Design and Verification Languages

Stephen A. Edwards
Department of Computer Science
Columbia University, New York, New York

November, 2004

Abstract

After a few decades of research and experimentation, register-transfer dialects of two standard languages—Verilog and VHDL—have emerged as the industry standard starting point for automatic large-scale digital integrated circuit synthesis. Writing RTL descriptions of hardware remains a largely human process and hence the clarity, precision, and ease with which such descriptions can be coded correctly has a profound impact on the quality of the final product and the speed with which the design can be created.

While the efficiency of a design (e.g., the speed at which it can run or the power it consumes) is obviously important, its correctness is usually the paramount issue, consuming the majority of the time (and hence money) spent during the design process. In response to this challenge, a number of so-called verification languages have arisen. These have been designed to assist in a simulation-based or formal verification process by providing mechanisms for checking temporal properties, generating pseudorandom test cases, and for checking how much of a design's behavior has been exercised by the test cases.

Through examples and discussion, this report describes the two main design languages—VHDL and Verilog—as well as SystemC, a language currently used to build large simulation models; SystemVerilog, a substantial extension of Verilog; and OpenVera, e, and PSL, the three leading contenders for becoming the main verification language.

Contents

1 Introduction

Hardware description languages—HDLs—are now the preferred way to enter the design of an integrated circuit, having supplanting graphical schematic capture programs in the early 1990s. A typical design methodology in 2004 starts with some back-of-the-envelope calculations that lead to a rough architectural design. This architecture is refined and tested for functional correctness by implementing it as a large simulator written in C or C++ for speed. Once this high-level model is satisfactory, it is passed to designers who re-code it in a register-transfer-level (RTL) dialect of VHDL or Verilog—the two industry-dominant HDLs. This new model is often simulated to compare it to the high-level reference model, then fed to a logic synthesis system such as Synopsys' Design Compiler, which translates the RTL into an efficient gate-level netlist. Finally, this netlist is given to a place-and-route system that ultimately generates the list of polygons that will become wires and transistors on the chip.

None of these steps, of course, is as simple as it might sound. Translating a C model of a system into RTL requires adding many details, ranging from protocols to cycle-level scheduling. Despite many years of research, this step remains stubbornly manual in most designs. Synthesizing a netlist from an RTL dialect of an HDL has been automated, but is the result of many years of university and industrial research, as are all the automated steps after it.

Verifying that the C or RTL models are functionally correct presents an even more serious challenge. At the moment, simulation remains the dominant way of raising confidence in the correctness of these models, but has many drawbacks. One of the more serious is the need for simulation to be driven by appropriate test cases. These need to exercise the design, preferably the difficult cases that expose bugs, and be both comprehensive and relatively short since simulation takes time.

Knowing when simulation has exposed a bug and estimating how complete a set of test cases actually is are two other major issues in a simulation-based functional verification methodology. The verification languages discussed here attempt to address these problems by providing facilities for generating biased, constrained random test cases, checking temporal properties, and checking functional coverage.

2 History

Many credit Reed [44] with the first hardware description language. His formalism, simply a list of Boolean functions that define the inputs to a block of flip-flops driven by a single clock (i.e., a synchronous digital system), captures the essence of an HDL: a semi-formal way of modeling systems at a higher level of abstraction. Reed's formalism does not mention the wires and vacuum tubes that would actually implement his systems, yet it makes clear how these components should be assembled.

In the five decades since Reed, both the number and the need for hardware description languages has increased. In 1973, Omohundro [39] could list nine languages and dozens more have been proposed since.

The main focus of HDLs has shifted as the cost of digital hardware has dropped. In the 1950s and 60s, the cost of digital hardware remained high and was used primarily for general-purpose computers. Chu's CDL [13] is representative of languages of this era: it uses a programming-language-like syntax; has a heavy bias toward processor design; and includes the notions of arithmetic, registers and register transfer, conditionals, concurrency, and even microprograms. Bell and Newell's influential ISP (described in their 1971 book [6]) was also biased toward processor design.

The 1970s saw the rise of many more design languages [12, 10]. One of the more successful was ISP'. Developed by Charles Rose and his student Paul Drongowski at Case Western Reserve in 1975–76, ISP' was based on Bell and Newell's ISP and used in a design environment for multiprocessor systems called N.mPc [42]. Commercialized in 1980 (and since owned by a variety of companies), it enjoyed some success, but starting in 1985, the Verilog simulator (and accompanying language) began to dominate the market.

The 1980s brought Verilog and VHDL, which remain the dominant HDLs to this day (2004). Initially successful because of its superior gate-level simulation speed, Verilog started life in 1984 as a proprietary language in a commercial product, while VHDL, the VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language, was designed at the behest of the US Department of Defense as a unifying representation for electronic design [17].

While the 1980s was the decade of the widespread commercial use of HDLs for simulation, the 1990s brought them an additional role as input languages for logic synthesis. While the idea of automatically synthesizing logic from an HDL dates back to the 1960s, it was only the development of multi-level logic synthesis in the 1980s [11] that made them practical for specifying hardware, much as compilers for software require optimization to produce competitive results. Synopsys was one of the first to release a commercially successful logic synthesis system that could generate efficient hardware from register-transfer-level Verilog specifications and by the end of the 1990s, virtually every large integrated circuit was designed this way.

Hardware description languages continue to be important for providing inputs for synthesis and modeling for simulation, but their importance as aids to validation recently has grown substantially. Long an important part of the design process, the use of simulation to check the correctness of a design has become absolutely critical, and languages have evolved to help perform simulation quickly, correctly, and judiciously.

Clearly articulated in features recently added to SystemVerilog, it is now common to automatically generate simulation test cases using biased random variables (e.g., to generate random input sequences in which reset occurs very little), check that these cases throughly exercise the design (e.g., by checking whether certain values or transitions have been overlooked), and by checking whether invariants have been violated during the simulation process (e.g., making sure that each request is followed by an acknowledgement). HDLs are expanding to accommodate such methodologies.

3 Design Languages

3.1 Verilog

The Verilog Hardware Description language [28, 29, 1] was designed and implemented by Phil Moorby at Gateway Design Automation in 1983–84 (see Moorby's history of the language [10]). The Verilog product was very successful, buoyed largely by the speed of its "XL" gate-level simulation algorithm. Cadence bought Gateway in 1989 and largely because of pressure from the competing, open VHDL language, made the language public in 1990. Open Verilog International (OVI) was formed shortly thereafter to maintain and promote the standard, IEEE adopted it in 1995, and ANSI in 1996.

The first Verilog simulator was event-driven and very efficient for gate-level circuits, the fashion of the time, but the opening of the Verilog language in the early 1990s paved the way for other companies to develop more efficient compiled simulators, which traded up-front compilation time for simulation speed.

Like tree rings, the syntax and semantics of the Verilog language embodies a history of simulation technologies and design methodologies. At its conception, gate- and switch-level simulation were in fashion, and Verilog contains extensive support for these modeling styles that is now little-used. (Moorby had worked with others on this problem before designing Verilog [21].)

Like many HDLs, Verilog supports hierarchy, but was originally designed assuming modules would have at most tens of connections. Hundreds or thousands of connections are now common, and Verilog-2001 [29] added a more succinct connection syntax to address this problem.

Procedural or behavioral modeling, once intended mainly for specifying testbenches, was pressed into service first for register-transfer-level specifications, and later for so-called behavioral specifications. Again, Verilog-2001 added some facilities to enable this (e.g., `always @*` to model combinational logic procedurally) and SystemVerilog has added additional support (e.g., `always_comb`, `always_ff`).

The syntax and semantics of Verilog are a compromise between modeling clarity and simulation efficiency. A "reg" in Verilog, the variable storage class for behavioral modeling, is exactly a shared variable. This means it simulates very efficiently (e.g., writing to a reg is just an assignment to memory), but also means that it can be misused (e.g., when written to by two concurrently-running processes) and misinterpreted (e.g., its name suggests a memory element such as a flip-flop, but it often represents purely combinational logic).

Thomas and Moorby [47] has long been the standard text on the language (Moorby was the main designer), and the language reference manual [28], since it was adopted from the original Verilog simulator user manual, is surprisingly readable. Other references include Palnitkar [40] for an overall description of the language, and Mittra [37] and Sutherland [46] for the programming language interface (PLI). Smith [45] compares Verilog and VHDL. French et al. [22] present a clever way of compiling Verilog simulations and also discuss more traditional ways.

3.1.1 Coding in Verilog

A Verilog description is a list of modules. Each module has a name; an interface consisting of a list of named ports, each with a type, such as a 32-bit vector, and a direction; a list of local nets and regs; and a body that can contain instances of primitive gates such as ANDs and ORs, instances of other modules (allowing hierarchical structural modeling), continuous assignment statements, which can be used to model combinational datapaths, and concurrent processes written in an imperative style.

Figure 1 shows the various modeling styles supported in Verilog. The two-input multiplexer circuit in Figure 1(a) can be represented in Verilog using primitive gates (Figure 1(b)), a continuous assignment (Figure 1(c)), a user-defined primitive (a truth table, Figure 1(d)), and a concurrent process (Figure 1(e)). All of these models exhibit roughly the same behavior (minor differences occur when some inputs are undefined) and can be mixed freely within a design.

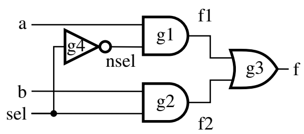
One of Verilog's strengths is its ability to also represent testbenches within the model being tested. Figure 1(f) illustrates a testbench for this simple mux, which applies a sequence of inputs over time and prints a report of the observed behavior.

Communication within and among Verilog processes takes place through two distinct types of variables: *net*s and *reg*s. Nets model wires and must be driven either by gates or by continuous assignments. Regs are exactly shared memory locations and can be used to model memory elements. Regs can be assigned only by imperative assignment statements that appear in *initial* and *always* blocks. Both nets and regs can be single bits or bit vectors, and regs can also be arrays of bit vectors to model memories. Verilog also has limited support for integers and floating-point numbers. Figure 2 shows a variety of declarations.

The distinction between regs and nets in Verilog is pragmatic: nets have slightly more complicated semantics (e.g., they can be assigned a capacitance to model charge storage and they can be connected to multiple tri-state drivers to model busses), but regs behave exactly like memory locations and are therefore easier to simulate quickly. Unfortunately, the semantics of regs make it easy to inadvertently introduce nondeterminism in the language (e.g., when two processes simultaneously attempt to write to the same reg, the result is undefined). This will be discussed in more detail in the next section.

Figure 3 illustrates the syntax for defining and instantiating models. Each module has a name and a list of named ports, each of which has a direction and a width. Instantiating such a module consists of giving the instance a name and listing the signals or expressions to which is connected. Connections can be made positionally or by port name, the latter being preferred for modules with many (perhaps ten or more) connections.

Continuous assignments are a simple way to model both Boolean and arithmetic datapaths. A continuous assignment uses Verilog's comprehensive expression syntax to define a function to be computed and its semantics are such that the value of the expression on the right of a continuous expression is always copied to the net on the left (regs are not allowed on the left of a continuous assignment). Practically, Verilog sim-



(a)

```
module mux(f,a,b,sel);
output f;
input a, b, sel;

and g1(f1, a, nsel),
    g2(f2, b, sel);
or  g3(f, f1, f2);
not g4(nsel, sel);

endmodule
```

(b)

```
module mux(f,a,b,sel);
output f;
input a, b, sel;

assign f = sel ? a : b;

endmodule
```

(c)

```
primitive
   mux(f,a,b,sel);
output f;
input a, b, sel;

table
  1?0 : 1;
  0?0 : 0;
  ?11 : 1;
  ?01 : 0;
  11? : 1;
  00? : 0;
endtable
endprimitive
```

(d)

```
module
   mux(f,a,b,sel);
output f;
input a, b, sel;
reg f;

always
  @(a or b or sel)
   if (sel) f = a;
   else f = b;

endmodule
```

(e)

```
module testbench;
  reg a, b, sel;
  wire f;

  mux dut(f, a, b, sel);

  initial begin
    $display("a,b,sel->f");
    $monitor($time,,
             "%b%b%b -> ",
             a, b, sel, f);
    a = 0; b = 0 ; sel = 0;
    #10 a = 1;
    #10 sel = 1;
    #10 b = 1;
    #10 sel = 0;
  end
endmodule
```

(f)

Figure 1: Verilog examples. (a) A multiplexer circuit. (b) The multiplexer as a Verilog structural model. (c) The multiplexer using continuous assignment. (d) A user-defined primitive for the multiplexer. (e) The multiplexer in imperative code. (f) A testbench for the multiplexer.

```
wire a;                   // Simple wire
tri [15:0] dbus;          // 16-bit tristate bus
tri #(5,4,8) b;           // Wire with delay
reg [-1:4] vec;           // Six-bit register
trireg (small) q;         // Wire stores charge
integer imem[0:1023];     // Array of 1024 integers
reg [31:0] dcache[0:63];  // A 32-bit memory
```

Figure 2: Various Verilog *net* and *reg* definitions.

```
module mymod(out1, out2, in1, in2);
output out1;       // Outputs first by convention
output [3:0] out2; // four-bit vector
input in1;
input [2:0] in2;

// Module body: instances,
// continuous assignments,
// initial and always blocks

endmodule

module usemymod;
reg a;
reg [2:0] b;
wire c, e, g;
wire [3:0] d, f, h;

      // simple instance
mymod m1(c, d, a, b);
      // instance with part-select input
mymod m2(e, f, c, d[2:0]),
      // connect-by-name
      m3(.in1(e), .in2(f[2:0]),
         .out1(g), .out2(h));

endmodule
```

Figure 3: Verilog structure: An example of a module definition and another module containing three instances of it.

ulators implement this by recomputing the expression on the right whenever any variable it references changes. Figure 4 illustrates some continuous assignments.

Behavioral modeling in Verilog uses imperative code enclosed in *initial* and *always* blocks that write to reg variables to maintain state. Each block effectively introduces a concurrent process that is awakened by an event and runs until it hits a delay or a wait statement. The example in Figure 5 illustrates basic behavioral usage.

Figure 6 shows a more complicated behavioral model, in this case a simple state machine. This example is written in a common style where the combinational and sequential parts of a state machine are written as two separate processes. The first process is purely combinational. The `@(a or b or state)` directive triggers its execution when signals a, b, or state change. The code consists of a multiway choice—a case statement—and performs procedural assignments to the o and nextState registers. This illustrates one of the odder aspects of Verilog modeling: both are declared reg, yet neither corresponds to the output of a latch or flip-flop. Instead, this reflects the Verilog requirement that procedural assignment can only be performed on regs.

The second process models a pair of flip-flops that holds the state between cycles. The `@(posedge clk or reset)` directive makes the process sensitive to the rising edge of the clock or a change in the reset signal. At the positive edge of the clock, the process captures the value of the `nextState` variable and copies it to `state`.

The example in Figure 6 illustrates the two types of behavioral assignments. The assignments used in the first process are so-called blocking assignments, written =, and take ef-

```
module add8(sum, a, b, carryin);
output [8:0] sum;
input [7:0] a, b;
input carryin;

// unsigned arithmetic
assign sum = a + b + carryin;
endmodule

module datapath(addr_2_0, icu_hit, psr_bm8, hit);
output [2:0]  addr_2_0
output        icu_hit
input         psr_bm8;
input         hit;
wire   [31:0] addr_qw_align;
wire   [3:0]  addr_qw_align_int;
wire   [31:0] addr_d1;
wire          powerdown;
wire          pwdn_d1;

// part select, vector concatenation is {}
assign addr_qw_align =
   { addr_d1[31:4], addr_qw_align_int[3:0] };

// if-then-else operator
assign addr_offset =
    psr_bm8 ? addr_2_0[1:0] : 2'b00;

// Boolean operators
assign icu_hit = hit & !powerdown & !pwdn_d1;

// ...

endmodule
```

Figure 4: Verilog modules illustrating continuous assignment. The first is a simple eight-bit full adder producing a nine-bit result. The second is an excerpt from a processor datapath.

fect immediately. Non-blocking assignments are written <= and have somewhat subtle semantics. Instead of taking effect immediately, the right-hand-sides of non-blocking assignments are evaluated when they are executed, but the assignment itself does not take place until the end of the current time instant. Such behavior effectively isolates the effect of non-blocking assignments to the next clock cycle, much like the output of a flip-flop is only visible after a clock edge. In general, non-blocking assignments are preferred when writing to state-holding elements for exactly this reason. See Figure 7 and the next section for a more extensive discussion of blocking vs. non-blocking assignments.

3.1.2 Verilog Shortcomings

Compared to VHDL, Verilog does a poor job at protecting users from themselves. Verilog's `reg`s are exactly shared variables and the language permits all the standard pitfalls associated with them, such as races and nondeterministic behavior. Most users avoid such behavior by following certain rules (e.g., by restricting assignments to a shared variable to a single concurrent process), but the Verilog allows more dangerous usage. Tellingly, a number of EDA companies exist solely to provide lint-like tools for Verilog that report such poor coding practices. Gordon [23] provides a more detailed discussion of the semantic challenges of Verilog.

```
module behavioral;
reg [1:0] a, b;

initial begin
  a = 'b1;
  b = 'b0;
end

always begin
  #50 a = ~a;  // Toggle a every 50 time units
end

always begin
  #100 b = ~b;  // Toggle b every 100 time units
end

endmodule
```

Figure 5: A simple Verilog behavioral model. The code in the *initial* block runs once at the beginning of simulation to initialize the two registers. The code in the two *always* blocks runs periodically: once every 50 and 100 time units respectively.

Non-blocking assignments are one way to ameliorate most problems with nondeterminism caused by shared variables, but they, too, can lead to bizarre behavior. To illustrate the use of shared variables, consider a three-stage shift register. The implementation in Figure 7(a) appears to be correct, but it fact may not behave as expected because the language says the simulator is free to execute the three *always* blocks in any order when they are triggered. If the processes execute top-to-bottom, the module becomes a one-stage shift register, but if they execute bottom-to-top, the behavior is as intended.

Figure 7(b) shows a correct implementation of the shift register that uses non-blocking assignments to avoid this problem. The semantics of these assignments are such that the value on the right hand side of the assignment is captured when the statement runs, but the actual assignment of values is only done at the "end" of each instant in time, i.e., after all three blocks have finished executing. As a result, the order in which the three assignments are executed does not matter and therefore the code always behaves like a three-stage shift register.

However, the use of non-blocking assignments can also be deceptive. In most languages, the effect of an assignment can be felt by the instruction immediately following it, but the delayed-assignment semantics of a non-blocking assignment violates this rule. Consider the erroneous decimal counter in Figure 8(a). Without knowing the subtle semantics of Verilog non-blocking assignments, the counter would appear to count from 0 to 9, but in fact it counts to 10 before being reset because test of `o` by the *if* statement gets the value of `o` from the previous clock cycle, not the results of the `o <= o + 1` statement. A corrected version is shown in Figure 8(b), which uses a local variable `count` to maintain the count, blocking assignments to touch it, and finally a non-blocking assignment to `o` to sent the count outside the module.

Coupled with the rules for register inference, the circuit implied by the counter in Figure 8(b) is actually just fine. While both `o` and `count` are marked as `reg`, only the `count` variable will actually become a state-holding element.

```
module FSM(o, a, b, reset);
output o;
reg o;  // declared reg: o is assigned procedurally
input a, b, reset;
reg [1:0] state; // only "state" holds state
reg [1:0] nextstate;

// Combinational logic block: sensitive to changes
// on all inputs and outputs.
// o and nextState always assigned

always @(a or b or state)
 case (state)
    2'b00: begin
      o = a & b;
      nextState = a ? 2'b00 : 2'b01;
    end
    2'b01: begin
      o = 0; nextState = 2'b10;
    end
    default: begin
      o = 0; nextState = 2'b00;
    end
 endcase

// Sequential block: sensitive to clock edge

always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;

endmodule
```

Figure 6: A Verilog behavioral model for a state machine illustrating the common practice of separating combinational and sequential blocks.

```
module bad_sr(o, i, clk);
output o;
input i, clk;
reg a, b, o;

always @(posedge clk) a = i;
always @(posedge clk) b = a;
always @(posedge clk) o = b;
endmodule
```

(a)

```
module good_sr(o, i, clk);
output o;
input i, clk;
reg a, b, o;

always @(posedge clk) a <= i;
always @(posedge clk) b <= a;
always @(posedge clk) o <= b;

endmodule
```

(b)

Figure 7: Verilog examples illustrating the difference between blocking and non-blocking assignments. (a) An erroneous implementation of a three-stage shift register that may or may not work depending on the order in which the simulator chooses to execute the three *always* blocks. (b) A correct implementation using non-blocking assignments, which make the variables take on their new values after all three blocks are done for the instant.

3.2 VHDL

Although VHDL and Verilog have grown to be nearly interchangeable, they could not have had more different histories. Unlike Verilog, VHDL was deliberately designed to be a standard hardware description language after a lengthy, deliberate process. As Dewey explains [17], VHDL was created at the behest of the U. S. Department of Defense in response to the desire to incorporate integrated circuits (specifically very high speed integrated circuits, hence the name of the program from which VHDL evolved, VHSIC) in military hardware. Starting with a summer study at Woods Hole, Massachusetts in 1981, requirements for and scope of the language were first established, then after a bidding process, a contract to develop the language was awarded in 1983 to three companies: Intermetrics, which was the prime contractor for Ada, the software programming language developed for the U. S. military in the early 1980s; Texas Instruments; and IBM. Dewey and de Geus describe this history in more detail [18].

The VHDL language was created in 1983 and 1984, essentially concurrent with Verilog, and first released publicly in 1985. Interest in an IEEE standard hardware description language was high at the time, and VHDL was eventually adapted and adopted as IEEE standard 1076 in 1987 [31] and revised in 1993 [27]. Verilog, meanwhile, remained proprietary until 1990. The standardization and growing popularity of VHDL was certainly instrumental in Cadence's decision to make Verilog public.

The original objectives of the VHDL language [17] were to provide a means of documenting hardware (i.e., as an alternative to imprecise English descriptions) and of verifying it through simulation. As such, a VHDL simulator was developed along with the language.

The VHDL language is vast, complicated, and has a verbose syntax obviously derived from Ada. While its popularity as a means of formal documentation is questionable, it has succeeded as a modeling language for hardware simulation, and, like Verilog, more recently as a specification language for register-transfer-level logic synthesis.

Although there are many features absent in Verilog that are unique to VHDL (and vice versa), in practice most designers use a nearly identical subset because this is what the synthesis tools accept. Thus although the syntax of the two languages differs greatly, and the semantics appear different, their usage has converged to a common core.

Unlike Verilog, VHDL has spawned a plethora of books discussing its proper usage. Basic texts include Lipsett, Schaefer, and Ussery [35] (one of the earliest), Dewey [19], Bhasker [8], Perry [43], and Ashenden [3, 4]. More advanced is Cohen [15], which suggests preferred idioms in VHDL, and Harr and Stancluescu [26], which discusses using VHDL for a variety of modeling tasks, not just RTL.

```
module bad_counter(o, clk);
output o;
input clk;
reg [3:0] o;

always @(posedge clk) begin
  o <= o + 1;
  if (o == 10)
    o <= 0;
end

endmodule
```

(a)

```
module good_counter(o, clk);
output o;
input clk;
reg [3:0] o;
reg [3:0] count;

always @(posedge clk) begin
  count = count + 1;
  if (count == 10)
    count = 0;
  o <= count;
end

endmodule
```

(b)

Figure 8: Verilog examples illustrating a pitfall of non-blocking assignments. (a) An erroneous implementation of a counter, which counts to 10, not 9. (b) A correct implementation using a combination of blocking and non-blocking assignments.

3.2.1 *Coding in VHDL*

Like Verilog, VHDL describes designs as a collection of hierarchical modules. But unlike Verilog, VHDL splits them into interfaces—called entities—and their implementations—architectures. In addition to named input and output ports, entities also define compile-time parameters (generics), types, constants, attributes, use directives, and others.

Figure 9 shows code for the same two-input multiplexer roughly equivalent to Verilog examples in Figure 1. Figure 9(a) is the entity declaration for the multiplexer, which defines its input and output ports. Figure 9(b) is a purely structural description of the multiplexer: it defines internal signals, the interface to the Inverter, AndGate, and OrGate components, and instantiates four of these gates. The name of the architecture, "`structural`" is arbitrary; it is used to distinguish among different architectures. The Verilog equivalent in Figure 1(b) did not define the internal signals, relying instead on Verilog's rule of automatically considering undeclared signals to be single-bit wires. Furthermore, the Verilog example used the built-in gate-level primitives; VHDL itself does not know about logic gates, but can be taught about them.

Figure 9(c) illustrates a dataflow model for the multiplexer with each logic gate made explicit. VHDL does have built-in logical operators. Figure 9(d) shows an even more succinct implementation, which uses the multi-way *when* conditional operator.

```
entity mux2 is
  port (a, b, c : in Bit; d : out Bit);
end;
```

(a)

```
architecture structural of mux2 is

  signal cbar, ai, bi : Bit;

  component Inverter
    port (a:in Bit; y: out Bit);
  end component;
  component AndGate
    port (a1, a2:in Bit; y: out Bit);
  end component;
  component OrGate
    port (a1, a2:in Bit; y: out Bit);
  end component;

begin
    -- connect by name
  I1: Inverter port map(a => c, y => cbar);
    -- connect by position
  A1: AndGate  port map(a, c, ai);
  A2: AndGate  port map(a1 => b, a2 => cbar,
                        y => bi);
  O1: OrGate   port map(a1 => ai, a2 => bi,
                        y => d);
end;
```

(b)

```
architecture dataflow1 of mux2 is
  signal cbar, ai, bi : Bit;
begin
  cbar <= not c;
  ai   <= a and c;
  bi   <= b and cbar;
  d    <= ai or bi;
end;
```

(c)

```
architecture dataflow2 of mux2 is
begin
  d <= a when c = '1' else b;
end;
```

(d)

```
architecture behavioral of mux2 is
begin
  process(a, b, c)  -- sensitivity list
  begin
    if c = '1' then
      d <= a;
    else
      d <= b;
    end if;
  end process;
end;
```

(e)

Figure 9: VHDL code for a two-input multiplexer. (a) The entity definition for the multiplexer. (b) A structural implementation instantiating primitive gates. (c) A dataflow implementation with an expression for each gate. (d) A direct dataflow implementation. (e) A behavioral implementation.

Finally, Figure 9(e) shows a behavioral implementation of the mux. It defines a concurrently-running process sensitive to the three mux inputs (a, b, and c) and uses an if-then-else statement (VHDL provides most of the usual control-flow statements) to select between copying the a and the b signal to the output d.

One of the design philosophies behind the VHDL language was to maximize its flexibility by making most things user-definable. As a result, unlike Verilog, it has only the most rudimentary built-in types (e.g., Boolean variables, but nothing to model four-valued logic), but has a much more powerful type system that allows such types to be defined. The `Bit` used in the examples in Figure 9 is actually a predefined part of the standard environment, i.e.,

```
type BIT is ('0', '1');
```

which is a character enumeration type whose two values are the characters 0 and 1. (VHDL is case-insensitive; `Bit` and `BIT` are equivalent.)

There are advantages and disadvantages to this approach. While it allows more things to be added to the language later, it also makes for a few infelicities. For example, the predicate in the *if* statement in Figure 9(e) must be written "`c = '1'`" instead of just "`c`" because the argument must be of type Boolean, not Bit. While not a serious issue, it is yet another thing that contributes to VHDL's verbosity.

Figure 10 is a more elaborate example showing an implementation of the classic traffic light controller from Mead and Conway [36]. This is written in a synthesizeable dialect, using the common practice of separating the output and next-state logic from the state-holding element. Specifically, the first process is sensitive only to the clock signal. The *if* statement in the first process checks for an event on the clock (VHDL signals have a variety of attributes; *event* is true whenever the value has changed) and the clock being high, i.e., the rising edge of the clock. The second process is sensitive only to the inputs and present state of the machine, not the clock, and is meant to model combinational logic. It illustrates the multi-way conditional *case* statement, constants, and bit vectors. It employs types (i.e., `std_ulogic` and `std_ulogic_vector`) and operators from the ieee.std_logic_1164 library, an IEEE standard library [32] for modeling logic that can represent unknown values ("X") as well as 0s and 1s.

3.2.2 *VHDL Shortcomings*

One shortcoming of VHDL is its obvious verbosity: the use of begin/end pairs instead of braces, the need to separate entities and their architectures, the need to spell out things like ports, its lengthy names for standard logic types (e.g., `std_ulogic_vector`), and its requirement of enclosing Boolean values and vectors in quotes. Many of these are artifacts of its roots in the Ada language, another fairly verbose language commissioned by the U. S. Department of Defense, but others are due to questionable design decisions. Consider the separation of entity/architecture pairs. While separating these concepts is a boon to abstraction and simplifies the construction of simulations of the same system in different configurations (e.g., to run a simulation using a gate-level archi-

```
library ieee;
use ieee.std_logic_1164.all;
entity tlc is
  port (
    clk, reset       : in  std_ulogic;
    cars, short long : in  std_ulogic;
    highway_yellow   : out std_ulogic;
    highway_red      : out std_ulogic;
    farm_yellow      : out std_ulogic;
    farm_red         : out std_ulogic;
    start_timer      : out std_ulogic);
end tlc;

architecture imp of tlc is
signal current_state, next_state :
    std_ulogic_vector(1 downto 0);
constant HG : std_ulogic_vector := "00";
constant HY : std_ulogic_vector := "01";
constant FY : std_ulogic_vector := "10";
constant FG : std_ulogic_vector := "11";
begin

P1: process (clk)    -- Sequential process
begin
  if (clk'event and clk = '1') then
    current_state <= next_state;
  end if;
end process P1;

-- Combinational process: sensitive to input changes
P2: process (current_state, reset, cars, short, long)
begin
  if (reset = '1') then
    next_state <= HG;
    start_timer <= '1';
  else
    case current_state is
      when HG =>
        highway_yellow <= '0'; highway_red <= '0';
        farm_yellow    <= '0'; farm_red    <= '1';
        if (cars = '1' and long = '1') then
          next_state <= HY; start_timer <= '1';
        else
          next_state <= HG; start_timer <= '0';
        end if;
      when HY =>
        highway_yellow <= '1'; highway_red <= '0';
        farm_yellow    <= '0'; farm_red    <= '1';
        if (short = '1') then
          next_state <= FG; start_timer <= '1';
        else
          next_state <= HY; start_timer <= '0';
        end if;
      when FG =>
        highway_yellow <= '0'; highway_red <= '1';
        farm_yellow    <= '0'; farm_red    <= '0';
        if (cars = '0' or long = '1') then
          next_state <= FY; start_timer <= '1';
        else
          next_state <= FG; start_timer <= '0';
        end if;
      when FY =>
        highway_yellow <= '0'; highway_red <= '1';
        farm_yellow    <= '1'; farm_red    <= '0';
        if (short = '1') then
          next_state <= HG; start_timer <= '1';
        else
          next_state <= FY; start_timer <= '0';
        end if;
      when others =>
        next_state     <= "XX"; start_timer <= 'X';
        highway_yellow <= 'X'; highway_red    <= 'X';
        farm_yellow    <= 'X'; farm_red       <= 'X';
    end case;
  end if;
end process P2;
end imp;
```

Figure 10: The traffic-light controller from Mead and Conway [36] implemented in VHDL, illustrating the common practice of separating combinational and state-holding processes.

tecture in place of a behavioral one for more precise timing estimation), in practice most designers only ever write a single architecture for a given entity and such pairs are usually written together.

The flexibility of VHDL also has advantages and disadvantages. Its type system is much more flexible than Verilog's, providing things such as aggregate types and overloaded functions and operators, but this flexibility also comes with a need for standardization and also tends to increase the verbosity of the language. Some of the need for standardization was recognized early, resulting in libraries such as the widely-supported IEEE 1164 library for multi-valued logic modeling. However, a standard for signed and unsigned arithmetic on logic vectors was slower in coming (it was eventually standardized in 1997 [33]), prompting both Synopsys and Mentor to each introduce similar but incompatible and incomplete versions of a similar library.

Fundamentally, many of the problems stem from a desire to make the language too general. Aspects of the type system suffer from this as well. While the ability to define new enumerated types for multi-valued logic modeling is powerful, it seems a little odd to require virtually every VHDL program (since its main use has long been specification for RTL synthesis) to include one or more standard libraries. This also leads to the need to be constantly comparing signals to the literal `'1'` instead of just using a signal's value directly, and requires a user to carefully watch the types of subexpressions.

3.3 SystemC

SystemC is a relative latecomer to the HDL wars. Developed at Synopsys in the late 1990s, primarily by Stan Liao, SystemC was originally called Scenic [34] and was intended to replace Verilog and VHDL as the main system description language for synthesis (see Arnout [2] for some of the arguments for SystemC). SystemC is not so much a language as a C++ library along with a set of coding rules, but this is exactly its strength. It evolved from the common practice of first writing a high-level simulation model in C or C++, refining it, and finally recoding it in RTL Verilog or VHDL. SystemC was intended to smooth the refinement process by removing the need for a separate hardware description language.

SystemC can be thought of as a dialect of C++ for modeling digital hardware. Like Verilog and VHDL, it supports hierarchical models whose blocks consist of input/output ports, internal signals, concurrently running imperative processes, and instances of other blocks. The SystemC libraries make two main contributions: an inexpensive mechanism for running many processes concurrently (based on a lightweight thread package; see Liao, Tjiang, and Gupta [34]), and an extensive set of types for modeling hardware systems, including bit vectors and fixed-point numbers. A SystemC model consists of a series of class definitions, each of which define a block. Methods defined for such a class become concurrently-running processes, and the constructor for each class starts these processes running by passing them to the simulation kernel. Simulating a SystemC model starts by calling the constructors for all blocks in the design, then invoking the scheduler, which is responsible for executing each of the concurrent processes as needed.

The computational model behind earlier versions of SystemC was cycle-based instead of the event-driven model of Verilog and VHDL. This meant that the simulation was driven by a collection of potentially asynchronous, but periodic clocks. Later versions (SystemC 2.0 and higher) adopted an event-driven model much like VHDL's.

SystemC books have only appeared recently. Grötker et al. [24] provide a nice introduction to SystemC 2.0. Bhasker [9] is also an introduction. The volume edited by Muller et al. [38] surveys more advanced SystemC modeling techniques.

3.3.1 Coding in SystemC

Figure 11 shows a small SystemC model for a 0–99 counter driving a pair of seven segment displays. It defines two modules (the `decoder` and `counter` *structs*) and an `sc_main` function that defines some internal signals, instantiates two decoders and a counter, and runs the simulation while printing out what it does.

The two modules in Figure 11 illustrate two of the three types of processes possible in SystemC. The `decoder` module is the simpler one: it models a purely combinational process by defining a method (called, arbitrarily, "`compute`") that will be invoked by the scheduler every time the `number` input changes, as indicated by the `sensitive << number;` — statement beneath the definition of `compute` as an `SC_METHOD`.

The second module, `counter` is an `SC_CTHREAD` process: a method (here called "`tick`") that is invoked in response to a clock edge (here, the positive edge of the `clk` input, as defined by the `SC_CTHREAD(tick, clk.pos());` statement) and can suspend itself with the `wait()` statement. Specifically, the scheduler resumes the method when the clock edge occurs, and the method runs until it encounters a `wait()` statement, at which point its state is saved and control passes back to the scheduler.

This example illustrates only a very small fraction of the SystemC type libraries. It uses unsigned integers (`sc_uint`), bit vectors (`sc_bv`), and a clock (`sc_clock`). The non-clock types are wrapped in `sc_signals`, which behave like VHDL signals. In particular, when an `SC_CTHREAD` method assigns a value to a signal, the effect of this assignment is felt only after all the processes triggered by the same clock edge have been run. Thus, such assignments behave like blocking assignments in Verilog to ensure that the nondeterministic order in which such processes are invoked (the scheduler is allowed to invoke them in any order) does not affect the ultimate outcome of simulating the system.

3.3.2 SystemC Shortcomings

Like many languages, the most common use of SystemC has diverged from its designers' original intentions—an input for hardware synthesis in the case of SystemC. A number of synthesis tools for the language do exist, but SystemC is now used primarily (and quite successfully) for system modeling. This does mean, however, that it does not solve the "separate language for synthesis problem."

```
#include "systemc.h"
#include <stdio.h>

struct decoder : sc_module {
  sc_in<sc_uint<4> > number;
  sc_out<sc_bv<7> > segments;

  void compute() {
    static sc_bv<7> codes[10] = {
      0x7e, 0x30, 0x6d, 0x79, 0x33,
      0x5b, 0x5f, 0x70, 0x7f, 0x7b };
    if (number.read() < 10)
      segments = codes[number.read()];
  }

  SC_CTOR(decoder) {
    SC_METHOD(compute);
    sensitive << number;
  }
};

struct counter : sc_module {
  sc_out<sc_uint<4> > tens;
  sc_out<sc_uint<4> > ones;
  sc_in_clk clk;

  void tick() {
    int one = 0, ten = 0;
    for (;;) {
      if (++one == 10) {
        one = 0;
        if (++ten == 10) ten = 0;
      }
      ones = one;
      tens = ten;
      wait();
    }
  }

  SC_CTOR(counter) {
    SC_CTHREAD(tick, clk.pos());
  }
};

int sc_main(int argc, char *argv[])
{
  sc_signal<sc_uint<4> > ones, tens;
  sc_signal<sc_bv<7> >
     ones_segments, tens_segments;
  sc_clock clk;

  decoder decoder1("decoder1");
  decoder1(ones, ones_segments);
  decoder decoder2("decoder2");
  decoder2(tens, tens_segments);

  counter counter1("counter1");
  counter1(tens, ones, clk);

  for (int i = 0 ; i < 12 ; i++) {
    sc_start(clk, 1);
    printf("%d %d %x %x\n",
           (int)tens.read(), (int)ones.read(),
           (int)(sc_uint<7>)tens_segments.read(),
           (int)(sc_uint<7>)ones_segments.read());
  }

}
```

Figure 11: A SystemC model for a two-digit decimal counter driving two seven-segment displays.

A big disadvantage of SystemC is that C++ was never intended for modeling digital hardware and as a result is even more lax about enforcing rules than Verilog. The syntax, similarly, is somewhat awkward and relies on some very tricky macro preprocessor definitions. On detailed models, the simulation speed of a good compiled-code Verilog or VHDL simulator may be better, although SystemC is much faster for higher-level models. For such systems, which consist of complex processes, SystemC should be superior since the simulation becomes nearly a normal C++ program. However, the context-switching cost in SystemC is higher than that of a good Verilog or VHDL simulator when running a more detailed model, so a system with many small processes would not simulate as quickly.

Another issue is the ease with which a SystemC model can inadvertently be made nondeterministic. Although carefully following a discipline of only communicating among processes through signals will ensure the simulation is nondeterministic, any slight deviation from this will cause problems. For example, library functions that use a hidden global variable may cause nondeterminism if called from different processes. Accidentally holding state in an `SC_METHOD` process (easily done if class variables are assigned) can also cause problems since such method are invoked in an undefined order.

Many argue that nondeterministic behavior in a language can be desirable for modeling nondeterministic systems, which certainly exist and need to be modeled. However, the sort of nondeterminism in a language such as SystemC or Verilog creeps in unexpectedly and is difficult to use as a modeling tool. For the simulation of a nondeterministic model to be interesting, there needs to be some way of seeing the different possible behaviors, yet a nondeterministic artifact such as an `SC_METHOD` process that holds state provides no mechanism for ensuring that it is not, in fact, predictable. As a result, a designer has a hard time answering whether a model of a nondeterministic system can exhibit undesired behavior, even through a careful selection of test cases.

4 Verification Languages

Thanks to dramatic improvements in integrated circuit fabrication technology, it is now possible to build very large, complex integrated circuits. Hardware description languages, logic synthesis, and automated place-and-route technology have similarly made it possible to design such complicated systems. Unfortunately, technology to validate these designs, i.e., to identify design errors, has had a hard time keeping pace with these trends.

The time required to validate a design now greatly outstrips the time required to design or fabricate it. Although many novel techniques have been proposed to address the validation problem, simulation remains the preferred method. So-called formal verification techniques, which amount to efficient exhaustive simulation, have been gaining ground, but suffer from capacity problems.

Simulation applies a stimulus to a model of a design to predict the behavior of the fabricated system. Naturally, there is a trade-off between highly detailed models that can predict many attributes, say, logical values, timing, and power consumption, and simplified models that can only predict logical behavior but execute much faster.

Because the size of the typical design has grown exponentially over time, functional simulation, which only predicts the logical behavior of a synchronous circuit at clock-cycle boundaries, has become the preferred form of simulation because of its superior speed. Furthermore, designers have shied away from more timing-sensitive circuitry such as gated clocks and transparent latches because they require more detailed simulation models and are therefore more costly to simulate.

Simulation-based validation raises three important questions: what the stimulus should be, whether it exposes any design errors, and whether the stimulus is long and varied enough to have exposed "all" design errors. Historically, these three questions have been answered manually, i.e., by having a test engineer write test cases, check the results of simulation, and make some informed guesses about how comprehensive the test suite actually is.

A manual approach has many shortcomings. Writing test-cases is tedious and the number necessary for "complete" verification grows faster than the size of the system description. Manually checking the output of simulation is similarly tedious and subtle errors can be easily overlooked. Finally, it is difficult to judge quantitatively how much of a design has really been tested.

More automated methodologies, and ultimately languages, have evolved to address some of these challenges, although the verification problem remains one of the most difficult. Biased random test case generation has become standard practice, although it has only supplemented manual test case generation, not completely supplanted it. Designer-inserted assertions, long standard practice for software development, have also become standard for hardware, although the sort of assertions needed in hardware are more complicated than the typical "the argument must be non-zero" sort of checking that works well in software. Finally, automated "coverage" checking, which attempts to quantify how much of a design's behavior has been exercised in simulation, has also become standard.

All of these techniques are improvements, not a panacea. While biased random test case generation can quickly generate many interesting tests, it provides no guarantee of completeness; bugs may go unnoticed. Because they must often check temporal properties (e.g., "acknowledge arrives within three cycles of every request"), assertions in hardware systems are more difficult to write than those for software (which most often check data structure consistency) and again, there is no way to know when enough assertions have been added, and it is possible that the assertions themselves have flaws (e.g., they let bugs by). Finally, test cases that achieve 100% coverage can also let bugs by because the criteria for coverage is necessarily weak. Coverage typically checks what states particular variables have been in, but it cannot consider all combinations because they grow exponentially quickly with design size. As a result, certain important combinations may not be checked even though coverage checks report "complete coverage."

While the utility of biased random test generation and coverage metrics is mostly limited to simulation, assertion specification techniques are useful for, and have been heavily influenced by, formal verification. Pure formal techniques consider all possible behaviors by definition and therefore do not require explicit test cases (implicitly, they consider all possible test cases) and also do not need to consider coverage. But knowing what behavior is unwanted is crucial for formal techniques, whose purpose is to either expose unwanted behavior or formally prove it cannot occur.

Recently, a sort of renaissance has occurred in verification languages. Temporal logics, specifically Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), form the mathematical basis for most assertion checking, but their mathematical syntax is awkward for hardware designers. Instead, a number of more traditional computer languages, which combine a more human-readable syntax for the bare logic with a lot of "syntactic sugar" for more naturally expressing common properties, have been proposed for expressing properties in these logics. Two industrial efforts from Intel (ForSpec) and IBM (Sugar) have emerged as the most complete.

Meanwhile, some EDA companies have produced languages designed for writing testbenches and checking simulation coverage. Vera, originally designed by Systems Science and since acquired by Synopsys, and e, designed and sold by Verisity, have been the two most commercially successful. Bergeron [7] discusses how to use the two languages.

All four of these languages have recently undergone extensive cross-breeding. Vera has been made public, rechristened OpenVera, had Intel's ForSpec assertions grafted onto it, and added almost in its entirety to SystemVerilog. Sugar, meanwhile, has been adopted by the Accelera standards committee, rechristened the Property Specification Language (PSL), and also added in part to SystemVerilog. Verisity's e has changed the least, only having recently been made public.

There are obvious advantages in having a single industry-standard language for assertions, so it seems likely that most of these languages will eventually disappear, but as of this writing (2004), there is no obvious winner.

The sections that follow describe languages that are currently public. As mentioned above, most started as proprietary in-house or commercial.

4.1 OpenVera

OpenVera began life around 1995 as Vera, a proprietary language implemented by Systems Science, Inc. mainly for creating testbenches for Verilog simulations. As such, its syntax was heavily influenced by Verilog. VHDL support was added later. Synopsys bought the company in 1998, released the language to the public in 2001, and rechristened it OpenVera.

OpenVera is a concurrent, imperative language designed for writing testbenches. It executes in concert with a Verilog or VHDL simulator and can both provide stimulus to the simulator and observe the results. In addition to the usual high-level imperative language constructs, such as conditionals, loops, functions, strings, and associative arrays, it provides extensive facilities for generating biased random patterns (designed to be applied to the hardware design under test) and monitoring what values particular variables take on during the simulation (for checking coverage).

```
class Bus {
  rand bit[15:0] addr;
  rand bit[31:0] data;

  constraint world_align { addr[1:0] == '2b0; }
}

program demo {
  Bus bus = new();
  repeat (50) {
    integer result = bus.randomize();
    if (result == OK)
      printf("addr = %16h data = %32h\n",
             bus.addr, bus.data);
    else
      printf("randomization failed\n");
  }
}
```

Figure 12: A simple Vera program illustrating its ability to generate biased random variables and its mix of imperative and object-oriented styles. From the OpenVera 1.3 LRM.

In the process of making OpenVera public, Synopsys added the assertion specification capabilities of Intel's ForSpec language, making it easy to check whether certain behaviors ever appear during the simulation.

In a further nod to cross-breeding, much of Vera was incorporated into SystemVerilog 3.1 around 2003. In particular, its style of generating biased random variables and checking for behavior coverage have been adopted more-or-less verbatim.

The following is a quick overview of OpenVera 1.3, currently the latest version (September 2004). References for the language include the OpenVera language reference manual, available from the OpenVera website, and Haque et al. [25].

OpenVera has three main facilities that separate it from more traditional programming languages: biased random variable generation subject to constraints, monitoring facilities for reporting coverage of state variables, and the ability to specify and check temporal assertions.

Figure 12 shows a simple OpenVera program that demonstrates the language's ability to generate biased random variables. First, a Java/C++-like class is defined containing the sixteen-bit field `addr` and the thirty-two bit field `data`. These are marked `rand`, meaning their values will be set by a call to the `randomize()` method implicitly defined for every class.

Following the definition of the fields, a constraint (named arbitrarily `word_align`) is defined for objects of this class. Such constraints restrict the possible values the `randomize()` method may assign to various fields. This particular constraint simply restricts the two lowest bits of `addr` to be zero, i.e., aligned on a four-byte boundary.

The "demo" program creates a new `Bus` object then, for fifty times, invokes `randomize()` to generate a new set of "random" values (in fact, they are taken from a pseudorandom sequence guaranteed to be the same each time the program runs) for the two fields in the `bus` object that conform to the given constraint.

Overly constrained or inconsistent constraints may lead `randomize()` to fail. It signals this with a non-OK return value (OK and FAIL are reserved words in Vera), here assigned to the local integer variable `result`. For this simple example, randomization will always succeed. The constraint solver guarantees that it will only fail if there is no consistent solution to the supplied constraints.

```
bit clk;
bit [15:0] addr;
bit [7:0] data;

coverage_group MyChecker {
  sample_event = @(posedge clk);
  sample addr, data;
}

program demo_coverage {
  MyChecker checker = new();
  ...
}
```

Figure 13: A simple Vera program containing a single coverage group that monitors which values appear on the addr and data state variables. From the OpenVera 1.3 LRM.

This example, of course, only illustrates a fraction of OpenVera's facilities for biased random variable generation. It can also add constraints on-the fly, impose set membership constraints, follow user-defined distributions, impose conditional constraints, impose constraints between variables, selectively randomize and disable randomization of user-specified variables, and generate random sequences from a language specified by a grammar.

OpenVera supports so-called functional coverage checking, which can monitor state variables and state transitions (unlike software coverage, which usually monitors which statements and branches have been executed). It uses a bin model: each bin represents a particular state or transition and when a matching activity occurs, the counter for that bin is incremented. The number of bins that remain empty after simulation therefore give a rough idea of what behavior has yet to be exercised.

The `coverage_group` construct defines a type of monitor. Each specifies a set of variables to monitor and an event that triggers a check of the variables, typically the positive edge of a clock. Like a class, these constructs must be explicitly instantiated and there may be multiple copies of each, especially useful when a coverage group has parameters. Figure 13 illustrates a simple coverage group.

Like the earlier example, Figure 13 shows only the most basic coverage functionality. OpenVera can also monitor cross coverage, i.e., the combinations of values taken by two or more variables; selectively disable coverage checking, e.g., during system reset; allow the user to explicitly specify bins and the values mapped to them; and monitor transition coverage, i.e., combinations of values taken by the same variable in successive cycles;.

OpenVera's assertions, which were adapted from Intel's ForSpec language, provides a way to check temporal properties such as "an acknowledge signal must occur within three clock cycles after any request." Because of its source, the syntax for assertions is a little unusual.

```
/* Checks for the sequence p, 6, 9, 3 */
unit mychecker
  #(parameter integer p = 0)
  (logic en, logic clk, logic [7:0] result);

  clock posedge (clk) {
    event e0: (result == p);
    event e1: (result == 6);
    event e2: (result == 9);
    event e3: (result == 3);
    event myseq:
      if (en) then (e0 #1 e1 #1 e2 #1 e3);
  }

  assert myassert: check(myseq, "Missed a step.");
endunit

/* Watch for the sequence 4, 6, 9, 3 on outp */
bind instances cnt_top.dut:
  mychecker myinst #(4) (enable, clk, outp);
```

Figure 14: A sample of OpenVera's assertion language. It defines a checker that watches for the sequence p, 6, 9, 3 and creates an instance of it that checks for the sequence 4, 6, 9, 3 on the outp bus. From the OpenVera 1.3 LRM.

Figure 14 shows the assertion language. It defines a checker ("mychecker") that takes a single integer parameter *p*, an enable signal, a clock, and an eight-bit bus and looks for the sequence *p*, 4, 6, 9, 3 on the bus. The `clock` construct defines a collection of events (patterns) synchronized to the positive edge of the clock signal. Events `e0` through `e3` are simple properties: they look for patterns on the result signal. The `myseq` pattern is the interesting one: it looks for the appearance of the four events separated by a single clock cycle (a one-cycle delay is written `#1`). The `assert` directive means to check for the `myseq` event and report and error otherwise.

Finally, `bind` indicates where to instantiate the checker, gives it the name `myinst`, passes the parameter 4, and connects the checker to the enable, clock, and result signals.

Once more, the example in Figure 14 barely scratches the surface. Sequences can also contain consecutive and non-consecutive repetition, explicit delays, simultaneous and disjoint sequences, and sequence containment.

4.2 The e Language

The e language was developed by Verisity as part of its Specman product as a tool for efficiently writing testbenches. Like Vera, it is an imperative object-oriented language with concurrency, the ability to generate constrained random values, mechanisms for checking functional (variable value) coverage, and a way to check temporal properties (assertions). Books on e include Palnitkar [41] and Iman and Joshi [30].

The syntax of e is unusual. First, all code must be enclosed in `<'` and `'>'` symbols, otherwise it is considered a comment. Unlike C, e declarations are written "name : type." The syntax for fields in compound types (e.g., structs) includes particles such as `%` and `!`, which indicate when a field is to be driven on the device-under-test and not randomly computed respectively.

Figure 15 shows a fragment of an e program that defines an abstract test strategy for a very simple microprocessor, specif-

```
Instruction encoding for a very simple processor
<'
type opcode: [ADD, SUB, ADDI, JMP, CALL] (bits: 4);
type reg: [REG0, REG1, REG2, REG3] (bits: 2);

struct instr {
  %op  : opcode;     // Four-bit opcode
  %op1 : reg;        // Two-bit operand

  kind : [imm, reg]; // Second operand type
  when reg instr { %op2 : reg; }  // reg operand
  when imm instr { %op2 : byte; } // imm operand

  // Constrain certain instructions to reg or imm
  keep op in [ ADD, SUB ] => kind == reg;
  keep op in [ ADDI, JMP, CALL ] => kind == imm;

  // Constrain JMP, CALL second operand
  when imm instr {
    keep opcode in [JMP, CALL] => op2 < 16;
  }
};

extend sys {
  // Add a non-generated field called instrs
  !instrs : list of instr;
};
'>
```

Figure 15: e code defining instruction encoding for a simple 8-bit microprocessor. After an example in the Specman tutorial.

ically how to generate instructions for it. It illustrates the type system of the language as well as the utility of constraints. It defines two enumerated scalar types, `opcode` and `reg`, and defines the width of each. The `struct instr` defines a new compound type (`instr`) that represents a single instructions. First is the `op` field, which is one of the opcodes defined earlier. It, the `op1`, and the `op2` fields are marked with `%`, indicating that they should be considered by the `pack` built-in procedure, which marshals data to send to the simulation.

The `kind` field is also a enumerated scalar, but is used here as a type tag. It is not marked with `%`, which means that its value will not be included when the structure is packed and sent to the simulation. The two `when` directives define two subtypes, i.e., "`reg instr`" and "`imm instr`." Such subtypes are similar to derived classes in object-oriented programming languages. Here, the value of the `kind` field, which can be either `imm` or `reg`, determines the subtype.

The two `keep` directives impose constraints between the `kind` field and the opcode, ensuring, e.g., that `ADD` and `SUB` instructions are of the `reg` type. Although these constraints are simple, e is able to impose much more complicated constraints on the values of fields in a struct.

The final `when` directive further constrains the `JMP` and `CALL` instructions, i.e., by restricting what values the `op2` field may take for these instructions.

The `extend sys` directive adds a field named `instrs` to the `sys` built-in structure, which is the basic environment. The leading `!` makes the system create an empty list of instructions, which will be filled in later.

Figure 16 illustrates how the definition of Figure 15 can be used to generate tests that exercise the `ADD` and `ADDI` instruc-

```
<'
extend instr {
  keep opcode in [ADD, ADDI];
  keep op1 == REG0;
  when reg instr { keep op2 == REG1; }
  when imm instr { keep op2 == 0x3; }
};

extend sys {
   keep instrs.size() == 10;
};
'>
```

Figure 16: e code that uses the instruction encoding of Figure 15 to randomly generate ten instructions. After an example in the Specman tutorial.

tions. It first adds constraints to the instr class (the template for instructions defined in Figure 15) that restrict the opcodes to either ADD or ADDI, then imposes a constraint on the top level (`sys`) that makes it generate exactly ten instructions. Running the source code of Figure 15 and Figure 16 together makes the system generate a sequence of ten pseudorandom instructions.

4.3 PSL

The Property Specification Language, PSL, evolved from the proprietary Sugar language developed at IBM. Its focus is narrower than either OpenVera or e, since its goal is purely to specify temporal properties to be checked in hardware designs, but is more disciplined and has more formal semantics.

Beer et al. [5] provide a nice introduction to an earlier version of the language, which they explain evolved over many years. It has been used within IBM in the RuleBase formal verification system since 1994 and was also pressed into service as a checker generator for simulators in 1997. Accelera, an EDA standards group, adopted it as their formal property language in 2002. Cohen [16] provides a tutorial.

PSL is based on Computation Tree Logic (CTL) [14], a powerful but rather cryptic temporal logic for specifying properties of finite-state systems. It is able to specify both safety properties ("this bad thing will never happen") as well as liveness properties ("this good thing will eventually happen"). Liveness properties can only be checked formally because it makes a statement about all the possible behaviors of a system while safety properties can also be tested in simulation. Linear Temporal Logic (LTL), a subset of CTL, expresses only safety properties, can therefore be turned into checking automata meant to be run in concert with a simulation to look for unwanted behavior. PSL carefully defines which subset of its properties are purely LTL and are therefore candidates for use in simulation-based checking.

PSL is divided into four layers. The lowest, Boolean, consists of instantaneous Boolean expressions on signals in the design under test. The syntax of this layer follows that of the HDL to which PSL is being applied, and can be Verilog, SystemVerilog, VHDL, or GDL. For example, `a[0:3] & b[0:3]` and `a(0 to 3) and b(0 to 3)` represent the bit-wise *and* of the four most significant bits of vectors `a` and `b` in the Verilog and VHDL flavors respectively.

The second layer, temporal, is where PSL gets interesting. It allows a designer to state properties that hold across multiple clock cycles. The `always` operator, which states that a Boolean expression holds in every clock cycle, is one of the most basic. For example, `always !(ena & enb)` states that the signals `ena` and `enb` will never be true simultaneously in any clock cycle.

More interesting are operators that specify delays. The `next` operator is the simplest. The property `always (req -> next ack)` states that in every cycle that the `req` signal is true, the `ack` signal is true in the next cycle. The `->` symbol denotes implication, i.e., if the expression to the left left is true, that on the right must also be true. The `next` operator can also take an argument, e.g., `always req -> next[2] ack` means that `ack` must be true two cycles after each cycle in which `req` is true.

PSL provides an extended form of regular expressions convenient for specifying complex behaviors. Although it would be possible to write `always (req -> next (ack-> next !cancel))` to indicate that `ack` must be true after any cycle in which `req` is true, and `cancel` must be false in the cycle after that, it is easier to write `always {req ; ack ; !cancel}`. This illustrates a basic principle of PSL: most operators are actually just "syntactic sugar;" the set of fundamental operators is quite small.

PSL draws a clear distinction between "weak" operators, which can be checked in simulation (i.e., safety properties) and "strong" operators, which express liveness properties and can only be checked formally. Strong operators are marked with a trailing exclamation point (`!`), and some operators come in both strong and weak varieties.

The `eventually!` operator illustrates the meaning of strong operators. The property `always (req -> eventually! ack)` states that after `req` is asserted, `ack` will always be asserted eventually. This is not something that can be checked in simulation: if a particular simulation saw `req` but did not see `ack`, it would be incorrect to report that this property failed because running that particular simulation longer might have produced `ack`. The is the fundamental difference between safety and liveness properties: safety states something bad never happens; liveness states something good eventually happens.

Another subtlety is that it is possible to express properties in which times moves backward through a property. A simple example is `always ((a && next[3](b)) -> c`, which states that when `a` is true and `b` is true three clock cycles later, `c` is true in the first cycle, i.e., when `a` was true. While it is possible to check this in simulation (for each cycle in which `a` is true, remember whether `c` is true and look three clock cycles later for `b`), it is more difficult to build automata that check such properties in simulation.

The third layer of PSL, the verification layer, instructs a verification tool what tests to perform on a particular design. It amounts to a binding between properties defined with expressions from the Boolean and temporal layer, and modules in the design under test. The following simple example

```
vunit ex1a(top_block.i1.i2) {
  A1: assert never (ena && enb);
}
```

declares a "verification unit" called `ex1a`, binds it to the instance named `top\_block.i1.i2` in the design under test, and declares (the assertion named `A1`) that the signals `ena` and `enb` in that instance are never true simultaneously.

In addition to `assert`, verification units may also include `assume` directives, which allow the tool to assume a property; `assume_guarantee`, which both assumes and tests a property; `restrict`, which constrains the tool to only consider those behaviors that have a property; `cover`, which asks the tool to check whether a certain property was ever observed; and `fairness`, which instructs the tool to only consider paths in which the given property occurs infinitely often, e.g., only when the system does not wait indefinitely.

The fourth (modeling) layer of PSL allows Verilog, SystemVerilog, VHDL, or GDL code to be included inline in a PSL specification. The intention here is to include additional details about the system under test in the PSL source file.

4.4 SystemVerilog

Recently, many aspects of the Vera, Sugar, and ForSpec verification languages have been merged into Verilog along with the higher-level programming constructs of Superlog [20] (which were taken nearly verbatim from C and C++) to produce SystemVerilog [1]. As a result, Verilog has become the English of the HDL world: voraciously assimilating parts of other languages and making them its own.

The C- and C++-like features added to SystemVerilog read like the list of features in those languages. SystemVerilog adds enumerated types, record types (*struct*s), *typedef*s, type casting, a variety of operators such as +=, operator overloading, control-flow statements such as `break` and `continue`, as well as object-oriented programming constructs such as classes, inheritance, and dynamic object creation and deletion. At the very highest level, it also adds strings, associative arrays, concurrent process control (e.g., fork/join), semaphores, and mailboxes, giving it features only found in concurrent programming languages such as Java.

Perhaps the most interesting features added to SystemVerilog are those directly related to verification. Specifically, SystemVerilog includes constrained, biased random variable generation, user-defined functional coverage checking, and temporal assertions, much like those in Vera, e, PSL, and ForSpec.

Figure 17 illustrates some of the random test-generation constructs in SystemVerilog, which were largely taken from Vera. Compare this with Figure 12.

Figure 18 illustrates some of the coverage constructs in SystemVerilog. In general, one defines "covergroups," which are collections of bins that sample values on a given event, typically a clock edge. Each covergroup defines the sorts of values it will be observing (e.g., values of a single variable, combinations of multiple variables, and sequences of values on a single variable) and rules that define the "bins" each of these values will be placed in. In the end, the simulator reports which bins were empty, indicating that none of the matching behavior was

```
class Bus;
  rand bit[15:0] addr;
  rand bit[31:0] data;

  constraint world_align { addr[1:0] = 2'b0; }
endclass

initial begin
  Bus bus = new;

  repeat (50) begin
    if (bus.randomize() == 1)
      $display("addr = %16h data = %h\n",
               bus.addr, bus.data);
    else
      $display("overconstrained\n");
  end
end

typdef enum { low, mid, high } AddrType;

class MyBus extends Bus;
  rand AddrType atype; // Random variable

  // Additional address constraint
  constraint addr_range {
    (atype == low ) -> addr inside { [0:15] };
    (atype == mid ) -> addr inside { [16:127] };
    (atype == high) -> addr inside { [128:255] };
  }
endclass

task exercise_bus;
  int res;

  // Restrict to low addresses
  res = bus.randomize() with { atype == low; };

  // Restrict to particular address range
  res = bus.randomize()
        with { 10 <= addr && addr <= 20 };

  // Restrict data to powers of two
  res = bus.randomize() with
     { data & (data - 1) == 0 };

  // Disable word alignment
  bus.word_align.constraint_mode(0);

  res = bus.randomize with { addr[0] || addr[1] };

  // Re-enable word alignment
  bus.word_align.constraint_mode(1);
endtask
```

Figure 17: Constrained random variable constructs in SystemVerilog. The example starts with a simple definition of a Bus class that constraints the two least-significant bits of the address to be zero, then invokes the `randomize` method to randomly generate address/data pairs and print the result. Next is a refined version of the Bus class that adds a field taken from an enumerated type that further constrains the address depending on its value. The example ends with a task that illustrates various ways to control the constraints. After examples in the SystemVerilog LRM [1]

```
enum { red, green, blue } color;

bit [3:0] adr, offset;

covergroup g2 @(posedge clk);
  Hue:    coverpoint color;
  Offset: coverpoint offset;

  // Consider (color, adr) pairs, e.g.,
  // (red, 3'b000), (red, 3'b001),  (blue, 3'b111)
  AxC:    cross color, adr;

  // Consider (color, adr, offset) triplets
  // Creates 3 * 16 * 16 = 768 bins
  all:    cross color, adr, Offset;
endgroup

g2 g2_inst = new; // Create a watcher

bit [9:0] a; // Takes values 0--1023

covergroup cg @(posedge clk);

  coverpoint a {
    // place values 0--63 and 65 in bin a
    bins a = { [0:63], 65 };

    // create 65 bins, one for 127, 128, ..., 191
    bins b[] = { [127:150], [148:191] };

    // create three bins: 200, 201, and 202
    bins c[] = { 200, 201, 202 };

    // place values 1000--1023 in bin d
    bins d = { [1000:$] };

    // place all other values
    // (e.g., 64, 66, .., 126, 192, ...)
    // in their own bin
    bins others[] = default;
  }

endgroup

bit [3:0] a;

covergroup cg @(posedge clk);
  coverpoint a {
    // Place the sequences 4 -> 5 -> 6, 7 -> 11,
    // 8 -> 11, 9 -> 11, 10 ->11, 7 -> 12, 8 -> 12,
    // 9 -> 12, and 10 -> 12 into bin sa.
    bins sa = (4 => 5 => 6), ([7:9],10 => 11,12);

    // Create separate bins for 4 -> 5 -> 6,
    // 7 -> 10, 8 -> 10, and 9 -> 10
    bins sb[] = (4 => 5 => 6), ([7:9] => 10);

    // Look for the sequence 3 -> 3 -> 3 -> 3
    bins sc = 3 [* 4];

    // Look for any of the sequences
    // 5 -> 5, 5 -> 5 -> 5, or 5 -> 5 -> 5 -> 5
    bins sd = 5 [* 2:4];

    // Look for any sequence of the form
    // 6 -> ... -> 6 -> ... -> , where "..."
    // represents any sequence excluding 6
    bins se = 6 [-> 3];
  }
endgroup
```

Figure 18: SystemVerilog coverage constructs. The example begins with a definition of a “covergroup” that considers the values taken by the color and offset variables as well as combinations. Next is a covergroup illustrating the variety of ways “bins” may be defined to classify values for coverage. The final covergroup illustrates SystemVerilog’s ability to look for and classify sequences of values, not just simple values. After examples in the SystemVerilog LRM [1]

```
// Make sure req1 or req2 is true
// if we are in the REQ state
always @(posedge clk)
  if (state == REQ)
    assert (req1 || req2);

// Same, but report the error ourselves
always @(posedge clk)
  if (state == REQ)
    assert (req1 || req2)
    else
      $error("In REQ; req1 || req2 failed (\%0t)",
             $time);

property req_ack;
  @(posedge clk) // Sample req, ack at rising edge
    // After req is true, between one and three
    // cycles later, ack must have risen.
    req ##[1:3] $rose(ack);
endproperty

// Assert that this property holds,
// i.e., create a checker
as_req_ack: assert property (req_ack);

// The own_bus signal goes high in 1 to 5 cycles,
// then the breq signal goes low one cycle later.
sequence own_then_release_breq;
  ##[1:5] own_bus ##1 !breq
endsequence

property legal_breq_handshake;
  @(posedge clk)      // On every clock,
  disable iff (reset) // unless reset is true,
  // once breq has risen, own_bus should rise
  // and breq should fall.
  $rose(breq) |-> own_then_release_breq;
endproperty

assert property (legal_breq_handshake);
```

Figure 19: SystemVerilog assertions. The first two *always* blocks check simple safety properties, i.e., that `req1` and `req2` are never true at the positive edge of the clock. The next property checks a temporal property: that `ack` must rise between one and three cycles after each time `req` is true. The final example shows a more complex property: when reset is not true, a rising `breq` signal must be followed by `own_bus` rising between one and five cycles later and `breq` falling.

observed. Again, much of this machinery was taken from Vera (cf. Figure 13).

Figure 19 shows some of SystemVerilog's assertion constructs. In addition to signaling an error when an "instantaneous" condition does not hold (e.g., a set of variables are taking on mutually-incompatible values), SystemVerilog has the ability to describe temporal sequences such as "`ack` must rise between one and five cycles after `req` rises" and check whether they appear during simulation. Much of the syntax comes from PSL/Sugar.

5 Conclusions

VHDL and Verilog remain the dominant hardware description languages and will likely be with us for a long time, although perhaps they will become like assembly language has become to programming: a part of the compilation chain, but not generally written manually. Both have deep flaws, but these can be largely avoided by adhering to coding conventions, and in practice are quite practical design entry vehicles.

The future of the verification languages discussed in this report is less certain. Clearly, there is a need to automate the validation process as much as possible, and these languages do provide useful assistance in the form of biased constrained pseudorandom test case generation, temporal property assertions, and coverage estimates. However, none has clearly proven itself essential to modern IC design, and the plethora of variants and derivatives suggests that their evolution is not complete. An even more serious question is whether the complexity of these languages, especially in their specification of temporal properties, creates more problems than they solve.

The fundamental burdens of specifying digital hardware and verifying its correctness will continue to fall on design and verification languages. Even if those in the future bear little resemblance to those described here, the current crop forms a strong foundation on which to build.

References

[1] Accelera, 1370 Trancas Street #163, Napa, CA 94558. *SystemVerilog 3.1a Language Reference Manual: Accellera's Extensions to Verilog*, May 2004.

[2] Guido Arnout. SystemC standard. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 573–578, Yokohama, Japan, January 2000.

[3] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, San Francisco, California, 1996.

[4] Peter J. Ashenden. *The Student's Guide to VHDL*. Morgan Kaufmann, San Francisco, California, 1998.

[5] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic Sugar. In *Proceedings of the 13th International Conference on Computer-Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367, Paris, France, 2001. Springer-Verlag.

[6] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, 1971.

[7] Janick Bergeron. *Writing Testbenches: Function Verification of HDL Models*. Kluwer, Boston, Massachusetts, second edition, 2003.

[8] J. Bhasker. *A VHDL Synthesis Primer*. Star Galaxy Publishing, Allentown, Pennsylvania, second edition, 1998.

[9] J. Bhasker. *A SystemC Primer*. Star Galaxy Publishing, Allentown, Pennsylvania, second edition, 2004.

[10] Dominique Borrione, Robert Piloty, Dwight Hill, Karl J. Lieberherr, and Philip Moorby. Three decades of HDLs: Part ii, Colan through Verilog. *IEEE Design & Test of Computers*, 9(3):54–63, September 1992.

[11] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[12] Yaohan Chu, Donald L. Dietmeyer, James R. Duley, Fredrick J. Hill, Mario R. Barbacci, Charles W. Rose, Greg Ordy, Bill Johnson, and Martin Roberts. Three decades of HDLs: Part i, CDL through TI-HDL. *ieeedtc*, 9(2):69–81, June 1992.

[13] Youhan Chu. An ALGOL-like computer design language. *Communications of the ACM*, 8(10):607–615, October 1965.

[14] Edmund M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[15] Ben Cohen. *VHDL Coding Styles and Methodologies*. Kluwer, Boston, Massachusetts, second edition, 1999.

[16] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. *Using PSL/Sugar for Formal Verification*. Vhdl-Cohen Publishing, PO Box 2362, Palos Verdes Peninsula, California, 2004.

[17] Al Dewey. VHSIC hardware description (VHDL) development program. In *Proceedings of the 20th Design Automation Conference*, pages 625–628, Miami Beach, Florida, June 1983.

[18] Allen Dewey and Aart J. de Geus. VHDL: Toward a unified view of design. *IEEE Design & Test of Computers*, 9(2):8–17, April 1992.

[19] Allen M. Dewey. *Analysis and Design of Digital Systems with VHDL*. Brooks/Cole Publishing (Formerly PWS), Pacific Grove, California, 1997.

[20] Peter L. Flake and Simon J. Davidmann. Superlog, a unified design language for system-on-chip. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 583–586, Yokohama, Japan, January 2000.

[21] Peter L. Flake, Philip R. Moorby, and G. Musgrave. An algebra for logic strength simulation. In *Proceedings of the 20th Design Automation Conference*, pages 615–618, Miami Beach, Florida, June 1983.

[22] Robert S. French, Monica S. Lam, Jeremy R. Levitt, and Kunle Olukotun. A general method for compiling event-driven simulations. In *Proceedings of the 32nd Design Automation Conference*, pages 151–156, San Francisco, California, June 1995.

[23] Michael J. C. Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, San Diego, California, June 1995.

[24] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer, Boston, Massachusetts, 2002.

[25] Faisal Haque, Jonathan Michelson, and Khizar Khan. *The Art of Verification with Vera*. Verification Central, www.verificationcentral.com, 2001.

[26] Randolph E. Harr and Alec G. Stanculescu, editors. *Applications of VHDL to Circuit Design*. Kluwer, Boston, Massachusetts, 1991.

[27] IEEE Computer Society, 345 East 47th Street, New York, New York. *IEEE Standard VHDL Language Reference Manual (1076–1993)*, 1994.

[28] IEEE Computer Society, 345 East 47th Street, New York, New York. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364–1995)*, 1996.

[29] IEEE Computer Society, 345 East 47th Street, New York, New York. *IEEE Standard Verilog Hardware Description Language (1364–2001)*, September 2001.

[30] Sasan Iman and Sunita Joshi. *The e Hardware Verification Langauge*. Kluwer, Boston, Massachusetts, 2004.

[31] The Institute of Electrical and Electronics Engineers (IEEE), 345 East 47th Street, New York, New York. *IEEE Standard VHDL Reference Manual (1076–1987)*, 1988.

[32] The Institute of Electrical and Electronics Engineers (IEEE), 345 East 47th Street, New York, New York. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*, 1993.

[33] The Institute of Electrical and Electronics Engineers (IEEE), 345 East 47th Street, New York, New York. *IEEE Standard VHDL Synthesis Packages (1076.3–1997)*, 1997.

[34] Stan Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th Design Automation Conference*, Anaheim, California, June 1997.

[35] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer, Boston, Massachusetts, 1989.

[36] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Massachusetts, 1980.

[37] Swapnajit Mittra. *Principles of Verilog PLI*. Kluwer, Boston, Massachusetts, 1999.

[38] Wolfgang Muller, Wolfgang Rosenstiel, and Jurgen Ruf, editors. *SystemC: Methodologies and Applications*. Kluwer, Boston, Massachusetts, 2003.

[39] Wayne E. Omohundro and James H. Tracey. Flowware—a flow charting procedure to describe digital networks. In *Proceedings of the First International Conference on Computer Architecture (ISCA)*, pages 91–97, Gainesville, Florida, December 1973.

[40] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[41] Samir Palnitkar. *Design Verification with e*. Prentice Hall, Upper Saddle River, New Jersey, 2003.

[42] Federic I. Parke. An introduction to the N.mPc design environment. In *Proceedings of the 16th Design Automation Conference*, pages 513–519, San Diego, California, June 1979.

[43] Douglas L. Perry. *VHDL*. McGraw-Hill, New York, third edition, 1998.

[44] Irving S. Reed. Symbolic synthesis of digital computers. In *Proceedings of the ACM National Meeting*, pages 90–94, Toronto, Canada, September 1952.

[45] Douglas J. Smith. VHDL & Verilog compared & contrasted — plus modeled examples written in VHDL, Verilog, and C. In *Proceedings of the 33rd Design Automation Conference*, pages 771–776, Las Vegas, Nevada, June 1996.

[46] Stuart Sutherland. *The Verilog PLI Handbook*. Kluwer, Boston, Massachusetts, 1999.

[47] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston, Massachusetts, fifth edition, 2002.