Why SystemVerilog?

Peter Flake
Elda Technology Ltd
Lewes, UK
flake@elda.demon.co.uk

***Abstract*—SystemVerilog is a hardware design and verification language (HDVL) that combines the features of several domain-specific languages for digital hardware design and verification with a general purpose object-oriented programming language. The paper discusses the functional and performance requirements of each domain, and the history of why particular language design choices were made. SystemVerilog is also compared with SystemC and VHDL.**

Keywords—HDVL; Verilog; SystemVerilog; domain-specific language;

I. INTRODUCTION

SystemVerilog is a hardware design and verification language (HDVL) that has been an IEEE 1800 standard since 2005, and was an Accellera standard before then. SystemVerilog is a superset of the previous Verilog® hardware description language standard IEEE 1364 [1]. The next (2009) version of the 1800 standard incorporated the content of the 1364 standard, and this was revised in 2012 [2].

Many people criticize SystemVerilog for looking like multiple languages stuck together. This paper shows why this has happened and why it is necessary. The philosophy is similar to that of Perl, which has been called a "post-modern language" [3] that does not aim for simplicity but aims for usefulness.

II. DOMAIN-SPECIFIC LANGUAGES

A. General

The idea of a domain-specific language [4] is that it has semantics that reflect the concepts of a particular domain such as modeling. The semantics are in turn supported by syntax that reflects the rules of combination of the concepts, such as how a model is constructed. Together these provide several benefits:

- Ease of reading and writing for domain experts.
- Concise description of e.g. models in the domain. The number of bugs introduced in coding is strongly related to the amount of code [5], so conciseness is a huge benefit, provided it improves readability.
- Early error reporting, with messages that are meaningful to domain experts.
- Capturing the user intent to aid analysis and transformation of the code.
- Optimization of data structures to simplify analysis and improve performance.

All these improve user productivity.

B. Domains for an HDVL

A digital hardware design and verification language has to represent digital hardware at various levels of abstraction as well as the testbench, which must model scenarios of the environment and assertions of correct behavior. In particular the language must represent:

1. Netlist: A netlist describes logic gates and switches (pass transistors) connected by wires, organized into a hierarchical structure of modules, with a naming system. The leaf modules are cells (in ASIC designs). Each type of cell has its logic function (such as a 2 input NAND gate) and delays between each input and output. Since these delays depend on the physical layout, it should be possible to set them separately from the netlist description itself. The netlist normally has regular structures, so arrays of gates, modules and wires are convenient to reduce the amount of coding. The most general way to manipulate these arrays is by loops that build the netlist.

2. Register transfer: At register transfer level, a logic design consists of Boolean and integer arithmetic expressions and concurrent assignments to clocked registers. The expressions that represent combinational logic may be evaluated by asynchronous data flow and should not contain loops (cycles). In the concurrent assignments, all the expressions on the right hand side must be evaluated before the left hand sides are updated. To describe complex and regular Boolean expressions, it is convenient to be able to use procedural code including user-defined functions, like in a general-purpose programming language.

3. General programming: To model the scenarios of the environment for simulation, and to instrument the code for debugging, it is necessary to have I/O functions like in a general purpose programming language, as well as concurrent processes with waiting for times or events. It is useful to be able to link to other code, such as a third party tool.

4. Testbench: A very concise way of generating a variety of test cases is to use constrained randomized data and sequences. For example a data packet can be constructed with various

header fields and a payload. There are constraints on the fields and these must be provided to a constraint solver. Such a packet must be created and then kept for a while in the simulation, so it requires heap storage. Variations of the packet are needed for testing, so an object-oriented or aspect-oriented language is convenient. The testbench also needs a flexible connection to the design under test (DUT).

5. Temporal properties: Many of the bugs in hardware design are caused by incorrect sequences of signals. Specifying and checking a hardware communication protocol requires the expression of temporal properties. These are most conveniently written like the regular expressions of a finite state grammar, with sequences of Boolean expressions and timing.

6. Functional coverage: It is important to check that verification has included all control states, significant data values, and combinations of these. For this purpose it is necessary to specify when they are sampled and what values or ranges are significant.

III. EVOLUTION

Before 1980, before Verilog and VHDL, most logic simulators had four simple languages: netlist, behavioral modeling for static components (often in C), test-bench/waveform, and simulation control/debug. Register transfer languages were new, and there were IEEE conferences on the subject of computer hardware description languages.

In 1981 HILO [6] introduced a real time register transfer language which included a netlist with gates, switches and wires combined with combinational or latched logic in Boolean expressions and concurrent event-triggered register assignments. This avoided the need for two languages to describe hardware. Its syntax used square brackets for arrays and had a type name followed by an instance list like C.

In 1986, Verilog [7] combined static sequential processes, which could wait for time or event, with a netlist of gates, switches and wires and combinational or latched logic in Boolean expressions. The processes were written in a C-like language and could call static non-reentrant tasks and express simple test-benches. A range of built-in tasks provided control and debug, so only one language was needed for simulation. A programming language interface to C allowed third party tools such as delay calculators or waveform displays to be efficiently coupled with a simulator. The Verilog-XL® simulator had better functionality and speed than its competitors and, by copying the established HILO netlist syntax, Verilog quickly became popular even though it was a proprietary language at first. In 1989 Open Verilog International was founded to make the language public.

At around the same time VHDL [8] was designed to model hardware with an approach that owed much more to general-purpose programming languages. There are no gates or switches built into the language, and even the set of logic values was not standardized at first. Stack and heap memory is provided, but until recently there was no C interface to the simulator (VHPI). The two phase (evaluate/update) simulation algorithm is more deterministic than the intermingled one of Verilog, but slower. Thus VHDL did not facilitate either the timing accuracy or the performance of Verilog simulators.

However, writing complex testbenches is much more of a programming task than describing hardware. Enumerations, structures/records, multi-dimensional arrays and dynamic memory are useful. VHDL has these features and so has an advantage over Verilog. Indeed Verilog is a poor language for general-purpose programming.

The original objectives of Superlog [9], developed from 1997 to 2002, were to provide a single language for system modeling, hardware design and testbenches. Verilog was, and still is, the leading language for ASIC design [10], and therefore had to be the basis for Superlog. Since C was well established in system modeling, and to some extent in testbenches, the language also needed to provide the features of C, and easy interfacing with C code such as the operating system.

The earliest version of Superlog was not a strict superset of Verilog, because of a desire to "tidy up" Verilog. It was pointed out that this would be a major obstacle to adoption, so Superlog was revised to be a strict superset of IEEE 1364-2001, which was the result of a parallel effort to enhance Verilog, e.g. with the **generate** construct.

In 2001, the hardware design part of Superlog was donated to Accellera to become a standard, and this was eventually named SystemVerilog [11], as a contrast to SystemC [12]. SystemC had appeared in 1999 as a challenge to the idea of a domain-specific language, by providing a set of macros and a C++ library for hardware description.

In the testbench area, pseudo-random test case generation was becoming popular. Domain-specific hardware verification languages Vera [13] and e [14] had appeared. In 2002 Vera was donated to Accellera and the SystemVerilog committee worked on incorporating its features of randomized classes with constraints, random sequences, and functional coverage.

Similarly, assertion based verification was being adopted, and the Open Vera Assertion language [15] was also donated to Accellera for incorporation into SystemVerilog. A separate Accellera committee [16] chose a different assertion language which became PSL. Both are based on temporal logic, and there has been work to harmonize the semantics of the two languages, though their syntaxes differ.

IV. SYSTEMVERILOG FEATURES

The following discussion groups the features according to the domains previously set out. Of course, some features are used in multiple domains.

A. Netlist

The netlist uses named nets, which can be in arrays. A net can have a built-in resolution function, such as AND, or can store a value, or can have a user-defined resolution function. The most common net type is a **wire**, whose value may have up to 7 strength levels of 0/1/X as well as high impedance Z.

These values support switch level simulation of complicated transistor networks. Like Verilog, SystemVerilog has a wide range of built-in gates and switches, and also allows users to define their own primitives in a tabular format. Such a primitive can have feedback, and therefore can model a flip-flop. Each gate or primitive has an optional instance name and a list of nets to which it is connected.

Gates can be grouped into a module, with a type name, optional parameters and an optional port list (input, output or inout). The module can also include path delays from input to output. These delays may be state sensitive.

A module can contain other modules, each of which has an instance name and may have actual parameters and port connections, matched by position or by name. A connection may be a slice of a net array or a concatenation of nets. A wild card allows automatic name matching if the port and its connecting net have the same name. The instances may be in **generate** blocks, which allow iterative and conditional structures to be built using **for/if/case** constructs controlled by parameters.

Nets can be bundled together in an **interface**, like variables are bundled in a structure. The interface can be passed through the levels of hierarchy as a single name and the nets accessed as *interfaceinstance.netname*. A **modport** declaration allows **input/output/inout** to be specified for each net or net array in an interface port.

The wild card and the interface construct were both introduced in SystemVerilog to reduce the amount of text required for a netlist description, and the number of design entry errors.

B. Register Transfer

The continuous assignment allows a net or net array to be driven by an expression, which may include a function call. This is like the dataflow concurrent statement in VHDL.

A process contains procedural statements. These are executed once for an **initial** process or a **final** process and repeatedly for an **always** process. Procedural statements can wait for an event, a delay, or a condition. Procedural assignments are to variables, which have data types. The data types can be built-in or user defined. A variable can be used instead of a net for the output port of a module, or connected to the input port of a module instance, or on the left hand side of a continuous assignment, provided there is only one continuous driver. Thus the distinction in use between nets and variables differs from the distinction between signals and variables in VHDL and SystemC.

The built-in data types used for RTL design are **bit**, which has two values 0 and 1, and **logic**, which has four values 0, 1, X (unknown) and Z (high impedance). These can be assembled into packed arrays, which can be treated as single objects for arithmetic. For example, a packed array of 32 bits can be used as the built-in type **int**, without any type conversion. Such variables can also be assembled into packed structures, which again can be treated as single objects for arithmetic.

This illustrates the approach to data type conversion, which resembles that of C in being much less strict than VHDL. When a logic expression is used as a condition, 0, X and Z are treated as false and 1 as true. This treatment of X can cause problems, since an unknown value can sometimes be a 1.

Enumerated types can have a user-defined coding, like in C. There are also unpacked structures, unions, and unpacked arrays, typically used for modeling memories, where only one element can be accessed at a time, as in C. Different memory layouts can be used to implement packed and unpacked arrays and structures, allowing different performance optimizations.

A procedural assignment can be blocking or non-blocking. A blocking assignment happens immediately, before the next statement is executed. A non-blocking assignment is deferred until later in that time slot, if no delay is specified, or after the time delay specified. The next statement is therefore executed before the assignment. Non-blocking assignments are used to model clocked registers.

A function can return a value and must contain procedural statements that execute without delay. A function call that returns a value is an expression, and SystemVerilog does not allow expressions to wait.

A task does not return a value and may wait for a delay or event, like a procedure in VHDL. A task call is a procedural statement.

To assist synthesis, there are some extra keywords. The **always_comb**, **always_latch**, and **always_ff** keywords identify the intent of the process, so that a synthesis tool can detect user errors [17]. The **unique** and **priority** keywords qualify **case** and **if** statements to check for overlapping conditions or uncovered conditions.

C. General Programming

As discussed above, system level modeling and test-benches tend to need higher level programming features. There are features, borrowed from scripting languages, such as dynamically sized arrays, associative arrays, queues, and strings. These are built into the language rather than library functions as in C++ or VHDL.

Whereas Superlog followed C and VHDL in having explicit heap de-allocation, SystemVerilog follows Java and Vera in relying on garbage collection.

As well as static processes, SystemVerilog allows dynamic processes. The fork keyword spawns concurrent processes of execution which can terminate themselves or be terminated by a disable keyword. There is a built-in process class for fine grain process control.

Processes can synchronize by named events, and communicate by mailboxes, which can have messages of any data type. Semaphores can be used to serialize access to resources.

Transaction level modeling can be done using the interface construct. As well as nets, it can contain variables, tasks, functions and processes. A modport can import or export a

task or function. This allows a transaction level model to be connected together like a netlist.

SystemVerilog has two methods of interfacing to C, the direct programming interface (DPI) and the so-called system tasks and functions, which are a legacy of Verilog. The DPI allows SystemVerilog to call imported C functions, and C to call exported SystemVerilog functions and tasks. Since a task can wait, such a call requires the SystemVerilog context of the previous imported call, which is in effect a "C task". The DPI arguments are limited to data types that are compatible with C. The system tasks and functions have names that begin with a $ and also call C code, but with a different mechanism for passing arguments. This is more cumbersome but also more general than that used by the DPI.

The legacy Verilog programming interface (VPI) is for accessing the simulation data structure, either for navigation or for setting values or callbacks. These callbacks may either relate to phases of the simulation such as advancing time, or to particular value change events. This feature allows third party tools such as delay calculators or waveform viewers to be linked in.

There are many built-in system tasks and functions for I/O and other utilities, and these can be extended using the VPI.

D. Testbench

Variables that are class members can be declared with a **rand** keyword, and named constraints can be given in the class. These constraints can include set membership, range restrictions, or relationships between variables. Constraints can guarantee that a random packet is correctly formatted, for example. The **randc** keyword provides cyclic values to reduce duplication of test cases.

An object of the class can be randomized again, using *object.randomize* with a new constraint, or with the earlier constraint disabled. Inheritance can be used to create a range of classes for various test cases such as error conditions.

Random control is just as important as random data, and SystemVerilog provides two mechanisms for this. The **randcase** keyword gives a weighted random multi-way branch. The **randsequence** keyword introduces a weighted random generative grammar.

Although these randomization features can be implemented in a general-purpose language with a constraint solver library, it requires much more coding and therefore a much greater chance of introducing bugs.

Connecting a testbench to a design requires care in the timing between the data and the clock to ensure that hazards are not introduced. SystemVerilog has a **clocking** block to allow accurate timing of the testbench.

It is convenient to be able to switch a test between similar parts of a design. A virtual interface is a handle to an actual interface, which provides the ability to get and set variables in different instances of the same interface type.

E. Assertions

Assertions are of two types: immediate assertions and concurrent assertions. An immediate assertion has a keyword (**assert**, **assume** or **cover**), a Boolean expression and pass/fail actions. An immediate assertion is a procedural statement, but its execution can be deferred till later in the time slot. The default fail action is to call **$error**, which will print useful information such as filename, line number, hierarchical instance name, and simulation time.

A concurrent assertion has a similar keyword, a property that is a clocked sequence of Boolean expressions, and pass/fail actions. The sequence may be encapsulated in a sequence declaration, which may have arguments. The assertion may be in a clocked procedure, in which case it can infer the clock, or it may stand alone, in which case it may use the default clock.

Because an assignment is also an expression, like in C, it is easy to capture data in one part of the sequence and compare it in a later part.

It is important to ensure that the values sampled in the assertions are stable, and not just glitches in the simulation time slot. The expressions are evaluated after the design code has been executed for that time slot, in the observe region, and the pass/fail actions are executed in the reactive region (see below section V.C).

Assertions may be included in interfaces as a convenient way of checking protocols. Alternatively they may be coded separately from the design and bound to specific module instances using the **bind** construct.

There are utilities to disable and enable assertions so that they can be turned off during a reset sequence, for example.

F. Functional Coverage

SystemVerilog coverage groups provide coverage of nets, variables and expressions as well as cross coverage between them. They have bins for sets of values, value sequences or combinations of values. They have various ways of specifying when the values are sampled. The user can provide weights and numerical goals to give an overall measurement of coverage.

As well as coverage groups, assertions with the keyword **cover** can be used to check that particular complex sequences have been verified.

V. IMPLEMENTATION

A. Parsing

SystemVerilog has a macro processor, **include** files, and compiler directives like C. Unlike C, Verilog was designed to be parsed again for each run, and the source files concatenated. Unlike VHDL, the modules could be parsed in any order, and library search was by source file.

SystemVerilog introduced packages and features to support separate compilation, so this may no longer apply to a particular tool.

B. Elaboration

For SystemVerilog (and VHDL) the elaboration phase is very important, because it creates the data structure that represents the design, including the name of each element. This data structure is extensively checked and meaningful error messages produced. A user can put a check in a generate block and call $error if there is a problem with the static structure.

With the VPI to access the data structure, third party tools can be attached to improve the thoroughness of the checking. The simulation run time code can be generated from the elaborated data structure, allowing global optimizations across module instance boundaries.

With SystemC, the data structure must be built in a single pass, whereas an HDL elaborator can use several passes. Because the SystemC elaboration constructs are not distinct from the run time constructs, it is more difficult to have different approaches to compiling and debugging them.

C. Initialization and scheduling

The initial state of a net defaults to X, which represents an unknown value. This gives a consistent initial state for gate level simulation, and guarantees that a reset sequence which clears all X values will reset the real circuit, because the X propagation is pessimistic. Logic variables also default to X, but they can have initializers in the declaration. Because procedural code is often not pessimistic, some people do not recommend relying on X values.

The **initial** and **always** procedures are started in any order.

The event scheduler has a stratified event queue in each time slot. First the design events are executed in the active region, with the non-blocking assignments delayed until other events have been executed. Then the observed region samples the values for assertion expressions. Then the assertion actions are executed in the reactive region. If this changes any design variables, the whole cycle is repeated.

There are also special regions just before and after time is advanced, called postponed and preponed. The postponed region is used for printing values of variables, for example.

D. Simulation performance

The performance of a gate-level simulator is optimized by having a compact bipartite graph structure built from all the modules connected together ("flattened hierarchy"), with the hierarchy and naming information kept separate to optimize data access locality and cache hit rate. Table based lookup for gate functions and event queue can reduce the instruction count per event. Verilog allows user-defined tables to provide very fast models of simple ASIC cells.

For register transfer simulation, merging small processes can improve performance. This is more difficult with a language that requires a delay (even if only a delta) for inter-process communication, such as VHDL or SystemC. Verilog defaults to zero delay for inter-process communication. For optimization, the communication data structure may be removed, at the expense of debug and API visibility. If the code generation is deferred till after elaboration, process merging can even cross module boundaries.

Switching between processes is much slower if a stack has to be saved and restored. Verilog does not have any dynamic data, so no persistent model stack is needed apart from task calls. SystemVerilog has dynamic data in a minority of processes because there are fewer of them in the testbench than in the design.

Defining the commonly used data types in the language rather than in a library allows the simulator to have optimized storage without the user being aware of it. For example, a 32 element packed logic vector can be stored with one bit of each element in one word and one bit in another, rather than as the two bits of each element together to code 0/1/X/Z. This allows much faster logic and arithmetic operations but does not sacrifice any of the indexing or slicing features of the language. However the code complexity required to make these kind of optimizations is considerable.

E. Synthesis

The extra keywords to assist synthesis and the use of interfaces to model buses make it easier to code the RTL.

The fact that the RTL input and gate level netlist output are in the same language makes it easier to re-use the testbench, especially if there are "don't touch" modules in the RTL.

F. Formal Verification

There are two kinds of formal verification in common use: equivalence checking and property checking. Equivalence checking can be used between RTL and gate netlist to verify that the synthesis flow is correct, or between gate netlists to verify that modifications such as scan chain insertion are correct.

Property checking takes the **assume** properties to be true and tries to prove that each **assert** property is true, or to generate a counterexample waveform. Note that in simulation both kinds of properties are treated similarly.

Property checking is currently limited to fragments of a design, making the assumptions quite demanding, but it is an active research area.

VI. Concluding Remarks

Domain-specific languages capture models concisely and intuitively. When a number of domains are combined, this leads to multiple ways of expressing the same concept. It is important for the user to know which constructs fit which domain, so as to obtain the best performance.

Because of the global optimization, SystemVerilog simulators can perform better than SystemC simulators at the same level of modeling abstraction.

For linking C models of virtual platform components, it is possible to use SystemVerilog, but SystemC is much more natural.

For smaller designs which do not require complex testbenches, VHDL is adequate, and is still the most popular language for design and verification of FPGAs [10].

SystemVerilog is a big language, because it includes all six domains of an HDVL, and provides a rich set of features in each one. It has been through various revisions, and these have increased the number of features. As a result, SystemVerilog offers a very productive environment for design and verification, and its use is increasing [10].

Unfortunately, there are many tools which do not implement the whole language. Coding style guidelines are useful to avoid language support problems in particular tool chains.

Parsers and elaborators for SystemVerilog (and for VHDL) are now commercially available. This has reduced the barrier to creating new tools.

Much of the time of both design and verification engineers is spent debugging their code. SystemC can be challenging because it is only a domain-specific library, and so user bugs can manifest themselves in the library, which makes it difficult for novices. Domain-specific languages can be much more user-friendly in this respect.

It is possible to design ASICs using VHDL for the RTL, e for the testbench, and Verilog for the ASIC libraries, but the differences in syntax and semantics between the languages make it even harder to understand and debug the interaction between the design and testbench.

The effort to learn the subset and style of SystemVerilog needed for a particular job is substantial, but is less than the effort to switch between multiple languages that are more restricted. SystemVerilog includes enough features for both design and verification of today's digital designs.

REFERENCES

[1] IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language, IEEE Std 1364-1995, IEEE Computer Society, 1995.

[2] IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[3] L. Wall, "Perl, the first postmodern computer language", http://www.perl.com/pub/1999/03/pm.html

[4] M. Mernick, J. Heering, A.M. Sloan,"When and how to develop domain-specific languages" *ACM Computing Surveys*, Vol. 37, No. 4, December 2005, pp. 316–344.

[5] M. Lipow "Number of faults per line of code", IEEE Trans. Software Engineering, Vol. No. 4, July 1982 pp. 437-439.

[6] P. L. Flake, P. R. Moorby, G. Musgravc, "HILO Mark 2 Hardware Description Language" *Proc. ACM/IEEE CHDL*, Kaiserslautern, Germany, 1981.

[7] D. E. Thomas, P. R. Moorby *The Verilog® Hardware Description Language,* Kluwer Academic Publishers, 1991

[8] IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987, IEEE Computer Society, 1988.

[9] P.L. Flake and S.J. Davidmann "Superlog, a unified design language for system-on-chip," *Proc. ASP-DAC 2000*, pp 583-586.

[10] H. Foster "Wilson Research Group Functional Verification Study 2012", http://blogs.mentor.com/verificationhorizons/blog/author/hfoster/

[11] S. Sutherland, S. Davidmann, P. Flake, *SystemVerilog for Design, Second Edition*, Springer, 2006.

[12] IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2011, IEEE Computer Society, 2011.

[13] OpenVera® Language Reference Manual: Testbench ftp://ftp.synopsys.com/pub/OpenVera_LRM_Testbench_Version_1.4.3/OpenveraTBlrm143.pdf

[14] Sasan Iman and Sunita Joshi, *The e Hardware Verification Language,* Springer, 2004

[15] OpenVera® Language Reference Manual: Assertions ftp://ftp.synopsys.com/pub/OpenVera_LRM/ova_lrm141.pdf

[16] D. I. Rich, "The evolution of SystemVerilog", IEEE Design and Test of Computers, Vol. 20 No. 4, July-August 2003.

[17] S. Sutherland, D. Mills "Synthesizing SystemVerilog: busting the myth that SystemVerilog is only for verification", SNUG Silicon Valley 2013. http://www.sutherland-hdl.com/papers/2013-SNUG-SV_Synthesizable-SystemVerilog_paper.pdf