describes both hardware and software components. This speeds up co-simulation by eliminating the use of complex PLIs or RPCs: the designer can easily exchange data between components. SystemC also provides the necessary hardware constructs so that simulation can be carried out at various, possibly mixed, levels of abstraction. Second, for implementation, SystemC allows the user to successively refine his model—without translating to an HDL. Once sufficient implementation details are available, the design can then be handed to a synthesis tool for circuit generation.

In this paper, we present the design of the SystemC Class Library [1], a brief overview of its syntax and usage, and some experiences of using it for modeling realistic systems of medium-to-high complexity. We will describe the main features of SystemC—concurrency, reactivity, data-types, and hierarchy—for modeling hardware in Section II. In In Section III we will discuss behavioral and RTL synthesis directly from SystemC code. Finally, Section IV presents two examples, an MPEG-2 video decoder and an 8051 processor, that serve to illustrate its practical usage.

II. Design of SystemC

A. Requirements for Modeling Hardware

A hardware system is typically modeled as *reactive system*: a system in continuous interaction with its environment. That is, we think of and express hardware as a set of non-terminating processes that react continuously to events in their environment [2]. The notion of reactivity is realized in Verilog and VHDL as signals and events (changes in signal values), and the ability to detect and respond to events. Most existing HDLs incorporate reactivity in using an event-driven model, and it is generally recognized that such a model is sufficient to describe most hardware

CODES 2000 San Diego CAUSA

systems at various levels of abstraction: from algorithms to gate-level circuits [7].

A.1 Reactivity

Support for reactivity requires the following:

Concurrency, or Parallelism. Hardware is inherently parallel. Concurrency in operations can be modeled using support for program threads and co-routines in the form of libraries. SystemC encapsulates concurrency in abstract C++ classes, thereby presenting to the user an easily usable interface. One can then build *non-terminating hardware processes* by employing the subtyping facilities of C++. SystemC provides three kinds of processes with performance–expressiveness trade-offs:

1. An `SC_METHOD` process executes its body from the beginning to the end every time it is invoked. A list of signals to which the process is sensitive needs to be declared, and the process is triggered whenever any of these signals changes value. `SC_METHOD` does not allow implicit execution state, but it offers the best simulation performance. It is suitable for describing combinational circuits and RTL explicit finite state machines (Section III).
2. An `SC_THREAD` process can be suspended at any point and resumed at that point the next time it is entered. Like `SC_METHOD`, it can have any arbitrary sensitivity list. It is useful for writing testbenches. The performance is usually somewhat slower than `SC_METHOD` due to context-switching overhead.
3. An `SC_CTHREAD` process is triggered by a transition (on either a positive edge or a negative edge, but not both) of a clock signal. Like `SC_THREAD`, it can be suspended and resumed at any point, and has the same context-switching overhead. However, unlike the other two kinds of processes, it is completely synchronized to a clock edge. Typically, this kind of process is used for writing and synthesizing descriptions at the behavioral level.

Ports, Signals, and Events. In a software-programming environment, communication between processes can be done through global variables or direct access to one another's internal state—as is often done at the functional level and above. However, for hardware implementation it is required that a module be self-contained and communications occur strictly through its *ports*. Furthermore, to ensure determinism in communication, signals (instead of plain variables) are the media over which data transmission occurs. (Other media, such as FIFOs, are also possible.)

Waiting and Watching. Hardware processes interact (in terms of control) primarily through signals and events. Thus they need the ability to wait or watch for a particular condition or event. Waiting refers to a blocking action that can be associated with a condition on a signal. Watching refers to a non-blocking action that runs in parallel with a specified behavior (as in "`do` *p* `watching` *s*"). This construct is used typically to handle preemptions [3]; the semantics is such that regardless of the state of execution of *p*, whenever *s* occurs, *p* is terminated. In SystemC, events on a signal can be detected by use of the `event` method; for example, `a.event()` returns `true` if there was an event on the signal or port `a`. This value can then be used to control the execution of other statements. In addition, SystemC's `SC_CTHREAD` provides mechanisms for waiting on a condition (`wait_until`) and for watching for a condition while executing a region of code (`W_BEGIN` ... `W_END`). For details of their implementation, see [7].

A.2 Data Types

A software programming language usually provides a small set of data types in terms of bit-widths, e.g., 8, 16, and 32. But a hardware designer must not be bound by this limitation—three sizes do not fit all in hardware.

SystemC provides another set of data types that gives the user the flexibility of specifying integers of any size. For example, `sc_int<12>` declares a 12-bit signed integer type and `sc_uint<25>` a 25-bit unsigned integer type. The precision of operations on these types are limited to the size of a double-word (typically 64 bits). Arbitrary-precision versions are also available: `sc_bigint` and `sc_biguint`. Fixed-point arithmetic packages such as FRIDGE [13] can be easily incorporated into SystemC.

In addition, SystemC provides 4-valued logic scalar and vector types, and the corresponding calculus. These data types are useful for modeling busses, for instance.

A.3 Structure and Hierarchy

Finally, to model hardware we must be able to express structures. In SystemC, the library abstract class `sc_module` provides the basic building block for expressing structures and design hierarchy. Processes cannot exist other than inside a module. A user-defined class derived from `sc_module` is characterized by:

1. *Ports.* Ports may be input, output, or bidirectional (inout). The template types `sc_in`, `sc_out`, and `sc_inout` are used to construct ports of different data types, e.g., `sc_in<int>` is a port whose data type is an `int`. There are other specialized ports, e.g., clocks.
2. *Internal signals.* These signals may be used to connect submodules or may be used by processes in the current module to communicate among themselves.
3. *Submodules.* A module may contain submodules.
4. *Internal member variables.* These variables may be used for saving any kind of data. One use of internal member variables is to keep the state of an RTL finite state machine.
5. *Member functions*, or *methods*. Some member functions are declared to be one of the three kinds of processes; others are used only as subroutines.
6. A *constructor.* The constructor contains declarations of member functions as processes and sensitivity lists.

It also contains the instantiation of submodules, and initialization of internal signals and internal member variables, if necessary.

B. SystemC Syntax and Usage

Since its inception SystemC has evolved substantially. At the present stage, one of the most notable changes from previous versions is the greatly simplified syntax. Ports are explicitly declared as in-ports, out-ports, or inout-ports, and the user need not explicitly (and tediously) bind ports to signals. Also, multiple processes can be declared succinctly inside a module.

Consider the following example:

```
#include "systemc.h"

struct my_module : sc_module {
  sc_in_clk  clk;
  sc_in<bool> rst;
  sc_in<int>  a;
  sc_in<int>  b;
  sc_out<int> c;
  sc_out<int> d;

  void multiply()
  {
    c = a * b;
  }
  void latched_add()
  {
    d = a + b;
    wait();
  }
  SC_CTOR(my_module)
  {
    SC_METHOD(multiply);
    sensitive << a << b;
    SC_CTHREAD(latched_add, clk.pos());
    watching(rst.delayed() == true);
  }
};

int sc_main(int ac, char* av[])
{
   sc_clock clk;
   sc_signal<bool> rst;
   sc_signal<int> a, b, c, d;

   my_module mm("mm");
   mm(clk, rst, a, b, c, d);
   stimulus st("st");
   st(clk, rst, a, b, c, d);

   sc_start(100);
   return 0;
}
```

This fragment defines a `struct` (synonymous with `class`) derived from the library base class `sc_module`, allowing it to inherit features from the latter. Then follow the port declarations—this module has six ports: an input clock port, a reset port of type `bool`, two input ports of type `int`, and two output ports of also type `int`. Two member functions, `multiply` and `latched_add`, are defined. By themselves they are just ordinary functions; an explicit statement declaring them to be SystemC processes is required. Hence, we need to define a *constructor*, using the `SC_TOR` macro (which generates code to manage hierarchical names properly). Inside this constructor, we declare `multiply` to be an `SC_METHOD` process, sensitive to changes to either of the inputs `a` or `b`; and `latched_add` to be an `SC_CTHREAD` process, clocked on the positive edge of `clk`. The `watching` statement prescribes that this process start over when the value of `rst` is `true`. Note that there is a `wait()` statement in the process `latched_add`. This statement is analogous to `@posedge` in Verilog. In general there may be more than one `wait()` statement in an `SC_CTHREAD` process. Note also that two processes are defined inside this module. A module can also contain other modules. In that case, we simply declare a member variable whose type is the submodule type, and inside the constructor instantiate it and bind its ports.

In the top-level `sc_main` function, we define the clock `clk` and the signals `rst` and `a` through `d`. We also instantiate two modules, `mm` and `st` (definition of class not shown). The statement `mm(clk, rst, a, b, c, d)` uses the positional form of port-signal binding—thus each signal is bound to the corresponding port in the order the port is declared in the module. A runtime type-check ensures that each port is bound to a signal of the correct type. Finally, `sc_start(100)` instructs the simulator to run for 100 time units.

III. Synthesis—Behavioral and RTL

Providing synthesis capability from SystemC can significantly improve designer productivity because the step of translation into HDL is eliminated. An *executable specification* is generally not ready for synthesis. Refinement is the process of adding just enough implementation details and constraints to make it an *implementable specification*, i.e., a specification that is synthesizable and can achieve good quality of results. The amount of implementation detail added depends on the degree of control over synthesis that the designer desires. Typically, the more control the designers want, the closer to register-transfer level (datapath + finite-state machine) of detail they have to specify. If a higher level of abstraction, such as behavioral level is desired, then the designer loses some control over the architecture because the synthesis tool selects it (though the designer has some control over directing the tool towards the right architecture, e.g., by specifying constraints on resources and timing).

Refinement consists of three steps. The first step is called data refinement, whereby C/C++ built-in types are replaced by data types of the right precision determined by the designer. One example of such refinement is the refinement of floating-point types to fixed-point types.

The second step is control refinement. The most important aspect of control refinement is specifying the input/output behavior of each block in the design, i.e., when inputs are sampled and when outputs are produced. During control refinement, a designer may decide what level of abstraction she desires to synthesize from. For a behavioral level of abstraction, defining the I/O behavior and setting the design constraints is all that is required. To refine a design to RTL, the designer has to create the finite-state machine and the datapath herself.

The third step in refinement is to ensure that the appropriate coding policies have been followed and that all synthesizable models use constructs from the synthesizable subset. The synthesizable subset for C/C++ consists of the entire C language except the obvious constructs that do not have well-defined hardware semantics, namely dynamic memory allocation, arbitrary pointers, arbitrary gotos, recursion, etc.

Since the C/C++ language can support various levels of abstraction, there are few restrictions on what can be specified for synthesis. The hardware semantics of C/C++ operators and expressions involving such operations are similar to that of most HDLs. Semantics of control flow statements such as if-then-else, switch-case, while, etc., are also well defined. A structural datapath consisting of arithmetic operators can be specified as easily as a finite-state machine. A behavioral model (one without an architecture) can specified more easily.

Currently we are working on synthesis tools for SystemC, both for behavioral and for RTL. These tools are much more than simple translators. They involve both language-independent and -dependent analyses and optimizations. Not only are we implementing applicable state-of-the-art compiler optimizations (see [9]), we have also discovered synthesis-specific optimization opportunities and are implementing algorithms to solve them. These tools have been successfully integrated into the existing design flow.

IV. Practical Experiences with SystemC

In this section we will describe some practical experiences with SystemC in modeling some realistic systems of medium-to-high complexity. These exercises confirm the efficiency of SystemC, and provide us with insights for laying the foundations of future design methodologies.

A. An MPEG-2 Video Decoder

The MPEG-2 video coding standard specifies the syntax of the video bit-stream and the corresponding video decoding process. One of the main applications of the MPEG-2 video coding standard is the encoding of interlaced video at TV and HDTV resolution.

The design flow in this exercise starts with a model of the entire decoder, including the testbench, in the Synopsys COSSAP environment. The synthesizable decoder part is then split into a control-flow part, modeled with the Synopsys Protocol Compiler (PC) tool, and a signal-processing part, modeled in synthesizable C++ code based on the SystemC Class Library.

At the COSSAP system level, the rather complex MPEG-2 decoder algorithm is split into several building blocks of lower algorithmic complexity. These blocks may be written in any of the HDLs, or SystemC (as in the present exercise), at a high-level of abstraction. Afterwards they may be easily replaced with synthesizable SystemC code or pre-compiled intellectual property (IP) blocks, and simulated against previous models.

The control-flow dominated part of the MPEG-2 decoder consists of the bit-stream parser and is modeled for synthesis using Protocol Compiler. PC allows for modeling the video parser at an abstraction level very close to that of the official MPEG-2 specification, and can generate bit- and cycle-true SystemC code for simulation as well as for synthesis.

Finally, the signal-processing part is modeled in SystemC for synthesis with the Synopsys SystemC Compiler. The following algorithms in this part are implemented: inverse quantization (IQ), inverse discrete cosine transform (IDCT), and motion compensation. At the algorithmic level, no details about timing, concurrency, reset behavior, and bit-true types were employed. To progress towards synthesis, the designer refines the system by adding more and more implementation details until a synthesizable behavioral or RTL form is reached. SystemC's hardware constructs makes possible this refinement in the a single language.

B. An 8051 Processor

We now present the co-simulation of an 8051 processor with a bus-functional model [10]. A bus-functional model (BFM) is a key component in any co-verification solution; it is a useful abstraction for the verification and evaluation of a processor-based design. The BFM provides a programming interface that can be used by the software directly. Since in the beginning stages of the design process the software runs on the host processor (on which development is done), this model is untimed because software execution time is inaccurate. In later stages of the design process, an instruction-set simulator (ISS) needs to be used in conjunction with the BFM in order to execute the instructions for the target processor. The ISS makes use of the same programming interface to communicate with the BFM. Since this ISS can be cycle-accurate, one can perform cycle-accurate simulations at this stage.

In the SystemC environment, a BFM is a module that is derived from SystemC library class `sc_module`. The ports of this module correspond to the pins of the processor. The BFM class has several methods that provide a programming interface to the software or to the ISS. The methods provided depend on the type of communication between hardware and software. The functionality of the BFM itself is modeled as a set of finite-state machines (that can

execute in parallel). In the SystemC environment, the programming interface to BFMs is more or less fixed, i.e. the interface methods have the same prototype for all BFMs, though some BFMs may support methods that are not supported in others. This allows the user to swap one processor model for another easily, without having to change the C/C++ source code. This capability is important because it allows the user to explore different architectures with different processors. The various pre-defined types for handling addresses, data, registers, etc., can be specialized inside each BFM, thereby allowing complete freedom for each BFM to define these types appropriately. The BFM interface consists of methods for accessing memory-mapped I/O, interrupt-driven I/O, configuration ports, internal registers, timers, and serial ports. In addition, it has a set of performance-estimation functions.

A system consisting of the Synopsys DesignWare DW8051 core [11] has been modeled using SystemC. It has been tested with a set of software programs exercising various features of the BFM. At this level (bus-functional-accurate), the simulation performance is up to three times faster than co-simulation with HDL-based commercial tools. This results primarily from the simplification of the communication between hardware and software.

Finally, a cycle-accurate ISS for a subset of the DW8051 architecture has also been developed and integrated with the BFM. This ISS is implemented as a single process that fetches, decodes, and executes the 8051 instructions. Several techniques (reducing activity on memory bus and gating clocks), have been used to optimize the ISS so that, even with cycle-accuracy, the performance is just 10% slower than host-based execution of software. This figure would typically increase for more-complex architectures (e.g., superscalar, pipelined architectures).

V. Conclusion

Increasingly complex systems and higher level of integration require us to reconsider our design methodology. In this paper we have addressed and, presented a solution for, a significant part of the puzzle, namely, that of a system-level design language. Our approach is pragmatic; we take an existing and widely-used language, C++, and provide a simulation environment that is both fast and robust. Our experiences with the MPEG-2 decoder and the 8051 processor have been encouraging, and we have also made tremendous progress on synthesis from SystemC at both the behavioral and RTL levels. We believe SystemC is an important step towards a new standard for system-level design.

Acknowledgments

The author would like to thank Luc Séméria and Abhijit Ghosh for providing the results and insights from the 8051 experiments, and to Norman Weyrich, Ulrich Holtmann, and Rocco Jonack for the MPEG-2 video decoder example.

References

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

[2] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[3] G. Berry. Preemption in concurrent systems. In *Proc. FSTTCS'93, Lecture Notes in Computer Science*, volume 761, pages 72–93. Springer-Verlag, 1993.

[4] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design & Test of Computers*, pages 64–75, December 1993.

[5] F. Balarin et al. *Polis: A Design Environment for Control-Dominated Embedded Systems*. See *http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html*.

[6] R. K. Gupta and G. De Micheli. A Co-Synthesis Approach to Embedded System Design Automation. *Design Automation for Embedded Systems*, 1(1-2), January 1996.

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

[8] Mentor Graphics Corp. *Seamless Co-Verification*. See *http://www.mentorgrahics.com/seamless*.

[9] S. Muchnick. *Advance Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[10] L. Séméria and A. Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings of the Asia South-Pacific Design Automation Conference*, Yokohama, Japan, January 2000.

[11] Synopsys, Inc. *DesignWare DW8051 Macrocell Solution. http://www.synopsys.com/products/designware/8051_ds.html*.

[12] Synopsys, Inc. *Eagle Tools*. See *http://www.synopsys.com/eagle*.

[13] M. Willems, V. Bursgens, H. Keding, T. Groetker, and H. Meyr. System-level fixed-point design on an interpolative approach. In *Proceedings of the Design Automation Conference*, pages 293–298, June 1997.

[14] W. Wolf. Hardware-Software Co-design of Embedded Systems. *IEEE Proceedings*, 82(7):965–989, July 1994.