Subject: Issue #266 - Version 6 - May 4, 2005
(Version 6 is attached - Page numbers added)

Includes a few friendly amendments from various reviewers
(Arturo Salz - personal email sent 5/3/2005 - friendly amendments - see: 11.4.2, 11.5, 12.7, 13.11, 16.13)
(John Havlicek - email sent 5/4/2005 - friendly amendment - see: 18.15)
(Eduard Cerny - email to Karen Peiper sent 5/3/2005 - friendly amendment - see: 18.11.2, 18.15 (18.15 is the same comment as John Havlicek))

We have passed Issue #266 in the following committees:
SV-BC
SV-EC - (officially passed in meeting of 5/4/2005)
SV-AC - (per Surrendra Dudani (5/3/2005) & Bassam Tabbara (5/4/2005))

The SV-CC committee met today and per the sv-cc minutes of 5/4/2005, all proposed changes appear to be correct. It looks like the SV-CC committee will take an official vote on this proposal on Friday, 5/6/2005 (but so far all looks good)

The Champions Committee will have one last opportunity to make additional changes before passing the proposed resolution on to the SystemVerilog main committee.

NOTE - Thank you all for taking time to review and propose updates to this proposal. (Of course this is a non-normative expression of gratitude!) ;-)

Regards - Cliff

I have been reviewing a negative vote, Issue #266, from entity #6 on behalf of all of the SystemVerilog subcommittees. This issue is related to the informative use of note throughout the SystemVerilog LRM (negative-ballot comments shown below).

Regards - Cliff

Issue #266 Comments

In many places throughout the LRM, the usage of "Note" and "Notes" is not correct. In IEEE standards, notes are informative and not a required part of the standard. It is clear that in many places in the LRM there are notes that are both intended to be normative and are required for proper implementation of the standard.

I believe that IEEE standards also require that notes be in a separate paragraph and use a smaller font. In many places in the LRM, there are sentences (often in the middle of a paragraph) that begin with "Note..." but are not in a smaller font. It is unclear as to whether these notes are meant to be an informative notes, or an important normative fact that users and/or implementers need to observe. All usage's of the word "Note" "note," "NOTE," "Notes," "notes" and "NOTES" should be reviewed and brought into IEEE standards compliance.

Clauses that should be reviewed as to whether the terms "note" or "notes" are correctly used as only informative text include:

3.8

WAS: Note that the C-like alternative '{1, 1.0, 2, 2.0} is not allowed.

PROPOSED: The C-like alternative '{1, 1.0, 2, 2.0} for the preceding example is not allowed.

4.3.3

WAS: Note that the signed keyword is part of Verilog.

PROPOSED: The signed keyword in the preceding example is part of Verilog.

4.7.2

WAS: Note: str.putc(j, x) is semantically equivalent to str[j] = x.

PROPOSED: The putc method assignment str.putc(j, x) is semantically equivalent to str[j] = x.

4.7.3

WAS: Note: x = str.getc(j) is semantically equivalent to x = str[j].

PROPOSED: The getc method assignment x = str.getc(j) is semantically equivalent to x = str[j].

4.9 (multiple occurrences)
A type can be used before it is defined, provided it is first identified as a type by an empty **typedef**:

```
typedef foo;
foo f = 1;
typedef int foo;
```

WAS: Note that this does not apply to enumeration values, which must be defined before they are used.
PROPOSED: An empty typedef shall not be allowed with enumeration values. Enumeration values must be defined before they are used.

Sometimes a user-defined type needs to be declared before the contents of the type has been defined. This is of use with user-defined types derived from **enum**, **struct**, **union**, and **class**. For an example, see 12.24. Support for this is provided by the following forms for **typedef**: ...

WAS: Note that, while this is useful for coupled definitions of classes as shown in 12.24, it cannot be used for coupled definitions of structures, since structures are statically declared and there is no support for pointers to structures.
PROPOSED: While an empty user-defined type declaration is useful for coupled definitions of classes as shown in 12.24, it cannot be used for coupled definitions of structures, since structures are statically declared and there is no support for pointers to structures.

4.11
The text preceding this note is about packed unions (?? should "normal" union and "normal" structures both be changed to "unpacked"??)

WAS: Note that writing one member and reading another is independent of the byte ordering of the machine, unlike a normal union of normal structures, which are C-compatible and have members in ascending address order.
PROPOSED: With packed unions, writing one member and reading another is independent of the byte ordering of the machine, unlike a unpacked union of unpacked structures, which are C-compatible and have members in ascending address order.

4.13 - *first paragraph clarification and Note fixed to be normative text*

4.13 Singular and aggregate types

WAS: Data types are categorized as either *singular* or *aggregate*. A singular type shall be any data type except an unpacked structure, union, or array (see Clause 5 on arrays). An aggregate type shall be any unpacked structure, union, or array data type. A singular variable or expression represents a single value, symbols, or handle. Aggregate expressions and variables represent a set or collection of singular values. Integral types are always singular even though they can be sliced into multiple singular values.

...

Note that although a class is a type, there are no variables or expressions of class type directly, only class object handles that are singular. So classes need not be categorized in this manner ~~(see Clause 12 on classes).~~

PROPOSED: Data types are categorized as either *singular* or *aggregate*. A singular type shall be any data type except an unpacked structure, unpacked union, or unpacked array (see Clause 5 on arrays). An aggregate type shall be any unpacked structure, unpacked union, or unpacked array data type. A singular variable or expression represents a single value, symbols, or handle. Aggregate expressions and variables represent a set or collection of singular values. Integral types are always singular even though they can be sliced into multiple singular values.

...

Although a class, as described in Clause 12, is a type, there are no variables or expressions of class type directly, only class object handles that are singular. So classes need not be categorized in this manner.

4.14

WAS: Note that the **bit** data type loses X values. If these are to be preserved, the **logic** type should be used instead.

No change required - this is just an informative reminder of the behavior of the bit type, which is adequately defined elsewhere in the standard.

4.15

WAS: Note: $cast is similar to the dynamic_cast function available in C++, but, $cast allows users to check if the operation will succeed, whereas dynamic_cast always raises a C++ exception.

No change required - this is just an informative comparison to the dynamic casting capability of C++.

5.2

The preceding text in this paragraph is:

When assigning to an unpacked array, the source and target must be arrays with the same number of unpacked dimensions, and the length of each dimension must be the same. Assignment to an unpacked array is done by assigning each element of the source unpacked array to the corresponding element of the target unpacked array. The leftmost element of the source array corresponds to the leftmost element of the target array. ...

WAS: Note that an element of an unpacked array can be a packed array.

PROPOSED: Each element of an unpacked array that is assigned to the corresponding element of another unpacked array can itself be a packed array.

5.3

WAS: Note that the dimensions declared following the type and before the name ([3:0][7:0] in the preceding declaration) vary more rapidly than the dimensions following the name ([1:10] in the preceding declaration).

PROPOSED: In a multidimensional declaration, the dimensions declared following the type and before the name ([3:0][7:0] in the preceding declaration) vary more rapidly than the dimensions following the name ([1:10] in the preceding declaration).

5.6.2

WAS: ~~Note:~~ The size method is equivalent to $size(addr, 1).

PROPOSED: The size dynamic array method is equivalent to $size(addr, 1) array query system function (see Clause 24.7).

5.8
WAS: ~~Note that unsized dimensions can occur in dynamic arrays and in formal arguments of import DPI functions.~~ If one dimension of a formal is unsized, then any size of the corresponding dimension of an actual is accepted.

PROPOSED: If one dimension of a formal is unsized (unsized dimensions can occur in dynamic arrays and in formal arguments of import DPI functions) then any size of the corresponding dimension of an actual is accepted.

5.14.1 (multiple occurrences)
WAS: — An invalid index (i.e., a 4-state expression with X's or Z's, or a value that lies outside 0...$+1) shall cause a write operation to be ignored and a run-time warning to be issued. Note that writing to Q[$+1] is legal.
PROPOSED: — An invalid index (i.e., a 4-state expression with X's or Z's, or a value that lies outside 0...$+1) shall cause a write operation to be ignored and a run-time warning to be issued; however, writing to Q[$+1] is legal.

WAS:
5.14.1 Queue Operators
(1st paragraph)
Queues support the same operations that can be performed on unpacked arrays, and using the same operators and rules except as defined below:
(last paragraph - add this to the beginning of the first paragraph in the section)
~~NOTE:~~ Queues and dynamic arrays have the same assignment and argument passing semantics.

PROPOSED:
5.14.1 Queue Operators
(new, modified 1st paragraph - includes the NOTE from the end of the section)
Queues and dynamic arrays have the same assignment and argument passing semantics. Also, queues support the same operations that can be performed on unpacked arrays, and using the same operators and rules except as defined below:

6.6 (multiple occurrences) - See Mantis #646
WAS: Note that in SystemVerilog, data can be declared in unnamed blocks as well as in named blocks.
PROPOSED: In SystemVerilog, data can be declared in unnamed blocks as well as in named blocks.

WAS: Note that automatic or dynamic variables cannot be written with nonblocking or continuous assignments. Automatic variables and dynamic constructs—objects handles, dynamic arrays, associative arrays, strings, and event variables—shall be limited to the procedural context.

PROPOSED: Fixed by Mantis #646.

6.7
WAS: Note that a SystemVerilog variable cannot have an implicit continuous assignment as part of its declaration, the way a net can. An assignment as part of the logic declaration is a variable initialization, not a continuous assignment. For example:
PROPOSED: Unlike SystemVerilog nets, a SystemVerilog variable cannot have an implicit continuous assignment as part of its declaration. An assignment as part of the logic declaration is a variable initialization, not a continuous assignment. For example:

6.9
WAS: Note that there is no category for identical types defined here because there is no construct in the SystemVerilog language that requires it. For example, as defined below, **int** can be interchanged with **bit signed** [31:0] wherever it is syntactically legal to do so. Users can define their own level of type identity by using the $typename system function (see 24.3), or through use of the PLI.
PROPOSED: SystemVerilog does not require a category for identical types to be defined here because there is no construct in the SystemVerilog language that requires it. For example, as defined below, **int** can be interchanged with **bit signed** [31:0] wherever it is syntactically legal to do so. Users can define their own level of type identity by using the $typename system function (see 24.3), or through use of the PLI.

6.9.1
WAS: Two array types match if they have the same number of unpacked dimensions and their slowest-varying dimensions have matching types and the same left and right range bounds. Note that the type of the slowest varying dimension of a multidimensional array type is itself an array type.
PROPOSED: Two array types match if they have the same number of unpacked dimensions and their slowest-varying dimensions have matching types and the same left and right range bounds. The type of the slowest varying dimension of a multidimensional array type is itself an array type.

6.9.2
WAS: Note that if any bit of a packed structure or union is 4-state, the entire structure or union is considered 4-state.
PROPOSED: *(This is an informational note. No change to the note - except use a smaller font for the note and separate into a separate paragraph)*
Note that if any bit of a packed structure or union is 4-state, the entire structure or union is considered 4-state (see Clause 4.11).
(Adding the supporting section number would be useful)

8.3
The preceding text in this paragraph is:
In SystemVerilog, an expression can include a blocking assignment, provided it does not have a timing control.

WAS: Note that such an assignment must be enclosed in parentheses to avoid common mistakes such as using a=b for a==b, or a|=b for a!=b.
PROPOSED: These blocking assignments must be enclosed in parentheses to avoid common mistakes such as using a=b for a==b, or a|=b for a!=b.

8.12
WAS: Note that unlike bit concatenation, the result of a string concatenation or replication is not truncated.
PROPOSED: Unlike bit concatenation, the result of a string concatenation or replication is not truncated.

8.13.2

WAS: Note that the **default** keyword applies to members in nested structures or elements in unpacked arrays in structures.

PROPOSED: Use of the **default** keyword applies to members in nested structures or elements in unpacked arrays in structures.

8.16 (multiple occurrences)

For example, suppose there is a structure type float:

```
  typedef struct {
    bit sign;
    bit [3:0] exponent;
    bit [10:0] mantissa;
  } float;
...
bind + function float fcopyf(float);     // unary +
bind + function float fcopyi(int);       // unary +
bind + function float fcopyr(real);      // unary +
bind + function float fcopyr(shortreal); // unary +
...
```

WAS: Note that the function prototype does not need to match the actual function declaration exactly. If it does not, then the normal implicit casting rules apply when calling the function. For example the fcopyi function can be defined with an **int** argument:

PROPOSED: A function prototype does not need to match the actual function declaration exactly. If it does not, then the normal implicit casting rules apply when calling the function. For example the fcopyi function can be defined with an **int** argument:

(Pertinent text)

```
bind = function float fcopyi(int); // cast int to float
bind = function float fcopyr(real); // cast real to float
bind = function float fcopyr(shortreal); // cast shortreal to float
```

WAS: The operators that can be overloaded are the arithmetic operators, the relational operators and assignment. Note that the assignment operator from a float to a float cannot be overloaded here because it is already legal. Similarly, equality and inequality between floats cannot be overloaded.

PROPOSED: The operators that can be overloaded are the arithmetic operators, the relational operators and assignment. The assignment operator from a float to a float cannot be overloaded above because it is already legal in the three preceding bind statements. Similarly, equality and inequality between floats cannot be overloaded.

8.19
PROPOSED: Fixed by Mantis #410

9.4
WAS: Note: by specifying **unique** or **priority**, it is not necessary to code a **default** case to trap unexpected case values. For Example:
PROPOSED: *(keep this as a note - smaller font and make this a separate note-paragraph)*
Note: by specifying **unique** or **priority**, it is not necessary to code a **default** case to trap unexpected case values.
Consider the following example:

9.4.1
WAS: In pattern-matching, the value *V* of an expression is always matched against a pattern. Note that static type checking ensures that *V* and the pattern have the same type. The result of a pattern match is:
PROPOSED: In pattern-matching, the value *V* of an expression is always matched against a pattern and static type checking ensures that *V* and the pattern have the same type. The result of a pattern match is:

9.4.1.1
WAS: Example (same as previous example, but note that the first inner **case** statement involves only structures and constants but no tagged unions):
PROPOSED: Example (same as previous example, except that the first inner **case** statement involves only structures and constants but no tagged unions):

9.6
WAS: Note that SystemVerilog does not include the C **goto** statement.
PROPOSED: *(keep this as a note - smaller font and keep it as a separate note-paragraph ... unless everybody would like to add a goto statement ☺)*
Note that SystemVerilog does not include the C **goto** statement.

9.10 (multiple occurrences)

```
module latch (output logic [31:0] y, input [31:0] a, input enable);
  always @(a iff enable == 1)
    y <= a; //latch is in transparent mode
endmodule
```

WAS: The event expression only triggers if the expression after the **iff** is true, in this case when enable is equal to 1. Note that such an expression is evaluated when a changes, and not when enable changes. Also note that **iff** has precedence over **or**. This can be made clearer by the use of parentheses.

PROPOSED: The event expression only triggers if the expression after the **iff** is true, in this case when enable is equal to 1. This type of expression is evaluated when a changes, and not when enable changes. Also in similar event expressions of this type, **iff** has precedence over **or**. This can be made clearer by the use of parentheses.

11.4.2

(From 11.4.1)

For example, calling the function bellow copies 1000 bytes each time the call is made.

```
  function int crc( byte packet [1000:1] );
    for( int j= 1; j <= 1000; j++ ) begin
      crc ^= packet[j];
    end
  endfunction
```

…

(From 11.4.2 - Pass by reference)

For example, the example above can be written as:

```
  function int crc( ref byte packet [1000:1] );
    for( int j= 1; j <= 1000; j++ ) begin
      crc ^= packet[j];
    end
  endfunction
```

WAS: Note that in the example, no change other than addition of the **ref** keyword is needed. The compiler knows that packet is now addressed via a reference, but users do not need to make these references explicit either in the callee or at the point of the call. That is, the call to either version of the crc function remains the same:

PROPOSED: As shown in the preceding example , no change other than addition of the **ref** keyword is needed. The compiler knows that packet is now addressed via a reference, but users do not need to make these references explicit either in the callee or at the point of the call. That is, the call to either version of the crc function remains the same:

11.4.3

WAS: The *default_value* is an expression. The expression is evaluated in the scope of the caller each time the subroutine is called. The elements of the expression must be visible at the scope of subroutine and, if used, at the scope of the caller. If the default_value is not used, the expression is not evaluated and need not be visible at the scope of the caller. Note that default values are only allowed with the ANSI style declaration.

PROPOSED: The *default_value* is an expression. The expression is evaluated in the scope of the caller each time the subroutine is called. The elements of the expression must be visible at the scope of subroutine and, if used, at the scope of the caller. If the default_value is not used, the expression is not evaluated and need not be visible at the scope of the caller. The use of default values shall only be allowed with the ANSI style declarations.

11.5 (multiple occurrences)

WAS: Several SystemVerilog functions can be mapped to the same foreign function by supplying the same *c_identifier* for several *fnames*. Note that all these SystemVerilog functions must have identical argument types, as defined in the next paragraph.

PROPOSED: Several SystemVerilog functions can be mapped to the same foreign function by supplying the same *c_identifier* for several *fnames*. The corresponding SystemVerilog functions must have identical argument types, as defined in the next paragraph.

WAS: Note that **import** "DPI" functions declared this way can be invoked by hierarchical reference the same as any normal SystemVerilog function.

PROPOSED: All **import** "DPI" functions declared this way can be invoked by hierarchical reference the same as any normal SystemVerilog function.

WAS: Import context functions can have side effects and can use other SystemVerilog interfaces (including but not limited to VPI). However, ~~note that~~ declaring an import context function does not automatically make any other simulator interface available. For VPI access (or any other interface access) to be possible, the appropriate implementation-defined mechanism must still be used to enable these interface(s). Note also that DPI calls do not automatically create or provide any handles or any special environment that might be needed by those other interfaces. ...

PROPOSED: Import context functions can have side effects and can use other SystemVerilog interfaces (including but not limited to VPI). However, declaring an import context function does not automatically make any other simulator interface available. For VPI access (or any other interface access) to be possible, the appropriate implementation-defined mechanism must still be used to enable these interface(s). Also, SystemVerilog DPI calls do not automatically create or provide any handles or any special environment that might be needed by those other interfaces. ...

WAS: To access functions defined in any other scope the foreign code shall have to change DPI context appropriately. Attempting to invoke an exported SystemVerilog function from a scope in which it is not directly visible shall result in a runtime error. How such errors are handled shall be implementation dependent. If an imported function needs to invoke an exported function that is not visible from the current scope, it needs to change, via svSetScope, the current scope to a scope that does have visibility to the exported function. This is conceptually equivalent to making a hierarchically qualified function call in SystemVerilog. The current SystemVerilog context shall be preserved across a call to an exported function, even if current context has been modified by an application. Note that context is not defined for non-context imports and attempting to use any functionality depending on context from non-context imports can lead to unpredictable behavior.
PROPOSED: To access functions defined in any other scope the foreign code shall have to change DPI context appropriately. Attempting to invoke an exported SystemVerilog function from a scope in which it is not directly visible shall result in a runtime error. How such errors are handled shall be implementation dependent. If an imported function needs to invoke an exported function that is not visible from the current scope, it needs to change, via svSetScope, the current scope to a scope that does have visibility to the exported function. This is conceptually equivalent to making a hierarchically qualified function call in SystemVerilog. The current SystemVerilog context shall be preserved across a call to an exported function, even if current context has been modified by an application. For non-context imports the context is not defined and attempting to use any functionality depending on context from non-context imports can lead to unpredictable behavior.

12.6

12.6 Object methods

An object's methods can be accessed using the same syntax used to access class properties:

```
Packet p = new;
status = p.current_status();
```

WAS: Note that the assignment to status is not:

```
status = current_status(p);
```

PROPOSED: The above assignment to status cannot be written as:

```
status = current_status(p);
```

12.7

SystemVerilog provides a mechanism for initializing an instance at the time the object is created. When an object is created, for example

```
Packet p = new;
```

The system executes the **new** function associated with the class:

```
class Packet;
  integer command;
  function new();
    command = IDLE;
  endfunction
endclass
```

WAS: Note that **new** is now being used in two very different contexts with very different semantics. The variable declaration creates an object of class Packet. In the course of creating this instance, the **new** function is invoked, in which any specialized initialization required can be done. The **new** function is also called the *class constructor*.

PROPOSED: As shown above, **new** is now being used in two very different contexts with very different semantics. The variable declaration creates an object of class Packet. In the course of creating this instance, the **new** function is invoked, in which any specialized initialization required can be done. The **new** function is also called the *class constructor*.

12.8

```
class Packet ;
  static integer fileId = $fopen( "data", "r" );
```

Now, fileID shall be created and initialized once. Thereafter, every Packet object can access the file descriptor in the usual way:

```
Packet p;
c = $fgetc( p.fileID );
```

WAS: Note that static class properties can be used without creating an object of that type.
PROPOSED: The static class properties can be used without creating an object of that type.

12.10

The x is now both a property of the class and an argument to the function **new**. In the function **new**, an unqualified reference to x shall be resolved by looking at the innermost scope, in this case the subroutine argument declaration. To access the instance class property, it is qualified with the **this** keyword, to refer to the current instance. ...

WAS: Note that in writing methods, members can be qualified with **this** to refer to the current instance, but it is usually unnecessary.
PROPOSED: *(keep this as a note - smaller font and keep it as a separate note-paragraph)*

12.11

Declaring a class variable only creates the name by which the object is known. Thus:

```
Packet p1;
```

creates a variable, p1, that can hold the handle of an object of class Packet, but the initial value of p1 is **null**. The object does not exist, and p1 does not contain an actual handle, until an instance of type Packet is created:

```
p1 = new;
```

Thus, if another variable is declared and assigned the old handle, p1, to the new one, as in:

```
Packet p2;
p2 = p1;
```

then there is still only one object, which can be referred to with either the name p1 or p2.

...

WAS: Note, **new** was executed only once, so only one object has been created.

PROPOSED: In this example, **new** was executed only once, so only one object has been created.

12.17

A **protected** class property or method has all of the characteristics of a **local** member, except that it can be inherited; it is visible to subclasses.

WAS: Note that within the class, a local method or class property of the class can be referenced, even if it is in a different instance. For example:

PROPOSED: Within a class, a local method or class property of the same class can be referenced, even if it is in a different instance of the same class. For example:

```
class Packet;
  local integer i;
  function integer compare (Packet other);
    compare = (this.i == other.i);
  endfunction
endclass
```

A strict interpretation of encapsulation might say that other.i should not be visible inside of this packet, since it is a local class property being referenced from outside its instance. Within the same class, however, these references are allowed. In this case, this.i shall be compared to other.i and the result of the logical comparison returned.

12.24 (multiple occurrences)

Sometimes a class variable needs to be declared before the class itself has been declared. For example, if two classes each need a handle to the other. When, in the course of processing the declaration for the first class, the compiler encounters the reference to the second class, that reference is undefined and the compiler flags it as an error.

This is resolved using **typedef** to provide a forward declaration for the second class:

```
typedef class C2; // C2 is declared to be of type class
class C1;
  C2 c;
endclass
class C2;
  C1 c;
endclass
```

WAS: In this example, C2 is declared to be of type **class**, a fact that is re-enforced later in the source code. Note that the **class** construct always creates a type, and does not require a **typedef** declaration for that purpose (as in **typedef class** …). This is consistent with common C++ use.
PROPOSED: In this example, C2 is declared to be of type **class**, a fact that is re-enforced later in the source code. In SystemVerilog, the **class** construct always creates a type, and does not require a **typedef** declaration for that purpose (as in **typedef class** …). This is consistent with common C++ use.

WAS: Note that the class keyword in the statement typedef class C2; is not necessary, and is used only for documentation purposes. The statement typedef C2; is equivalent and shall work the same way.
PROPOSED: In the preceding example, the class keyword in the statement typedef class C2; is not necessary, and is used only for documentation purposes. The statement typedef C2; is equivalent and shall work the same way.

13.4.3

WAS: It is important to note that the **inside** operator is bidirectional, thus, the second example above is equivalent to a == b || a == c.
PROPOSED: In SystemVerilog, the **inside** operator is bidirectional, thus, the second example above is equivalent to a == b || a == c.

13.4.11 (multiple occurrences)
WAS: Unlike the count_ones function, more complex properties, which require temporary state or unbounded loops, may be impossible to convert into a single expression. The ability to call functions, thus, enhances the expressive power of the constraint language and reduces the likelihood of errors. Note that the two constraints above are not completely equivalent; C2 is bidirectional (length can constrain v and vice-versa), whereas C1 is not.
PROPOSED: Unlike the count_ones function, more complex properties, which require temporary state or unbounded loops, may be impossible to convert into a single expression. The ability to call functions, thus, enhances the expressive power of the constraint language and reduces the likelihood of errors. The two constraints, C1 and C2, from above are not completely equivalent; C2 is bidirectional (length can constrain v and vice-versa), whereas C1 is not.

... For example:

```
class B;
  rand int x, y;
  constraint C { x <= F(y); }
  constraint D { y inside { 2, 4, 8 } ; }
endclass
```

WAS: Forces y to be solved before x. Thus, constraint D is solved separately before constraint C, which uses the values of y and F(y) as state variables. Note that the behavior for variable ordering implied by function arguments differs from the behavior for ordering specified using the "**solve**...**before**..." constraint; function argument variable ordering subdivides the solution space thereby changing it. Since constraints on higher-priority variables are solved without considering lower-priority constraints at all this subdivision can cause the overall constraints to fail. Within each prioritized set of constraints, cyclical (**randc**) variables are solved first.
PROPOSED: Forces y to be solved before x. Thus, constraint D is solved separately before constraint C, which uses the values of y and F(y) as state variables. In SystemVerilog, the behavior for variable ordering implied by function arguments differs from the behavior for ordering specified using the "**solve**...**before**..." constraint; function argument variable ordering subdivides the solution space thereby changing it. Since constraints on higher-priority variables are solved without considering lower-priority constraints at all this subdivision can cause the overall constraints to fail. Within each prioritized set of constraints, cyclical (**randc**) variables are solved first.

13.5.3

WAS: **13.5.3 Randomization methods notes**

PROPOSED: **13.5.3 Behavior of Randomization methods**

13.11

For example:

```
  module stim;
    bit [15:0] addr;
    bit [31:0] data;
    function bit gen_stim();
      bit success, rd_wr;
      // call std::randomize
      success = randomize( addr, data, rd_wr );
      return rd_wr ;
    endfunction
    ...
  endmodule
```

WAS: The function gen_stim calls std::randomize() with three variables as arguments: addr, data, and rd_wr. Thus, std::randomize() assigns new random variables to those variables that are visible in the scope of the gen_stim function. Note that addr and data have module scope, whereas rd_wr has scope local to the function. The above example can also be written using a class:

PROPOSED: The function gen_stim calls std::randomize() with three variables as arguments: addr, data, and rd_wr. Thus, std::randomize() assigns new random variables to those variables that are visible in the scope of the gen_stim function. In the preceding example, addr and data have module scope, whereas rd_wr has scope local to the function. The preceding example can also be written using a class:

14.3.7

WAS: Note that calling peek() can cause one message to unblock more than one process. As long as a message remains in the mailbox queue, any process blocked in either a peek() or get() operation shall become unblocked.

PROPOSED: Calling the peek() method can also cause one message to unblock more than one process. As long as a message remains in the mailbox queue, any process blocked in either a peek() or get() operation shall become unblocked.

15.2

WAS: The SystemVerilog language is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this clause to provide a context to describe the meaning and valid interpretation of SystemVerilog constructs. These resulting definitions provide the standard SystemVerilog reference algorithm for simulation, which all compliant simulators shall implement. Note that there is a great deal of choice ~~in the definitions that follow,~~ and differences in some details of execution are to be expected between different simulators. In addition, SystemVerilog simulators are free to use different algorithms than those described in this clause, provided the user-visible effect is consistent with the reference algorithm.

PROPOSED: The SystemVerilog language is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this clause to provide a context to describe the meaning and valid interpretation of SystemVerilog constructs. These resulting definitions provide the standard SystemVerilog reference algorithm for simulation, which all compliant simulators shall implement. Within the following event execution model definitions, there is a great deal of choice and differences in some details of execution are to be expected between different simulators. In addition, SystemVerilog simulators are free to use different algorithms than those described in this clause, provided the user-visible effect is consistent with the reference algorithm.

16.13
WAS: Wait for the falling edge of the specified 1-bit slice dom.sign[a]. ~~Note that the index a is evaluated at runtime.~~
@(**negedge** dom.sign[*a*]);

(separate the note, place the note after the example code, make the font smaller - This can be an informative note because there is a sentence shortly before this list of examples that states, "Slices can include dynamic indices, which are evaluated once, when the @ expression executes.")

PROPOSED: Wait for the falling edge of the specified 1-bit slice dom.sign[a].
@(**negedge** dom.sign[*a*]);
Note: the dynamic index a is evaluated at runtime.

17.4
WAS: Since the program schedules events in the Reactive region, the clocking block construct is very useful to automatically sample the steady-state values of previous time steps or clock cycles. Programs that read design values exclusively through clocking blocks with #0 input skews are insensitive to read-write races. It is important to note that simply sampling input signals (or setting non-zero skews on clocking block inputs) does not eliminate the potential for races. Proper input sampling only addresses a single clocking block. With multiple clocks, the arbitrary order in which overlapping or simultaneous clocks are processed is still a potential source for races. The program construct addresses this issue by scheduling its execution in the Reactive region, after all design events have been processed, including clocks driven by nonblocking assignments.
PROPOSED: Since the program schedules events in the Reactive region, the clocking block construct is very useful to automatically sample the steady-state values of previous time steps or clock cycles. Programs that read design values exclusively through clocking blocks with #0 input skews are insensitive to read-write races. It is important to understand that simply sampling input signals (or setting non-zero skews on clocking block inputs) does not eliminate the potential for races. Proper input sampling only addresses a single clocking block. With multiple clocks, the arbitrary order in which overlapping or simultaneous clocks are processed is still a potential source for races. The program construct addresses this issue by scheduling its execution in the Reactive region, after all design events have been processed, including clocks driven by nonblocking assignments.

18.2

WAS: Note: The assertion control system tasks are described in 24.9.

PROPOSED: *(keep this as a note - smaller font and keep it as a separate note-paragraph)*

Note: The assertion control system tasks are described in 24.9.

18.3

The sampled values are used to evaluate value change expressions or boolean subexpressions that are required to determine a match of a sequence.

WAS: Note:

— It is important to ensure that the defined clock behavior is glitch free. Otherwise, wrong values can be sampled.

— If a variable that appears in the expression for clock also appears in an expression with an assertion, the values of the two usages of the variable can be different. The current value of the variable is used in the clock expression, while the sampled value of the variable is used within the assertion.

PROPOSED: For concurrent assertions:

— It is important to ensure that the defined clock behavior is glitch free. Otherwise, wrong values can be sampled.

— If a variable that appears in the expression for clock also appears in an expression with an assertion, the values of the two usages of the variable can be different. The current value of the variable is used in the clock expression, while the sampled value of the variable is used within the assertion.

18.6
A sequence is declared with optional formal arguments. When a sequence is instantiated, actual arguments can be passed to the sequence. The sequence gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. Semantic checks are performed to ensure that the expanded sequence with the actual arguments is legal.

An actual argument can replace an:
— identifier
— expression
— event control expression
— upper delay range or repetition range if the actual argument is $

WAS: ~~Note that~~ variables used in a sequence that are not formal arguments to the sequence are resolved according to the scoping rules from the scope in which the sequence is declared.
PROPOSED: Variables used in a sequence that are not formal arguments to the sequence are resolved according to the scoping rules from the scope in which the sequence is declared.

18.7.3
When intermediate optional arguments between two arguments are not needed, a comma must be placed for each omitted argument. For example,

```
$past(in1, , enable);
```

WAS: Here, a comma is specified to omit *number_of_ticks*. The default of one is used for the empty *number_of_ticks* argument. Note that a comma for the omitted *clocking_event* argument ~~is not needed,~~ as it does not fall within the specified arguments.
PROPOSED: Here, a comma is specified to omit *number_of_ticks*. The default of one is used for the empty *number_of_ticks* argument. There is no need to include a comma for the omitted *clocking_event* argument as it does not fall within the specified arguments.

18.8

In a single cycle, there can be multiple matches of a sequence instance to which ended is applied, and these matches can have different valuations of the local variables. The multiple matches are treated semantically the same way as matching both disjuncts of an **or** (see below). In other words, the thread evaluating the instance to which ended is applied will fork to account for such distinct local variable valuations.

WAS: Note that when a local variable is a formal argument of a sequence declaration, it is illegal to declare the variable, as shown below.
PROPOSED: When a local variable is a formal argument of a sequence declaration, it is illegal to declare the variable, as shown below.

```
sequence sub_seq3(lv);
  int lv; // illegal since lv is a formal argument
  (a ##1 !a, lv = data_in) ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
```

18.11.2

The use of implication when multi-clock sequences and properties are involved is explained in 18.12.

WAS: The following example illustrates a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which irdy is asserted and either trdy or stop is asserted. Note that an asserted signal ~~here~~ implies a value of low.
PROPOSED: The following example illustrates a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which irdy is asserted and either trdy or stop is asserted. In this example, an asserted signal implies a value of low.

18.12.3
The scope of a clocking event does not flow into the reset condition of **disable iff**.

WAS: ~~Note that~~ juxtaposing two clocking events nullifies the first of them:
PROPOSED: Juxtaposing two clocking events nullifies the first of them; therefore, the following two-clocking-event statement

```
@(d) @(c) x
```

is equivalent to

```
@(c) x
```

because the flow of clock *d* is immediately overridden by clock *c*.

18.13.1
WAS: ~~Note:~~ The pass and fail statements are executed in the Reactive region. The regions of execution are explained in the scheduling semantics Clause 15.
PROPOSED: The pass and fail statements of an **assert** statement are executed in the Reactive region. The regions of execution are explained in the scheduling semantics Clause 15.

18.13.2
WAS: Note that **assume** does not provide an action block, as the actions for an assumption serve no purpose.
PROPOSED: The **assume** statement does not provide an action block, as the actions for an assumption serve no purpose.

18.15 (multiple occurrences)
Example of binding a program instance to a module:

```
  bind cpu fpu_props fpu_rules_1(a,b,c);
```

Where:
— cpu is the name of the target module.
— fpu_props is the name of the program to be instantiated.
— fpu_rules_1 is the program instance name to be created in the target scope.
— An instance named fpu_rules_1 is instantiated in every instance of module cpu.
...

WAS: — The first three ports of program fpu_props get bound to objects a, b, and c in module cpu. ~~Note that~~ these objects are viewed from module cpu's point of view. They are completely distinct from any objects named a, b, and c that are visible in the scope that contains the **bind** directive.
PROPOSED: — The first three ports of program fpu_props get bound to objects a, b, and c in module cpu (these objects are viewed from module cpu's point of view and they are completely distinct from any objects named a, b, and c that are visible in the scope that contains the **bind** directive).

WAS: It is legal for more than one **bind** statement to bind a *bind_instantiation* into the same target scope. However, it shall be an error for a *bind_instantiation* to introduce an instance name that clashes with another name in the module name space of the target scope (See 19.13). This applies to both pre-existing names as well as instance names introduced by other **bind** statements. ~~Note that~~ the latter situation will occur if the design contains more than one instance of a module containing a **bind** statement.
PROPOSED: It is legal for more than one **bind** statement to bind a *bind_instantiation* into the same target scope. However, it shall be an error for a *bind_instantiation* to introduce an instance name that clashes with another name in the module name space of the target scope (See 19.13). This applies to both pre-existing names as well as instance names introduced by other **bind** statements. The latter situation will occur if the design contains more than one instance of a module containing a **bind** statement.

19.6
WAS: This allows the same module name, e.g. and2, to occur in different parts of the design and represent different modules. ~~Note that~~ an alternative way of handling this problem is to use configurations.
PROPOSED: This allows the same module name, e.g. and2, to occur in different parts of the design and represent different modules. An alternative way of handling this problem is to use configurations.

19.7
WAS: To support separate compilation, extern declarations of a module can be used to declare the ports on a module without defining the module itself. An extern module declaration consists of the keyword **extern** followed by the module name and the list of ports for the module. Both list of ports syntax (possibly with parameters), and original Verilog style port declarations can be used. Note that the potential existence of defparams precludes the checking of the port connection information prior to elaboration time even for list of ports style declarations.
PROPOSED: *(Separate the note -use a smaller font for the note)*
To support separate compilation, extern declarations of a module can be used to declare the ports on a module without defining the module itself. An extern module declaration consists of the keyword **extern** followed by the module name and the list of ports for the module. Both list of ports syntax (possibly with parameters), and original Verilog style port declarations can be used.

Note that the potential existence of defparams precludes the checking of the port connection information prior to elaboration time even for list of ports style declarations.

19.12.2
WAS: Note that where the data types differ between the port declaration and connection, an initial value change event can be caused at time zero.
PROPOSED: If there is a data type difference between the port declaration and connection, an initial value change event can be caused at time zero.

20.2.1
WAS: This example shows a simple bus implemented without interfaces. Note that the logic type can replace wire and reg if no resolution of multiple drivers is needed.
PROPOSED: This example shows a simple bus implemented without interfaces. The logic type, as used in this example, can replace wire and reg if no resolution of multiple drivers is needed.

20.3
WAS: Note: Because the instantiated interface names do not match the interface names used in the memMod and cpuMod modules, implicit port connections cannot be used for this example.
PROPOSED: In the preceding example, the instantiated interface names do not match the interface names used in the memMod and cpuMod modules; therefore, implicit port connections cannot be used for this example.

20.4 (multiple occurrences)
WAS: The syntax of interface_name.modport_name reference_name gives a local name for a hierarchical reference. Note that this can be generalized to any interface with a given modport name by writing **interface**. modport_name reference_name.
PROPOSED: The syntax of interface_name.modport_name reference_name gives a local name for a hierarchical reference. This technique can be generalized to any interface with a given modport name by writing **interface**. modport_name reference_name.

WAS: Note that if no **modport** is specified in the module header or in the port connection, then all the nets and variables in the interface are accessible with direction **inout** or **ref**, as in the examples above.
PROPOSED: Adding modports to an interface does not require that any of the modports be used when the interface is used. If no **modport** is specified in the module header or in the port connection, then all the nets and variables in the interface are accessible with direction **inout** or **ref**, as in the examples above.

20.6.4

WAS: For a read task, only one module should actively respond to the task call, e.g. the one containing the appropriate address. The tasks in the other modules should return with no effect. Only then should the active task write to the result variables.

Note multiple export of functions is not allowed, because they must always write to the result.

PROPOSED: For a read task, only one module should actively respond to the task call, e.g. the one containing the appropriate address. The tasks in the other modules should return with no effect. Only then should the active task write to the result variables.

Unlike tasks, multiple export of functions is not allowed, because they must always write to the result.

20.8.1

20.8.1 Virtual interfaces and clocking blocks

Clocking blocks and interfaces can be combined to represent the interconnect between synchronous blocks. Moreover, because clocking blocks provide a procedural mechanism to assign values to both nets and variables, they are ideally suited to be used by virtual interfaces. For example:

```
  interface SyncBus( input bit clk );
    wire a, b, c;

    clocking sb @(posedge clk);
      input a;
      output b;
      inout c;
    endclocking
  endinterface

  typedef virtual SyncBus VI; // A virtual interface type

  task do_it( VI v ); //handles any SyncBus via clocking sb
    if( v.sb.a == 1 )
      v.sb.b <= 0;
    else
      v.sb.c <= ##1 1;
endtask
```

WAS: In the preceding example, interface SyncBus includes a clocking block, which is used by task do_it to ensure synchronous access to the interface's signals: a, b, and c. Note that changes to the storage type of the

interface signals (from net to variable and vice-versa) requires no changes to the task. The interfaces can be instantiated as shown below.
PROPOSED: In the preceding example, interface SyncBus includes a clocking block, which is used by task do_it to ensure synchronous access to the interface's signals: a, b, and c. A change to the storage type of the interface signals (from net to variable and vice-versa) requires no changes to the task. The interfaces can be instantiated as shown below.

There is no 20.10.1

21.10
WAS: It is important to note that the cumulative coverage considers the union of all significant bins, thus, it includes the contribution of all bins (including overlapping bins) of all instances.
PROPOSED: It is important to understand that the cumulative coverage considers the union of all significant bins, thus, it includes the contribution of all bins (including overlapping bins) of all instances.

22.2
WAS: Note that function $isunbounded is used for checking the validity of the actual arguments.
PROPOSED: *(This is an informational usage note. No change to the note - except use a smaller font for the note)*

24.17
WAS: Note that the diagram would be identical if one or more of the unpacked dimension declarations were reversed, as in:
reg [31:0] mem [2:0][0:4][8:5]
PROPOSED: The above diagram would be identical if one or more of the unpacked dimension declarations were reversed, as in:
reg [31:0] mem [2:0][0:4][8:5]

25.2

WAS: Note that the current VCD format does not indicate whether a variable has been declared as **signed** or **unsigned**.

PROPOSED: *(This is an informational note. No change to the note - except use a smaller font for the note)*

28.3

Every task or function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every task or function exported from SystemVerilog defines a global symbol. Thus the tasks and functions imported to and exported from SystemVerilog have their own global name space of linkage names, different from compilation-unit scope name space. Global names of imported and exported tasks and functions must be unique (no overloading is allowed) and shall follow C conventions for naming; specifically, such names must start with a letter or underscore, and can be followed by alphanumeric characters or underscores. Exported and imported tasks and functions, however, can be declared with local SystemVerilog names. Import and export declarations allow users to specify a global name for a function in addition to its declared name. Should a global name clash with a SystemVerilog keyword or a reserved name, it shall take the form of an escaped identifier. The leading backslash (\ ) character and the trailing white space shall be stripped off by the SystemVerilog tool to create the linkage identifier. ...

WAS: Note that after this stripping, the linkage identifier so formed must comply with the normal rules for C identifier construction.

PROPOSED: After this stripping, the linkage identifier so formed must comply with the normal rules for C identifier construction.

28.4.1.1

WAS: Note that imported tasks can consume time, similar to native SystemVerilog tasks.

PROPOSED: *(This is an informational note. No change to the note - except use a smaller font for the note)*

28.4.1.4

WAS: NOTE—In this last scenario, a block of memory is allocated and freed in the foreign code, even when the standard functions malloc and free are called directly from SystemVerilog code.

PROPOSED: *(This is an informational note. No change to the note - except use a smaller font for the note)*

28.4.3 (multiple occurrences)
WAS: Only calls of context imported tasks or functions are properly instrumented and cause conservative optimizations; therefore, only those tasks or functions can safely call all tasks or functions from other APIs, including PLI and VPI functions or exported SystemVerilog tasks or functions. For imported tasks or functions not specified as **context**, the effects of calling PLI or VPI functions or SystemVerilog tasks or functions can be unpredictable and such calls can crash if the callee requires a context that has not been properly set. However note that declaring an import context task or function does not automatically make any other simulator interface automatically available. For VPI access (or any other interface access) to be possible, the appropriate implementation defined mechanism must still be used to enable these interface(s). Note also that DPI calls do not automatically create or provide any handles or any special environment that can be needed by those other interfaces. It is the user's responsibility to create, manage or otherwise manipulate the required handles/environment(s) needed by the other interfaces.
PROPOSED: Only calls of context imported tasks or functions are properly instrumented and cause conservative optimizations; therefore, only those tasks or functions can safely call all tasks or functions from other APIs, including PLI and VPI functions or exported SystemVerilog tasks or functions. For imported tasks or functions not specified as **context**, the effects of calling PLI or VPI functions or SystemVerilog tasks or functions can be unpredictable and such calls can crash if the callee requires a context that has not been properly set; however, declaring an import context task or function does not automatically make any other simulator interface automatically available. For VPI access (or any other interface access) to be possible, the appropriate implementation defined mechanism must still be used to enable these interface(s). Realize also that DPI calls do not automatically create or provide any handles or any special environment that can be needed by those other interfaces. It is the user's responsibility to create, manage or otherwise manipulate the required handles/environment(s) needed by the other interfaces.

28.4.4 (multiple occurrences)

An import declaration specifies the task or function name, function result type, and types and directions of formal arguments. It can also provide optional default values for formal arguments. Formal argument names are optional unless argument binding by name is needed. An import declaration can also specify an optional task or function property. Imported functions can have the properties context or pure; imported tasks can have the property **context**.

WAS: Note that an import declaration is equivalent to defining a task or function of that name in the SystemVerilog scope in which the import declaration occurs, and thus multiple imports of the same task or function name into the same scope are forbidden. Note that this declaration scope is particularly important in the case of imported context tasks or functions, see 28.4.3; for non-context imported tasks or functions the declaration scope has no other implications other than defining the visibility of the task or function.

PROPOSED: *(The first "Note that" should be changed to "Since" as shown below, but the second note is an informational note and should be placed in a separate paragraph with smaller font for the note)*

Since an import declaration is equivalent to defining a task or function of that name in the SystemVerilog scope in which the import declaration occurs, and thus multiple imports of the same task or function name into the same scope are forbidden.

Note that this declaration scope is particularly important in the case of imported context tasks or functions, see 28.4.3; for non-context imported tasks or functions the declaration scope has no other implications other than defining the visibility of the task or function.

WAS: ~~Note that~~ multiple declarations of the same imported or exported task or function in different scopes can vary argument names and default values, provided the type compatibility constraints are met.

PROPOSED: It is permitted to have multiple declarations of the same imported or exported task or function in different scopes; therefore, argument names and default values can vary, provided the type compatibility constraints are met.

28.4.6
WAS: — packed one dimensional arrays of type **bit** and **logic**
~~Note however, that~~ every packed type, whatever ~~is~~ its structure, is eventually equivalent to a packed one dimensional array. Therefore practically all packed types are supported, although their internal structure (individual fields of structs, multiple dimensions of arrays) shall be transparent and irrelevant.
PROPOSED: — packed one dimensional arrays of type **bit** and **logic**
Since every packed type, whatever its structure, is eventually equivalent to a packed one dimensional array. Therefore practically all packed types are supported, although their internal structure (individual fields of structs, multiple dimensions of arrays) shall be transparent and irrelevant.

28.6 (multiple occurrences)
WAS: Note that class member functions cannot be exported, but all other SystemVerilog functions can be exported.
PROPOSED: One important restriction exists. Class member functions cannot be exported, but all other SystemVerilog functions can be exported.

WAS: *c_identifier* is optional here. It defaults to *function_identifier*. For rules describing *c_identifier*, see 28.3. ~~Note that all export functions are always context functions.~~ No two functions in the same SystemVerilog scope can be exported with the same explicit or implicit *c_identifier*. The export declaration and the definition of the corresponding SystemVerilog function can occur in any order. Only one export declaration is permitted per SystemVerilog function.
PROPOSED: *c_identifier* is optional here. It defaults to *function_identifier*. For rules describing *c_identifier*, see 28.3. No two functions in the same SystemVerilog scope can be exported with the same explicit or implicit *c_identifier*. The export declaration and the definition of the corresponding SystemVerilog function can occur in any order. Only one export declaration is permitted per SystemVerilog function and all export functions are always context functions.

28.8 (multiple occurrences)
WAS: An imported task or function is said to be in the disabled state when a **disable** statement somewhere in the design targets either it or a parent for disabling. Note that the only way for an imported task or function to enter the disabled state is immediately after the return of a call to an exported task or function. An important aspect of the protocol is that disabled import tasks and functions must programmatically acknowledge that they have been disabled. A task or function can determine that it is in the disabled state by calling the API function svIsDisabledState().
PROPOSED: An imported task or function is said to be in the disabled state when a **disable** statement somewhere in the design targets either it or a parent for disabling. An imported task or function can only enter the disabled state immediately after the return of a call to an exported task or function. An important aspect of the protocol is that disabled import tasks and functions must programmatically acknowledge that they have been disabled. A task or function can determine that it is in the disabled state by calling the API function svIsDisabledState().

WAS: Note that if an exported task itself is the target of a disable, its parent imported task is not considered to be in the disabled state when the exported task returns. In such cases the exported task shall return value 0, and calls to svIsDisabledState() shall return 0 as well.
PROPOSED: If an exported task itself is the target of a disable, its parent imported task is not considered to be in the disabled state when the exported task returns. In such cases the exported task shall return value 0, and calls to svIsDisabledState() shall return 0 as well.

29.3.1
WAS: NOTES
1—As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.
2—Unnamed assertions cannot be found by name.
PROPOSED: IMPORTANT DETAILS
1—As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.
2—Unnamed assertions cannot be found by name.

29.3.2.1 (multiple occurrences)
WAS: Assertions can occur in modules and interfaces: for assertions defined in modules, the instance field in the s_vpi_assertion_info structure shall contain the handle to the appropriate module or interface instance. Note that VPI does not currently define the information model for interfaces and therefore the interface instance handle shall be implementation dependent.
PROPOSED: *(The note is an informational note and should be placed in a separate paragraph with smaller font for the note)*
Assertions can occur in modules and interfaces: for assertions defined in modules, the instance field in the s_vpi_assertion_info structure shall contain the handle to the appropriate module or interface instance.
Note that VPI does not currently define the information model for interfaces and therefore the interface instance handle shall be implementation dependent.

WAS: NOTE: a single call returns all the information for efficiency reasons.
PROPOSED: *(This is an informational note. No change to the note - except use a smaller font for the note)*

29.4.2
WAS: NOTES
1—In a failing transition, there shall always be at least one element in the expression array.
2—Placing a step callback results in the same callback function being invoked for both success and failure steps.
3—The content of
PROPOSED: IMPORTANT DETAILS
1—In a failing transition, there shall always be at least one element in the expression array.
2—Placing a step callback results in the same callback function being invoked for both success and failure steps.
3—The content of

29.5.1
WAS: vpiAssertionSysEnd discard all attempts in progress and disables any further assertions from starting. All assertion callbacks currently installed shall be removed. Note that once this control is issued, no further assertion related actions shall be permitted.
PROPOSED: vpiAssertionSysEnd discard all attempts in progress and disables any further assertions from starting. All assertion callbacks currently installed shall be removed. Once this control is issued, no further assertion related actions shall be permitted.

29.5.2
PROPOSED: Proposal already entered as Mantis #432

30.2.2.1 (multiple occurrences)
WAS: ‘SV_COV_START
If possible, starts collecting coverage information in the specified hierarchy. No effect if coverage is already being collected. Note that coverage is automatically started at the beginning of simulation for all portions of the hierarchy enabled for coverage.
PROPOSED: ‘SV_COV_START
If possible, starts collecting coverage information in the specified hierarchy. No effect if coverage is already being collected. Coverage is automatically started at the beginning of simulation for all portions of the hierarchy enabled for coverage.

WAS: ‘SV_COV_CHECK
Checks if coverage information can be obtained from the specified hierarchy. ~~Note the possibility of having~~ coverage information does imply that coverage is being collected, as the coverage could have been stopped.

PROPOSED: *(Typo?? Did the above note mean to say that "coverage information does **NOT** imply that coverage is being collected?" If not, the Note is very confusing to me)*
‘SV_COV_CHECK
Checks if coverage information can be obtained from the specified hierarchy. The existence of coverage information does not imply that coverage is being collected, as the coverage could have been stopped.

WAS: NOTE—Definition names are represented as strings, whereas instance names are referenced by hierarchical paths. A hierarchical path need not include any . if the path refers to an instance in the current context (i.e., normal Verilog hierarchical path rules apply).
PROPOSED: *(This is an informational note. No change to the note - the note already uses a smaller font)*

30.2.2.2
WAS: NOTE—This value is proportional to the design size and structure, so it also needs to be constant through multiple independent simulations and compilations of the same design, assuming any compilation options do not modify the coverage support or design structure.

PROPOSED: *(This is an informational note. No change to the note - the note already uses a smaller font)*

30.2.2.5
WAS: NOTES
1—The coverage database format is implementation-dependent.
2—Mapping of names to actual directories/files is implementation-dependent. There is no requirement that a coverage name map to any specific set of files or directories.
PROPOSED: DETAILS
1—The coverage database format is implementation-dependent.
2—Mapping of names to actual directories/files is implementation-dependent. There is no requirement that a coverage name map to any specific set of files or directories.

30.4.3 (multiple occurrences)
WAS: Returns the number of times each coverable entity referred by the handle has been covered. Note that this is only easily interpretable when the handle points to a unique coverable item (such as an individual statement); when handle points to an item containing multiple coverable entities (such as a handle to a block statement containing a number of statements), the result is the sum of coverage counts for each of the constituent entities.
PROPOSED: Returns the number of times each coverable entity referred by the handle has been covered. The handle coverage information is only easily interpretable when the handle points to a unique coverable item (such as an individual statement); when handle points to an item containing multiple coverable entities (such as a handle to a block statement containing a number of statements), the result is the sum of coverage counts for each of the constituent entities.

WAS: Returns the number of coverable entities pointed by the handle. Note that this shall always return 1 (one) when applied to an assertion or FSM state handle.
PROPOSED: Returns the number of coverable entities pointed by the handle. The number returned shall always be return 1 (one) when applied to an assertion or FSM state handle.

30.4.4
WAS: Controls the collection of coverage on the given instance or assertion. Note that statement, toggle and FSM coverage are not individually controllable (i.e., they are controllable only at the instance level and not on a per statement/signal/FSM basis).
PROPOSED: Controls the collection of coverage on the given instance or assertion. Statement, toggle and FSM coverage are not individually controllable (i.e., they are controllable only at the instance level and not on a per statement/signal/FSM basis).

31.8.3
WAS: Note that loading the object means loading the object from a database into memory, or marking it for active use if it is already in the memory hierarchy.
PROPOSED: "Loading the object" means loading the object from a database into memory, or marking it for active use if it is already in the memory hierarchy.

31.8.4
WAS: vpiHandle trvsHndl = vpi_handle(vpiTrvsObj, object_handle);
Note that the user (or tool) application can create more than one value change traverse handle for the same object, thus providing different views of the value changes.
PROPOSED: vpiHandle trvsHndl = vpi_handle(vpiTrvsObj, object_handle);
A user (or tool) application can create more than one value change traverse handle for the same object, thus providing different views of the value changes.

31.10
WAS: The SystemVerilog tool the user application is running under is responsible for loading the appropriate extension, i.e. the reader API library in the case of the read API. The extension name is used for this purpose, following a specific policy, for example, this extension name can be the name of the library to be loaded. Once the reader API library is loaded all VPI function calls that wish to use the implementation in the library shall be performed using the returned p_vpi_extension pointer as an indirection to call the function pointers specified in s_vpi_extension or the extended vendor specific structure as described above. Note that, as stated earlier, in the case the application is using the built-in routine implementation (i.e. the ones provided by the tool (e.g. simulator) it is running under) then the de-reference through the pointer is not necessary.
PROPOSED: The SystemVerilog tool the user application is running under is responsible for loading the appropriate extension, i.e. the reader API library in the case of the read API. The extension name is used for this purpose, following a specific policy, for example, this extension name can be the name of the library to be loaded. Once the reader API library is loaded all VPI function calls that wish to use the implementation in the library shall be performed using the returned p_vpi_extension pointer as an indirection to call the function pointers specified in s_vpi_extension or the extended vendor specific structure as described above. As stated earlier, in any case that the application is using the built-in routine implementation (i.e. the ones provided by the tool (e.g. simulator) it is running under) then the de-reference through the pointer is not necessary.

32.2 through 32.49 - many of these diagrams have supporting descriptions included in a normative "NOTES" section at the bottom of each section. Globally change "NOTES" to "DETAILS" and change "NOTE" to "DETAIL" to fix the "informative" problem.

WAS: NOTES:
1) **vpiMemory** shall return array variable objects rather than **vpiMemory** objects. The IEEE P1364 standard has made a similar update to the Verilog VPI (refer to note 1 in P1364, 26.6.9)
PROPOSED: DETAILS
1) **vpiMemory** shall return array variable objects rather than **vpiMemory** objects. The IEEE P1364 standard has made a similar update to the Verilog VPI (refer to note 1 in P1364, 26.6.9)

(Wait to see if the P1364 LRM changes before making a proposal)

... (same global solution until section 32.9)

(32.9 under Notes)
WAS: 2) The **vpiImport** iterator shall return all objects imported into the current scope via import statements. Note that only objects actually referenced through the import shall be returned, rather than items potentially made visible as a result of the import. Refer to 19.2.2 for more details.
PROPOSED: 2) The **vpiImport** iterator shall return all objects imported into the current scope via import statements. Only objects actually referenced through the import shall be returned, rather than items potentially made visible as a result of the import. Refer to 19.2.2 for more details.

... (same global solution until section 32.14)

(32.14 under Notes)
WAS: 19) Note that:
logic var == reg
var bit == reg bit
array var == reg array
PROPOSED: 19) In the above diagram:
logic var == reg
var bit == reg bit
array var == reg array

C.2
WAS: The mailbox class is described in 14.3 and its prototype is:
Note: *dynamic_singular_type* below represents a special type that enables run-time type-checking.
class mailbox
PROPOSED: The mailbox class is described in 14.3 and its prototype is:
The *dynamic_singular_type* below represents a special type that enables run-time type-checking.
class mailbox

E.5
WAS: **E.5 Semantic constraints**
Note that the constraints expressed here merely restate those expressed in 28.4.1.
PROPOSED: *(This is an informational note. No change to the note - except use a smaller font for the note)*

E.5.7
WAS: NOTE—In this last scenario, a block of memory is allocated and freed in C code, even when the standard functions malloc and free are called directly from SystemVerilog code.
PROPOSED: *(This is an informational note. No change to the note)*

E.6.1
WAS: NOTE—The actual argument can have both packed and unpacked parts of an array; either can be multidimensional.
PROPOSED: *(This is an informational note. No change to the note)*

E.6.4
WAS: Note that input mode arguments of type **byte unsigned** and **shortint unsigned** are not equivalent to **bit**[7:0] or **bit**[15:0], respectively, since the former are passed as C types **unsigned char** and **unsigned short** and the latter are both passed by reference as svBitPackedArrRef
PROPOSED: The input mode arguments of type **byte unsigned** and **shortint unsigned** are not equivalent to **bit**[7:0] or **bit**[15:0], respectively, since the former are passed as C types **unsigned char** and **unsigned short** and the latter are both passed by reference as svBitPackedArrRef

---

E.6.6

WAS: 1) If a packed part of an array has more than one dimension, it is linearized as specified by the equivalence of packed types (see E.6.5 and 6.9.3).

2) A packed array of range [L:R] is normalized as [abs(L-R):0]; its most significant bit has a normalized index abs(L-R) and its least significant bit has a normalized index 0.

3) The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range [L:R], the element with SystemVerilog index min(L,R) has the C index 0 and the element with SystemVerilog index max(L,R) has the C index abs(LR).

~~NOTE —~~The above range mapping from SystemVerilog to C applies to calls made in both directions, i.e., SystemVerilog calls to C and C-calls to SystemVerilog.

PROPOSED: 1) If a packed part of an array has more than one dimension, it is linearized as specified by the equivalence of packed types (see E.6.5 and 6.9.3).

2) A packed array of range [L:R] is normalized as [abs(L-R):0]; its most significant bit has a normalized index abs(L-R) and its least significant bit has a normalized index 0.

3) The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range [L:R], the element with SystemVerilog index min(L,R) has the C index 0 and the element with SystemVerilog index max(L,R) has the C index abs(LR).

The above range mapping from SystemVerilog to C applies to calls made in both directions, i.e., SystemVerilog calls to C and C-calls to SystemVerilog.

E.8.1

WAS: Note that all DPI export tasks and functions require that the context of their call is known.

PROPOSED: All DPI export tasks and functions require that the context of their call is known.

E.8.2

WAS: Note that context is transitive through imported and export context tasks and functions declared in the same scope. That is, if an imported task or function is running in a certain context, and if it in turn calls an exported task or function that is available in the same context, the exported task or function can be called without any use of svSetScope(). For example, consider a SystemVerilog call to a native function f(), which in turn calls a native function g(). Now replace the native function f() with an equivalent imported context C function, f'(). The system shall behave identically regardless if f() or f'() is in the call chain above g(). g() has the proper execution context in both cases.

(Modified per email from Doug Warmke - 4/18/2005 - 04:26 PM)

PROPOSED: The context property is transitive through imported and export context tasks and functions declared in the same scope. That is, if an imported task or function is running in a certain context, and if it in turn calls an exported task or function that is available in the same context, the exported task or function can be called without any use of svSetScope(). For example, consider a SystemVerilog call to a native function f(), which in turn calls a native function g(). Now replace the native function f() with an equivalent imported context C function, f'(). The system shall behave identically regardless if f() or f'() is in the call chain above g(). g() has the proper execution context in both cases.

E.8.3 (multiple occurrences)
WAS: To achieve shared data storage, a related set of context imported tasks and functions should all use the same user-Key. To achieve unique data storage, a context import task or function should use a unique key. Note that it is a requirement on the user that such a key be truly unique from all other keys that could possibly be used by C code.
...
PROPOSED: To achieve shared data storage, a related set of context imported tasks and functions should all use the same user-Key. To achieve unique data storage, a context import task or function should use a unique key and it is a requirement on the user that such a key be truly unique from all other keys that could possibly be used by C code.
...

WAS: Note that it is never possible to share user data storage across different contexts. For example, if a Verilog module m declares a context imported task or function f, and m is instantiated more than once in the SystemVerilog design, then f shall execute under different values of svScope.
PROPOSED: It is never possible to share user data storage across different contexts. For example, if a Verilog module m declares a context imported task or function f, and m is instantiated more than once in the SystemVerilog design, then f shall execute under different values of svScope.

E.8.5
WAS: There is no specific relationship defined between DPI and the existing Verilog programming interfaces (VPI and PLI). Programmers must make no assumptions about how DPI and the other interfaces interact. In particular, note that a vpiHandle is not equivalent to an svOpenArrayHandle, and the two must not be interchanged and passed between functions defined in two different interface standards.
PROPOSED: There is no specific relationship defined between DPI and the existing Verilog programming interfaces (VPI and PLI). Programmers must make no assumptions about how DPI and the other interfaces interact. For example, a vpiHandle is not equivalent to an svOpenArrayHandle, and the two must not be interchanged and passed between functions defined in two different interface standards.

E.11
WAS: Formal arguments specified as open arrays allows passing actual arguments of different sizes (i.e., different range and/or different number of elements), which facilitates writing more general C code that can handle SystemVerilog arrays of different sizes. The elements of an open array can be accessed in C by using the same range of indices and the same indexing as in SystemVerilog. Plus, inquiries about the dimensions and the original boundaries of SystemVerilog actual arguments are supported for open arrays.
NOTE—Both packed and unpacked array dimensions can be unsized.
PROPOSED: *(This is an informational note. No change to the note)*

E.11.1
WAS: If a formal argument is specified as a sized array, then it shall be passed by reference, with no overhead, and is directly accessible as a normalized array. If a formal argument is specified as an open (unsized) array, then it shall be passed by handle, with some overhead, and is mostly indirectly accessible, again with some overhead, although it retains the original argument boundaries.
NOTE—This provides some degree of flexibility and allows the programmer to control the trade-off of performance vs. convenience.
PROPOSED: *(This is an informational note. No change to the note)*

E.11.4
WAS: If the actual layout of the SystemVerilog array passed as an argument for an open unpacked array is different than the C layout, then it is not possible to access such an array as a whole; therefore, the address and size of such an array shall be undefined (zero (0), to be exact). Nonetheless, the addresses of individual elements of an array shall be always supported.
NOTE—No specific representation of an array is assumed here; hence, all functions use a generic pointer void *.
PROPOSED: If the actual layout of the SystemVerilog array passed as an argument for an open unpacked array is different than the C layout, then it is not possible to access such an array as a whole; therefore, the address and size of such an array shall be undefined (zero (0), to be exact). Nonetheless, the addresses of individual elements of an array shall be always supported.
PROPOSED: *(This is an informational note. No change to the note)*

G
WAS: — An user might want to switch between selections or provide additional code. This-use case is covered by providing a set of tool switches to define the corresponding information, although it might also use the bootstrap file approach.
NOTE—This annex defines a set of switch names to be used for a particular functionality.
PROPOSED: *(This is an informational note. No change to the note)*

G.2
WAS: The following conditions also apply.
— The compiled object code itself shall be provided in form of a shared library having the appropriate extension for the actual platform.
NOTE—Shared libraries use, for example, .so for Solaris and .sl for HP-UX; other operating systems might use different extensions. In any case, the SystemVerilog application needs to identify the appropriate extension.
PROPOSED: *(This is an informational note. No change to the note)*

H.3
WAS: The semantics of assertions and properties is defined via a relation of satisfaction by empty, finite, and infinite words over the alphabet $\Sigma = 2^{\mathbf{P}} \cup \{\top, \square\}$. Such a word is an empty, finite, or infinite sequence of elements of Σ. The number of elements in the sequence is called the *length* of the word, and the length of word *w* is denoted |*w*|. Note that |*w*| is either a non-negative integer or infinity.
PROPOSED: The semantics of assertions and properties is defined via a relation of satisfaction by empty, finite, and infinite words over the alphabet $\Sigma = 2^{\mathbf{P}} \cup \{\top, \square\}$. Such a word is an empty, finite, or infinite sequence of elements of Σ. The number of elements in the sequence is called the *length* of the word, and the length of word *w* is denoted |*w*|, where |*w*| is either a non-negative integer or infinity.
