Keeping Up with Chip — the Proposed SystemVerilog 2012 Standard Makes Verifying Ever-increasing Design Complexity More Efficient

Stuart Sutherland
SystemVerilog Trainer and Consultant
Sutherland HDL, Inc.
Portland, Oregon
stuart@sutherland-hdl.com

Tom Fitzpatrick
Verification Evangelist
Mentor Graphics, Corp.
Waltham, Massachussettes
tom_fitzpatrick@mentor.com

***Abstract*—The complexity and size of our hardware designs and verification code continues to increase at a rapid rate, and the SystemVerilog Design and Verification language is keeping pace. As soon as the SystemVerilog standards organization completed the SystemVerilog-2009 standard, they immediately began work on the next generation of the language, looking for ways to more efficiently and more effectively model and verify increasingly complex designs. In just three years — near record time for a complex IEEE standard — a SystemVerilog-2012 standard has been defined and is currently in the balloting process. New and powerful language features include multiple class inheritance, user-defined net types, additional assertion capabilities, and much more. You won't need to wait long to use SystemVerilog-2012; EDA companies have already begun adding SystemVerilog-2012 features to their software tools.**

This paper presents the many new features in the proposed SystemVerilog standard, and discusses how key new language capabilities can enable more efficiently verifying designs that are continually increasing in size and complexity. The paper also discusses how new features such as multiple inheritance can benefit the UVM verification methodology. SystemVerilog is "keeping up with Chip" — your chip.

***Keywords*—Verilog, SystemVerilog, hardware design, hardware verification, UVM**

I. INTRODUCTION

SystemVerilog has become a primary language for the design and verification of digital hardware designs. SystemVerilog was first introduced in 2002 as an Accellera standard that specified a large number of extensions to the Verilog-2001 Hardware Description Language[1]. These extensions added both new modeling and verification capabilities to Verilog. In 2005, the IEEE standardized these extensions as the 1800-2005 SystemVerilog standard[2]. A brief description of these extensions can be found in [3] and [4]. The base Verilog language remained a separate standard, IEEE 1364-2005. The 1800-2005 SystemVerilog standard only specified enhancements to the base Verilog language.

In 2009, the IEEE approved the 1800-2009 SystemVerilog standard[5]. SystemVerilog-2009 merged the Verilog HDL standard into the SystemVerilog standard, and officially ended the Verilog standard. SystemVerilog-2009 standard also added a number of additional features to the language (see [6] and [7]).

2009 was not the end of the evolution of SystemVerilog. Design size and complexity continues to rapidly evolve. A design and verification language must evolve to keep pace with designs. As soon as the IEEE SystemVerilog standards committee completed SystemVerilog-2009, work began on defining the next generation of SystemVerilog, currently referred to as IEEE P1800-2012 proposed SystemVerilog-2012[8]. Wish lists of new language features were developed, and from those a "top 10" list of new features was created for categories such as design modeling and testbench modeling.

Furthermore, ambiguities in the SystemVerilog standard, and occasional typographical errors, were identified as Electronic Design Automation (EDA) companies implemented SystemVerilog-2009 in various types of software tools. Along with specifying new language features for the next generation of SystemVerilog, the IEEE 1800 standards committee made a number of clarifications and minor corrections to the standard.

The work on specifying new features and clarification for SystemVerilog-2012 was completed in December 2011. A pre-ballot draft of the new standard was prepared and reviewed during the month of January 2012. At the time this paper was written, a ballot draft was in the process of being prepared, with the IEEE balloting process set to begin in February 2012 and close in March 2012.

The data base used to track changes to the

SystemVerilog standard is called “Mantis.” The Mantis data base lists 162 changes for the proposed SystemVerilog-2012 standard. Of these 162 changes:

- 31 are new features that have been added to SystemVerilog.
- 60 are clarifications of how existing features in the standard should behave.
- 71 are minor corrections to fonts, punctuation, etc. (plus a number of minor editorial corrections, such as fixing a font, that were not recorded in the data base).

This focus of this paper is on the 31 new language features, and how those features can help make writing complex verification testbenches simpler or more efficient.

II. NEW FEATURES IN SYSTEMVERILOG-2012

A. *OOP enhancements*

Three of the new features in the proposed SystemVerilog 2012 standard affect Object Oriented Programming (OOP). One is a convenience enhancement that does not add new functionality. Another adds important functionality for helping OOP code avoid race conditions with procedural code. The third enhancement is significant — a form of multiple inheritance.

1. *Typed new() constructors (Mantis 3001)*

Previous versions of the SystemVerilog standard required that the type of an object to be constructed must match the type of handle variable of that object’s `new()` constructor. Once constructed, a child handle can then be assigned to a handle of its parent type. To construct an object and assign the handle to a parent type requires at least three lines of code. For example:

```
class base_trans; ... endclass

class reset_trans extends base_trans;... endclass

base_trans  t_base;
reset_trans t_reset t_reset = new;
t_base = t_reset;
```

The typed `new()` constructor enhancement adds a class scope immediately before the `new` keyword, specifying the constructed object’s type independently of the assignment target. This reduces the three lines of code above to a single line:

```
base_trans  t_base = reset_trans::new;
```

This new feature in SystemVerilog is a convenience enhancement. It does not add new functionality, but can help reduce the lines of code and make code more self-documenting.

2. *Nonblocking assignments to class properties (Mantis 2112)*

Previous versions of SystemVerilog did not allow nonblocking assignments to class properties. The proposed SystemVerilog-2012 standard removes this restriction.

```
class base_trans;
  int data;
  bit resetN;
endclass

initial begin
  resetN <= 0; // assert reset in NBA region
  ...
end
```

While nonblocking assignments are primarily a hardware modeling construct used in RTL models, they are also useful in verification code. Nonblocking assignments allow verification engineers a way to utilize SystemVerilog’s internal event scheduling regions to control the order in which concurrent processes are evaluated. When and why nonblocking assignments should be used is beyond the scope of this paper, but it is important. This enhancement enables writing better verification code.

3. *Multiple inheritance (Mantis 1356)*

This new feature is one of the most significant efficiency enhancements in the proposed SystemVerilog-2012 standard. Multiple inheritance allows a child class to inherit properties and methods from more than one parent class. The proposed SystemVerilog-2012 uses Java-like *interface classes* to do multiple inheritance.

In brief, a SystemVerilog interface class can define:

- Parameter constants
- User-defined types (typedefs)
- Pure virtual method prototypes

A regular class can then *implement* one or more interface classes. The full syntax, semantic rules and mechanics of interface classes is beyond the scope of this paper. A simple usage of interface classes and multiple inheritance is illustrated in the following example (bold text shows the important new features):

```
interface class Put;
  pure virtual function void put(int a);
endclass

interface class Get;
  pure virtual function int get();
endclass

//inherit method prototypes from multiple parents
class Fifo implements Put, Get;
  ... // implementations of inherited methods
endclass
```

Section III of this paper discusses how multiple inheritance might be used in a UVM testbench to help make verification more efficient.

B. *Constrained randomization enhancements*

Two new features in the proposed SystemVerilog-2012 standard provide a means to more efficiently model constraints for random value generations. Both are major enhancements.

1. *Soft constraints (Mantis 2987)*

All constraints in previous versions of the SystemVerilog standard are referred to as *hard constraints*. All hard constraints must be met, or an error results. This can be problematic and inefficient in complex verification code. For example, if a transaction class has default constraints specified, but a specific test requires and specifies a different constraint, an error can occur if the specific constraint conflicts with the built-in constraint. The programmer developing the special test must be aware of the potential conflict, and write extra code to first disable the built-in constraint — and then remember to re-enable the constraint after the test is complete.

The proposed SystemVerilog-2012 standard adds an important new feature — *soft constraints*. A soft constraint is ignored if it conflicts with another constraint. This allows for a more efficient coding style. A class can define default soft constraints that are used most of the time. A specific test can specify a different constraint, which will override — instead of conflict with — the default constraint. For example:

```
class Packet;
  rand int size;
    constraint dflt {soft size inside {32,1024};}
endclass

Packet p = new();
p.randomize with {size == 1512;}
```

In this example, the `randomize with()` constraint conflicts with the built-in constraint. No extra coding is required to prevent a constraint solution error, because the `randomize with()` constraint takes precedence over the soft constraint, and so the soft constraint is ignored.

2. *Uniqueness constraints (Mantis 3028)*

In previous versions of the SystemVerilog standard, in was difficult to specify constraints so that a list of several variables — or all the members of an array — had different random values, so that no two members of the list or array had the same value. A short list or very small array could be specified with multiple constraints, but this would be impractical for larger lists or arrays.

The proposed SystemVerilog-2012 standard adds a uniqueness constraint that efficiently — as in a single line of code — models generating random values where all variables in a list or an array receive unique values.

```
class Transaction;
  rand int  a, b, c;
  rand byte data_array[16];

  constraint c1 { unique {a,b,c}; }
  constraint c2 { unique {data_array}; }
endclass
```

Constraint `c1` ensures that, whenever random values are generated, the values of `a`, `b` and `c` will be different. Constraint `c2` ensures that whenever random values are generated, every element of `data_array` will have a different value.

C. *General programming enhancements*

This paper classifies 10 of the new features in the proposed SystemVerilog-2012 as general programming enhancements. Most of these new features help make SystemVerilog more efficient by simplifying or reducing the amount of code required to verify complex designs.

1. *User defined nets (Mantis 3398)*

SystemVerilog allows programmers to create new variable types using `typedef` declarations. User-defined variable types is a powerful modeling construct that is widely used in both design and verification. Unfortunately, previous versions of the SystemVerilog standard did not allow users to define new net data types that could be used to connect design blocks together.

The proposed SystemVerilog-2012 standard extends this capability to also create user-defined net types, using `nettype` declarations. Unlike variables, nets require both a value set and resolution functions for multiple drivers of the net. The new `nettype` declaration can specify both value sets and multi-driver resolution functions.

The built-in net types in SystemVerilog, such as wire, can only work with 4-state values sets. User-defined nets significantly enhance this, and allow a net to be comprised of:

- 4-state integral (vector) types, including packed arrays, packed structures or packed unions
- 2-state integral types (`bit`, `byte`, `int`, etc.)
- `real` and `shortreal` types
- Fixed-size unpacked arrays, unpacked structures, and unpacked unions (each element must have a valid data type for a net of a user-defined net type)

The full syntax and rules for user-defined net types is beyond the scope of this paper. Only a simple example, adapted from the proposed SystemVerilog-2012 standard, is shown to illustrate the concept of user-defined nets.

```
typedef struct {
  real field1;
  bit field2;
} T;

function automatic T Tsum(input T driver[]);
  Tsum.field1 = 0.0;
  foreach (driver[i])
    Tsum.field1 += driver[i].field1;
endfunction

nettype T wTsum with Tsum;
```

This `nettype` declaration defines a net whose data type is `T` (which has a mix of real and 2-state values), and which uses the `Tsum` function to resolve multiple drivers.

The ability to have nets with `real` (floating point) values is an important new capability. It allows accurately modeling and simulating mixed-signal (analog and digital) designs completely within the SystemVerilog language rather than requiring co-simulation environments. Co-simulation can be expensive, difficult, and slow. Keeping within the SystemVerilog language can simplify verification and be much more efficient, both in the amount of coding required and the run-time performance of simulation.

2. Typeless connections in netlists (Mantis 3724)

SystemVerilog modules, interfaces and programs can be parameterized to have port types or sizes for each instance of the block. Configurations and conditional compilation allow different models to be selected at compile/elaboration time. These capabilities provide a great deal of design and verification flexibility, but at the same time require that the netlist connecting design blocks be modeled work with a wide variety of data types. This was difficult to do in previous versions of SystemVerilog.

The proposed SystemVerilog-2012 standard adds a new typeless net, referred to as an *interconnect net*, that makes it much easier and efficient to model generic netlists that can work with different types of nets. Interconnect nets separate the specification of the netlist connections from the data types of the nets. The netlist is generic, and the types are determined by the internal types of actual components being connected (which can be selected using configurations or conditional compilation). The syntax for typeless interconnect nets is simple; the keyword `interconnect` is used instead of a specific net type.

Note that interconnect nets can only be connected to ports where the other side of the port is a net type or another interconnect net. Interconnect nets cannot be connected to ports with a variable on the other side.

```
nettype real rwire;
nettype int  iwire;

module adder #(parameter type DTYPE=wire [15:0])
              (input  DTYPE a, b,
               output DTYPE sum);
  ...
endmodule

module top;
  interconnect a, b, c, d, e, f, r1, r2, r3;

  // 16-bit 4-state adder
  adder                     i1 (a, b, r1);

  // 32-bit 2-state adder
  adder #(.DTYPE(iwire))    i2 (c, d, r2);

  // floating point adder
  adder #(.DTYPE(rwire))    i3 (e, f, r3);
endmodule
```

3. Parameterized tasks and functions (Mantis 696)

A popular — and efficient — coding style in SystemVerilog (and part of the original Verilog HDL) is parameterized modules. It allows a single version of module to be written, and then reconfigured for each usage. SystemVerilog classes can also be parameterized and reconfigured (referred to as “specializing”) for each instance in a similar way.

A limitation in previous versions of SystemVerilog, however, is that tasks, functions and class methods cannot be parameterized. If, for example, a design could work with 16-bit, 32-bit, or 64-bit bus sizes, a different task would need to be written for each bus size. Redundant code such as this is inefficient and error-prone.

It has been a long-standing enhancement request to add parameterized tasks and functions to SystemVerilog. Unfortunately, the syntax used to redefine module and class parameters is not compatible with the syntax of task/function calls. As a result, the SystemVerilog standards committee has left this enhancement request on a back burner for each new version of the SystemVerilog standard.

The proposed SystemVerilog-2012 standard adds this long-desired efficient coding style of parameterized tasks and functions. A simple and novel way was found to overcome the parameter redefinition syntax incompatibility — using static methods within a parameterized class. Parameters in a class can be redefined for each usage, and static class methods can be called from anywhere using the class scope name, and parameters in the class scope can be redefined for each call.

The following example is adapted from the proposed SystemVerilog-2012 standard:

```
virtual class C
  #(parameter  DECODE_W,
    localparam ENCODE_W = $clog2(DECODE_W) );
  static function logic [ENCODE_W-1:0] ENCODER_f
    (input logic [DECODE_W-1:0] DecodeIn);
    ENCODER_f = '0;
    for (int i=0; i<DECODE_W; i++) begin
      if (DecodeIn[i]) begin
        ENCODER_f = i[ENCODE_W-1:0];
        break;
      end
    end
  endfunction

  static function logic [DECODE_W-1:0] DECODER_f
    (input logic [ENCODE_W-1:0] EncodeIn);
    DECODER_f = '0;
    DECODER_f[EncodeIn] = 1'b1;
  endfunction
endclass

module test;
  ...
  // Redefine DECODE_W for each function call
  encoder_out = C#(8)::ENCODER_f(8'b0100_0000);
  decoder_out = C#(4)::DECODER_f(2'b11);
  ...
endmodule
```

In this example, functions `ENCODER_f` and `DECODER_f` serve as template functions that can work with any size of data. The specific return size and input argument size is specified for each call to these functions by redefining the `DECODE_W` class parameter for each call.

In a strict sense, parameterized tasks and functions is a clarification of previous versions of SystemVerilog, rather than a new feature. No new syntax or semantic rules were added to the SystemVerilog standard, but using parameterized classes to specialize each call to a task or function was not documented in previous versions of the SystemVerilog standard.

4. *Parameterized user-defined types (Mantis 1504)*

SystemVerilog user-defined types are powerful and widely used in verification code. A limitation, however, is that a `typedef` definition applies to all instances of that type. It cannot be customized to be different each place the user-defined type is used.

The proposed SystemVerilog-2012 allows user-defined types to be parameterized so that each usage of the type can be specialized based on the parameter values. This is accomplished by defining the user-defined type within a parameterized class, in a similar manner to the parameterized tasks and functions described in enhancement II.3, above. For example:

```
virtual class P#(parameter SIZE = 1);
  typedef struct packed {
    logic [63:0] source_addr;
    logic [63:0] dest_addr;
    logic [31:0] payload [0:SIZE-1];
  } pakcet_t;
endclass

module test;
  // Redefine SIZE for each usage of packet_t
  P#(16)::pakcet_t small_pkt;
  P#(1024)::pakcet_t large_pkt;
...
endmodule
```

Defining class `P` as virtual in this example means that the class itself cannot be constructed, but static definitions within the class, such as `typedef`, can be referenced at anytime using the class scope name.

Parameterized user-defined types is more of a clarification of previous versions of SystemVerilog, rather than a new feature. No new syntax or semantic rules were added to the SystemVerilog standard, but using parameterized classes to specialize each instance of a user-defined type was not documented in previous versions of the SystemVerilog standard.

5. *Explicit untyped arguments in let constructs (Mantis 2835)*

The SystemVerilog `let` construct allows defining macro code. The macro can be used anywhere procedural code is used, and the body of the macro is effectively expanded at that usage point. The let construct is similar to the `‘define` text substitution compiler directive, but without the inherent dangers of compiler directives.

The `let` construct was first introduced in the SystemVerilog-2009 standard. Its syntax allows for the data types of its formal arguments to either be explicitly specified or to implicitly inherit the type of an actual argument. A mix of typed and untyped arguments is allowed in SystemVerilog-2009, but only if the untyped arguments are specified first. It is not possible to have a mix of explicitly typed arguments followed by implicitly typed arguments.

The proposed SystemVerilog-2012 standard allows a mix of typed and untyped `let` formal arguments to be in any order, by explicitly specifying untyped arguments using the `untyped` keyword. The syntax is the same as with assertion properties.

```
let OK(event clk, untyped a) =
  assert ($stable(a,clk));

module test;
  logic [31:0] d;
  real         r;
  bit          clock;

  task do_something;
    ...
    OK(@(posedge clk), d) ...
    OK(@(negedge clk), r);
    ...
  endtask
endmodule
```

6. *$countbits system function (Mantis 2476)*

Previous versions of the SystemVerilog standard provide several convenience system functions that return information about the value of a vector: `$countones`, `$onehot`, `$onehot0`, and `$isunknown`.

The proposed SystemVerilog-2012 standard adds another convenience system function, `$countbits`. This function returns the number of bits that have a specific set of values (e.g., 0, 1, X, Z) in a bit vector. This new system function makes it easier and more efficient to write verification code that examines the individual bits of a vector. Without `$countbits`, it would be necessary to code a loop that iterates through each bit of a vector.

```
assert (!$isunknown(data)
else $error (“data has %0d bits with X or Z”,
  $countbits (data, 'x, 'z) );
```

This enhancement also makes several clarifications regarding the semantic rules of the bit-value system functions that were in previous versions of the standard.

7. *ref arguments with variable sized dimensions (Mantis 2929)*

SystemVerilog allows module ports and task/function formal arguments to be reference (in essence a pointer) to an actual argument. A reference port or formal argument is declared as `ref` instead of `input`, `output` or `inout`.

In previous versions of the SystemVerilog standard, `ref` ports or arguments could reference an array with fixed size dimensions, but not an array with variable size dimensions. The proposed SystemVerilog-2012 standard adds this capability.

```
package subroutines;
  task put_data (input value, ref d[$] );
    d.push_back(value);
  endtask
endpackage

module stack (input clk, input int data, ...);
  int data_q[$]; // variable size queue array

  always @(posedge clock)
    put_data(data, data_q);

endmodule
```

This example would be illegal in previous versions of SystemVerilog. The work-around would be to hard code the array name within the task, which would have made the task more difficult to re-use within the same design project or other projects.

8. *var type() in for-loop variable declarations (Mantis 2901)*

SystemVerilog added to the original Verilog language the ability to declare a for-loop iterator within the for-loop. In previous versions of SystemVerilog, the data type of this iterator had to be hard coded. For example:

```
for (int i; i<=255; i++) ...
```

The proposed SystemVerilog-2012 standard provides more flexibility by allowing the for-loop iterator data type to be calculated at compilation/elaboration time. This is done using a `var type()` declaration. `var type()` was already part of the SystemVerilog standard, but previously was not allowed in for-loop declarations. For example:

```
paramenter SIZE=64; // redefinable parameter
logic [SIZE-1:0] a, b;

for (var type({a,b}) i; i<=255; i++) ...
```

In this example, assuming `SIZE` is not redefined, variable `i` will be declared as a `logic [128:0]` type. If `SIZE` is redefined, variable `i` will adjust accordingly.

9. *`begin_keywords 1800-2012 (Mantis 3750)*

As with each new version of SystemVerilog, the proposed SystemVerilog-2012 reserves new keywords that are used by some of the new language features. The four new keywords are: `implements`, `interconnect`, `nettype`, `soft`.

SystemVerilog provides a compiler directive that can be used to inform software compilers about the version of SystemVerilog to which the code was written. This directive only affects the set of reserved keywords — it is not a backward compatibility directive that affects syntax and semantics. The directives for the two previous versions and the proposed 2012 version of SystemVerilog are:

```
  ‘begin_keywords 1800-2005
  ‘begin_keywords 1800-2009
  ‘begin_keywords 1800-2012
```

D. Coverage enhancements

There are three significant new features in the proposed SystemVerilog-2012 that enhance the ability to specify functional coverage. These enhancements were specified in one Mantis item in the SystemVerilog standard committee’s data base, but are listed separately in this paper (that same Mantis item also made many clarification changes affecting SystemVerilog functional coverage).

1. *Coverpoint variables (Mantis 2506)*

In previous versions of SystemVerilog, a coverpoint could have an optional *label*. This label was a simple name that had limited usage; primarily just in cross coverage specifications and in coverage reports. An example is:

```
covergroup cg5;
  Hue:  coverpoint pixel_hue;
  Addr: coverpoint pixel_addr + offset;
endgroup
```

The proposed SystemVerilog-2012 standard changes the label to a variable name. The data type of this variable can be explicitly specified or it can be inferred from context by a software tool. As a variable, the coverpoint can now be used to describe much more complex coverpoints, coverpoint bins, and cross coverage. If no data type is specified, the syntax for a coverpoint looks the same as in previous versions (and is therefore syntactically backward compatible). The underlying semantics of a coverpoint in the proposed SystemVerilog-2012 standard is quite different from previous versions. This difference should be transparent to users of the language, but will require changes within software tools.

```
covergroup cg;
  int       Hue:  coverpoint pixel_hue;
  bit [7:0] Addr: coverpoint pixel_addr + offset;
endgroup
```

2. *Coverage bin...with() expressions (Mantis 2506)*

The proposed SystemVerilog-2012 standard adds a `with()` clause to the coverpoint bin and cross bin definitions. The `bins`...`with()` construct can be used to exclude values in a bin that would not be of interest in a test. The bin will only count values that evaluate as true in the `with()` clause. The `bins`...`with()` construct uses a similar syntax to a `with()` clause in SystemVerilog's array locator methods. An implicit variable called `item` is used to represent a candidate value. This variable can be used as part of an expression that evaluates to true or false. If false, then the candidate value is ignored and excluded from the values counted in the bin.

In the following example, the bin definition covers all values of data from 0 to 255 that are evenly divisible by 16.

```
a: coverpoint data {
  bins mod16[] = {[0:255]} with (item % 16 == 0);
}
```

The name of the coverpoint containing the bin can be used in place of the value range to denote all possible values of the coverpoint.

```
int b: coverpoint data {
  bins lt1024 = b with (item < 1024);
}
```

3. *Function calls in covergroup expressions (Mantis 2506)*

The specification of coverage points and cross coverage in a large design can be very complex. The specification could be tedious and involve many lines of code in previous versions of SystemVerilog. Often this code would need to be repeated for multiple coverpoints. The proposed SystemVerilog 2012 standard adds the ability to define and call functions within a coverpoint or cross point. This allows complex coverpoints and cross coverage to be described in a much more efficient and re-usable manner. There are several restrictions regarding what is allowed in a function used in coverage expressions. These restrictions are beyond the scope of this paper.

The example that follows is adapted from the proposed SystemVerilog-2012 standard. Variables `a` and `b` are 32-bit vectors, and therefore a cross of the coverpoints for `a` and `b` would infer a very large set of cross bins. The `bins`...`with()` construct is used to limit the number of bins to values of interest, but that could still be an unwieldy and tedious amount of code that would be difficult to re-use. In this example, a function is used to limit the cross bins to a range of values that is of interest for a specific test. The value limits are specified when the covergroup is constructed, making this definition concise, efficient and easily re-used.

```
logic [31:0] a, b;

covergroup cg (int lower_limit, upper_limit);
  coverpoint a;
  coverpoint b;
  aXb : cross a, b {
    bins of_interest = f(cg_lim);
    function f myFunc(logic [31:0] f_lim);
      for (logic [31:0] i = 0; i < f_lim; ++i)
      f.push_back('{2*i,2*i});
    endfunction
  }
endgroup
```

E. Assertion enhancements

Ten of the new features in the proposed SystemVerilog-2012 standard enhance or provide new capabilities in SystemVerilog assertions.

1. *Additional data types in assertions (Mantis 2328)*

Previous versions of SystemVerilog did not allow assertions to use real (floating point) values, or dynamic arrays such as strings and queues. The proposed SystemVerilog-2012 standard removes these restrictions. The following simple example references a queue element as part of an assertion — something that could not be done this way in earlier versions of SystemVerilog.

```
byte q[$];
property p1;
  $rose(write) |-> q[0];
endproperty
```

2. *Static class properties in assertions (Mantis 2353)*

Previous versions of SystemVerilog did not allow concurrent assertions to access any property variables with a class object. The proposed SystemVerilog-2012 standard relaxes this restriction just a little, and allows concurrent assertions to access static properties within a class.

3. *Additional data types in sampled values (Mantis 3213)*

A moment in time in SystemVerilog is subdivided into several event regions. The SystemVerilog `$sampled`, `$past`, and other sampled value system functions return the value of a variable or expression that existed at the beginning of a moment in simulation time, in the Preponed region. Assertions and some other SystemVerilog constructs also sample current simulation values in the Preponed region.

Previous versions of the SystemVerilog standard restricted the data types that `$sampled` and other value sample functions could reference. Most notable is that real (floating point) and automatic variables were not allowed. This restriction made it difficult and inefficient to work with certain types of data, such as the real values that might be used in an analog/digital mixed signal model. The proposed SystemVerilog-2012 standard removes most of the restrictions on the data types that can be used with sampled value functions.

In the following example, the `$past` sampled value function references the automatic for-loop control variable `i`. This was not allowed in previous versions of the standard.

```
always @(posedge clk)
  for (int i = 0; i < 4; i ++)
    if (cond[i]) reg1[i] <= $past(b[i]);
```

Backward compatibility concern — the proposed SystemVerilog-2012 standard changes how sampled value functions work with free variables in checkers. The value that is returned is not backward compatible with previous versions of SystemVerilog. (Checkers and free variables were first introduced in the SystemVerilog-2009 standard). The value returned in previous versions was not correct, and considered an erratum. The proposed SystemVerilog-2012 corrects this incorrect specification.

4. *New rules for $global_clock resolution (Mantis 3069)*

The SystemVerilog-2009 standard introduced the concept of a global clock definition for use in formal verification assertion constructs. In SystemVerilog-2009, there could only be a single global clock definition, which encompassed the entire design. This rule proved to make it difficult to verify entire designs and/or sub-blocks of designs where multiple clock domains were involved. The `$global_clock` system function could be used anywhere in the hierarchy of a design to refer to the one global clock definition.

The proposed SystemVerilog-2012 standard changes the global clocking rules, and allows each hierarchical block of a design to specify a global clock for that scope. Although more than one global clocking declaration may appear in different parts of the design hierarchy, at most one global clocking declaration is effective at each point in the elaborated design hierarchy. The `$global_clock` system function refers to the global clock definition in the scope containing the call to `$global_clock`.

```
module master (...);
  ...
  global clocking @(posedge master_clock);
  endclocking
  ...
  property @($global_clock)
    ...
  endproperty
  ...
endmodule

module slave (...);
  ...
  global clocking @(posedge slave_clock);
  endclocking
  ...
  property @($global_clock)
    ...
  endproperty
  ...
endmodule
```

Backward compatibility concern — the proposed SystemVerilog-2012 standard is not fully backward compatible with SystemVerilog-2009. SystemVerilog-2009 allowed a global clocking definition to exist in a non-top-level module and use `$global_clock` outside that sub-hierarchy to reference the definition. This style will result in an error in the proposed SystemVerilog-2012 standard.

5. *Inferred clocks in sequences (Mantis 2412)*

In previous versions of the SystemVerilog standard, a named assertion sequence would only infer a clock when used in a property. Sequences specified in checkers or called from procedural code did not infer a clock from context. This limited the ability to use named sequences in checkers and other contexts without explicitly defining or passing in the sequence clock. The proposed SystemVerilog-2012 standard enhances named sequences to use the same clock inference rules as $sampled and other value sample functions. This allows writing more efficient, reusable checkers.

The following example, adapted from the justification for this proposed enhancement, would not work in previous versions of SystemVerilog because the sequence would not infer its sampling clock.

```
sequence following(e1, e2);
  e1 ##1 e2;
endsequence
```

```
checker check_mutex(
         input sequence s1,
         input cond, event clk=$inferred_clock);
  default clocking cb @clk; endclocking
  let r = s1.triggered; // not allowed in SV-2009
  a1: assert property (cond |=> r);
endchecker
```

6. Sequence methods with sequence expressions (Mantis 3191)

Previous versions of the SystemVerilog standard only allowed the `triggered` and `matched` sequence methods to be used on instances of a sequence. They could not be used with a sequence that was passed in as a sequence expression argument. This limited the usefulness of these methods, and made it difficult to model efficient, reusable code. The proposed SystemVerilog-2012 standard enhances the `triggered` and `matched` sequence methods by allowing them to be used on both sequence instances and sequence expressions.

```
property p1 (sequence s);
  s.triggered; // not allowed in SV-2009
endproperty
assert property p1(a ##1 b);
```

7. Final deferred immediate assertions (Mantis 3206)

The earliest versions of SystemVerilog included immediate assertions, which executed as simple programming statements, the same as an if...else statement. While very useful, an immediate assertion can inadvertently trigger multiple times in the same moment of simulation time if the procedural code containing the assertion glitches. These glitches could result in false assertion failures occurring within a moment in time.

SystemVerilog-2009 introduced deferred immediate assertions to reduce the risk of glitches. However, in order to allow the same flexibility in assertion action blocks that immediate assertions have, deferred immediate assertions still have a possibility, albeit greatly reduced, of glitching.

The proposed SystemVerilog-2012 standard adds a final deferred assertion construct. Final immediate assertions execute in the Postponed region of a simulation time step (the same region used by the $strobe and $monitor print tasks). The Postponed region severely restricts what can be done in an action block. In essence, messages can be printed, but no variables can be assigned. A final deferred immediate assertion is not as flexible as a deferred immediate assertion, but is guaranteed to be glitch free. Final assertions are specified with one of the keyword pairs: `assert final`, `assume final` or `cover final`.

In the following example, assertion `A1` is an immediate assertion and has the highest risk of false failures, should the `always_comb` trigger multiple times in a single moment of time (due to glitches on the signals read by the procedure). Assertion `A2` is a deferred immediate assertion and is less likely to have false failures, but glitches could occur. Assertion `A3` is a final deferred assertion, and is glitch free, but cannot do actions such as incrementing an error account (or even contain a `begin`...`end` block).

```
module test (...);
  ...
  always_comb
    A1: assert (!$isunknown state) else begin
          err_cnt++;
          $error("bad state");
        end
    A2: assert #0 (!$isunknown state) else begin
          err_cnt++;
          $error("bad state");
         end
    A3: assert final (!$isunknown state)
         else $error("bad state");
endmodule
```

8. Fine-grained assertion controls (Mantis 3295)

Previous versions of SystemVerilog provide the ability to control assertions using `$assertkill`, `$assertoff`, and `$asserton` system tasks. These tasks provide a medium level of granularity on which assertions are affected. The proposed SystemVerilog-2012 standard adds a new `$assertcontrol` system task that provides a much more fine level of control granularity. This system task can enable, disable or kill the assertions based on the assertion type (concurrent, immediate, or deferred immediate) or directive type (`assert`, `assume`, `cover`, `expect`). The task can also enable or disable action block execution of assertions and expect statements. The full syntax of `$assertcontrol` is not shown in this paper, but the usage is illustrated with a simple example.

```
enum {
  LOCK=1, UNLOCK=2, ON=3, OFF=4, KILL=5,
  CONCURRENT=1, IMMEDIATE=2, D_IMMEDIATE=12,
  EXPECT=16, ASSERT=1, COVER=2, ASSUME=4
} controls;
$assertcontrol(OFF, CONCURRENT, COVER|ASSUME, 0);
```

9. Output arguments for checkers (Mantis 2093)

The SystemVerilog-2009 standard added a checker construct to SystemVerilog. Checkers are a verification building block that are used to encapsulate related sequences, properties, assertions, and coverage. A checker can instantiate other checkers, but, in SystemVerilog-2009, checkers could only have input arguments. This made it difficult to build up complex checkers from other checkers. The proposed SystemVerilog-2012 standard enhances checkers by also allowing checker output arguments.

10. *New capabilities in checkers (Mantis 3033)*

The checker construct introduced in SystemVerilog-2009 was intended to encapsulate related assertion and coverage definitions. The checker could then serve as a reusable building block in verification code. The checker construct is useful, but the limitations on what a checker could contain made it difficult to encapsulate more complex verification building blocks. The proposed SystemVerilog-2012 standard significantly extends the capabilities of checkers by allowing a greater number of constructs within a checker. These new features include:

- `always_comb`, `always_latch`, and `always_ff` procedures
- Procedural blocking assignments
- Continuous assignments of checker variables
- Procedural conditional and looping statements
- Immediate assertions
- Task calls
- `let` declarations

These new capabilities within a checker enable modeling more efficient and reusable verification building blocks.

Backward compatibility — The SystemVerilog-2009 standard permitted `always` procedures in checkers, but did not allow `always_comb`, `always_latch` and `always_ff`. The proposed SystemVerilog-2012 standard is just the opposite, and makes the general `always` procedure illegal. The proposed SystemVerilog-2012 standard also changes the semantic rules for when checker arguments and external variables are sampled, which can result in a different simulation behavior between the two versions of the standard.

F. VPI enhancements

The proposed SystemVerilog2012 standard adds four major features to the Verification Procedural Interface (VPI), along with a number of errata and clarification corrections. These primary enhancements are:

- VPI support for soft constraints (Mantis 3884)
- VPI access added to the built-in process class (Mantis 3193)
- VPI transition to typespecs added to named events (Mantis 3116)
- VPI join type property added to the Scope diagram (Mantis 3188)

A full discussion of these new features is outside the scope of this paper.

III. USING INTERFACE CLASSES WITH UVM

All of the new features in the proposed SystemVerilog-2012 standard discussed in this paper are applicable to verification programming, and can potentially reduce the lines of code or enable more robust verification techniques. Perhaps the most intriguing of these new features is how multiple inheritance might enhance and make testbenches more efficient. This section focusses on multiple inheritance using interface classes might be useful in a UVM test environment.

The Universal Verification Methodology (UVM) relies on parameterization of interfaces in order to provide as much compile-time type safety as possible when assembling verification IP and environments. The most common use of parameterization is to specify the type of transaction being passed across an interface. Take, for example, the sequence/sequencer/driver parameterization.

The driver and sequencer communicate via the uvm_seq_item_port interface.

```
class my_driver extends uvm_driver
  #(type REQ = my_item);
  ...
  task run_phase(uvm_phase phase);
    seq_item_port.get_next_item(req);
    m_addr = req.get_addr();
    m_data = req.get_data();
    drive_bus(m_addr,m_data);
  endtask
endclass

class my_sequence extends uvm_sequence
  #(type REQ = my_item);
  ...
endclass

class my_agent extends uvm_agent;
  my_driver m_driver;
  uvm_sequencer #(type REQ = my_item) m_seqr;
  ...
  function void connect_phase(uvm_phase phase);
    m_driver.seq_item_port.connect(
      m_seqr.seq_item_export);
    ...
  endfunction
endclass
```

Note that we extend the `uvm_driver` to include user-defined functionality to specify exactly how the driver will convert the sequence item it receives through the `seq_item_port.get_next_item()` call into pin-level activity to communicate to the DUT. It is usually not necessary to extend the `uvm_sequencer` because the default functionality of the `uvm_sequencer` is sufficient for most applications.

The sequence itself communicates with the driver via the `start_item()`/`finish_item()` (and optionally `get_response()`) methods, each of which relies on the type of the REQ parameter. When the driver calls

```
  seq_item_port.get_next_item(req);
```

the item returned is of the parameterized type `REQ`. This allows the driver writer to rely on the type of the request transaction to know what methods to call to get information. It also constrains the driver to be able only to communicate with a sequence that generates items of the particular type, or extensions thereof. Of course, it also means that the sequence itself can only generate items of one particular type.

Because the driver can handle extensions of a base type, it is possible to create two sequences, each of which generates extensions of a common base item type, and then run them in parallel on the sequencer so that they interleave transactions to the driver.

```
class my_item extends uvm_sequence_item;
  ...
  addr_t m_addr;
  data_t m_data;
  virtual function addr_t get_addr();
    return m_addr;
  endfunction
  virtual function data_t get_data();
    return m_data;
  endfunction
  ...
endclass
```

```
class my_err_item extends my_item;
  ...
  function data_t get_data();
    addr_t tmp;
    tmp = super.get_data();
    return tmp+1; // could randomize the error
  endfunction
  ...
endclass
```

```
class my_test extends uvm_test;
  ...
  task run_phase(uvm_phase phase);
      ...
    my_seq = my_sequence::type_id::create
             ("my_seq");
    my_err_seq = my_err_sequence::type_id:create
                 ("my_err_seq");
    fork
      my_seq.start(m_env.m_agent.m_seqr);
      my_err_seq.start(m_env.m_agent.m_seqr);
    join
  endtask
  ...
endclass
```

The proposed SystemVerilog 2012 standard introduces the concept of interface classes that permit greater flexibility in the use of UVM sequences. An interface class contains a set of pure virtual methods, type declarations and parameter declarations that define a common set of behaviors that can be shared across multiple classes that are not necessarily derived from each other.

```
interface class my_driver_intf;
  pure virtual function addr_t get_addr();
  pure virtual function data_t get_data();
endclass
```

The driver would then be parameterized in terms of the interface class that defines how it will extract information from the item it receives from the sequence.

```
class my_driver extends uvm_driver
  #(my_driver_intf);
  ...
  task run_phase(uvm_phase phase);
    seq_item_port.get_next_item(req);
    m_addr = req.get_addr();
    m_data = req.get_data();
    drive_bus(m_addr,m_data);
  endtask
endclass
```

Note that the internals of the driver did not change. It still uses the `get_addr()` and `get_data()` methods to extract the address and data components of the transaction. This allows us to have a sequence that can generate any item type, as long as it implements the desired interface.

```
class my_item_1 extends uvm_sequence_item
                implements my_driver_intf;
  ...
endclass
```

```
class my_item_2 extends uvm_sequence_item
                implements my_driver_intf;
  ...
endclass
```

The sequence is also parameterized by the interface class, as was the driver.

```
class my_intf_seq extends uvm_sequence
    #(my_driver_intf);
  ...
  my_item_1 item1;
  my_item_2 item2;
  task body();
    item1 = my_item_1::type_id::create("item1");
    item2 = my_item_2::type_id::create("item2");
    fork
      for(int i = 0; i < 10; i++) begin
        start_item(item1);
        finish_item(item1);
      end
      for(int j = 0; j < 10; j++) begin
        start_item(item2);
        finish_item(item2);
      end
    join
  endtask
endclass
```

If the structure of the items being generated is similar, it would be possible to model the same basic pattern by extending `item1` and `item2` from the same abstract base class. This would require the driver, sequencer and sequence to be parameterized in terms of the abstract base class. Using the interface class, however, the sequence items being generated by the sequence do not need to be related to each other in any way, other than both implementing the interface class.

While it is possible that interface classes could be used to simplify the implementation of some of the internals of UVM, there is no pressing need to do so. Particularly in the area of the TLM interfaces, the use of interface classes could eliminate the need for some macros that are currently used along with the wrapper pattern to model similar functionality. However, since not all simulators support interface classes yet, and the savings would be more aesthetic than functional, this is not something the Accellera VIP-TSC, the committee that oversees the UVM standard, has explored at this point.

IV. ENHANCEMENTS THAT MIGHT BE SYNTHESIZABLE

All of the enhancements listed in this paper are useful in verification testbenches. Of these, the following enhancements might be synthesizable and, therefore, also useful in modeling hardware designs:

- User defined nets
- Generic, typeless connections in netlists
- var in for-loop variable declarations
- Parameterized tasks and functions
- Parameterized user-defined types
- Omit default type in parameterized type definition
- `begin_keywords 1800-2012

Note that there is no official synthesis subset defined for SystemVerilog. At the time this paper was written, no major commercial synthesis compilers were supporting the constructs listed in this section. However, these constructs could represent hardware, and might be implemented by synthesis compilers in the future.

V. CONCLUSIONS

It is critical for a hardware design and verification language to evolve with the complexity of hardware engineering projects. The proposed SystemVerilog-2012 standard is an important update to the SystemVerilog language. Thirty-one new design and verification capabilities have been added to SystemVerilog, along with numerous clarifications to the previous standard.

At the time this paper was written, The specification of the proposed SystemVerilog-2012 standard was complete, and the IEEE balloting and approval process about to begin. Final approval of the proposed SystemVerilog-2012 standard is expected later in 2012. Many Electronic Design Automation tool vendors have already begun implementing the new features in this proposed standard.

A primary goal of the new features in SystemVerilog is to help make design and verification more efficient. This efficiency is primarily achieved by providing ways to model complex design and verification code more accurately and concisely. Some of the proposed enhancements for SystemVerilog-2012 might also improve the run-time efficiency of software tools.

The IEEE 1800 SystemVerilog standards committee paid careful attention maintaining backwards compatibility with previous versions of SystemVerilog. The rare exceptions which were noted in this paper. These exceptions are behind the scenes, and should have minimal impact, if any, on existing SystemVerilog code.

The IEEE 1800 SystemVerilog standards committee has been proactive in ensuring that SystemVerilog is keeping pace with the engineering projects for which the language is intended. The chip you are designing is evolving quickly, and SystemVerilog is indeed “keeping up with Chip” — your chip.

VI. REFERENCES

[1] *“SystemVerilog 3.1: Accellera’s Extensions to Verilog”*, Accellera, Napa, CA, 2003.

[2] *“1800-2005 IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language”*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN: 0-7381-4811-3.

[3] *“SystemVerilog Is For Everyone (not just system designers)”* white paper by Stuart Sutherland, published 2004. Available at www.sutherland-hdl.com/papers.

[4] *“SystemVerilog Interoperability Checklist”* paper by Stuart Sutherland, presented at DVCon-2005, San Jose, California. Available at www.sutherland-hdl.com/papers.

[5] *“P1800-2009 IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language”*, IEEE, Pascataway, New Jersey. Copyright 2009. ISBN: 978-0-7381-6129-7.

[6] *“SystemVerilog Is Getting Even Better! An Update on the Proposed 2009 SystemVerilog Standard, Part 1”* presentation by Cliff Cummings, presented at DAC-2009, San Diego, California. Available at www.sutherland-hdl.com/papers.

[7] *“SystemVerilog Is Getting Even Better! An Update on the Proposed 2009 SystemVerilog Standard, Part 2”* presentation by Stuart Sutherland, presented at DAC-2009, San Diego, California. Available at www.sutherland-hdl.com/papers.

[8] *“P1800-2012/D5 Draft Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (ballot draft)”*, IEEE, Pascataway, New Jersey. Copyright 2012. ISBN: (not yet assigned).

VII. ABOUT THE AUTHORS

Stuart Sutherland is a well-known Verilog and SystemVerilog expert, with more than 23 years of experience using these languages for design and verification. His company, Sutherland HDL, specializes in training engineers to become true wizards using SystemVerilog. Stuart is active in the IEEE SystemVerilog standards process, and has been a technical editor for every version of the IEEE Verilog and SystemVerilog Language Reference Manuals since the IEEE standards work began in 1993. Prior to founding Sutherland HDL, Mr. Sutherland worked as an engineer on high-speed graphics systems used in military flight simulators. In 1988, he became a corporate applications engineer for Gateway Design Automation, the founding company of Verilog, and has been deeply involved in the use of Verilog and SystemVerilog ever since. Mr. Sutherland has authored several books and conference papers on Verilog and SystemVerilog. He holds a Bachelors Degree in Computer Science with an emphasis in Electronic Engineering Technology and a Masters Degree in Education with an emphasis on eLearning. You can contact Mr. Sutherland at stuart@sutherland-hdl.com.

Tom Fitzpatrick is a Verification Technologist at Mentor Graphics Corp., where he brings over two decades of design and verification experience to bear on developing advanced verification methodologies, particularly using SystemVerilog, and educating users on how to adopt them. He has been actively involved in the standardization of SystemVerilog, starting with his days as a member of the Superlog language design team at Co-Design Automation through its standardization via Accellera and then the IEEE, where he has served as chair of the 1364 Verilog Working Group, as well as a Technical Champion on the SystemVerilog P1800 Working Group. At Mentor Graphics, Tom was one of the original designers of the Advanced Verification Methodology (AVM), and later the Open Verification Methodology (OVM), and is the editor of Verification Horizons, a quarterly newsletter. He is a charter member and key contributor to the Accellera Verification IP Technical Subcommittee. He has published multiple articles and technical papers about SystemVerilog, verification methodologies, assertion-based verification, functional coverage, formal verification and other functional verification topics.