A Tale of Two Languages - SystemVerilog and SystemC

By David C Black, Senior MTS and Trainer at Doulos America Inc
Austin, Texas

Abstract

There is a lot of confusion on why there are two EDA languages with the word "System" in their title. Many believe these are in competition with each other. Some believe duplication means one or the other is redundant, and thus should be removed. Yet another group resents the possibility of competition or the possibility they might have to learn more complicated syntax. This paper examines these issues and puts forth conclusions on the need for both languages.

Introduction

This paper hopes to dispel myths of conflict between two of the dominant EDA languages. The goal is to educate the reader to understand the need for two languages and see strengths in both. The author hopes also to stimulate the industry to produce more tools that help with the development of models and promote a methodology that encourages system-level modeling.

In particular, we examine, contrast and compare the two EDA languages SystemVerilog and SystemC. The terms SystemVerilog and SystemC have often confused folks because they both have the word "system" in their titles. In fact at various points in time, both have been considered for use as "system-level" modeling languages, but as we shall see, they are really very different in their application.

To get the proper perspectives, we will first describe each "language" separately and then compare them directly.

SystemVerilog History

SystemVerilog comes from a long history of several hardware description and verification languages including Verilog, Vera, Superlog, PSL and even draws ideas from VHDL and SystemC. Fundamentally SystemVerilog is an extension of a solid RTL hardware design language (I.e. Verilog) adding features that allow for robust verification with a relatively concise syntax. Some would say that in pursuing a single language to "do it all", the SystemVerilog committee in fact produced a single standard that encompasses three different languages: a robust RTL design language, a hardware verification language, and an assertion language (SVA).

1984	Verilog debuts
1987	Commercial RTL Synthesis - Synopsys Design Compiler
1987	IEEE 1076-1987 - VHDL standard released
1990	Verilog-XL acquired by Cadence Design Systems Inc.
1996	IEEE 1364-1995 - Verilog standard released
1998	VERA appears and acquired by Synopsys
2000	Accellera formed from OVI & VHDL International
2001	IEEE 1364-2001 - Verilog improvements
2002	Synopsys acquires Superlog by acquiring Co-Design
2005	IEEE 1800-2005 SystemVerilog standard released
2009	IEEE 1800-2009 SystemVerilog updated
2012	IEEE 1800-2012 expected

Table 1: SystemVerilog History

SystemVerilog Features, Strengths & Weaknesses

As a Hardware Design and Verification Language (HDVL), SystemVerilog has many strongpoints. First as a design implementation language it directly supports RTL synthesis with specialized constructs such as **always_comb**, **always_latch**, and **always_ff**. These specify user intent and allow simulators to properly model the behavior of RTL to reduce the chances of simulation mismatches between the RTL and final gate-level netlist. There are also keywords such as **unique** and **priority** to allow simulations to match synthesis directives and ensure correct design.

Consider figure 1 where the always_comb statement demands that simulators verify the contained logic is combinatorial and the separate if clauses should be checked and implemented as parallel cases.

```
module mux (
  input a, b, c,
  input [1:0] sel,
  output logic f);
  // combinational
  always_comb
    // parallel case for synthesis
    unique if ( sel == 2'b10 )
      f = a;
    else if ( sel == 2'b01 )
      f = b;
    else
      f = c;
endmodule
```

Figure 1: SystemVerilog RTL feature

SystemVerilog also added a handful of features to ease coding by drawing on the familiar syntax of C. For instance in figure 2 there are two value data types (**int** and **byte**) rather than the traditional four value logic. SystemVerilog allows for optional initialization and the ability to declare variables inside loops and code blocks without labeling. Also loop controls, **continue** and **return**, were added freeing us from the awkward use of the **disable** statement for the same purpose.

```
// C-style for-loop
for (int i = 0; i < 8; i++)
begin
  // initialization of 2-state logic
  byte count = 8;
  if (input[i]) begin
    // pre-decrement operator
    --count;
    // C-style loop control
    continue;
  end
  if (count == 0)
    return i; //< C-style return
end
```

Figure 2: SystemVerilog C-style conveniences

While the RTL improvements are exciting, the real power in SystemVerilog comes from the great wealth of features added for verification. Concepts added include Packages, User defined types, Interfaces, Assertions, Object Oriented Programming (OOP), functional coverage, constrained randomization, dynamic processes, dynamic arrays, queues, associative arrays, mailboxes and semaphores. Figure 3 illustrates some of these features.

```
package pkg;//< VHDL style package
 // user-defined type
 typedef int T[8];
 typedef enum bit[1:0] {
   RESET, IDLE, READ, WRITE } E;
 // C-style struct with extensions
 typedef struct signed packed {
   logic [3:0] a;
   logic [7:0] b;
 ) S;
 //
 function void puts(
   const ref S s
 );
   $display("%p",s);
 endfunction :puts
endpackage

module ...
  import pkg::*;
  E e = IDLE;
  // initialize struct
  S s = '{4'd2, 8'bx};
  T t; //< create array of 8 int's
  ...
endmodule
```

Figure 3: SystemVerilog packages and data types

First, interfaces provide a common rallying point between RTL and verification. From an RTL point of view, interfaces greatly simplify interconnection of buses. For verification, interfaces provide clean access point to allow greater independence and reuse of verification IP. Clocking blocks provide additional isolation from the effects of gate-level timing. Figure 4 is an example of a SystemVerilog interface declaration. This example of a trivial bus has a clock, address and data. The RTL has access to all the signals via RTL specific modports (views), but with directionality corrected depending on whether the RTL represents master or slave processing. A testbench modport (tb) is provided for read-only access by a verification environment.

```
interface Bus_if;
  logic clock;
  logic [15:0] addr;
  logic [7:0] data;
  clocking cb @(posedge clock);
    output #1 addr;
    input  #1step data;
  endclocking
  modport rtl_master(
    input  clock,
    output addr,
    inout  data);
  modport rtl_slave(
    input clock, addr,
    inout data);
  modport tb(clocking cb);
endinterface
module CPU (Bus_if.master bus);
module Dut (Bus_if.slave  bus); …

Bus_if inst ();

Dut dut1 ( .iport(inst.rtl) );
```

Figure 4: SystemVerilog Interfaces

Assertions aid the creation of automated checking (esp. protocols) as illustrated by the syntax in figure 5. This example checks to see that a request is following by grant within 1 to 5 clocks and rant is held high for 2 to 4 clocks before returning low.

```
interface pins;
  logic clock;
  logic req, grant;
  assert property (
    @(posedge clock)
    req |-> #[1:5] grant
  );
  assert property (
    @(posedge clock)
    $rose(grant)
         |-> grant [*2:4] #1 !grant
  );
endinterface
```

Figure 5: SystemVerilog assertions

Functional coverage assures designers that important features have been exercised for which sample syntax in figure 6 is shown. This illustrates gathering data to ensure all opcodes, modes and specific data ranges are managed.

```
class instruction;
  bit [2:0] m_opcode;
  bit [1:0] m_mode;
  shortint unsigned m_data;
  covergroup cg @(posedge clk);
    coverpoint m_opcode;
    coverpoint m_mode;
    coverpoint m_data {
      bins tiny    [8] = {[0:7]  };
      bins moderate[8] = {[8:255]};
      bins huge    [8] = {[256:$]};
    }
  endgroup
  ...
endclass: instruction
```

Figure 6: SystemVerilog Functional Coverage

Constrained randomization ensures that a variety of valid stimuli are used to check the design. Figure 7 illustrates some of the constrained random syntax. In this we randomize three fields, and ensure that **m_mode** is 3 when bit 2 of **m_opcode** is zero. We also force the mode to 2 when **m_data** is less than 256.

```
class instruction;
  rand bit [2:0] m_opcode;
  rand bit [1:0] m_mode;
  rand shortint unsigned m_data;
  ...
  constraint assert_mode {
    m_opcode[2]==0 -> m_mode==2'b11;
  }
  constraint assert_data {
    m_mode == 2'b10 -> m_data < 256;
  }
  ...
endclass: instruction
```

Figure 7: SystemVerilog Constrained Random

Inheritance and other OOP concepts help to make verification IP reusable and flexible with a minimum of effort. Figure 8 shows inheritance declaration.

```
class jump_instruction
            extends instruction;
  constraint jump {
    m_opcode[2:1] == 2'b11;
  }
endclass
```

Figure 8: SystemVerilog Class Inheritance

All of this makes for an excellent language to perform both RTL design implementation and high level verification. In other words, you can design an entire SoC (System on a Chip) using one language.

On the downside is the somewhat unavoidable problem that the SystemVerilog language is huge. With the upcoming 201x standard there will be 245 keywords, and the standards reference manual weighs in at over 1,200 pages! Never-the-less, the engineering community has been rushing to embrace SystemVerilog as its hardware verification language of choice. Consider that UVM, the Universal Verification Methodology provides an open-source implementation in SystemVerilog. EDA vendors eagerly rush out new versions of all types tools to support the demand.

SystemC History

SystemC was developed originally as a joint effort with the University of California at Irvine and Synopsys. Other companies joined such as Frontier Design, Infineon, and IMEC. Eventually, work was migrated to the Open SystemC Initiative (OSCI). Their work resulted in an open-source "proof-of-concept" implementation that was made available for free download from systemc.org. More recently, OSCI merged with Accellera to become the Accellera Systems Initiative (ASI), and current versions may be downloaded from accellera.org. Here is more detailed history.

1999	SystemC v0.9
2000	OSCI formed
2002	SystemC v2.0.1
2005	SystemC v2.1
2005	IEEE 1666-2005 released
2006	SystemC v2.2
2011	ASI merges Accellera & OSCI
2011	IEEE 1666-2011 released w/ TLM 2.0
2012	SystemC v2.3

Table 4: SystemC History

This brief history does not cover the fact that several side groups are actively working on elements that may enter the standard at a future date. Active committees include the Configuration, Control and Inspection (CCI) working group, the SystemC Analog/Mixed Signal (AMS) working group, the SystemC Synthesis working group, and the SystemC Verification working group.

SystemC Features, Strengths and Weaknesses

As a System-Level design language, SystemC has many strengths and weaknesses. First, as an architectural design language, its C++ nature lends itself well to the natural incorporation of algorithms and other software that is already in C or C++. With relatively simple additions, algorithms may be wrapped inside SystemC containers to resemble hardware blocks with communication abstractions ranging from transaction-level to pin-level and incorporate timing annotations. This may be used to obtain rough estimates of performance that guide the selection of hardware and software architectures early in the design process.

The availability of an open-source implementation has been crucial to SystemC's success. This has made SystemC more accessible to the general software community (esp. academic), which in turn provides ideas and software components that help the SystemC community. This has been met with mixed feelings by the EDA vendor's community since their success depends on their ability to sell tools at premium prices to support their large R&D investments. The prices would preclude the software and academic communities from participation. On the other hand, there are plenty of opportunities for good SystemC design tools to augment the modeling design process.

Many early SystemC efforts seemed to focus on an RTL representation. An unfortunate problem for SystemC is that too many see it as an RTL description language. Instead, SystemC descriptions are more successful at focusing on everything above RTL. Although, SystemC can do RTL, it does not do so eloquently nor efficiently. More importantly, SystemC is not intended for RTL design as a primary focus. Rather the RTL features of SystemC are intended to allow interworking between high-level architectural/behavioral models and lower-level RTL. RTL can also serve as glue or be used in situations where RTL predates the modeling effort. Figure 9 is an example of SystemC RTL that implements a simple registered adder.

```
SC_METHOD(accumulate_method);
sensitive << clock.pos() << reset;
…
void accumulate_method() {
  if (reset) acc->write(0);
  else       acc->write(a + b);
}
```

Figure 9: SystemC RTL register

Behavioral modeling at higher levels of abstraction is where SystemC really shines. By not simulating the details of wires, pins and gates, SystemC realizes performance that can even be faster than realtime. We can also take advantage of the large number of C++ libraries available through such entities as Boost.org. For instance, it would be easy to model polygon processing using the boost geometry classes as the hypothetical code following suggests. The highlighted code is SystemC and Boost elements.

```
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>
BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)
typedef sc_fixed<16,8> data_t;
data_t fence_post_x() {…}
data_t fence_post_y() {…}
…
void polygon_thread() {
  vector<model::point> fence_posts;
  while (fence_post_available())
     fence_posts.push_back(model::point(fence_post_x(), fence_post_y()));
  sc_assert(fence_posts.size() > 2); // make sure it's legal
  model::polygon<model::d2::point_xy<data_t> > enclosure;
  append(enclosure, fence_posts);
  data_t fenced_area = area(enclosure);
  wait(AREA_CALCULATION_TIME_ESTIMATE);
  sc_assert(fenced_area > 0);
  area_display->write(fenced_area); //< put on operator display
  Get_beacon_data(); //< from TLM model
  boost::tuple<data_t, data_t> prisoner_locn
                              = boost::make_tuple(beacon_x(), beacon_y());
  wait(PROXIMITY_DETECTION_TIME_ESTIMATE);
  if (within(prisoner_locn, enclosure)) {
    SC_REPORT_INFO("","Prisoner OK");
  } else {
    SC_REPORT_WARNING("","Prisoner escaped!");
  }
}
```

Figure 10: SystemC Behavior using Boost Geometry Library

Boost contains many classes for modeling things such as matrix arithmetic, higher-level mathematics, state machines and data processing.

Because of SystemC's ability to abstract communication, SystemC can easily leverage models written in C++ (or even Java or Python) by the System Architect to ensure fidelity to the original concepts and specifications. With the standardization of the transaction level modeling API, known as TLM 2.0, SystemC facilitates the rapid creation of loosely-timed (LT) models that simulate fast enough to enable virtual platforms for software development.

Systems architects use the approximately-timed coding style of TLM 2.0 to provide timing details for performance analysis. This analysis is typically used to verify the design has adequate performance.

TLM 2.0 has enabled vendors to provide off-the-shelf IP models that are essentially plug-n-play. By enabling rapid modeling of virtual platforms, this in turn allows software to begin development early in

the project schedule before RTL has even materialized.

Because SystemC models can execute extremely quickly, it is possible to simulate an entire system. This is known as an Electronic System-Level (ESL) model. When a processor is involved and an Instruction Set Simulator (ISS) is provided, the model is called a Virtual Platform. A Virtual Platform should run fast enough that software development is reasonable.

The best scenario for use of SystemC is for the project team to adopt the principle that the ESL model or Virtual Platform is golden. Because co-simulation with SystemVerilog is available, updates to the RTL design are to be routinely compared to the ESL model. As necessary, the models are adjusted to ensure their behaviors match. The software team can rest assured their efforts will integrate with the final hardware because design fidelity is checked. This is a huge win for everyone.

Because the hardware is modeled as software, this also makes for excellent debug facilities that are not present on real hardware. This can even be used post-silicon when debugging extremely difficult situations as the scenario can be reproduced with a level of control and observability that is simply not possible in physical hardware.

Side by side comparisons

The following table compares syntactic features of modern SystemVerilog to SystemC. The point of this table is not to say that one syntactic feature is better than another, but rather to point out is just a matter of syntax. For the most part, every feature available in SystemVerilog can be reproduced in SystemC and vice versa. Similar comparisons might be made with VHDL, and the point is that engineers simply get used to a particular syntax. As previously noted, SystemC does not directly support SVA or coverage. These examples utilize the latest SystemVerilog, SystemC and C++ standards. Language keywords and features are highlighted in **bold**.

Feature	SystemVerilog	SystemC

Packaging	**package** my_stuff;
	endpackage : my_stuff
	import my_stuff::*;
		namespace my_stuff {
		}
		using namespace my_stuff;

Behavioral Process	**initial begin** :BLOCK
	  STATEMENTS…
	end
		SC_THREAD(BLOCK);
		void BLOCK(**void**) {
		  STATEMENTS…
		}

RTL Process	**always_ff** @(**posedge** clk)
	begin :REGS
	  q **<=** d;
	end
		SC_METHOD(REGS);
		sensitive << clk.**pos**();
		…
		void REGS(**void**) {
		  q->**write**(d);
		}

Module	**module** Design
	(**input** logic [7:0] d
	, **output** logic [7:0] q
	);
	  …
	endmodule: Design
		SC_MODULE(Design) {
		  sc_in <sc_lv<8> > d;
		  sc_out<sc_lv<8> > q;
		  …
		};

Data	**logic** [3:0] l;
	int i;
	bit b;
	string txt;
	typdef struct { **int** a, b; } S;
	S s = '{1,2};
	time t;
		sc_lv<4> l;
		int i;
		bool b;
		string txt;
		struct S { **int** a, b; };
		S s**{**1,2**}**;//C++11
		sc_time t;

Events	**event** e1, e2; **->**e1; ->> #5**ns** e2; **@**(e1 or e2);	**sc_event** e1, e2; e1.**notify**(); e2.**notify**(2, **SC_NS**); **wait**(e1 **|** e2);

Feature	SystemVerilog	SystemC

	`#5ns;`
	`wait(sig == 0);`
		`wait(5,SC_NS);`
		`while (not (sig == 0))`
		`  wait(sig.default_event);`

Classes (OOP)	`class C extends B;`
	`  int m_i;`
	`  extern function new(int i);`
	`  function int get;`
	`  return m_i;`
	`  endfunction`
	`  function void inc;`
	`  m_i += val;`
	`  endfunction`
	`  static task delay;`
	`  @(sync);`
	`  endtask`
	`  local static event sync;`
	`endclass: C`
	
	`C::delay();`
	`C obj1 = new(101);`
	`obj1.inc();`
	`C obj2;`
	`obj2 = new(102);`
	`$display(obj2.get());`
		`struct C : B {`
		`  int m_i;`
		`  C(int i);`
		`  int get(void) const {`
		`  return m_i;`
		`  }`
		`  void inc(void) {`
		`  m_i += val;`
		`  }`
		`  static void delay(void) {`
		`  wait(sync);`
		`  }`
		`private:`
		`  static sc_event sync;`
		`};`
		`C::delay();`
		`C obj1{101};`
		`obj1.inc();`
		`C* obj2;`
		`obj2 = new C(102);`
		`cout << obj2->get() << endl;`

Conditional	`if (EXPR) STATEMENT`
	`else STATEMENT`
	`case (EXPR)`
	`  EXPR1: STATEMENT`
	
	`  default: STATEMENT`
	`endcase`
		`if (EXPR) STATEMENT`
		`else STATEMENT`
		`switch (EXPR) {`
		`  case CONSTANT: STATEMENT`
		`  break;`
		`  default: STATEMENT`
		`}`

Loops	`while(EXPR) STATEMENT`
	`do STATEMENT while (EXPR);`
	`for(int i=0;i!=max;++i) STMT`
	`forever STMT`
	`foreach(CONTAINER[i]) STMT`
		`while(EXPR) STATEMENT`
		`do { STATEMENTS } while (EXPR);`
		`for(int i=0; i!=max; ++i) STMT`
		`for(;;) STMT`
		`for(auto i:CONTAINER) STMT`

Dynamic Processes	`fork`
	`  begin STATEMENTS… end`
	`  begin STATEMENTS… end`
	`join`
	
	`process h;`
	`fork`
	`  begin`
	`  h = process::self();`
	`  STATEMENTS…`
	`  end`
	`join_none`
	
	`wait(h.status !=`
	`  process::FINISHED);`
		`FORK`
		`  sc_spawn([&](){STATEMENTS…}),`
		`  sc_spawn([&](){STATEMENTS…})`
		`JOIN`
		
		`auto h = sc_spawn([&](){`
		`  STATEMENTS…`
		`});`
		
		`wait(h.terminated_event());`

Final	`final begin`
	`  …`
	`end`
		`void end_of_simulation(void) {`
		`  …`
		`}`

Containers	`T1 fixedArray[N];`
	`T1 dynamicArray[];`
	`T1 associativeArray[T2];`
		`std::array<T1,N> fixedArray;`
		`std::vector<T1>  dynamicArray;`
		`std::map<T2,T2>  associativeArray;`

Feature	SystemVerilog	SystemC
	`T1 queue[$];`	`std::deque<T1>  queue;`

Channels	`var  logic [1:0] sig;`
	`sig <= a + b;`
	
	`wire logic [7:0] w;`
	`assign w = a + b;`
		`sc_signal<sc_int<2>> sig;`
		`sig.write(a + b);`
		
		`sc_signal_rv<char> w;`
		`sc_spawn([&](){for(;;){`
		`  wait(a|b);`
		`  w.write(a + b);`
		`}});`

Table 5: Side by side syntax

There are features that one side or the other do not currently support natively; although, there is nothing fundamentally that cannot be solved. For instance, SystemC does not support SystemVerilog Assertions (SVA); although, one commercial company has provided a solution, and at least one instance of a internal solution have been demonstrated. (Also PSL is designed for SystemC). The issue with SVA is primarily that there has been no overwhelming need for temporal assertions for high level modeling efforts. SVA as implemented in SystemVerilog is focussed on verification of RTL issues such as pin-level bus protocols.

SystemC also is not able to efficiently simulate RTL because some of the tricks employed by SystemVerilog simulators are not germane to the standard C/C++ compilers used with SystemC. For instance, identifying all clocked code fragments and combining them into a single context switch is simply something no C/C++ compiler could have any understanding of.

SystemVerilog on the other hand does not trivially integrate into the world of C/C++ open-source. For instance, consider the pain required if one desired to use some of the algorithms present in the open-source BOOST libraries. Certainly, Direct Programming Interface feature has made it easier; however, it frequently requires data conversions and the semantics are generally pass-by-value. Consider that TLM 2.0 payload extensions are almost impossible. But then again, SystemVerilog was designed with RTL in mind; whereas, SystemC is all about melding in with the C/C++ coding world.

Applying Languages to Designs

So how should a project make use of these two languages? Part of the answer comes when a project is broken down into a schedule, and its needs are examined. For modern designs, software dominates the schedule, which dictates a need to be able to start coding early. Furthermore, hardware becomes a bottleneck for verification. If the product team can agree to designate an ESL model as golden, then a high-performance version can be used to develop software. The verification team can begin to verify this model as the basis for their test environment, while the hardware team develops the RTL. Best of all, the final design can use of co-simulation to verify the RTL against the original ESL model.

Is one language better than the other? No, not really. Each has strengths and weaknesses, but more importantly, each has a particular domain for which it is suited.

Eliminating FUD

Fear, Uncertainty and Doubt (FUD) can be hindrances to progress. In an uncertain world, engineers and managers alike sometimes see new design methodologies and tool technologies as a threat. They see it as a threat because they've just spent the last N years of their life mastering the current technology (e.g. Verilog RTL), and now they may have to re-enter the race as newbies learning new skills (e.g. SystemVerilog, SystemC, UVM or TLM 2.0). It may also mean the tools they have grown fond of, customized and use daily may be changing. The fact that upper level management wants products out the door faster than ever, does not help the stress level. It's challenging enough to master the technologies that are going into the next products without adding the potential stress of new tools or languages. What is an engineer to do? It is interesting to consider that engineers are facilitators of change, but in this situation, they often resist the change to their own world.

A lot of questions to be answered to determine what you need. The following list has some food for thought. You may choose to adjust the recommendations. Additionally, each question has a weight relative to the project itself. For example, if software dominates the project schedule then

obviously recommendations relative to it should be weighted higher.

#	Question	SystemVerilog	SystemC
	Do you need to start the software earlier in the schedule?	0	10
	Does the architecture need bandwidth considerations analysis?	3	7
	Does software need to evaluate its impact on power or performance?	1	9
	Does the specification include FSM descriptions?	6	4
	Do you plan to purchase external RTL IP for incorporation?	5	5
	Does the design incorporate a lot of legacy RTL IP?	8	2
	Does verification desire to reuse code from other designs?	8*	2
	Are simulation mismatches between synthesis and RTL a concern?	7	3
	Is a new standard protocol involved?	8	2
	Is a new custom protocol involved?	7	3
	Are you starting from pre-existing Verilog RTL?	10	0
	Is this for architectural purposes?	3	7
	Is this for verification of the implementation?	7*	2
	Is this for implementation of RTL specifications?	10	0
	Does verification IP exist?	10*	0
	Does TLM 2.0 IP exist?	0	7
	Does an Instruction Set Simulator exist?	0	7
	Are there asynchronous design blocks?	8	2
	Are there multiple clocks?	8	0
	Does the design include major new algorithms or complex design content?	3	7

* UVM should be part of the solution

Table 6: Questions Influencing Choice of Language

Another factor challenging practicing engineers is the fear of not becoming proficient quickly enough in a job market that favors the employer. Some recent graduates are coming out of some schools with training in these technologies and employers are snapping them up. There are several ways to alleviate this including getting professional training, obtaining good reference materials, proper tools and prolific use of web resources and users groups/forums. Let's look at some of these approaches to proficiency.

It is amazing how many engineers underestimate the learning challenges and choose to attempt independent learning. By choosing professional training, engineers can cut months off the learning curve. Of course practice makes perfect, and simply attending a training course won't make you proficient, but it does provide a better foundation.

Reference materials take several forms. The first should be the actual standards documents themselves. In some cases they can be obtained at no cost, but other require an investment. In many cases, employers have purchased copies available in a library or on-line electronically for use by their employees. Even though tedious, every engineer should have these available to consult. For learning a technology, it is probably best to also obtain a book on the subject matter. Some of these may be obtained from training classes and others purchased. If not in a library, many employers will reimburse employees for at least one or two books if requested. Older versions and second hand copies may also be

valuable as foundational material. Newer versions of standards are frequently presented at conferences accompanied by papers that announce the changes and improvements.

Obtaining the right tools can also make a difference it proficiency; although, tool vendors understandably overstate the tool's value versus the need for foundational knowledge of the technologies. Do not get confused between the language and the tools. It should be noted that evolving standards means that vendors may not support all the latest features yet, or perhaps they have interpreted the standard differently. When in doubt consult the standards document. Rolling your own tools is probably not the best use of your time. The real key to proficiency is a combination of education and tools.

Finally, it should be obvious that the web provides a wealth of information. It must be noted that information is only as good as its source and a lot of misinformation is out there. When you read a reply, consider its source. Does the answer match the definitions as stated in the relevant standards document? Is the source an expert or simply an amateur stating an opinion or vendor pushing a tool? Do others agree with the opinion? Is the answer simply a guideline or does it represent a particular methodology? Can you find other experts to back up the opinion?

Summary: Conclusions, Possible future directions, Recommendations

In summary, both SystemVerilog and SystemC are needed for modern designs and work in a complementary manner. They should be applied to each area of the project as appropriate. SystemC makes sense in creating Virtual Platforms to enable early architectural analysis, software development, high-level verification and as a reference model. SystemVerilog provides a solid framework for properly verifying RTL designs, enabling the use of structured verification methodologies such as UVM. SystemVerilog is also enables design engineers to reliably communicate their RTL intent and have better control over the output of synthesis.

It is clear that evolution of the standards will continue to occur as it always has. Designs will get more complex and raising the abstraction levels is a means to keeping complexity manageable.

References

[1] "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009, 2009.

[2] "IEEE Standard for Standard SystemC® Language Reference Manual," IEEE Std 1666-2011, 2012