Using a Programming Language for Digital System Design

RAJESH K. GUPTA
University of California, Irvine
STAN Y. LIAO
Synopsys

HDLs must satisfy important semantic requirements, especially when CAD tools are involved. Designers can meet these requirements by using the standard language constructs of a software-programming language to model hardware for simulation and synthesis.

AS DIGITAL CIRCUITS AND SYSTEMS become more complex, system designers are increasingly concerned about system-modeling tools and their impact on productivity and hardware design quality. In addition, they want to quickly produce a working hardware model, simulate it with the rest of the system, and synthesize and/or formally verify it for specific properties. Toward this end, designers are using textual languages based on high-level programming languages to express executable behaviors. Indeed, languages such as C, VHDL, and Verilog are common in large-scale system design and debugging. Undoubtedly, this growth in the use of textual programming languages stems from system designers' familiarity with general-purpose, high-level programming languages.

Using programming languages for hardware specification can significantly shorten the system designer's learning curve and enables simulation of complete systems for correct functionality. There are pitfalls, however, in following a pure software-programming-language description to model hardware—mainly, inefficient results from synthesis tools. Consequently, language developers often modify and extend software-programming languages to produce hardware description languages (HDLs) geared specifically to hardware modeling. Most semantic extensions concern structural components, exact event timing, and operational concurrency—concepts absent from most software-programming languages.

To be useful, a design language must help the system designer meet the following goals:

- correctly and unambiguously model hardware behavior at various levels of abstraction;
- simulate the hardware model along with the rest of the system, which may contain software components; and
- synthesize an efficient hardware solution, using existing CAD tools.

As we shall see, these requirements, combined with the diversity of hardware models and their uses, impose a significant burden on a modeling language. For instance, levels of abstraction may range from a gate-level view of the hardware to a behavioral view in which operations may move across cycles. Sometimes a description consists of a mixture of abstraction levels. Simulating such a multilevel hardware model is crucial in a design process that iteratively refines the language-level model as implementation proceeds in various parts of the system.

Generally, today's synthesis solutions transform register-transfer-level descriptions binding operations to specific cycles into gate-level hardware descriptions consisting of combinational and sequential circuit blocks. Recently, however, behavioral synthesis tools have become available. These tools translate limited behavioral descriptions into an RTL intermediate form before proceeding to gate-level hardware. To facil-

0740-7475/97/$10.00 © 1997 IEEE

itate these tools, a modeling language must meet certain semantic and methodological requirements.

Hardware modeling requirements

We can model a hardware system as a reactive system—that is, a system in continuous interaction with its environment. Thus, we would think of and describe hardware as a set of nonterminating processes that react continuously to events in their environment.[1] Kurshan[2] first introduced the concept of reactive behavior, and system developers have since applied it in process-control and real-time systems.[1,3] Language requirements for reactive systems fall into two broad categories: semantic, essential for correct and unambiguous hardware modeling, and pragmatic, dictated by marketplace needs and implementation issues.

Semantic requirements. The following language features are essential for correct hardware system modeling.

Abstraction. The language must provide an abstraction mechanism—that is, a means of building large systems from smaller, HDL-modeled components. The two main methods of composing large systems are procedure calls and declarative blocks. Designers typically use the latter to describe a netlist of blocks in which component instances are bound by shared variables. These blocks are useful in modeling structural components.

It may also be necessary for the designer to represent an external view of the component independent from its implementation(s). Thus, the interface specification must be separate from the component's body specification. Such a feature is useful for iteratively refining a hardware model through successive implementations, without altering the external view of the component.

Reactive programming. As mentioned earlier, hardware models are best represented as nonterminating programs in continuous interaction with their environment. It is easy to capture nonterminating interaction with programming languages that use control constructs such as while loops, but these structures severely constrain implementation and event handling in a concurrent environment. For hardware modeling, reactivity requires the basic mechanisms of synchronization and exception handling. The following are typical HDL constructs for modeling these mechanisms:

- `wait(`*condition*`)` can specify a fixed amount of delay or an event occurrence. Its implementation can either translate it as a busy-wait loop (`while(!`*condition*`);`) or associate it with a specific communication structure (for example, event broadcasting in the Esterel programming language[1]). Typically, designers use wait to synchronize operations across simultaneously active modules. In languages that do not assume an implicit communication structure, it also explicitly specifies communication structures for data transfers among hardware modules.
- `watching(`*port or condition*`) {`*block*`}` can be associated with an event (an action instance such as I/O, assignment, or condition setting) for a given block. Designers typically use `watching` to model the hardware's handling of interrupt events. HDLs that provide this construct are considerably more powerful than traditional programming languages and other models. For instance, Esterel provides `watching` by encasing a behavior in a `do {`*block*`} watching` *signal.* As the number of these signals (or interrupts) increases, an equivalent finite-state model's size increases exponentially, whereas the Esterel program grows only linearly.
- `disable(`*name*`)` provides limited exception handling, such as termination of a block of behavior associated with a specific exception condition. Hierarchical disable operations (in Verilog, for example) execute and/or terminate specific process blocks to prescribe the extent of the exception handling. The net effect of using `disable` to suspend execution of a simultaneously active process is to issue an interrupt on the named process. In other words, `disable` is a sender-side analog of `watching.`

Determinism. At any level of abstraction, system simulation should be predictable, yielding the same outputs from the same sequence of inputs. Although a modeled behavior may be nondeterministic or even indeterminate[4]—that is, we may not have complete information to predict this unique behavior—we can always decompose it into deterministic parts.

Simultaneity. We often need simultaneous actions to model hardware's inherent parallelism. One programming-language mechanism for modeling parallelism is concurrent compositions. In these compositions, concurrency control is critical. We usually introduce serialization in concurrent compositions if a dependency exists between concurrent actions or if one of the actions represents a break in the control flow. Serialization preserves the natural semantics of continue and break statements.

Using concurrency to model hardware parallelism is not very efficient for two main reasons. First, in process-level concurrency,[5] communication is a synchronizing activity. In contrast, parallel hardware actions often proceed independently, and even in cases where the actions communicate, rarely do they require explicit synchronization. Second, during interpretation, the time synchronization takes is un-

Timing uncertainty makes most concurrent programming languages a poor choice for modeling hardware systems.

known (and may even be unbounded). Since communication is related to synchronization, all communication, by definition, takes an unknown amount of time. This timing uncertainty makes most concurrent programming languages a poor choice for modeling hardware systems in which much of the synthesis (and verification) requires parametric determinism in time (and other cost measures).

Existing HDLs model simultaneity at varying degrees of operation granularity. HardwareC[6] and Verilog allow specification of nested sequential and simultaneous blocks, whereas VHDL allows simultaneity only at the process level. Processes in VHDL execute in parallel. The two-phase (signal update followed by process execution) event-timing model,[7] which provides a method of determining a unique total order of actions, effectively rules out temporal nondeterminism. In contrast, Verilog parallelism is convenient to program but difficult to debug (with sequential consistency in mind), and it makes nondeterministic simulation a possibility.

Time. Typically, HDLs support logical time. That is, time is a progress marker; the advancement of time represents simulation progress. In contrast, real time progresses with no relation to event processing.

Pragmatic requirements. In addition to the semantic requirements, an HDL must meet pragmatic and syntactical requirements.

Data types. Some languages, such as Verilog, support only "small" data types such as bit and Boolean. VHDL supports "big" types such as arrays, records, and pointers. General-purpose programming languages such as C++ support abstract and dynamic data types. Support of these data types provides improved abstraction and information hiding, but it may also make synthesis and verification tasks difficult. In addition to small and big types, language support for precise (bit-true) data and operations is sometimes important in modeling digital signal-processing algorithms, which often use fixed-point data types and/or saturation arithmetic.

Interface abstraction. VHDL and Ada provide an entity interface that can be totally independent of the component implementation. This abstraction, which allows fast, separate compilation, is also available through C++ classes. A disadvantage of a separate interface abstraction is loss of efficiency due to the preclusion of intercomponent optimization, which eventually affects simulation speed.

Communication model. The model can base intercomponent communication on shared variables—that is, component parameters bound to the same variable, as in VHDL. (Lately, dispute has arisen over whether shared variables should remain in VHDL.[8]) Such communication often requires an explicit communication architecture. Implicit communication architectures, such as the synchronous remote subprogram call or Esterel's instantaneous broadcast, are alternatives not often tried in hardware modeling.

Time and clock model. Most HDLs maintain a time model that equates global time with simulated time. In this model, physical time is frequently the same as logical time. The HDL may also support multiple clocks, although most hardware systems do not require distributed clocks or their explicit phase relationships.

Design tools. A design methodology builds upon a modeling formalism and supporting tools. Among the necessary tools are language analyzers, or front ends, that allow separate compilation, essential for large-scale system modeling and design. Another necessary tool is the execution engine, for which the definition of a virtual-machine interface is an important issue. In addition, specification checkers, verifiers, synthesizers, and presentation/debugging environments are critical to an HDL's hardware modeling effectiveness.

Multiple logic values. Support for multiple logic (tristate or quad-state) values is useful in modeling signal contention and indeterminism in sequential elements. To model signal contention, it is not necessary to know the conflicting drivers on a signal; its value indicates a conflict exists. This is useful in event management that isolates event generation from the event consumption module.

Simulation and synthesis

Event-driven simulation remains the primary means of executing the model of a reactive system. Traditional event-driven simulators such as those that implement HDLs maintain a notion of simulated time and track every write to a signal. Thus, we can schedule signal writes to occur at a future simulation time. A write that changes a signal's value generates an event that can resume waiting processes or divert the control flow of watching processes. An event-driven simulator requires a scheduler that maintains time, a

list of pending signal writes, and a list of waiting and watching processes for each signal.

Event-based modeling, which models hardware systems at various levels of detail by associating events to behaviors of interest, is quite powerful. However, this generality also imposes a significant burden on simulation efficiency due to the extra work needed for event maintenance and processing. Event processing often requires interpretation of event generation, propagation, and disposition by the simulation model. That is, a separate procedure provides a semantic interpretation of an object in the context of the simulation model, either statically at compilation time or at runtime. A separate module that holds the state of execution performs runtime interpretation; the simulation engine invokes this module whenever interpretation is needed.

In contrast, in native execution, the host machine (running the simulation) holds the state of execution. Since each call to the interpreter may require a context switch (to another simulated context, for instance), execution cost versus context-switching cost dictates the trade-off between native execution and interpretation. Often, designers use a hybrid approach that encapsulates native codes and allows interpretation at a coarse level.

HDLs support interpretation at either the action or process level. In action-level interpretation, each action statement requires a call to the interpreter. For instance, a VHDL model associates interpretation with events. Since each signal assignment potentially can generate an event, frequent calls to the interpreter (or the event manager) often slow system simulations. Process-level interpretation reduces this overhead by making sure that each process invocation requires a call to the interpreter. In VHDL, since event processing affects processes (not actions), we can turn event processing around—to the receiver instead of the sender. For example, we can stipulate that all processes synchronize to a common clock (typically required by hardware synthesis tools anyway). In this clock-synchronous execution model, the sensitivity list of a process is statically known. Once invoked, a process executes without interruption to the next clock boundary. No events are processed between clock boundaries.

In process-driven simulation, processes check for signal changes and conditions for which they are waiting or watching. In the extreme, this can become busy waiting, an expensive proposition. For synchronous systems, performing the checks during clock changes is often sufficient. Some cycle-based simulators use this approach.[9] Process-driven simulation, however, may incur many context-switching operations.

Event-driven simulations best suit hardware models in which each process is small, with only a few inputs, and event activity is low, with few events per unit of simulated time. For such models, the event-driven approach avoids unnecessary computation. For example, an event-driven simulator avoids evaluating a gate whose inputs have not changed. When event activity is high or when processes are large, scheduler overhead may overwhelm the savings from avoided computations. (In large processes, the probability that all inputs remain unchanged is low.)

In system and behavioral models in which large processes read and write many signals, cycle-based simulation offers higher speed. For this reason, microprocessor designers usually build a model in C and simulate it with their own cycle-based simulator.[9] In the following section, we describe a C++ library that combines the event-driven and process-driven approaches. The library has a simple scheduler that tracks only clock signals, but the checks on signal changes and conditions have shifted from processes to the scheduler to reduce context-switching overhead.

Synthesis is typically a concern in HDLs whose semantic constructs create embedded simulator directives. As a result, HDL developers have proposed language restrictions to ensure that HDL-modeled behavior can be synthesized. These restrictions, adopted by many HDLs, generally prohibit the use of runtime-created processes, permit statically resolved data types only (no interconverting arrays or pointers), and disallow type casts. In addition, the hardware modeler must follow several methodological restrictions to ensure the quality of synthesis results. For example, the modeler must assign initial values to the variables and default cases when using case statements. Without such assignments, the synthesis tool must assume the program language semantics associated with a variable, which keeps its value until assigned otherwise. The omission of assignments on default control paths may lead to excessive storage elements in the resulting hardware.

Another important issue is how a language treats different branches of a case statement. Some languages, such as Verilog, do not require different branches of a case statement to be mutually exclusive, and deterministic execution decides the priorities among branches. In these cases, the synthesis tool may generate the case statement control as a priority encoder structure rather than a multiplexer as in the case of mutually exclusive branches. In contrast, VHDL assumes mutually exclusive branches.

As we will show, designers can model hardware using an object-oriented software-programming language such as C++ without modifying the language constructs and without using interpretation of assignment operations.

Hardware modeling

Underlying the C++ library we present here is the idea that system design is essentially a programming activity. The library supports this activity by supplying a single language framework in which the designer describes both hardware and software components. We must stipulate only that hard-

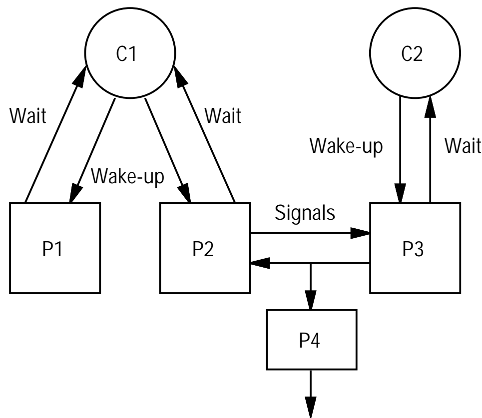


Figure 1. A system modeled as clocks (C1 and C2) and processes (P1–P4). A wait message suspends sequential processes P1–P3 by transferring control to their clocks; the processes continue when the clocks issue a wake-up message. Sequential processes communicate through signals. On the other hand, combinational process P4 computes its outputs only when its inputs change.

ware components be synchronous, a weak limitation since most hardware designs *are* synchronous. We use C++'s facilities to implement hardware-modeling features that a front end can easily map to a synthesizable intermediate representation. Designers can continue to use widely available standard compilers and debuggers to verify their designs, and commercially available synthesis tools to implement their hardware.

C++, a programming language unburdened by event-processing overhead, is familiar to most designers who write software. Our library uses subtyping and templates, C++'s object-oriented extension facilities, to implement reactivity in a manner natural to hardware designers. It also assists designers in modeling data types (for example, VHDL standard logic) and structural hardware elements (such as ports and port maps). Specifically, the library supports the following concepts:

- *Concurrency, or parallelism.* Hardware is inherently parallel. Designers can model program threads or coroutines in the form of libraries. If we encapsulate concurrency in a library base class, we can build nonterminating hardware processes by using C++'s subtyping and virtual-function facilities.
- *Signals and events.* Hardware processes require signals and events to communicate. C++ thread/concurrency libraries provide other communication primitives such as semaphores and critical regions, which make C++ attractive for writing software. But their use usually assumes that processes have easy access to one another's states—in other words, that it would be possible to refer to a process's internal variables. This assumption is inappropriate for hardware modeling and synthesis. Thus, we use signals on which events will be detected; signals are implemented with C++ templates.
- *Waiting and watching.* Hardware processes interact through events and signals. Thus, they need the ability to wait or watch for a particular condition or event. As we have seen, waiting is a blocking action (as in "`wait until (`*expression*`)`") that we can associate with an event. Watching is a nonblocking action that runs in parallel with a specified behavior (as in "`do` ***p*** `watching` ***s***"). We typically use watching to handle preemptions or interrupts.[1] The semantics of this construct govern that whenever *s* occurs, *p* terminates, regardless of *p*'s execution state.

Modeling interacting processes. Designers typically model a hardware system as a set of clocks and interacting processes, as shown in Figure 1. Processes are of two types: sequential and combinational. (We use the term *combinational* in the sense that these processes are not clocked and hence have no execution states, although the synthesis tool may generate latches.) Sequential processes are always synchronized by a clock; combinational processes have no delay and are evaluated whenever any input changes.

To implement concurrency in the C++ programming environment, we map each sequential process to a separate thread, which maintains the process's execution state. In contrast, a combinational process has no execution state, so it does not need mapping to a separate thread. For our purposes, a nonpreemptive thread library, such as QuickThreads,[10] is usually sufficient; the overhead of preemptive thread libraries, such as Solaris Threads, may well outweigh their benefits.

An associated clock synchronizes events generated by each process. The system identifies each event as a delayed signal assignment and passes the associated action to the clock. A process synchronizes with its clock by issuing it a wait message. The clocks are responsible for performing actions (signal updates) at the end of the cycle and then "waking up" the processes.

Modeling structures and data types. In addition to describing behaviors, a hardware model must describe structures. A structural description consists of component instances and their interconnection—in a netlist, for example. Two central concepts in structural modeling are ports and port maps. Ports are windows through which a process "sees" the outside world and receives information from its environs. We model ports using C++ references to signals; signals, in turn, are entities to which we map ports.

Port mapping takes place at object instantiation time, when the constructor of each process object (a process implemented in C++ as an object) binds signal arguments to the object's ports.

HDLs provide multiple-valued logic for representing unknown or don't-care values. We can represent these values in C++ by defining a new aggregate type **`std_ulogic`** and overloading the logic operators. We can easily incorporate a complete implementation of **`std_ulogic`** and **`std_ulogic_vector`**, as well as bit-vector-based signed and unsigned types, into the C++ class library.

Describing sequential processes. We declare a sequential process class by publicly deriving it from the library base class **`Process`**; the process thereby inherits fundamental capabilities defined in the class library, such as reactivity. We declare the process's inputs and outputs as member variables, specifically references to appropriate signals. Instantiating an object of the class initializes these references. The base class requires a clock (of type **`Clock`**). The member function **`entry( )`** separately specifies the process's behavior.

Figure 2 shows an example declaration and definition of a sequential process. Figure 2a shows the declaration of a counter that takes an input **`enable`** and produces an output **`iszero.`** The constructor **`Counter( )`** is responsible for connecting the input and output signals to the actual arguments supplied to it (**`EN`** and **`ZERO`**), as well as for taking a clock argument and passing it to the base-class initialization (**`Process(CLK)`**).

Figure 2b shows the body of the **`Counter`** process, specified in **`entry( )`**. This function demonstrates the use of two basic features of the **`Process`** class: **`write( )`** and **`next( )`**. Calling **`write( )`** places an event on the clock's list of actions. For instance, **`write(iszero, '1')`** schedules an update at the next clock edge for the signal **`iszero.`** The statement **`next( )`** synchronizes the process with the next clock edge. Because every **`Process,`** after its creation, has a handle to the clock on which it is synchronized, **`next( )`** does not need to explicitly specify this clock. The **`entry( )`** function behaves much like a VHDL process; that is, its body repeatedly executes, even though there is no explicit enclosing loop.

Describing combinational processes. We declare a combinational process by publicly deriving it from the library base class **`Combo`**. Like sequential processes, the process's inputs and outputs are declared member variables, and its function is specified in **`entry( )`**. However, because a combinational process is not clocked, to invoke the process when inputs change, the user-defined constructor must call **`decl_input( )`** or **`decl_output( )`** for each input or output signal. These function calls also allow the library to topologically order the combinational processes so that we can construct a static schedule or detect combinational cycles.

```
class Counter : public Process {
private:
        // clock is in the base class
        const  Signal<std_ulogic> & enable;      // input
               Signal<std_ulogic>& iszero;       // output
                         int count;              // state
public:
        Counter(
               // interface specification
                         Clock& CLK,
               const Signal<std_ulogic>& EN,
                    Signal<std_ulogic>& ZERO
               )
               // initializers - mapping ports
               : Process(CLK), enable(EN), iszero(ZERO)
        {
               count = 15;    // process initialization
        }
        void entry();
};
```

(a)

```
void Counter::entry()
{
        if (enable.read() == '1') {
           if (count == 0) {
                  write(iszero, '1');
                  count = 15;
           }
           else {
                  write(iszero, '0');
                  count—;
           }
        }
        next();
}
```

(b)

*Figure 2. A sequential process: declaration of the Counter class in C++ **(a)**; body of the Counter process **(b)**.*

Figure 3 (next page) shows the declaration and definition of a combinational process—a multiplexer. Unlike **`Counter, MUX`** is not clocked; hence, the constructor requires no clock argument. The body of **`MUX,`** specified in **`entry( )`**, writes the value of *a* to *d* if the select input *s* is 0, or the value of *b* to *d* otherwise.

Instantiating processes. We instantiate a process in C++ by defining a variable of the appropriate process class and supplying arguments to the constructor. We instantiate clocks and signals first, followed by processes. Then, we call **`Clock::press_start_button( )`** to begin the simulation.

```
class MUX : public Combo {
private:
        const Signal<std_ulogic>& a;
        const Signal<std_ulogic>& b;
        const Signal<std_ulogic>& s;
              Signal<std_ulogic>& d;
public:
        MUX( const Signal<std_ulogic>& A,
              const Signal<std_ulogic>& B,
              const Signal<std_ulogic>& S,
                    Signal<std_ulogic>& D)
              : a(A), b(B), s(S), d(D)
        {
              decl_input(a);
              decl_input(b);
              decl_input(s);
              decl_output(d);
        }
        void entry( )
        {
              if (s.read() == '0')
                  write(d, a);
              else
                  write(d, b);
        }
};
```

Figure 3. *A combinational process: declaration of a multiplexer in C++.*

```
int main()
{
        Signal<std_ulogic> enable, iszero;
        Signal<std_ulogic> select, out;

        Clock clk;
        Counter counter(clk, enable, iszero);
        MUX mux(iszero, enable, select, out);

        Clock::press_start_button(1000);
}
```

Figure 4. *Instantiating a process and beginning simulation.*

```
void Counter::entry()
{
        wait_until(enable == '1');
        if (count == 0) {
                write(iszero, '1');
                count = 15;
        }
        else {
                write(iszero, '0');
                count—;
        }
}
```

Figure 5. *Using delay-evaluated expressions.*

In the example in Figure 4, we create the signals **enable, iszero, select**, and **out**; a clock; a sequential process of class **Counter**; and a combinational process of class **MUX**. Then we begin a simulation of 1,000 cycles. (Of course, this code fragment serves only as an illustration and does nothing useful since there is no stimulus.)

It is important to note that declaring and defining a process class defines a behavior, whereas defining a variable of a particular class creates an instance. Therefore, we can conceivably create several instances of the same process class without duplicating code or explicitly passing objects as arguments. Object-oriented languages such as C++ offer a less cumbersome way to create instances than procedural languages.

Reactivity: waiting and watching. Our implementation of waiting introduces the notion of delay-evaluated expressions (DEEs). To implement watching, we use DEEs in conjunction with the C++ exception-handling mechanisms.

Waiting. A waiting process suspends itself until an event occurs. For example, after its initialization, a process may wait for assertion of the **start** signal before starting operation. In VHDL we can write: **wait until start = '1'**;.

Using this wait for a synchronous digital circuit requires that **start** be sampled at a clock edge. We accomplish this in VHDL as follows:

```
loop
  wait until clk'event and clk = '1';
  exit when start = '1';
end loop
```

The analogous expression in C++ is

```
do {next( );} while (start.read( ) !=
  '1');
```

Although this expression achieves the desired effect, it is not very efficient. Every invocation of **next( )** causes a context switch to the next process in the clock's process list or to the clock, and in some machines context switches can be expensive.

These context switches to and from the current process serve only to evaluate the expression *E* on which the process is waiting. If this evaluation can occur outside the current process, we can avoid many unnecessary context switches. Instead of unconditionally switching back to evaluate *E*, our decision to switch will depend on the value of *E*: We will switch only when *E* evaluates to true in the current cycle.

To permit other processes to evaluate expression *E*, a process must have a way of making *E* known externally. If we were to write the expression as shown here, it would be eval-

uated immediately, leaving us with a truth value, not an expression. Therefore, we must delay-evaluate *E*. The key is to create an object that encapsulates expression *E* and a method **`eval( )`** that allows anyone with a handle to the object to force its evaluation.

We provide the method **`wait_until( )`** to take advantage of DEEs. For instance, **`wait_until(start == '1');`** has the same semantics as the **`do-while`** loop shown earlier but is more efficient. Note that while we previously wrote **`start.read( ) == '1'`**, here we write **`start == '1'`**. This is because we have overloaded the operator == such that when a signal appears in the expression, a DEE is created. The DEE in turn serves as an argument for the method call to **`wait_until( )`**.

To gain simulation efficiency, we write the **`Counter::entry( )`** function using **`wait_until( )`** as shown in the example in Figure 5.

Watching. As mentioned earlier, an important property of reactive systems is the ability to react to preemptions or interrupts. One of the most commonly used preemptions in hardware design is the reset, which, regardless of the system's present state, always brings the system to a known initial state. It is the system's responsibility to watch for preemptions at all times.

VHDL and Verilog conspicuously lack the ability to handle interrupts such as the reset elegantly. For instance, even if all signals (including those carrying preemptions) external to processes are synchronized at clock boundaries, VHDL requires the designer to test for such signals at every clock boundary:

```
reset_loop: loop
  ...
  wait until clk'event and clk = '1';
  --VHDL
  exit reset_loop when reset = '1';
  ...
end loop
```

Here, **`reset_loop`** is the outermost loop that encloses the reset sequence and the main loop. For pre- and post-synthesis simulations to yield the same results, these statements are required. This would be tolerable if **`reset`** were the only preemption the process was watching for. However, as we add more preemptions to watch for, the code can become unwieldy. Furthermore, we would not be able to exploit the delayed evaluation described earlier because the preemption could occur before the awaited condition. We agree with Berry and Gonthier that support for preemption should be orthogonal with respect to other constructs.[1]

An elegant way to solve this problem is to use the C++ exception-handling mechanisms: **`try, catch`**, and **`throw`**. We associate each process with a watch list, to which we add DEEs during object instantiation. All the DEEs on the watch list will be evaluated on every clock edge. For instance, the statement in the user-defined constructor **`watching(reset == '1')`** creates a DEE corresponding to a **`reset`** condition, and registers this condition with the watch list. This mechanism allows a process to watch several preemptive conditions.

```
Counter::Counter()
{
        watching(reset == '1');
}

Counter::entry()
{
        try {
                wait_until(enable == '1');
                if (count == 0) {
                        write(iszero, '1');
                        count = 15;
                }
                else {
                        write(iszero, '0');
                        count—;
                }
        }
        catch(UserWatch&) {
                if (reset.read() == '1') {
                        count = 15;
                }
        }
}
```

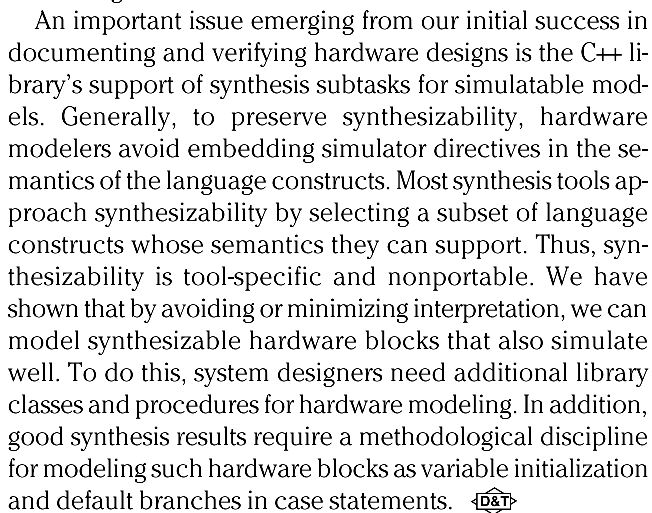
Figure 6. *Using C++ exceptions to handle reset.*

The **`entry( )`** function uses the **`try/catch`** construct to implement preemption handling. The **`next( )`** or **`wait_until( )`** functions may throw a C++ exception of type **`UserWatch`**. These functions receive a notice from the clock when one or more DEEs in the watch list evaluate to true. Instead of returning normally, they throw an exception that is caught by the **`catch`** block. The code within the **`catch`** block handles the preemptions.

In Figure 6, we extend the **`Counter`** from the previous examples to include a reset signal. The constructor **`Counter( )`** now consists of the method call to **`watching( )`**, which registers the **`reset`** condition. The **`entry( )`** method now uses **`try/catch`** to handle the reset. The **`try`** block specifies the process's normal behavior; the **`catch`** block specifies the action to be taken when **`reset`** is asserted.

OUR EXAMPLES SHOW that designers can accurately model hardware behavior with programming languages, with-

out using new semantic interpretations that would ultimately slow down simulation. Consequently, we can reasonably expect hardware design to someday be taught as a programming activity, bringing the hardware implementation task into the purview of the system architect. However, the characteristics of hardware and the limitations of simulation and synthesis tools make it important to implement reactivity and data types differently from high-level software implementations. Although questions remain about the viability of pure programming approaches for complex digital hardware design, ample evidence shows that HDLs can produce substantial gains over schematic-entry and gate-level design methods.



An important issue emerging from our initial success in documenting and verifying hardware designs is the C++ library's support of synthesis subtasks for simulatable models. Generally, to preserve synthesizability, hardware modelers avoid embedding simulator directives in the semantics of the language constructs. Most synthesis tools approach synthesizability by selecting a subset of language constructs whose semantics they can support. Thus, synthesizability is tool-specific and nonportable. We have shown that by avoiding or minimizing interpretation, we can model synthesizable hardware blocks that also simulate well. To do this, system designers need additional library classes and procedures for hardware modeling. In addition, good synthesis results require a methodological discipline for modeling such hardware blocks as variable initialization and default branches in case statements. D&T

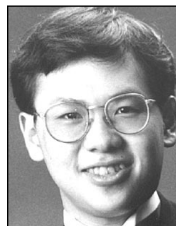
Acknowledgments

We acknowledge the support received by Rajesh K. Gupta from National Science Foundation Career Award MIP 95-01615, NSF/DARPA contract ASC-96-34947, and NSF grant EEC 89-43166.

References

1. G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, Vol. 19, No. 2, 1992, pp. 87-152.
2. R.P. Kurshan, "Reducibility in Analysis of Coordination," *Lecture Notes in Computer Science*, Vol. 103, 1987, pp. 19-39.
3. M.A. Ardis et al., "A Framework for Evaluating Specification Methods for Reactive Systems," *IEEE Trans. Software Engineering*, Vol. 22, No. 6, June 1996, pp. 378-389.
4. R.E. Filman and D.P. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, New York, 1984.
5. R. Milner, "Operational and Algebraic Semantics of Concurrent Processes" in *Handbook of Theoretical Computer Science*, Elsevier, New York, 1990.
6. D. Ku and G. De Micheli, *HardwareC—A Language for Hardware Design (Version 2.0)*, Tech. Report CSL-TR-90-419, Stanford Univ., Stanford, Calif., 1990.
7. P.J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann, San Mateo, Calif., 1996.
8. R. Goering, "VHDL Shared Variables Create Dissent—Spec Divides EDA," *Electronics Engineering Times*, No. 937, Jan. 20, 1997.
9. C. Hansen, "Hardware Logic Simulation by Compilation," *Proc. Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 712-715.
10. D. Keppel, *Tools and Techniques for Writing Fast Portable Threads Packages*, Tech. Report UW-CSE-93-05-06, Univ. of Washington, Seattle, 1993; available at ftp://ftp.cs.washington.edu/tr.

Rajesh K. Gupta is an assistant professor of computer science at the University of California, Irvine. His current research focuses on system-level design and CAD issues. Previously, he worked as an assistant professor at the University of Illinois, Urbana-Champaign. Still earlier, Gupta worked at Intel on design teams for three generations of microprocessor devices. He is the coauthor of a patent for a PLL-based clock circuit and the author of *Cosynthesis of Hardware and Software for Digital Embedded Systems*. The University of Illinois nominated Gupta for the NSF Presidential Faculty Fellow Award in 1996. He received the National Science Foundation Career Award in 1995 and two Intel departmental achievement awards. Gupta received the PhD from Stanford University and the MS from UC Berkeley.



Stan Y. Liao is a researcher in the Advanced Technology Group of Synopsys, Inc. His research emphasis is behavioral synthesis and high-level language compilers for hardware-software codesign of embedded systems. He has coauthored several papers on code generation and optimization for embedded processors. Liao received his SB, SM, and PhD degrees in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a member of the ACM and the IEEE.

Direct questions and comments about this article to Stan Y. Liao, Advanced Technology Group, Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043; stanliao.91@alum.mit.edu.