General

Sections 4.6.1 new[], 4.8 Arrays as arguments, 10.5.2 Pass by reference, 10.5.4 Argument passing by name, 12.6 In-line constraints - randomize() with, 13.3.5 get(), 13.4 Parameterized mailboxes, 13.8 $wait_var(), 15.2 Clocking domain declaration, 15.3 Input and output skews, 18.7 Module instances

Editor's Note: "parameter" is a Verilog keyword, and "parameterized" models refer to the usage of Verilog "parameters" (see Sections 11.21, 19.6 and 20). Use of the word "parameterized" in this context is not consistent with the Verilog LRM. Suggest using "arguments" (as in Verilog LRM), "formal arguments" or "formals".

Section 2.5 Time Literals

The time literal is interpreted as a **realtime** value scaled to the current time unit and rounded to the current time precision. Note that if a time literal is used as an actual parameter to a module or interface instance, the current time unit and precision are those of the module or interface instance.

Editor's Note: What is meant by "actual parameter" in the preceding paragraph? Is it referring to the Verilog parameter data type?

Section 3.4 Time data types

Editor's Note: BC08-05 says to "Remove section 3.4.1". There is no such section, and the change order does not have the requisite details of which version it referred to, or what text is to be deleted.

Section 3.8.9 atoi(), atohex(), atooct(), atobin()

function integer atoi()
function integer atohex()
function integer atooct()
function integer atobin()

Editor's Note: "integer" or "int"?

Section 3.10 User defined types

A **typedef** inside a **generate** may not define the actual type of a forward definition that exists outside the scope of the forward definition.

Editor's Note: Does "may not" in the preceding paragraph indicate a mandatory rule or an optional rule? If mandatory, then change to "shall". If optional, the sentence needs to be rephrased to not be ambiguous.

Section 7.12 Unpacked array expressions, 7.13 Structure expressions, 7.14 Aggregate expressions, 7.15 Conditional operator

Editor's Note: Both BC62a and BC65 added this new section. I used BC62a.

Section 8.10 Nonblocking event trigger

Editor’s Note: This new operator needs to be added to the operator precedence table and other sections describing operator rules (signed/unsigned/2-state/4-state/real operands, etc.).

Section 10 Tasks and Functions

— Importing and exporting functions through the Direct Programming Interface (DPI)

Editor’s Note: Previous bullet added by the editor. It was not actually specified in EC-C120.

Section 10.6 Import and Export Functions

For any given *cname*, all declarations, regardless of scope, must have exactly the same type signature. The type signature includes the return type, the number, order, direction and types of each and every argument. Type includes dimensions and bounds of any arrays/array dimensions. For **import** declarations, arguments can be open arrays. Open arrays are defined in Section 1.4.5 of the DPI LRM. Signature also includes the **pure**/**context** qualifiers that may can be associated with an **import** definition.

Editor’s Note: What is meant by "Signature"?

Section 11.20 Class scope resolution operator::

Editor’s Note: This new operator needs to be added to the operator precedence table and other sections describing operator rules (signed/unsigned/2-state/4-state/real operands, etc.).

Section 11.22 Parameterized classes

```
class vector #(parameter int size = 1;);
        bit [size-1:0] a;
        static int count = 0;
        function void disp_count();
                $display( "count: %d of size %d", count, size );
        endfunction
endclass
```

Editor’s Note: Verilog syntax is "int static". Is the "static int" above correct? NOTE: The Co-design SystemSim simulator allows both forms. I submitted a request to the BC committee to clarify if SystemVerilog was intended to also allow both forms. I do not know the result of that request.

Section 12.4.3 Set membership

inside

Editor’s Note: This new operator needs to be added to the operator precedence table and other sections describing operator rules (signed/unsigned/2-state/4-state/real operands, etc.).

Section 12.4.4 Distribution

dist

Editor’s Note: This new operator needs to be added to the operator precedence table and other sections describing operator rules (signed/unsigned/2-state/4-state/real operands, etc.).

The **:=** operator assigns the specified weight to the item, or if the item is a range, to every value in the range.

The **:/** operator assigns the specified weight to the item, or if the item is a range, to the range as a whole. Ifthere are `n` values in the range, the weight of each value is `range_weight / n`.

Editor’s Note: These new operators need to be added to the operator precedence table and other sections describing operator rules (signed/unsigned/2-state/4-state/real operands, etc.).

Section 12.4.5 Implication

`=>`

Editor’s Note: This new operator needs to be added to the operator precedence table and other sections describing operator rules (signed/unsigned/2-state/4-state/real operands, etc.).

Section 12.4.9 Static constraint blocks

Editor’s Note: Verilog syntax puts "static" after the data type, the opposite of C. Should the declaration for constraint be consistent with Verilog, or with C? Note that the Co-design SystemSim simulator allows the static declaration of SystemVerilog variables to be in either order. I submitted a request to the BC committee asking if the SystemVerilog LRM was intended to allow the same. I do not know the result of that request.

Section 12.10.1 $urandom

function unsigned int $urandom `[ (int` *seed* `) ] ;`

Editor’s Note: Verilog syntax is "function int unsigned" instead of "function unsigned int". Co-design’s SystemSim allows both Verilog and C styles.

Section 12.10.2 $urandom_range

function unsigned int $urandom_range`(` **unsigned int** *maxval*`,` **unsigned int** *minval* `= 0 );`

Editor’s Note: Verilog syntax is “function int unsigned” instead of “function unsigned int” ?

Section 12.10.3 $srandom

task $srandom`(` **int** *seed*`, [object` *obj*`] );`

Editor’s Note: Is “object” a data type? There is no keyword “object” anywhere else in the LRM.

Section 14.4 The PLI callback control points

There are two kinds of PLI callbacks, those that are executed immediately when some object changes value in an update event, and those that are explicitly registered as a one-shot evaluation event.

Editor's Note: The preceding paragraph does not account for all types of callbacks. For example: cbStmt, cbEnterInteractive, cbStartOfReset, etc. These and some other callbacks are not logic value related, and they may occur more than one time after being registered.

Section 18.7.3 Instantiation using implicit .name port connections

Editor's Note: BC42-24 said to make ".name" all bold. This was not done, because the bold text is used to designate a keyword, and "name" is not a keyword.

Section 18.8.3 Port connection rules for interfaces

See Section XX for more port connection rules with interfaces.

Editor's Note: What is the cross reference for above?

Section 22.6 Assertion control system tasks, 22.7 Assertion system functions

Editor's Note: Are these system tasks to be removed? The new 3.1 assertion section does not mention them.

Section 26.4.4 Import declarations

```
import "DPI" newQueue=function handle newAnonQueue(input string s=NULL);
```

Editor's Note: Is the uppercase "NULL" correct? The SystemVerilog keyword is in lowercase.

Section 26.4.6.1 Open arrays

Here are examples of types of formal arguments (empty square brackets `[]` denote open array):

```
logic
bit [8:1]
bit[]
bit [7:0] b8x10 [1:10] // b8x10 is a formal arg name
logic [31:0] l32x [] // l32x is a formal arg name
logic [] lx3 [3:1] // lx3 is a formal arg name
bit [] an_unsized_array [] // an_unsized_array is a formal arg name
```

Editor's Note: It is illegal in Verilog to start a name with a number (e.g. "132x". Does that rule apply here?

Section 27.1.2 Nomenclature

Assertion Temporal expression—A declarative expression (one or more clock cycles) describing the behavior of a system over time. // This is the "body" of the assertion.

Editor's Note: Why the underlined comment, above? Should it be regular text, or deleted?

Section 27.2 Extensions to VPI enumerations

These extension shall be merged into the contents of the `vpi_user.h` file, described in *IEEE Std 1364-2001*, Annex G. The numbers in the range `700 - 799` are reserved for the assertion portion of the VPI.

Editor's Note: Is "shall", which means mandatory, too strong here? It seems to infer that users or software vendors must modify the IEEE standard vpi_user.h header file, thereby making the file non IEEE compliant.

Section 27.2.3 Callbacks

27.2.2 Object properties

This section lists the object property VPI calls. The VPI reserved range for these call is `700 - 729`.

```
/* Directives as properties */
#define vpiSequenceAssertion 701
#define vpiAssertAssertion 702
#define vpiAssumeAssertion 703
#define vpiRestrictAssertion 704
#define vpiCoverAssertion 705
#define vpiCheckAssertion 705 /* inlined behavior assertion */
#define vpiOtherDirectiveAssertion 706 /* placeholder for other
assertion directive */
```

27.2.3 Callbacks

This section lists the system callbacks. The VPI reserved range for these call is `700 - 719`.

1) Assertion

```
#define cbAssertionStart 700
#define cbAssertionSuccess 701
#define cbAssertionFailure 702
#define cbAssertionStepSuccess 703
#define cbAssertionStepFailure 704
#define cbAssertionDisable 705
#define cbAssertionEnable 706
#define cbAssertionReset 707
#define cbAssertionKill 708
```

Editor's Note: This section is using some of the same constant values as the previous section.

Section 27.3.2 Obtaining static assertion information

Any assertion updates from the *SV-AC*.

— Assertion source information: the file, line, and column where the assertion is defined.

— Assertion clocking domain/expression2

Editor's Note: Item 6, above, does not seem appropriate in a standard. Should it be deleted?
Editor's Note: Are the two dashed-list lines above part of item 6?
Editor's Note: What is the "2" in "expression2", above?

Section 27.4.1 Placing assertion “system” callbacks

Editor’s Note: Why is “system” in quotes?.

Section 27.4.2 Placing assertions callbacks

`cbAssertionStepSucess`
the progress of one “thread” along an attempt. By default, step callbacks are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis, rather than on a per-assertion basis.

Editor’s Note: Why is “thread” in quotes?.

Section 27.5.2 Assertion control

For the following operator, the second argument shall be a valid assertion handle, the third argument shall be an attempt start-time (as a pointer to a correctly initialized `s_vpi_time` structure) and the fourth argument shall be a “step control” constant.

Editor’s Note: Why is “step control” in quotes?.

Section 28.2 System Verilog real-time coverage access

This section ...

Editor’s Note: Something appears to be missing in this section.

Section 28.2.2 Built-in coverage access system functions

This section ...

Editor’s Note: Something appears to be missing in this section.

Section 28.3.1 Specifying the signal that holds the current state

```
/* tool state_vector signal_name */
```

where `tool` and `state_vector` are required keywords. This pragma needs to be specified inside the module definition where the signal is declared.

Editor’s Note: Throughout this coverage API section, Verilog-2001 attributes should be used, instead of using obsolete pragmas that are hidden in comments!

Section 28.3.6 Specifying the possible states of the FSM

```
parameter [1:0] /* tool enum enumeration_name */
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

Editor’s Note: Can SystemVerilog enumerated types be used instead of parameter constants? What about ‘define macros

Section 28.4 VPI coverage extension, 28.4.1 VPI entity/relation diagrams related to coverage

28.4 VPI coverage extensions

This section ...

Editor's Note: Something appears to be missing in this section.

28.4.1 VPI entity/relation diagrams related to coverage

This section ...

Editor's Note: Something appears to be missing in this section.

Section 28.4.3 Obtaining coverage information

All **what?? use `vpi_get()` along with the appropriate properties and object handles.

Editor's Note: The preceding sentence needs to be fixed.

Section 28.4.4 Controlling coverage

28.4.4 Controlling coverage
Revise similar to Assertions

Editor's Note: What is this comment referring to?

Annex D.1 Overview

Formal arguments in SystemVerilog can be specified as open arrays solely in import declarations; exported SystemVerilog functions can not have formal arguments specified as open arrays. A formal argument is an open array when a range of one or more of its dimensions is unspecified (denoted in SystemVerilog by using empty square brackets (`[]`)). This corresponds to a relaxation of the DPI argument-matching rules (section 1.5.1). An actual argument shall match the corresponding formal argument regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized C code that can handle SystemVerilog arrays of different sizes.

Editor's Note: What is the correct cross reference, above?

Annex D.5 Semantic constraints

Note that the constraints expressed here merely restate those expressed in section 1.4.1.

Editor's Note: What is the correct cross reference, above?

Annex D.5.5 context and non-context functions

Also refer to section 1.4.3.

Editor’s Note: What is the correct cross reference, above?

Annex D.5.6 pure functions

See also 1.4.2.

Editor’s Note: What is the correct cross reference, above?

Annex D.5.7 Memory management

See also section 1.4.1.4.

Editor’s Note: What is the correct cross reference, above?

Annex D.1.2 Mapping between SystemVerilog ranges and normalized ranges

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 section ??).

Editor’s Note: What is the correct cross reference, above?

Annex D.3.2 Calling SystemVerilog functions from C

It can be done while preserving the binary compatibility, see Annex D.7.5 and section A.11.11.

Editor’s Note: What is the correct cross reference, above?

Annex D.3.5 Allocating actual arguments for SystemVerilog-specific types

compromising the portability (see section A.11.11). Such a technique does not work if a packed array is a part of another type.

Editor’s Note: What is the correct cross reference, above?

Annex D.3.7 input arguments

[There is a problem here: ‘int’ is the same as svBitVec32, long long is not the snae as svBitVect32[2], so how to return a value in the canonical representation as a function result, if this value is between 33 and 64 bits?]

Editor’s Note: Has the preceding note been taken care of?

Annex D.4 Context functions

A small set of DPI utility functions is available to assist programmers when working with context functions (see section A.8.3). If those utility functions are used with any non-context function, a system error will result.

Editor's Note: Has the preceding note been taken care of?

Annex D.6.2 DIrect access to unpacked arrays

Unpacked arrays, with the exception of formal arguments specified as open arrays, shall have the same layout as used by a C compiler; they are accessed using C indexing (see section A.6.6).

Editor's Note: What is the correct cross reference, above?

Annex D.7.1 Actual ranges

In the former case, all indices are normalized on the C side (i.e., `0` and up) and the programmer needs to know the size of an array and be capable of determining how the ranges of the actual argument map onto C-style ranges (see section A.6.6).

Editor's Note: What is the correct cross reference, above?