White Paper

An Overview of SystemVerilog 3.1

by Stuart Sutherland of Sutherland HDL, Inc.
stuart@sutherland-hdl.com

published in EEdesign, May 23, 2003
http://www.eedesign.com/features/exclusive/OEG20030521S0086

SystemVerilog is an extensive set of enhancements to the IEEE 1364 Verilog-2001 standard. These enhancements provide powerful new capabilities for modeling hardware at the RTL and system level, along with a rich set of new features for verifying model functionality.

The primary objectives of this article are to:

- Present an overview of the features in the SystemVerilog 3.1 standard
- Address concerns that perhaps SystemVerilog is not ready for use

It is impossible to cover all the aspects of SystemVerilog in one short article. Only some of the more significant features are presented. For those who would like to know more about the many exciting features of SystemVerilog, Accellera is presenting a free SystemVerilog workshop at DAC. The author of this article also presents in-depth training workshops on SystemVerilog.

A problem that needed solving. For many years, the behavioral coding features of Verilog, plus a few extras such as display statements and simulation control, gave Verilog-based design engineers all they needed to both model hardware and to define a testbench to verify the model. As design sizes have increased, however, the number of lines of RTL code required to represent the design have increased dramatically. Even more significant is the increase in the amount of verification code required to test these very large designs. While modeling large designs and verification routines in traditional RT-level HDLs is still possible, the amount of coding far exceeds what can be accomplished in a reasonable amount of time.

To address these problems, new design languages such as SystemC were created that could model full systems at a much higher level of abstraction, using fewer lines of code. Proprietary Hardware Verification Languages (HVLs) such as Verisity's *e* and Synopsys' VERA were created to more concisely describe complex verification routines [note: company and product names are trademarked names of their respective companies]. These proprietary languages solve a need, but at the cost of requiring engineering teams to work with multiple languages, and often at the expense of simulation performance.

The SystemVerilog standard currently being defined by Accellera takes a different approach to solving the design and verification needs of today's multi-million gate designs. Rather than re-invent the wheel with new languages, Accellera—the combined VHDL International and Open Verilog International organizations—has defined a set of high-level extensions to the IEEE 1364 Verilog-2001 language. The definition of the SystemVerilog 3.1 standard has been completed and is expected to be released in June of this year. Accellera plans to donate the SystemVerilog extensions to the IEEE 1364 Verilog Standards Group, where it is anticipated that the extensions will become part of the next generation of the IEEE 1364 Verilog standard.

SystemVerilog's roots. Accellera chose not to concoct these SystemVerilog enhancements to Verilog from scratch. That would have required re-inventing the wheel and creating a standard based on unproven, untested syntax and semantics. Instead, Accellera relied on donations of technology from a number of companies. These donations include high-level modeling constructs from the SUPERLOG language developed by Co-design, testbench constructs from the Open VERA language and VCS DirectC interface technology donated by Synopsys, and assertions work from several companies, including, to name just a few, OVA from Verplex, ForSpec from Intel, Sugar (renamed PSL) from IBM, and OVA from Synopsys.

Over the past two years, the Accellera SystemVerilog committee and subcommittees have met two to four times each month to standardize these donations. Members of the SystemVerilog committee include experts in simulation engines, synthesis compilers, verification methodologies, members of the IEEE 1364 Verilog Standards Group, and senior design and verification engineers.

Compatibility with Verilog-2001. A primary goal of the SystemVerilog standardization effort has been to ensure that SystemVerilog is fully compatible with the IEEE 1364-2001 Verilog standard. Each of the technology donations selected by the SystemVerilog committee was in a different language than Verilog. The committee carefully reviewed each and every construct and enhancement and made changes where ever necessary to ensure that all SystemVerilog enhancements were fully backward compatible with the Verilog language. All existing Verilog models should work with software tools that implement the SystemVerilog enhancements.

There is one caveat to this backward compatibility. SystemVerilog adds several new keywords to the Verilog language. There is the possibility that an existing model may have used one or more of these new keywords as a regular identifier. This is a relatively minor problem that software tools can easily deal with using compatibility switches.

Assertions. SystemVerilog provides special language constructs to verify design behavior. An assertion is a statement that a specific condition, or sequence of conditions, in a design is true. If the condition or sequence is not true, the assertion statement will generate an error message.

SystemVerilog assertions are an integration of PSL (originally called "Sugar"), OVA, OVL, ForSpec and other assertion technologies, all of which have been donated to Accellera. The result is a single assertion language that provides a convergence of the best features of each of these assertion methodologies.

A SystemVerilog assertion can test for a sequence of conditions that span multiple clock cycles. The following assertion example checks that in the `FETCH` state, `request` must be true immediately, and `grant` must become true one to three clock cycles later, followed by `request` becoming false by the next clock cycle and `grant` being false by the next clock cycle after that. Should this sequence not occur, the assert statement will automatically generate an error message.

```
sequence request_check;
   request ##[1:3] grant ##1 !request ##1 !grant;
endsequence

always @(posedge clock)
   if (State == FETCH)
      assert property request_check;
```

SystemVerilog allows much more complex assertion sequences to be easily constructed than what is shown in this simple example. Special functions are also provided to check for complex expressions, such as only a single bit being set in a one-hot state machine controller. Another important feature of SystemVerilog assertions is the ability to define assertions outside of Verilog modules, and then bind them to a specific module or module instance. This allows verification engineers to add assertions to existing Verilog models, without having to change the model in any way.

Interfaces. Verilog connects one module to another through module ports. This requires a detailed knowledge of the intended hardware design, in order to define the specific ports of each module that makes up the design. Several modules often have many of the same ports, requiring redundant port definitions for each module. Every module connected to a PCI bus, for example, must have the same ports defined.

SystemVerilog *interfaces* provide a new, high level of abstraction for module connections. An interface is defined independent from modules, between the keywords `interface` and `endinterface`. Modules can use an interface the same as if it were a single port. In its simplest form, an interface can be considered a bundle of wires. Interfaces go far beyond just representing bundles of interconnecting signals, however. An interface can also include functionality that is common to each module that uses the interface. In addition, an interface can include built-in protocol checking.

Global declarations. Verilog has an implicit global name space which holds the names of modules, primitives, etc. SystemVerilog makes this global space accessible to the user. Any declarations outside of a module boundary are in the global, or root, name space. All modules, anywhere in the design hierarchy, can refer to names declared in the root space. This allows global variables, type definitions, functions and other information to be declared, that are shared by all levels of hierarchy in the design.

Relaxed data type rules. The Verilog language has strict rules on where net data types (`wire`, `wand`, `wor`, etc.) must be used, and where variables (`reg`, `integer`, etc.) must be used. SystemVerilog relaxes these rules, allowing variable types to be used in almost any context. The relaxed rules make it much easier to write hardware models without concern about which data type class to use. Since variables do not have wired-logic resolution like net data types, semantic restrictions ensure that a variable cannot be driven by multiple output ports or multiple continuous assignments.

Data types. Verilog provides hardware-centric net and variable data types. These types represent 4-state logic values, and are used to model and verify hardware behavior at a detailed level. The net data types also have multiple strength levels and resolution functions for multiple drivers of the net. SystemVerilog adds several new data types, which allow hardware to be modeled at more abstract levels, using data types more intuitive to C programmers.

- **`class`** — an object-oriented dynamic data type, similar to C++ and Java.
- **`byte`** — a 2-state signed variable, that is defined to be exactly 8 bits.
- **`shortint`** — a 2-state signed variable, that is defined to be exactly 16 bits.
- **`int`** — a 2-state signed variable, similar to the `int` data type in C, but defined to be exactly 32 bits.
- **`longint`** — a 2-state signed variable, that is defined to be exactly 64 bits.
- **`bit`** — a 2-state unsigned data type of any vector width.
- **`logic`** — a 4-state unsigned data type of any vector width, equivelant to the Verilog `reg` data type.
- **`shortreal`** — a 2-state single-precision floating-point variable, the same as the `float` type in C.
- **`void`** — represents no value, and can be specified as the return value of a function.

User defined types. SystemVerilog provides a method to define new data types using `typedef`, similar to C. The user-defined type can then be used in declarations the same as with any data type.

```
typedef unsigned int uint;
uint a, b;
```

Enumerated types. SystemVerilog allows the creation of enumerated types, using a C-like syntax. An enumerated type has a value from a set of named values.

```
enum {red, green, blue} RGB;
enum logic [2:0] {WAIT=3'b001, LOAD=3'b010, DONE=3'b100} states;
```

SystemVerilog also provides several built-in methods for working with enumerated types. These methods allow, for example, incrementing to the next value in a type list, without having to know the name of that next value.

Structures and unions. SystemVerilog adds structures and unions, which allow multiple signals, of various data types, to be bundled together and referenced by a single name.

```
struct {
   bit [15:0] opcode;
   logic [23:0] addr;
} IR;

union {
   int i;
   shortreal r;
} N;
```

Fields within a structure or union are referenced using a period between the structure or union name and the field name, as in C.

```
IR.opcode = 1; //set the opcode field in IR
N.r = 0.0; //set N as floating point value
```

A structure can be assigned as a whole, using a list of values, the same as in C.

```
IR = {5, 200};
```

Casting. SystemVerilog adds the ability to change the type, vector size or signedness of a value using a cast operation. To remain backward compatible with the existing Verilog language, casting in SystemVerilog uses a different syntax than C.

```
int'(2.0 * 3.0)    //cast result to int
mytype'(foo)       //cast foo to the user-defined type of mytype
17'( x - 2)        //cast the operation to 17 bits
signed'(x)         //cast x to a signed value
```

Arrays. SystemVerilog enhances Verilog arrays in several significant ways, including the addition of *dynamic arrays* and *associative arrays*. Dynamic arrays are one-dimensional arrays where the size of the array can be changed dynamically. Built-in methods provide a means to set and change the size of dynamic arrays during run-time. Associative arrays are one-dimensional sparse arrays that can be indexed using values such as enumerated type names. Special built-in methods for working with associative arrays are provided: `exists()`, `first()`, `last()`, `next()`, `prev()` and `delete()`.

Classes. SystemVerilog adds object oriented classes to the Verilog language, similar to C++. A class can contain data declarations (referred to as "*properties*"), plus tasks and functions for operating on the data (referred to as "*methods*"). The properties and methods together define the contents and capabilities of an "*object*". Classes can have inheritance and public or private protection, as in C++.

Classes allow objects to be dynamically created, deleted and assigned values. Objects can be accessed via handles, which provide a safe form of pointers. SystemVerilog does not require the complex memory allocation and de-allocation of C++. Memory allocation, de-allocation and garbage collection are automatically handled, preventing the possibility of memory leaks.

An example of a SystemVerilog object definition is:

```
class Packet;
   bit [3:0] command;
   bit [39:0] address;
   bit [4:0] master_id;
   integer time_requested;
   integer time_issued;
   integer status;

   function new();
      command = 4'hA;
      address = 40'hFE;
      master_id = 5'b0;
   endfunction

   task clean();
      command = 4'h0; address = 40'h0;
      master_id = 5'b0;
   endtask
endclass
```

It is important to note that classes are different than other Verilog and SystemVerilog data types, in that classes are dynamic by nature, instead of static. The storage required by a class object is automatically created when needed, and deleted when no longer required. The dynamic nature of classes make them ideal for testbench modeling, whereas the static nature of other variable types make them appropriate for modeling hardware, which is also static in nature. Because of the dynamic nature of classes, they are not considered synthesizable constructs. Classes are intended for verification routines and highly abstract system-level modeling.

String data type. SystemVerilog adds a **`string`** data type, defined as a built-in class. The string data type contains a variable length array of ASCII characters. Each time a value is assigned to the string, the length of the array is automatically adjusted. This eliminates the need for the user to define the size of character arrays, or to be concerned about strings being truncated due to an array of an incorrect size.

String operations can be performed using standard Verilog operators: `=`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `{,}`, `{{}}`. In addition, a number of methods are defined to manipulate strings: `len()`, `putc()`, `getc()`, `toupper()`, `tolower()`, `compare()`, `icompare()`, `substr()`, `atoi()`, `atohex()`, `atooct()`, `atobin()`, `atoreal()`, `itoa()`, `hextoa()`, `octtoa()`, `bintoa()` and `realtoa()`.

Operators. SystemVerilog adds several new operators:

- `++` and `--` increment and decrement operators
- `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `<<<=` and `>>>=` assignment operators

Unique and priority decision statements. The Verilog `if-else` and `case` statements can be a source of mismatches between RTL simulation and how synthesis interprets an RTL model, if strict coding styles are not followed. The synthesis `full_case` and `parallel_case` pragmas can lead to further mismatches if improperly used, as they affect synthesis but not simulation.

SystemVerilog adds the ability to explicitly specify when each branch of a decision statement is unique or requires priority evaluation, using the keywords `unique` and `priority`. These keywords affect simulators, synthesis compilers, formal verifiers and other tools, ensuring that all tools interpret the model the same way.

```
priority casez(a)
   3'b00?: y = in1; // a is 0 or 1
   3'b0??: y = in2; //a is 2 or 3;
   default: y = in3; //a is any other value
endcase
```

Enhanced for loops. Verilog `for` loops can have a single initial assignment statement, and a single step assignment statement. The variable used as the loop control must be declared outside of the loop. SystemVerilog enhances `for` loops to allow the loop control variable to be declared as part of the `for` loop, and allows the loop to contain multiple initial and step assignments.

```
for (int i=1, shortint count=0; i*count < 125; i++, count+=3)
   ...
```

Bottom testing loops. Verilog has the `for`, `while` and `repeat` loops, all of which test to execute the loop at the beginning of the loop. SystemVerilog adds a `do-while` loop, which tests the loop condition at the end of executing code in the loop.

Jump statements. Verilog provides the ability to jump to the end of a named statement group using the `disable` statement. SystemVerilog adds the C `break` and `continue` keywords, which do not require the use of block names, and a `return` keyword, which can be used to exit a task or function at any point. SystemVerilog does not include the C `goto` statement.

Final blocks. Verilog has `initial` blocks that begin execution at the very beginning of simulation. SystemVerilog adds `final` blocks, which execute at the very end of simulation, just before simulation exits. Final blocks can be used in verification to print simulation results, such as code coverage reports.

Hardware-specific procedures. Verilog uses the `always` procedure as a general purpose construct to represent models of sequential logic, combinational logic and latched logic, as well as in testbenches and code that is not intended to be synthesized. Synthesis and other software tools must infer the intent of the `always` procedure from the sensitivity list and the statements within the procedure.

SystemVerilog adds three new procedures to explicitly indicate the intent of the logic:

- `always_ff` — the procedure is intended to represent sequential logic
- `always_comb` — the procedure is intended to represent combinational logic
- `always_latch` — the procedure is intended to represent latched logic

The following example illustrates modeling a multiplexer using these new procedures:

```
always_comb
   if (sel) y = a;
   else y = b;
```

Software tools can examine the procedure contents to ensure that the functionality matches the type of procedure. In the example above, since a tool knows the designers intent is to represent combinational logic, the tool can check that the procedure makes assignments to the same variables for every branch of logic, and that the branches cover every possible condition. If these conditions are not true, the tool can report that the procedure does not properly model the engineers intent. These explicit procedures also have special semantics that are different than the general purpose always procedure, which more accurately simulate the behavior of the specified type of hardware.

Task and function enhancements. SystemVerilog adds several enhancements to the Verilog task and function constructs. Only a few of the enhancements are highlighted in this article.

- Function return values can have a `void` return type. Void functions can be called the same as a Verilog task. The difference between a void function and a task is that Verilog functions have several semantic restrictions, such as no time controls.
- Functions can have any number of inputs, outputs and inouts, including none.

- Values can be passed to a task or function in any order, using the task/function argument names. The syntax is the same as named module port connections.
- Task and function input arguments can be assigned a default value as part of the task/function declaration. This allows the task or function to be called without passing a value to each argument.
- Task or function arguments can be passed by reference, instead of copying the values in or out of the task or function. Passing by reference allows the task or function to work directly with the value in the calling scope, instead of a local copy of the value. To use pass by reference, the argument direction is declared as a **`ref`**, instead of `input`, `output` or `inout`.

Enhanced fork—join. In the Verilog `fork—join` statement block, each statement is a separate thread, that executes in parallel with other threads within the block. The block itself does not complete until every parallel thread has completed. Therefore, any statements following a `fork—join` are blocked from execution until all the forked parallel threads have completed execution.

SystemVerilog adds `fork—join_none`, and `fork—join_any` blocks:

- **`join_none`** — statements that follow the `fork—join_none` are not blocked from execution while the parallel threads are executing. Each parallel thread is an independent, dynamic process.
- **`join_any`** — statements which follow a `fork—join_any` are blocked from execution until the first of any of the threads has completed execution.

Inter-process synchronization. SystemVerilog provides three powerful ways for synchronizing parallel activities within a testbench or abstract model: ***semaphores***, ***mailboxes***, and enhanced ***event*** types.

A *semaphore* is a built-in object class. Semaphores serve as a bucket with a fixed number of "*keys*". Processes using semaphores must procure one or more keys from the bucket before they can continue execution. When the process completes, it returns its keys to the bucket. If no keys are available, the process must wait until a sufficient number of keys have been returned to the bucket by other processes. The semaphore class provides several built-in methods for working with semaphores: `new()`, `get()`, `put()` and `try_get()`.

A *mailbox* is another built-in class, which allow messages to be exchanged between processes. A message can be added to the mailbox at anytime by one process, and retrieved anytime later by another process. If there is no message in the mailbox when a process tries to retrieve one, the process can either suspend execution and wait for a message, or continue and check again at a later time. Mailboxes behave like FIFOs (First-In, First-Out). When the mailbox is created, it can be defined to have a bounded (limited) size, or an unbounded size. If a process tries to place a message into a bounded mailbox that is full, it will be suspended until there is enough room. The mailbox class also provides several built-in methods: `new()`, `put()`, `tryput()`, `get()`, `peek()`, `try_get()` and `try_peek()`.

The Verilog `event` type is a momentary flag that has no logic value and no duration. The `event` type can be triggered, and other processes can be watching for the trigger. If a process is not watching when the event is triggered, the event will not be detected. SystemVerilog enhances the `event` data type by allowing events to have persistence throughout the current simulation time step. This allows the event to be checked after it is triggered.

Constrained random values. The Verilog standard includes a very basic random number function, called `$random`. This function, however, gives very little control over the random sequence and no control over the range of the random numbers. SystemVerilog adds two random number classes, **`rand`** and **`randc`**, using SystemVerilog's object class system. These classes provide methods to set seed values and to specify various constraints on the random values that are generated.

The following example creates a user-defined class called `Bus`, that can generate a random address and data value, with limits on the value sizes. A constraint on the address ensures that the lower two bits of random address value will always be zero. The class is then used to generate 50 random address/data value pairs, using the `randomize()` method, which is part of the `rand` class.

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;
   constraint word_align { addr[1:0] == 2'b0; }
endclass

//Generate 50 random data values with quad-aligned addresses
Bus bus = new;
repeat(50)
   begin
      int result = bus.randomize();
   end
```

Using the `rand` and `randc` classes and methods, much more elaborate random number generation is possible than what is shown in the preceding simple example.

Testbench program block. In Verilog the testbench for a design must be modelled using Verilog hardware modeling constructs. Since these constructs were primarily intended to model hardware behavior at various levels of abstraction, they have no special semantics to indicate how test values should be applied to the hardware. SystemVerilog adds a special type of code block, declared between the keywords `program` and `endprogram`. The program block has special semantics and syntax restrictions for modeling a testbench. A program block:

- Contains a single initial block
- Executes events in a "reactive phase" of the current simulation time, appropriately synchronized to hardware simulation events
- Can use a special **`$exit`** system task that will wait to exit simulation until after all concurrent program blocks have completed execution (unlike `$finish`, which exits simulation immediately)

Clocking domains. SystemVerilog adds a special clocking block, using the keywords `clocking` and `endclocking`. The clocking block identifies a "*clocking domain*", containing a clock signal and the timing and synchronization requirements of the blocks in which the clock is used. A testbench may contain one or more clocking domains, each containing its own clock plus an arbitrary number of signals. Clocking domains allow the testbench to be defined using a cycle-based methodology, rather than the traditional event-based methodology of defining specific transition times for each test signal.

A clocking domain can define detailed skew information for the relationship between a clock and one or more signals. *Input skews* specify the amount of time before a clock edge that signals should be sampled by a testbench. *Output skews* specify how many time units after a clock edge that signals should be driven. Note that "input" and "output" are relative to the testbench—that is, an output of the design under test is an input to the testbench. For example:

```
clocking bus @(posedge clk);
   default input #2ns output #1ns;      //default I/O skew
   input   enable, full;
   inout   data;
   output  empty;
   output  #6ns reset;                  //reset skew is different than default
endclocking

initial //Race conditions with Verilog; no races with SystemVerilog
   begin
      @(posedge clk)  input_vector = ...         //drive stimulus onto design
      @(posedge clk)  $display(chip_output);     //sample result
                      input_vector = ...         //drive stimulus onto design
      @(posedge clk)  $display(chip_output);     //sample result
                      input_vector = ...         //drive stimulus onto design
   end
```

In regular Verilog, which does not have clocking domains, the `initial` procedure in the preceding example would have race conditions with the design under test, if the design is using the same positive edge of the clock to store values in registers. By using SystemVerilog clocking domain skews, the testbench can reference a clock edge to sample a value or drive stimulus. The appropriate skew will automatically be applied, thus avoiding race conditions with the design. Clocking domains greatly simplify defining a testbench that does not have race conditions with the design being tested.

Direct Programming Interface (DPI). SystemVerilog provides a means for SystemVerilog code to directly call functions written C, C++ or SystemC, without having to use the complex Verilog Programming Language Interface (PLI). Values can be passed directly to the foreign language function, and values can be received from the function. The foreign language function can also call Verilog tasks and functions, which gives the foreign language functions access to simulation events and simulation time. The SystemVerilog DPI provides a bridge between high-level system design using C, C++ or SystemC and lower-level RTL and gate-level hardware design.

Is SystemVerilog ready for use? Savvy engineers will have no doubt seen many SystemVerilog features in this article that will help them in their design and verification work. This, then, raises two important questions: *"Is this new SystemVerilog standard really ready for software companies to implement, and how soon will software tools supporting SystemVerilog be available?"*.

The answer to *"Is the SystemVerilog standard ready?"* is a resounding YES! The many experts from EDA companies and the Verilog user community that participated in the development of SystemVerilog have voted that the standard is ready to be released. Their recommendation, along with the SystemVerilog 3.1 Language Reference Manual, have been sent to the Accellera board of directors for final approval. The Board will be voting in late May 2003 on ratifying SystemVerilog 3.1 (a subset of SystemVerilog which focussed primarily on enhanced hardware modeling constructs was approved by the Accellera board in June of 2002, and released to EDA vendors as SystemVerilog 3.0).

As to *"When will software tools support SystemVerilog?"*, there are two answers. First, several EDA vendors are aggressively adding SystemVerilog to their existing Verilog tool sets. Look for some exciting product announcements and suite demonstrations at the upcoming Design Automation Conference, or DAC (June 2-6 in Anaheim, California). Second, since most of the SystemVerilog extensions come from proven technology in commercial software tools, you are quite likely already using portions of SystemVerilog in your current designs, just under the guise of other proprietary names.

It is worth noting that one EDA company on the Accellera board has raised a number of concerns regarding the current state of the SystemVerilog standard, and feels SystemVerilog should not be released at this time. The concerns that have been expressed are not against extending Verilog, but rather about the syntax or semantics of the extensions. The Accellera SystemVerilog committee has thoroughly analyzed all of the concerns expressed, and found no reason to delay the release of the SystemVerilog standard. A small number of the concerns are legitimate, and are being addressed by the SystemVerilog committee. Many of this company's concerns are in regard to the difficulty of implementing certain SystemVerilog constructs. Since SystemVerilog is based on donations of proven technology, and since other EDA vendors either have implemented, or are implementing these SystemVerilog constructs, the concern about the difficulty level is largely unfounded. It is never a simple task to retrofit old software products with new features, but that is no reason to not provide users with essential new language features. Most of the concerns expressed by this EDA have to do with how certain enhancements are specified syntactically or semantically. For example, should the new `int`, `byte` and other data types be keywords or built-in object classes. These concerns are simply a matter of there being more than one good way to do the same thing. The majority of EDA companies and designers on the SystemVerilog committees voted to use one approach, while this one EDA vendor would have preferred a different approach. Concerns of this nature are simply a matter of preference, and have no bearing on the worthiness, correctness or readiness of the SystemVerilog standard.

Conclusion. SystemVerilog provides a major set of extensions to the Verilog-2001 standard. These extensions allow modeling and verifying very large designs more easily and with less coding. By taking a proactive role in extending the Verilog language, Accellera is providing a standard that can be implemented by simulator and synthesis companies quickly. It is expected that the IEEE Verilog standards group will adopt the SystemVerilog extensions as part of the next generation of the IEEE 1364 Verilog standard.

SystemVerilog extends the modeling aspects of Verilog, and adds a Direct Programming Interface which allows C, C++, SystemC and Verilog code to work together without the overhead of the Verilog PLI. SystemVerilog bridges the gap between hardware design engineers and system design engineers. SystemVerilog also significantly extends the verification aspects of Verilog by incorporating the capabilities of VERA and powerful assertion constructs.

Adding these SystemVerilog extensions to Verilog creates a whole new type of engineering language, an **HDVL**, or **H**ardware **D**escription *and* **V**erification **L**anguage. This unified language will greatly increase the ability to model huge designs, and verify that these designs are functionally correct.

The SystemVerilog standard is ready for you to use, and several EDA companies are working on adding the SystemVerilog extensions to their Verilog software tools.

References

"SystemVerilog 3.1, ballot draft: Accellera's Extensions to Verilog", Accellera, Napa, California, April 2003.

"Verilog 2001: A Guide to the new Verilog Standard", Stuart Sutherland, Kluwer Academic Publishers, Boston, Massachusetts, 2001.

About the author. Mr. Stuart Sutherland is a member of the Accellera SystemVerilog technical subcommittee that is defining SystemVerilog, and is the technical editor of the SystemVerilog Language Reference Manual. He is also a member of the IEEE 1364 Verilog Standards Group, where he serves as chair of the PLI task force. Mr. Sutherland is an independent Verilog consultant, and specializes in providing comprehensive expert training on Verilog, SystemVerilog and the PLI. Mr. Sutherland can be reached by e-mail at stuart@sutherland-hdl.com.