TEACHING COMPUTER ORGANIZATION AND ARCHITECTURE USING SYSTEMC*

Ed Harcourt
Dept. Mathematics
St. Lawrence University
Canton, NY 13617
edharcourt@stlawu.edu

ABSTRACT

Hardware simulation is often used in courses that contain a hardware component. We describe and introduce SystemC, a C++ library for designing, simulating, and analyzing digital systems. We compare and contrast the strengths and weaknesses of SystemC to other technologies used in hardware courses such as breadboards and other simulation technologies including schematic capture and traditional hardware description languages Verilog and VHDL. We ascertained the strengths and weaknesses of using SystemC as a teaching tool by having a student use SystemC to design a subset of the MIPS microprocessor described in the popular textbook [6].

1. INTRODUCTION

A variety of courses in a computer science curriculum contain a hardware component; courses such as digital design, computer organization and architecture, real-time (embedded) systems, and even breadth based introductory courses [7]. These courses will usually use either breadboards or hardware simulation to support student assignments and projects. If a decision has been made to use simulation then there are choices to make regarding the simulation paradigm to use and the tools, languages, and pedagogical infrastructure required to support that paradigm.

Simulation paradigms fall broadly into two categories; *schematic capture* and *language based*. Each paradigm has its own strengths and weaknesses that play a role in which to use. We compare and contrast the strengths and weaknesses of these two

---

paradigms and then focus on a relatively new entry in the language based camp, SystemC, and evaluate its efficacy as a teaching tool.

The rest of the paper proceeds as follows. In section 2 we compare technologies currently used in courses that have a hardware component and analyze their strengths and weaknesses. Section 3 provides a basic introduction to SystemC using canonical examples. Section 4 provides an analysis of SystemC's strength and weaknesses with relation to previously mentioned technologies. Section 5 concludes.

2. CURRENT PRACTICE

A variety of technologies are used to support hardware design with each technology having its own strengths and weaknesses. In a learning environment we are largely concerned with; *cost*, *ease-of-use*, *scalability*, and *abstraction*. *Cost* not only includes the dollar amount the student or university must incur but also any resources that the tool might consume, for example IT resources, dedicated workstations, or lab space. *Ease-of-use* refers to the overhead a student incurs in learning to effectively use the tool. For example, does the student spend more time learning the tool than they do learning about hardware? *Scalability* refers to the tool's ability to handle larger designs that a student might develop in a project based course or senior project. *Abstraction* refers to the tool's capability to allow the user to specify hardware at higher levels of abstraction rather than in terms of lower level gate level implementations. Scalability and abstraction are related but not the same. Technologies that support abstraction will almost always be scalable, but not all scalable technologies support abstraction.

2.1 Breadboards

Many courses use physical non-software logic environments consisting of breadboards, logic chips, and wires where students design and build real working digital circuits. These environments are excellent for learning introductory digital logic design and provide students with a hands-on lab experience that is satisfying and rewarding for the student while also being effective pedagogically [4]. Breadboards are also easy to use.

Breadboards are good when teaching lower-level hardware. However there are many situations where they may not be the best choice. For example, a breadth first introductory course that has a short digital logic component might find the overhead of acquiring and maintaining them prohibitive. Breadboards may or may not be portable requiring dedicated lab and storage space. While breadboards do not have to be expensive they still have a non-negligible cost especially when compared to the availability of free hardware simulation software. Furthermore if students don't have their own breadboard assigned to them for the duration of the semester or project then labs and exercises must be carefully constructed so that they can be completed in one lab session.

Breadboard environments do not support scalability. For example, if a student has constructed a four-bit register using four flip-flops they can not quickly duplicate that four bit register to make another register without rewiring four more physical flip-flops. In more advanced courses breadboard environments do not scale well to larger designs

that involve advanced digital concepts such as pipelines and caches. Maintaining, modifying, and debugging complicated digital designs is also difficult using breadboards.

2.2 Schematic Capture

Graphical circuit drawing environments, commonly called schematic capture, are popular as they provide a visual representation of the circuit analogous to the way circuits are drawn on paper. For smaller designs schematic capture tools are relatively easy to use. Users drop components on a palette, wire them together, and hit a simulate button. Students can quickly construct, test, and simulate simple hardware designs in a matter of minutes. At our university our computer organization course includes a three week digital logic component where students use MultiMedia Logic, a free schematic capture tool [8].

Figure 1 shows a simple graphical representation of a full-adder rendered in MultiMedia Logic.

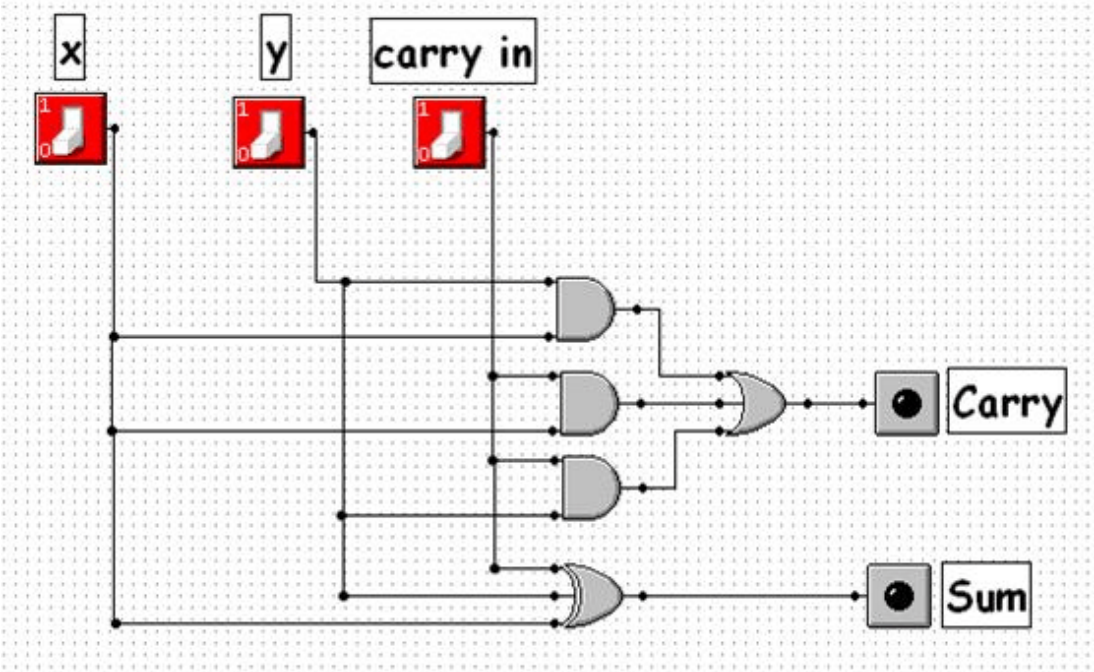

Figure 1: A three input adder in a schematic capture tool.

One issue all simulation environments have to deal with is how the simulator handles input and output with the user. As seen in figure 1 MultiMedia Logic contains some simple interactive devices such as switches and lights that are easy to understand, quick to insert into the circuit and easy to motivate pedagogically.

The major weakness of schematic capture tools is that they, similar to breadboard environments, do not scale well for larger designs though they do provide better support than breadboards. For example users can copy and paste portions of a circuit and many tools allow designs to be hierarchical with a single graphical component acting as a container for sub-components.

Schematic capture tools provide little or no support for abstraction. For example, in a finite state machine the next state function is, abstractly, a function which can be

implemented using functions, arrays, or conditional statements; constructs that schematic capture tools do not support.

Another consideration is that there is no standard graphical ``language" that all schematic capture tools implement, requiring special instruction for each tool. Additionally, if one is concerned about career relevance for students headed to industry we should point out that the use of schematic capture tools is on the decline. A recent survey of 137 engineers found that the use of graphical entry is waning and that the vast majority of engineers use a hardware description language. One out of three engineers plan to use SystemC within the next six months [2].

2.3 Language Based Design

Language based simulation refers to using textual languages that are similar to imperative programming languages but contain dedicated language constructs for designing and simulating hardware (rather than software). Language based environments are usually based on the *hardware description languages* (HDLs) Verilog [9] and VHDL [1]. Verilog and VHDL are both IEEE standards and are used heavily in industry. Many tools are available for designing, testing, debugging, and synthesizing an HDL based design. Language based design is now commonplace and is even used in the popular textbook by Hennessy and Patterson [6]. There are a variety of free simulators available as well as commercial versions available at little or no cost to universities.

Analogous to a chip, a block of hardware is best visualized as a black box with a pin interface. HDLs allow the designer to capture this block oriented nature of hardware. Systems are composed of interconnected blocks communicating through wires, where a wire is connected to a pin on a block. In HDLs a block is called a *module*, a pin is called a port, and a wire is called a *signal*. Figure 2 shows an example module hierarchy of how HDls support hierarchical block structure.

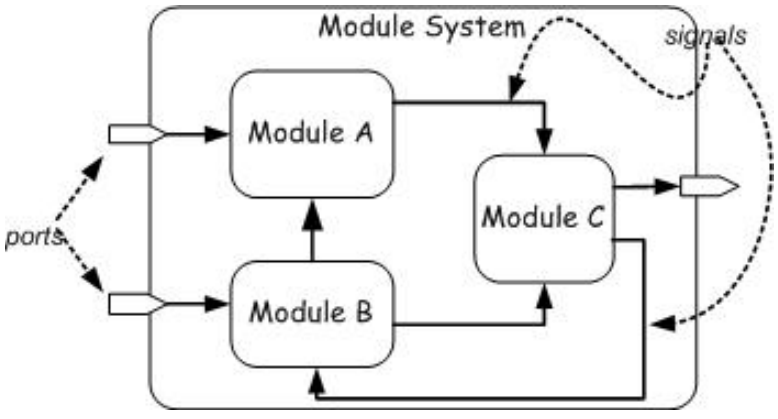

Figure 2: A hierarchical block diagram showing modules, ports, and signals.

Figure 3 shows a Verilog version of the adder of figure 1. The module `adder` is a top-level module composed of sub-modules (the gates). The gate wiring is specified by the arguments to the modules. Line 5 says that the output signal sum is the exclusive-or of `a`, `b`, and `cin`. Line 6 says that an internal signal ab is the output of a and-ed with b. If no bit-width is specified Verilog implicitly types the signals and ports to contain a single bit. The internal signals `ab`, `ac`, and `bc` are implicitly declared.

The main strengths of HDLs is their support for large designs (scalability) and abstraction. A module can be instantiated as many times as needed in the circuit. It is not uncommon for HDL designs to encompass tens of millions of gates. A circuit of this size would be unwieldy in a schematic capture tool.

HDLs also provide better support for abstraction than either schematic capture or breadboards, though as we will point out they still don't provide good support. HDLs provide many of the same programming constructs found in standard imperative languages such as loops, conditionals and functions. A function can be expressed using programming constructs as opposed to having to derive the logic equations for the function. For example the adder in figure need not be described in terms of gates but could instead be described using mathematical operators.

```
1   module adder(sum, cout, a, b, cin);
2       output sum, cout;
3       input a, b, cin;
4
5       xor (sum, a, b, cin);
6       and (ab, a, b);
7       and (ac, a, c);
8       and (bc, b, c);
9       or (cout, ab, ac, bc);
10  endmodule
```

Figure 3: A full adder in Verilog

HDLs are not easy to use. They are special purpose languages with a specific syntax and semantics that requires students learn a new language that is quite different than languages they already know. HDLs are large, complex, and have a large learning curve. Commercial tools that support HDLs are expensive (though are often discounted substantially for universities) but also require IT support to install and license them on a network.

HDLs still lack the kinds of abstraction mechanisms found in modern programming languages such as classes, polymorphism, and templates. Furthermore, as designs become more abstract the boundary between what part of a system is implemented in hardware and what part is in software becomes blurred. None of the technologies discussed so far have any support for implementing the software portion of a system.

To overcome these problems with HDLs it is common for high level hardware designs to be developed in C++ first. These designs are then used as a specification or reference implementation for a subsequent Verilog or VHDL implementation. It is especially common for high-level processor simulators to be developed in C or C++. These instruction set simulators are then used by both software and hardware engineers as a platform to develop software before the processor is actually built and as an executable specification for the hardware engineer. Another benefit of more abstract C++ designs is that they often simulate faster than their HDL counterparts.

There are three main disadvantages to using C++ to model hardware. First, the way hardware behaves and sequential programming languages behave is very different. Hardware is inherently parallel where an occurrence of an event (*e.g.*, a clock edge) at time t controls which portions of the hardware are updated at time $t + 1$. HDLs support this with a discrete event simulation semantics. Contrast this with an imperative

programming language such as C++ where control proceeds sequentially from one statement to the next with no notion of time. Another disadvantage is that C++ lacks hardware like data types (*e.g.*, bit vectors), and operations on those data types. The final disadvantage is that C++ does not provide the proper structuring mechanism to model hardware as a hierarchy of interconnected blocks. HDLs have dedicated syntax to express this, C++ does not.

3. SYSTEMC

In this section we briefly introduce SystemC using the adder example we have been using up to this point. In the next section we analyze SystemC in terms of its ease of use, scalability, and support for abstraction.

SystemC is a C++ library for simulating digital systems that contain both hardware and software components [3,5]. Strictly speaking SystemC is a C++ library, but we often describe SystemC as a new ``language''. This is because SystemC supports a style of programming that provides hierarchical block structure, discrete event simulation semantics, inter-process communication primitives, and a variety of hardware oriented data types. This leads to a programming paradigm that is quite different than traditional sequential programming in C++.

SystemC was developed and is currently administered under an industry sponsored consortium called the Open SystemC Initiative (OSCI) and has wide industrial support from both electronic design automation (EDA) tool vendors and hardware engineers who are designing and implementing real systems [5]. (See a variety of press releases on the EE Times web site www.eetimes.com.) Work has also begun on making SystemC an IEEE standard.

SystemC addresses the problems of using standard C++ by introducing HDL-like constructs in C++, an existing language already familiar to most students. C++ is a language that also provides good abstraction mechanisms; support for abstract data types, OOP, and generic programming. SystemC provides a discrete event simulation engine that keeps track of events and time.

SystemC allows the user to express block structure through the use of predefined classes that represent blocks, wires, and pins. SystemC also contains a large library of useful hardware oriented data types such as bits, bit vectors, four-valued logic types, fixed-point types, and sized integers. Furthermore, since SystemC is a C++ library SystemC development tools are just the C++ compilers, debuggers, and development environments that students are already familiar with. Hence, the ramp-up time to learn SystemC is lower than a traditional HDL such as Verilog or VHDL.

3.1 A Simple Example

Our first SystemC example is to implement the simple adder from figure 1. A gate is a module and the half-adder is a module composed of sub-modules. (The motivation for designing SystemC was not for describing gate-level circuits. Most would consider it an abuse to use it in this manner. However, smaller gate-level circuits are canonical and

provide simple examples that highlight the capabilities of SystemC as well as being easy to understand.)

We'll construct the adder bottom-up by first creating modules for the gates. Figure 4 shows an example And gate. A module is created by declaring a class that derives from the SystemC class `sc_module` (line 1).

```
1   class And : public sc_module {
2   public:
3       sc_in<bool> a;
4       sc_in<bool> b;
5       sc_out<bool> out;
6
7       SC_CTOR(And) {
8           SC_METHOD(run);
9           sensitive << a << b;
10      }
11
12      void run() {
13          out.write(a.read() && b.read());
14      }
15  };
```

Figure 4: A SystemC module for an and gate.

Modules have two kinds of ports, input ports and output ports. Ports are declared using the SystemC classes `sc_in` and `sc_out`. The port types `sc_in` and `sc_out` are template types parameterized with the type of value that is communicated on the port. In our example ports are all single bits which we represent using the predefined C++ type `bool`. Lines 3-5 show the port declarations.

SystemC supports an event-based style of programming where a module listens for events on ports. An event handler executes when an event occurs. In SystemC terminology, to listen for an event, a module declares that it is *sensitive* to events on particular ports. These sensitivity declarations are specified in the module constructor and use an overloaded input stream operator << (line 9). To indicate which event handler should execute on an event we use the SystemC macro `SC_METHOD` and specify the function name as an argument to the macro (line 8). Event handlers are just functions in the module class that the constructor specifies as such. Line 8 in the constructor indicates that the function run should get called when there is an event on either port `a` or `b`. Presumably the run function carries out the actual computation by anding the values on the a and b ports and putting the result on the port `out`.

SystemC provides a macro `SC_CTOR` to aid in declaring the module constructor (line 7).

The run function (lines 12-14) of the and-gate puts a value on the output port out by using the function `write` defined in the `sc_out` class. Values are read from an input port using the function `read` defined in the `sc_in` class.

An exclusive-or gate (not shown) is identical to the And gate except that the run function computes the exclusive-or of `a` and `b` rather than logical and. To remove code duplication we could use inheritance to construct a new base class `Gate` that contained the port definitions and a virtual function run that subclasses `And` and `Xor` would

override. Alternatively, rather than using inheritance we could use templates and factor out the computational portion of the gates and make it a template parameter.

3.2 Constructing the Adder

The final task is to build the adder from instances of `Xor`, `And`, and `Or` gates. Referring to figure 1, the adder has three input ports `a`, `b`, and `c`, and two output ports `sum` and `carry`. Lines 3-4 in figure 5 show the port declarations. Line 5 declares the internal signals needed to connect the gates. Lines 17-19 declare the instances of the gates.

All that remains is to connect the inputs, outputs, and signals to the gates. Connectivity is specified in the constructor (lines 8-12). For example, line 8 connects the `a`, `b`, and `c` inputs of the adder to the inputs of the `Xor` gate and the output of the adder sum to the output of the `Xor` gate.

Figure 5 shows the complete version of the adder.

```
1   class Adder : public sc_module {
2   public:
3       sc_in<bool> a, b, c;
4       sc_out<bool> sum, carry;
5       sc_signal<bool> tmp1, tmp2, tmp3;
6
7       SC_CTOR(Adder) {
8            x1(a, b, c, sum);
9            ab(a, b, tmp1);
10           ac(a, c, tmp2);
11           bc(a, b, tmp3);
12           o1(tmp1, tmp2, tmp3, carry);
13           sensitive << a << b << c;
14      }
15
16  private:
17      Xor x1;
18      And ab, ac, bc;
19      Or3 o1;
20  };
21
```

Figure 5: An adder constructed from And, Or, and an Xor gates.

3.3 Simulating the Design

As we mentioned earlier every simulation technology needs a way to get inputs to the design and observe outputs. In the case for schematic capture the example used faux switches and lights. For language based designs we test modules by constructing a *test bench*, another module that instantiates the *design under test*. In the case for the adder the test bench provides inputs (stimulus) to the at regular intervals (controlled by a clock). Figure 6 shows a test bench for the adder and a main program that instantiates the adder and runs the simulation for 10 time units. The test bench generates random stimulus to the adder on every clock pulse. The clock pulse is provided by a predefined SystemC clock module.

To supply values on the adders input ports we need to be able to write to them. But the class `sc_in` does not provide a `write` function, only `read`. To remedy this we connect a *wire* to the input ports of the half-adder and write to the wire instead. Wires in SystemC are called *signals* and are declared using the class `sc_signal`. This is analogous to hooking up input wires and switches to circuits in a breadboard environment.

```
1   class TestBench : public sc_module {
2   public:
3       sc_clock clock;
4       Adder fa;
5       sc_signal<bool> x, y, z, sum, carry;
6
7       SC_CTOR(TestBench) {
8            fa(x, y, sum, carry);
9            SC_METHOD(run);
10           sensitive << clock;
11      }
12
13      void run() {
14           x.write(random_bool());
15           y.write(random_bool());
16           z.write(random_bool());
17      }
18  };
19
20  int sc_main(int argc, char* argv[]) {
21      TestBench tb("tb");
22      sc_start(10);
23  }
```

Figure 6: A module that acts as a component to test the adder.

SystemC designs require the user provide an `sc_main` function (lines 19-22) rather than a `main`. The function (`main` is defined by the SystemC library to initialize the simulation and then call `sc_main`). The simulation is started by calling the function `sc_start` with the number of time units to simulate for (or -1 to simulate until there are no more events to process).

3.4 Debugging

Debugging event driven programs is more complicated than debugging sequential programs. Nevertheless since SystemC programs run in standard C++ development environments a C++ debugger can be used to step through designs, set breakpoints in modules, and view the values on signals and ports.

The values of ports and signals vary over time. Tools for viewing these value changes are called *waveform browsers*. SystemC provides a trace capability that outputs value changes on ports and signals to a VCD (*value change dump*) file, a standard IEEE file format supported by all waveform browsers. To create a trace file that tracks changes on the `carry` signal of the half adder we add the following statements to our main program.

```
sc_trace_file *tf =
      sc_create_vcd_trace_file("adder_out");
sc_trace(tf, tb.ha.carry, "carry");
```

The designer can have as many trace files open as needed and trace as many signals at a time as needed, mindful that file I/O can slow simulation speed considerably. Figure 7 shows a waveform for the half-adder signals using the free waveform browser from SynaptiCAD.

Figure 7: Viewing the half-adder signals using a waveform browser (SynaptiCAD).

3.5 More Abstract Models

Hardware design methodologies typically define levels of abstraction with gate models being the lowest level and behaviorial models being the highest level. At the behavioral level circuits are defined in terms of algorithms. Another level is emerging called the *system level* where systems are described without regard to whether they will be implemented in software, hardware, or both.

SystemC provides support for more abstract modules in various ways. One way to make models abstract is to use abstract data types rather than low-level types such a bool. For example, defining a 32-bit adder in terms of the predefined SystemC type `sc_int<32>` as opposed to instantiating 32 one-bit adders.

Ports and signal types need not be limited to simple scalar data types. Ports and signals are templated types that can carry any data type such as a type that defines a network packet.

```
struct NetworkPacket {
      char header[10];
      char body[255];
};
```

A port declaration can then use `NetworkPacket` as the type of a port; `sc_in<NetworkPacket>`. To some extent this manner of abstracting using higher-level data types is also available in HDLs (VHDL has aggregate types, Verilog does not).

Hardware blocks communicate and synchronize through events generated by writing to ports. This a low hardware like communication model. SystemC allows the designer to change the communication model by either replacing it with another predefined model (e.g., dataflow) or by having the user define their own communication model. For example, an abstract model of a block that represents a hardware FIFO might use blocking reads/writes rather than implementing the lower level read/write protocol in terms of pins/events. Abstracting a model in this way is unique to SystemC and is not supported at all in traditional HDLs.

A third way in which SystemC designs can be made more abstract is to leverage the abstraction mechanisms of C++. For example, using templates, the Standard C++ library, and concepts such as function objects, a user can create generic reusable hardware components.

4. ANALYSIS

In this section we analyze SystemC using our three criteria; ease of use, scalability, and abstraction. We'll tackle scalability and abstraction first. SystemC's support for scalability is excellent, similar to other HDLs. Modules are designed once and instantiated as many times as needed. Managing the size and complexity of the design is similar to managing the size and complexity of a large software design. Proper use of separate compilation, header files, and source control eases the burden. Large designs are constructed hierarchically from top-level modules using sub-modules.

Because SystemC is a C++ library SystemC's support for abstraction is unsurpassed. No current hardware modeling language in current use provides the kind of abstraction mechanisms that SystemC does (for example, support for OOP, polymorphism, generic programming, overloading, and operator overloading).

Because SystemC piggybacks on an already familiar programming language it was hoped that learning SystemC would be easier than either Verilog or VHDL. This is not necessarily the case. Even though C++ may be familiar the event driven, concurrent, and time based programming style needed to use SystemC effectively is still foreign to most undergraduate students. Furthermore there are several ways in which using C++ as the base programming language causes problems that users of other HDLs do not experience. These are outlined in the next section.

4.1 Pitfalls

While there are many advantages of using the SystemC library in modeling hardware there are also many problems a beginning user will encounter. These problems were uncovered by students in the course of constructing a subset of the MIPS as described in the [6].

Advanced C++ knowledge required. C++ is a complex language. The SystemC library uses many sophisticated features of C++ including heavy use of operator overloading and advanced template features such as partial template specialization. Even the simplest C++ model uses class inheritance and templates. Many of these advanced uses of C++ is transparent to the user however not all. At times it is necessary to consult the reference manual or the SystemC code directly to understand what classes and functions are available to the user. At these times a deeper understanding of C++ is needed.

Debugging the Simulator. When debugging a SystemC model using a C++ debugger it is easy to accidentally step into the SystemC library code (e.g., the simulator code). This can be confusing to the student. This is a common problem in C++ environment in general since most of the Standard C++ library is provided in header files studetns frequently find themselves stepping into library headers files.

Hardware design errors often manifest themselves as C++ errors or simulation time errors. The user's design is just a C++ program and the host C++ compiler knows nothing about SystemC or hardware. Hence errors that would normally be caught by a SystemC ``aware'' compiler show up as C++ syntax errors in the host C++ compiler or as run-time errors. For example, for efficiency the SystemC simulator passes values on ports by reference. To know this requires knowledge of how the SystemC simulator is implemented. Unaware, the user might try to pass a value by reference themselves. But in C++ a reference to a reference is illegal and the compiler will issue a syntax error pointing to a line not in the user's code but deep in a SystemC header file.

Limited availability of SystemC tutorials, textbooks and sample models. At this time, there are few SystemC tutorials, textbooks and sample models available for an instructor to use. While the SystemC library is free and there is a comprehensive user's guide and language reference manual, these are not targeted to students nor are they suitable for introductory courses. Contrast this with the plethora of tutorials and textbooks available for Verilog and VHDL. A reference implementation of SystemC and the language manuals are freely available from (www.systemc.org).

5. CONCLUSIONS

Most digital systems today have both a hardware and a software component. SystemC is the first design language to attract wide support that enables both the hardware and software portions of a digital system to be designed using a single, common standard language. While SystemC has many benefits and is very powerful, this power comes at a price. Instructors and students need to be keenly aware of the potential problems that using such a general infrastructure entails. For lower level digital courses where students may not yet be familiar with C++ or the advanced C++ features used in the SystemC library, SystemC may not be suitable. In advanced undergraduate and graduate courses SystemC can provide an excellent vehicle to support class or senior projects.

6. REFERENCES

[1] P. Ashenden. The Designer's Guide to VHDL. Morgan Kaufmann, second edition, 2002.

[2] R. Goering. 137 engineers sound off on verification tools. EE Times, June 2004. http://www.eedesign.com/news/showArticle.jhtml?articleId=21400956.

[3] T. Grotker, S. Liao, G. Martin, and S. Swan. System Design with SystemC. Kluwer, 2002.

[4] L. Ivanov. A hardware lab for the computer organization course at small colleges. The Journal of Computing Science in Colleges, 19(2), December 2003.

[5] Open SystemC Initiative, http://www.systemc.org. SystemC.

[6] D. Patterson and J. Hennessy. Computer Organization And Design, The Hardware/Software Interface. Morgan Kaufmann, 2005.

[7] G. M. Schneider and J. Gerstring. Invitation to Computer Science. Thomson, second edition, 2004.

[8] Softronics. Multimedia logic. available from http://www.softronix.com.

[9] D. E. Thomas and P. R. Moorby. The Verilog Hardware Description Language. Kluwer, ¯fth edition, 2002.