Clarification to Testbench Donation SytemVerilog 3.1

Addendum to Version 1.1
September 4, 2002

Table of Contents

Clarification to Testbench Donation
SystemVerilog 3.1

1 Introduction

VeraLite is a subset of verification constructs from the Vera language that has been submitted to the Accellera committee to become part of the test-bench extensions of SystemVerilog (3.1). Vera was initially designed as a set of enhancements to Verilog 1.0 (1995), thus, the lexical and syntactical elements of both languages are the same. While many of the test-bench constructs are unique to Vera, the semantics of the constructs common to both languages, including data-types and operators, remain largely the same. This makes VeraLite an ideal candidate for inclusion into SystemVerilog. Nonetheless, Vera, since its inception, has evolved as a separate language. Verilog too has evolved, first into the Verilog –2001 standard and more recently into SystemVerilog 3.0. As such, VeraLite and SystemVerilog exhibit several conflicts and areas of functional overlap. The purpose of this document is to identify areas of conflict between SystemVerilog and VeraLite, offer resolution to those conflicts, and suggest improvements that will simplify the resulting language. Since VeraLite is based largely on Verilog-1995, the first set of conflicts is resolved by updating Vera to be compatible with Verilog-2001. This will allow VeraLite to become a natural extension to SystemVerilog, an important point that is not stated in the donation.

1.1 Document Format

The following section provides a high level overview of the two languages, VeraLite and SystemVerilog. The rest of this document is organized in the same order as the VeraLite donation to Accellera. It is intended to be read as a companion to the original donation. We chose this format so that this document can be added to the original donation as an appendix, rather than as a re-write. This document provides a set of brief clarifications and recommendations in the spirit of cooperation with the committee in order to expedite the task before them. This document doesn't address or attempt to solve every inconsistency or problem. It merely resolves the most salient problems, leaving the more detailed issues to be worked out by the committee.

Some of the issues addressed by this document arise because the donation was derived from an earlier version of the VeraLite manual that was not in strict LRM form, and contains mistakes, omissions, and incomplete semantic detail. The donation is also missing several sections that were clearly intended to be included. This document addresses the confusion caused by references to missing items in the original donation by attaching the relevant sections. At the end of each section heading, a cross-reference to the corresponding pages in the donation is enclosed in parentheses.

Missing sections that have been attached are shown with a right-hand side bar, like this.

2 Languages Overview

2.1 SystemVerilog 3.0

SystemVerilog enhances Verilog for designers in the following broad areas:

- **Interfaces:** High level abstractions for module connections.
- **Enhanced Hierarchy:** Global declarations and statements (implicit top-level **$root**), nested modules for better encapsulation, and unnamed blocks with data declarations.
- **Enhanced Time Unit and Precision:** Physical time units (ns, ps,..) and precision can be specified in any module.
- **Abstract Data Types:** Predefined 2-state and 4-state data-types for easier modeling.
  - **char, int, shortint, longint, byte, bit, logic,** and **shortreal**
  - Type casting
  - **User Defined Types:**
    - **typedef** of user defined types
    - **enum, struct,** and **union**
- **Enhanced Arrays:** Packed and unpacked multidimensional arrays.
- **Dynamic Processes:** Enables multithreaded process creation
- **Enhanced Sequential Flow Control:** C-like loops and jump statements
  - **break, continue, return, do …while**
  - Additional C-like compound operators: **++, --, +=, -=, /=, *=, %=, …**
- **New Procedures:** Procedures that explicitly indicate the intent of the logic
  - **always_comb, always_ff, always_latch**

2.2 VeraLite

VeraLite enhances SystemVerilog in the following important areas:

- **Test-bench Functions:** Reusable, reactive test-bench data-types and functions.
- **Synchronization**: Mechanisms for process creation, control, and inter-process communication.
- **Classes:** Object-Oriented mechanism that provides abstraction, encapsulation, and safe *pointer* capabilities.
- **Dynamic Memory**: Automatic memory management in a re-entrant environment using a garbage collection mechanism that frees users from explicit de-allocation.
- **Cycle-Based Functionality:** Clocking domains and cycle-based attributes that help reduce development, ease maintainability, and promote reusability.

In addition, VeraLite provides the following aspects useful for test-benches:

- Predefined abstract data-types: **string, event**
- User defined data-types: **enum, class, associative-array**
- Enhanced Dynamic Process: **fork … join {all|any|none}**
- Process Synchronization: Semaphore, Mailbox, **wait_var(), wait_child()**
- Clocking domains and Associated Signal Drives and Expects

Note that VeraLite constructs are applicable only in the behavioral context. The usage of VeraLite makes sense and should be allowed only in initial or always blocks.

3 VeraLite: The Language

3.1 Lexical Elements

VeraLite lexical conventions are the same as in SystemVerilog. There are no known conflicts.

3.2 VeraLite Keywords (1-2)

Clarification: VeraLite recognizes the keywords shown in the table below. Keywords unique to VeraLite (not in SystemVerilog) are shown in **boldface**. Keywords that conflict with SystemVerilog are shown in **boldface and underlined**.

all	end	**interface**	repeat
any	**endclass**	join	**return**
async	**endprogram**	**local**	**static**
begin	**enum**	negedge	**string**
bit	**event**	**new**	**super**
break	**extends**	**none**	task
case	**extern**	**null**	**this**
casex	for	or	**typedef**
casez	fork	output	**var**
class	function	posedge	**void**
CLOCK	if	program	**virtual**
continue	inout	**protected**	while
default	input	**public**	
else	integer	reg	

3.2.1 VeraLite Predefined Constants

VeraLite introduces several predefined constants. The table below lists the predefined constant identifiers.

ALL	DELETE	NO_WAIT	ORDER
ANY	FIRST	OFF	SEMAPHORE
CHECK	HAND_SHAKE	ON	WAIT
COPY_NO_WAIT	MAILBOX	ONE_BLAST	
COPY_WAIT	NEXT	ONE_SHOT	

These predefined constants are defined using the following enumerated types:

```
enum TriggerModes { OFF, ON, ONE_SHOT, ONE_BLAST, HAND_SHAKE };
enum CheckMode { CHECK = 0 };
enum SyncModes { ALL = 1, ANY, ORDER };
```

```
enum AssocIdxModes { FIRST = 1, NEXT, DELETE };
enum MailboxModes { NO_WAIT, WAIT, COPY_NO_WAIT, COPY_WAIT };
enum AllocTypes { SEMAPHORE, MAILBOX };
```

3.2.2 VeraLite Predefined Clocking Identifiers

VeraLite recognizes four signal clocking identifiers. These are neither keywords nor predefined constants, but constant identifiers recognized only within the parsing context of a clocking-domain (see Section 7.1). The predefined clocking identifiers are:

```
NHOLD          NSAMPLE          PHOLD          PSAMPLE
```

3.3 Statement Blocks (1-3)

Conflict: VeraLite uses braces '**{**' and '**}**' to denote the start and end of a block. This includes execute blocks as well as declaration blocks (class, interface, end enum).[1] SystemVerilog uses **begin** and **end** for execute blocks, and some declarations have an implicit begin and a specialized end: **module … endmodule, task … endtask,** etc….
Resolution: VeraLite will adopt **begin** and **end** for execute blocks, and SystemVerilog syntax for **task** and **function** declarations. In addition, the syntax for **class** and **program** declarations will be changed in a manner consistent with SystemVerilog, as shown in the table below[2]:

Old Syntax	New Syntax
class *name* **{ . . . }**	**class** *name* **. . . endclass**
program *name* **{ . . . }**	**program** *name* **. . . endprogram**

Braces will continue to be used, but only for those constructs in which their use is consistent with SystemVerilog, these are: the concatenation and replication operators, array initialization, declaration of enumerated data-types, and declaration of clocking domains.

3.4 Strings (1-4)

In SystemVerilog string literals (character strings enclosed by double quotes) behave like packed arrays (of a width that is a multiple of 8 bits). In VeraLite a string literal behaves the same way. Unlike SystemVerilog, however, VeraLite also has the string data type to which a string literal can be assigned. In SystemVerilog, a string literal assigned to a packed array is truncated to the size of the array, whereas in VeraLite, the strings can be of arbitrary length and no truncation occurs. This does not represent a conflict, but, the donation doesn't differentiate between string (a VeraLite data-type) and string literal.

[1] By allowing braces '{' and '}' to denote blocks VeraLite creates a syntactical conflict with Verilog: The left-hand-side concatenation operator is ambiguous with the start of a block and was thus renamed `{}.

[2] The additional keywords needed for this change are already included in the table of Section 3.2.

Clarification: string literals behave exactly as in SystemVerilog, except that they are implicitly converted to the string type when assigned to a string type or used in an expression involving string type operands (see Section 3.18).

Limitation: VeraLite only accepts the 'C' notation '*ddd*' to denote an ascii character (where *d* is any octal digit). SystemVerilog also allows the notation '\x*hh*' to denote an ascii character (where *h* is a hexadecimal digit).
Resolution: VeraLite will accept both character notations.

3.5 Numbers (1-5)

Limitation: In SystemVerilog, a numerical constant that doesn't specify the size extends the leftmost digit to fill the variable it is assigned to (bit [2:0] = '1 is extended to 3'b111). VeraLite doesn't handle this type of extension.
Resolution: Remove this limitation. VeraLite will extend un-sized constants to the size of the left-hand side operand.

3.6 Data Types and Variable Declaration (1-6)

Clarification: The donation lists virtual port as one of the user-defined types. This is an error, the virtual port is not supported by VeraLite. Please disregard the section 3.6 of the VeraLite donation.

3.7 Standard Data Types

3.7.1 Integer (1-7)

Correction: The donation states "The upper limit for integer sizes is dependent on the host machine…". That statement is incorrect. **integer** is a 32-bit signed data-type on all implementations. This is the same as SystemVerilog.
Clarification: The donation doesn't specify the default value for an uninitialized integer. The default value is all X's, i.e., 32'bX
Conflict: A VeraLite integer does not allow setting individual bits to X or Z. Assigning any one bit to X or Z causes the entire integer to become X. SystemVerilog does allow setting individual bits of an integer to X or Z.
Resolution: VeraLite integer will behave exactly like a SystemVerilog integer.

3.7.2 bit (1-7)

Conflict: In VeraLite **bit** is a 4-state data-type for vectors of user-defined size. In SystemVerilog, **bit** is a 2-state data-type for vectors of user-defined size. VeraLite's **bit** data-type is equivalent to a SystemVerilog **reg** data-type.
Resolution: VeraLite will use **reg** instead of **bit**.

In addition, VeraLite will accept all the SystemVerilog types (**char, shortint, int, byte, longint, bit, logic**) plus the 2 Verilog-2001 types already supported (**reg** and **integer**). The remainder of the VeraLite documentation shall be amended to consider **bit** as **reg**.

Limitation: VeraLite (like Verilog-1995) only allows one-dimensional bit vectors while SystemVerilog allows multi-dimensional bit-vectors (packed arrays).
Resolution: Remove this limitation. VeraLite will support multidimensional packed arrays. Note that anywhere VeraLite accepts a numerical value, it will also accept a multi-dimensional packed array.

Limitation: Packed arrays in SystemVerilog can be specified using arbitrary indices, as in: **reg [MSB:LSB].** VeraLite bit-vectors can only be specified as **reg [MSB:0] (**LSB must be 0).
Proposal: Remove this limitation. VeraLite will accept arbitrary indices.

3.7.3 String (1-8)

Clarification: The donation doesn't specify the default value for an uninitialized string. The default value is the special constant **null,** which denotes an uninitialized string. Note that **null** is not the same as **"",** which is a zero-length string or empty string.

3.8 User-Defined Data Types

3.8.1 Enumerated types (1-10)

Conflict: The VeraLite syntax for declaring an enum is different from SystemVerilog's. VeraLite uses

enum *name*{ … };

which always creates an enum type called *name*, whereas SystemVerilog uses

enum { … } *vname*;

and only creates a type when used with a typedef construct:

typedef enum { … } *name*;

Resolution: VeraLite will support the SystemVerilog syntax. Also allow the VeraLite syntax as a shorthand for the SystemVerilog typedef form, in the same manner as C++ extends C. This enhancement requires the compiler to distinguish a new type name from an exiting type or size specification (see next point).

Conflict: In VeraLite all enumerated values are represented by 2-state integer values (like in C/C++). SystemVerilog allows the size and type (2-state/4-state) to be specified.
Resolution: Remove this limitation. VeraLite will support SystemVerilog enums. For unspecified enumerated types, the default is the same (2-state integer).

Conflict: VeraLite defines the `++, --, +=,` and `-=` operators to iterate over the enumerated values, SystemVerilog doesn't specify what thoee operators do when applied to an enum.
Resolution: SystemVerilog should adopt the VeraLite operator semantics.
VeraLite enumerated types are strongly typed, thus, a variable of type enum cannot be assigned a value that lies outside the enumeration set. This is a powerful type-checking aid that prevents users from accidentally assigning nonexistent values to variables of an enumerate type. This restriction only applies to an enumeration that is explicitly declared as a type. The enumeration values can still be used as constants in expressions, and the results can be assigned to any variable of a compatible integral type. For example:

```
typedef enum { RED, GREEN, BLUE } Color;
Color   co;
int     i;

co = RED;

co = RED + 3;        // not allowed
i = RED +3;          // allowed
if ( i > BLUE ) begin $display( "error" ); end    // allowed
```

Since enum types are strongly typed, the only two options available are to either disallow the ++ and -- operators (as in C++), or to attach a semantic meaning that is consistent with a strongly-typed enum type. VeraLite opts for the latter, providing a useful feature to iterate over the enumeration values.

Clarification: The donation doesn't specify the default value for an uninitialized enum. The default value is the value of the first element in the enumeration.

3.8.2 Arrays (1-11)

Correction: The examples list an array of **port_name** . This is an error. Please disregard.

Conflict: VeraLite abides by Verilog-1995 rules and hence explicitly disallows slicing of an array element (donation example showing memory[42][3:2]). SystemVerilog allows slicing of array elements.
Resolution: Remove this limitation. Allow VeraLite to slice array elements.

Conflict: SystemVerilog allows unpacked arrays to be sliced by any arbitrary number of dimensions. VeraLite does not allow unpacked array slices:

```
integer a_arr[10:0], b_arr[1:0];
b_arr = a_arr[2:1];
```

Resolution: Remove this limitation. VeraLite will support both array slicing, and allow arrays to be used on the left-hand-side of assignments (L-values).

3.8.3 Array Initialization (1-12)

Conflict: VeraLite does not allow concatenation or replication in array initialization. For example, the following is not allowed:

```
integer a[2][2] = {1,2, {2'b10,2'h01}, 3 };
```

This restriction arises because in VeraLite (like in C/C++) the braces used to group the elements of a dimension are optional. For example, the following are all valid in VeraLite:

```
integer a[2][2] = {1, 2, 3, 4 };
integer a[2][2] = {{1, 2}, 3, 4 };
integer a[2][2] = {{1, 2}, {3, 4} };
```

Since the braces are optional, they become ambiguous in this context; they can either be a concatenation or a new array dimension. Basically, this grammar construct is not context-free and the compiler cannot parse this grammar, thus it's disallowed in VeraLite.
In SystemVerilog all the dimensional braces are required, which does not remove the grammatical ambiguity, but does allow the compiler to distinguish concatenation from dimensional brackets.
Resolution: Remove this limitation . VeraLite will require that all dimensional braces be specified, like in SystemVerilog. This will remove the ambiguity and allow replication or concatenation as part of array initialization.

Correction: The donation states that "you cannot initialize an array in the declaration". This is an error. It is allowed.

3.8.4 Multi-dimensional Arrays (1-13)

Conflict: In VeraLite, array dimensions are specified like in 'C', with a single number denoting the number of elements in a given dimension. SystemVerilog uses the more general indexing notation **[msb:lsb]** that is also used for packed arrays (bit fields).
Resolution: Remove this limitation. VeraLite will accept the SystemVerilog syntax for both packed and unpacked arrays. Also, SystemVerilog should accept VeraLite's array declaration as a shorthand notation, that is: **[size]** becomes the same as **[size – 1:0]**.
For example:

```
int  Array[8][32];
```
is the same as:
```
int  Array[7:0][31:0];
```

Conflict: VeraLite does not allow slicing of (unpacked) arrays. SystemVerilog does.
Resolution: Enhance VeraLite to allow slicing of multi-dimensional arrays, both packed and unpacked.

Clarification: VeraLite will allow packed arrays of the same types as SystemVerilog, but it won't support packed arrays of additional types that are not part of SystemVerilog 3.0. This explicitly excludes strings, events, associative arrays, and objects.

3.8.5 Array Initialization (1-15)

See the correction in Section 3.8.3. The same corrections apply to multi-dimensional arrays.

3.9 Associative Arrays (1-16)

Correction: The examples incorrectly list an array of **port_name**. Please disregard.

Correction: The donation states that an associative array index "is an unsigned number with a maximum value of `2^64-2`". This is incorrect. Associative arrays can be declared with a specific index type or with no index type. Associative arrays that do not specify an index type have the following properties:

- **unspecified index type:** *type* `array_name[];`
  - The array can be indexed by any integral data type, including integers, bit-vectors of arbitrary length, and string literals. Since the indices can be of different sizes, the same numerical value may have multiple representations, each of a different size. VeraLite resolves this ambiguity by detecting the number of leading zeros and computing a unique length and representation for every value.
  - Indices are unsigned.
  - Indexing expressions are self-determined: signed indices are not sign extended.
  - 4-state indices containing X or Z result in a run-time error (see conflict below).
  - A string literal index is auto-cast to an equal size bit-vector.
  - The traversal order is numerical (smallest to largest).

An associative array that specifies an index type restricts the indexing expressions to a particular type. Currently, VeraLite only supports the following index types:

- **string index**: *type* `array_name[string];`
  - Indices can be strings or string literals of any length. Other types result in a compiler error.
  - A **null** index result in a run-time error.
  - The traversal order is lexicographical (lesser to greater).
- **object index**: *type* `array_name[some_Class];`
  - Indices can be objets of that particular type or derived from that type. Any other type results in a compiler error.
  - The traversal order is arbitrary.
  - A **null** index is valid and will be smaller than all other objects (for traversal).
- **integer index:** *type* `array_name[integer];`
  - Indices can be any integral expression.
  - Indices are signed.
  - Indices smaller that **integer** are sign extended to 32 bits.
  - Indices larger than **integer** are truncated to 32 bits.
  - Indices containing an X or Z result in a run-time error (see conflict below).
  - The traversal order is numerical.

Limitation: Associative arrays do not support arbitrary user-defined numerical types, such as `reg[21:2]`.
Resolution: VeraLite will support any arbitrary user-defined numerical type, provided that the type has been previously defined via a **typedef**. For example:

```
typedef bit [3:0] nibble;
integer arr[ nibble ];        // associative  array of integer indexed by bit[3:0]
```

Conflict: In SystemVerilog an index expression containing X or Z values does not result in a run-time error, instead, the result depends on the operation and the array type. If the array is of a 4-state type, a read returns X; for a 2-state array a read returns a 0 and issues a warning. A write always causes a warning to be issued and the operation is ignored.
Resolution: VeraLite associative arrays will be compatible with SystemVerilog arrays. When the index expression contains an X or Z, a read will return the default value for the corresponding array type (i.e., null for string or class). In addition, a write operation or a read operation from an associative array that is not a 4-state type will result in a warning.

3.10 Dynamic Arrays

Dynamic arrays complement static and associative arrays, the description is missing in the donation, and attached below.

3.10.1 Dynamic Arrays

Dynamic arrays are one-dimensional arrays whose size is set or changed at runtime. The space for a dynamic array doesn't exist until the array is explicitly created at runtime.

The syntax to declare a dynamic array is:

```
data_type array_name[*];
```

data_type
The data type of the array elements. Dynamic arrays support the same types as fixed-size arrays.

For example:

```
bit[3:0] nibble[*];     // Dynamic array of 4-bit vectors
integer mem[*];         // Dynamic array of integers
```

To set or change the size of the array, use the **new[]** operator. To get the current size of the array, use the **get_array_size()** function.

3.10.2 new[]

The syntax to set or change the size of a dynamic array is:

```
array_name = new[size] [(src_array)];
```

size

The number of elements in the array. Must be a non-negative integral expression.

src_array

Optional. The name of an array with which to initialize the new array. If *src_array* is not specified, the elements of *array_name* are left uninitialized. s*rc_array* must be a dynamic array of the same data type as *array_nam*e, but it doesn't have to be of the same size. If the size of *src_array* is less than *size*, the extra elements of *array_nam*e are left uninitialized. If the size of *src_array* is greater than *size*, the additional elements of *src_array* are ignored.

This parameter is very useful when growing or shrinking an existing array. In this situation, *src_array* is *array_name* so the previous values of the array elements are preserved. For example:

```
integer addr[*]; // Declare the dynamic array.
addr = new[100]; // Create a 100-element array .
...
   // Double the array size, preserving previous values.
addr = new[200] (addr);
```

3.10.3 get_array_size()

The syntax for the get_array_size() function is:

```
function integer get_array_size(array_name);
```

The function returns the current size of a dynamic array, or zero if the array has not been created.

3.11 Enumerated Types in Numerical Expressions (1-20)

Clarification: The donation seems to imply that it is not possible to assign a numerical value to an enumerated type. This is not true, enum variables can be assigned numerical values (arbitrary expressions) using the run-time **cast _assign** system function. That part of the manual was omitted from the donation. The relevant portion is attached below.

Assigning Values to Variables

VeraLite provides the **cast_assign()** system function to assign values to variables that might not ordinarily be valid because of differing data type.

The syntax for **cast_assign()** is:

```
function integer cast_assign(scalar dest_var,
    scalar source_exp [, CHECK]);
```

dest_var:

The *dest_var* is the variable to which the assignment is made. It can be any scalar (non-array) type (bit, integer, string, enumerated type, event, and object handle).

source_exp:

The *source_exp* is the expression that is to be assigned to the destination variable.

CHECK:

The pre-defined constant, CHECK, is optional. Its use determines how the function handles invalid assignments.

When the **cast_assign()** system function is called without CHECK, the function assigns the source expression to the destination variable. If the assignment is illegal, a fatal runtime error occurs.
When the **cast_assign()** system function is called with null specified, the function makes the assignment and returns a 1 if the casting is successful. If the casting is unsuccessful, the function does not make the assignment and returns a 0. In the latter case, no runtime error occurs, and the destination variable is set to its corresponding uninitialized value, depending on the data type.
Note:

The compiler only checks that the destination variable and source expression are scalars. Otherwise, no type checking is done at compile time.

This is an example of the **cast_assign()** system function:

```
cast_assign(enum_var, 12*7);
```

This example assigns the expression to the enumerated type. Without **cast_assign()**, this assignment is not allowed because of strong typing of enumerated types.

Resolution: Include the cast_asssign functionality in the donation.
Alternatively, users could specify this operation using a SystemVerilog cast operation:

EnumType enum_var = EnumType'(12 * 7)

However, SystemVerilog casts are all compile-time casts, whereas cast_assign is a run-time operation. Also, SystemVerilog casts do not provide for error checking (the CHECK variant of cast_assign), nor for the possibility of a run-time error due to a cast.

Note: **cast_assign** is similar to the **dynamic_cast** function available in C++, however, **cast_assign** allows users to check if the operation will succeed, whereas **dynamic_cast** always raises a C++ exception.

3.12 Operators (1-22)

Clarification: All VeraLite operators that are also defined in Verilog have the same semantics as described in SystemVerilog. Any extensions of these semantics for VeraLite data types not in Verilog 2001 are described in the corresponding section for that data type in this document. All the VeraLite operators are shown in the table below.
Operators that do not exist in SystemVerilog or behave different are shown in **boldface**.

Operator	*Description*	*Semantics*

{}	RHS numeric concatenation	Same as System Verilog
`{}	LHS numeric concatenation	Same as LHS {} in SystemVerilog[3]
{{}}	Numeric Replication	Same as SystemVerilog
{}	**String concatenation**	**Not in SystemVerilog[4]**
{{}}	**String replication**	**Not in SystemVerilog**
+ - * /	Arithmetic	Same as SystemVerilog
%	Modulus	Same as SystemVerilog
++ --	Increment/Decrement (post)	Same as SystemVerilog[5]
++ --	Increment/Decrement (pre)	Same as SystemVerilog
+= -= *= /= %= <<= >>= &= |= ^= ~&= ~|= ~^=	Compound assignment	Same as SystemVerilog
=	Simple Assignment	Same as Verilog 2001
< <= > >=	Relational	Same as Verilog 2001
! && || == !=	Logical operators	Same as Verilog 2001
=== !==	Case equality, inequality	Same as Verilog 2001
=?= !?=	**Wild equality, inequality**	**Not in SystemVerilog**
~	Bit-wise negation	Same as Verilog 2001
& | ^ ^~	Bit-wise binary operators	Same as Verilog 2001
&~ |~	**Bit-wise NAND, NOR**	**Not in SysytemVerilog**
& | ^ ~& ~| ~^	Reduction operators	Same as SystemVerilog
<< >>	Shift	Same as SystemVerilog
?:	**Conditional**	**Unlike SystemVerilog[6]**

VeraLite Operators

Clarification: The bitwise NAND (&~) and bitwise NOR (|~) have these meaning:

```
a &~ b is equivalent to: ~(a & b)
a |~ b is equivalent to: ~(a | b)
```

Conflict: Introducing the &+ and |~ operators changes the meaning of existing Verilog expressions that contain the & or | operators followed by ~ without any space in between. For example, the expression:

```
a = b &~ c;
```

Is interpreted as:

SystemVerilog: `a = b & (~c);  // b AND (NOT c)`

VeraLite: `a = b &~ c;  // b NAND c`

The situation is similar to the ambiguity involving the SystemVerilog xnor (^~) operator:

a ^~ b is different from a ^ ~b, which means a ^(~b)

Resolution: Remove these operators from the donation. They exist only for the sake of completeness, but are not vital.

[3] This was addressed before (see Statement Blocks (1-3)): Will be changed to use SystemVerilog's {}.
[4] These operators don't exist in SystemVerilog when applied to string variables (see 3.18).
[5] The behavior of ++ and -- is incompletely specified in SystemVerilog (see 3.17).
[6] The behavior of this operator is different as explained below (see Conditional Operator (1-27)).

Clarification: The donation doesn't specify the data types allowed by the wild equality `=?=` and wild inequality `!?=` operators. These operators accept any integral expression, that is, bit, integer, or enumerated type.

Limitation: The operators in the table below exist in SystemVerilog, but not in VeraLite.

Operator	*Description*	*Semantics*
`<<< >>>`	`arithmetic shift`	`sign preserving shift`
`<<<= >>>=`	`Compound assignment`	`arithmetic shift assign`
`**`	`power`	`exponentiation`

Resolution: Remove this limitation. VeraLite will support all SystemVerilog operators.

3.13 Operator Precedence (1-23)

Clarification: The donation doesn't explicitly list operator associativity, as well as distinguish clearly between unary and binary operators. The following table lists the precedence and associativity of all VeraLite operators. Highest precedence operators are listed first.

Operator	**Associativity**
`() [] .`	left
Unary `! ~ ++ -- & ~& | ~| ^ ~^`	right
`* / %`	left
`+ -`	left
`<< >>`	left
`< <= > >=`	left
`== != === !== =?= !?=`	left
`& &~`	left
`^ ~^`	left
`| |~`	left
`&&`	left
`||`	left
`? :`	right
`= += != *= /= %= &= |= ^= <<= >>=`	none

3.14 Arithmetic Operators (1-24)

Clarification: The donations states "If an operand has any bit with a value of x, the entire result is x.". That statement is incomplete. It should say "If an operand has any bit with a value of x or z, the entire result is x."

3.15 Bitwise Operators (1-26)

Clarification: The donation does not include the table for the bitwise NAND (&~) and bitwise NOR (|~) operators. These are included below.

&~	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x

|~	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x

3.16 Conditional Operator (1-27)

Clarification: The donation incorrectly describes the semantics of the conditional operator. The correct description is:

```
expression1 ? expression2 : expression3
```

If expression1 evaluates to true (known value other than 0) then expression2 is evaluated and used as the result. If expression1 evaluates to false (0) then expression3 is evaluated and used as the result. If expression1 evaluates to X or Z, it is treated as false.

The behavior of this operator was modified to behave as in 'C' in order to avoid having to evaluate both expressions (as stated in the donation). This is because expression2 or expression3 (or both) may contain side effects that require that only one of them should be evaluated. Consider the following examples:

```
a = (b) ? d++ : e++;                  // increment d or e but not both!
x = (p == null) ? -1 : p.value;       // do not dereference null handle
```

Conflict: The conditional operator is not compliant with SystemVerilog when the conditional expression evaluates to an ambiguous value (X or Z). In that case, SystemVerilog evaluates both expression2 and expression3 and the results are combined bit by bit (expanding the shorter operand by zero-filling). VeraLite considers an ambiguous value the same as a false and evaluates expression3 only.
Resolution: VeraLite will support SystemVerilog semantics. It may be desirable to introduce another conditional operator (perhaps ??) that behaves like C.

3.17 Side effecting operators: Increment and Decrement

The behavior of the pre/post increment/decrement operators is not completely defined in SystemVerilog. This can lead to unexpected behavior when a single statement modifies the same variable more than once. For C/C++, the ANSI-C standard states that the behavior is undefined so every compiler is free to do it differently, and indeed they do. For example, the following C/Vera code fragment produces the output shown below:

```
int i = 1;
printf( "%d  %d  %d  %d  %d  %d\n", i++, i++, ++i, --i, i--, i-- );
```

vera	1 2 4 3 3 2	gcc -g	1 2 4 3 3 2
cc (solaris)	1 2 1 1 3 2	gcc -O2	1 2 1 1 3 2
cc (dec)	1 1 2 1 1 1	cc (linux)	0 -1 -1 -2 0 1

Vera defines the semantics for computing all arguments and operands. Arguments with the same precedence are evaluated in strict left-to-right order. In addition, the pre and post increment operators operate on their corresponding variable as they are evaluated. Thus, the semantics of *post* and *pre* increment (++) is roughly equivalent to the code shown below (decrement is analogous).

```
function integer pre_inc (var integer a); begin   // ++a
 a += 1;
 pre_inc = a;
end
endfunction

function integer post_inc (var integer a); begin  // a++
 post_inc = a;
 a += 1;
end
endfunction
```

Clarification: The above section does not represent a conflict. It merely states a possible semantic definition for the ++ and -- operators. VeraLite's semantics are compatible with Verilog operators, which are also left to right associative, and may have side-effects. For example:

$display(f(a) + g(b));

If functions f() and g() have side effects on a or b, Verilog must enforce the left-to-right semantics to avoid the ambiguous results.

3.18 Operators for manipulating strings (1-28)

In VeraLite a string literal is *implicitly* converted to string type when it is assigned to a string type variable or is used in an expression involving string type operands. A string literal and a concatentation or replication of string literals are the only types of **regs** that are allowed to be assigned to a string type variable.

For example:

```
reg [15:0] r;
integer i = 1;
string a = {"Hi", b};
string b = "";
r = a;          // valid
b = r;          // Error
```

```
b = "Hi";         // valid
b = {5{"Hi"}};    // valid
a = {i{"Hi"}};    // valid
r = {i{"Hi"}};    // invalid in Verilog
a = {i{b}};       // valid
a = {a,b};        // valid
a = {"Hi",b};     // valid
```

The basic operators defined on the string data type are listed in the following table.

Operator	Meaning
Str_1 **==** Str_2	Checks if the two strings are equal. Result is 1 if they are equal and 0 if they are not. Both strings may be of type string. Or one of them may be a string literal. Note that if both are string literals, the expression is the same Verilog equality operator for numeric types.
Str_1 **!=** Str_2	Logical Negation of ==
{Str_1,Str_2,...,Str_n**}**	Each Str_i may be of type string or may be a string literal (it will be implicitly converted to string). If at least one Str_i is of type string, then the expression evaluates to the concatenated string and is of type string. Note that if all the Str_i are string literals, then the expression behaves like Verilog concatenation for numeric types (if the result is used in another expression involving string types, it is implicitly converted to string type).
{multiplier**{**Str**}}**	Str may be of type string or may be a string literal. Multiplier is of numeric type and can be non-constant. If Str is a literal and the multiplier is constant, the expression behaves like numeric replication in Verilog (if the result is used in another expression involving string types, it is implicitly converted to string type).
Str.method(...)	The dot (.) operator is used to invoke a specified method on strings. See Section 3.18.1 for detailed descriptions of the various string methods available

3.18.1 Methods on String (1-28)

Clarification: This section was omitted from the donation. It is attached below.

VeraLite supports the following methods on the **string** data type:

```
function integer len()
```

- str.len() returns the number of characters in the string excluding any terminating null character
- str.len() returns 0, if str is null

```
task putc(integer i, string s)
task putc(integer i, integer c)
```

- str.putc(i, c) replaces the i^{th} character in str with the given value.
- str.putc(i, s) replaces the i^{th} character in str with the first character in s. s can be any expression that can be assigned to a string.
- str.putc leaves str unchanged, if i < 0 or i >= str.len()

```
function integer getc(integer i)
```

- str.getc(i) returns the ASCII code of the i^{th} character in str
- str.getc(i) returns 0, if i < 0 or i >= str.len()

```
function string toupper()
function string tolower()
```

- str.tolower() and str.toupper() return strings with characters in str converted to lowercase and uppercase respectively. str is unchanged.

```
function compare(string s)
function icompare(string s)
```

- str.compare(s) compares str and s character by character and returns the difference between the ASCII codes of the first character in which they differ; returns 0 if the string are equal. str.icompare(s) behaves similarly but the comparison is case insensitive

```
function string substr(integer i, integer j)
```

- str.substr(i, j) returns a sub string formed by characters in position i through j of str if 0 <= i <= j < str.len(); returns "" (not null), otherwise

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

- str.atoi() returns the integer corresponding to the ASCII decimal representation in str. Example. str = "123"; i = str.atoi() assigns 123 to i. The string is converted to the first non-digit. atohex, atooct and atobin are similar except that instead of decimal the string is interpreted as hexadecimal, octal and binary respectively.

```
task itoa(integer i)
```

- str.itoa(i) writes the ASCII decimal representation of i into str.

Limitation: The donations doesn't list support for parsing real numbers.
Resolution: The following string function will be added:

```
function real atoreal()
```

3.19 Concatenation (1-29)

Clarification: The donation does not explicitly state that the arguments to the replication operator can be concatenations, as in SystemVerilog. They can, as shown in the second example of the donation: `{4 {addr,data}}`.

3.20 variable Assignment (1-31)

Conflict: VeraLite does not support assignment recursion:

```
a = b = c;
```

This form is not supported by SystemVerilog either, but it does support a modified form that uses parenthesis in order to avoid common mistakes:

```
a = (b = c);
```

Likewise, assignments within conditionals are not allowed by VeraLite:

```
if( a = b ) a = c + d;
```

but SystemVerilog does allow the modified form:

```
if( (a = b) ) a = c + d;
```

Resolution: VeraLite will accept SystemVerilog's form.

Clarification: In the presence of side-effecting operators, SystemVerilog is not specific as to what the result should be. For example:

```
a = 1;
a = (b = a++);
```

What is the value a? 1 or 2 ? (see Section 3.17 for more puzzlers).

3.21 Expressions and Operators (General)

Clarification: In SystemVerilog, the type and size of all expressions is determined at compile time. This allows the compiler to compute the size of all expression temporaries and apply strict rules regarding extension, truncation, and sign extension. The same rules apply to VeraLite. Although, the VeraLite donation does not state so explicitly, the arguments to the concatenation, replication, and slicing operators must be constants. If that were not so, it would not be possible to compute all expression sizes at compile time.

Limitation: SystemVerilog provides a syntax for specifying fixed-size, variable position slices: `[position +: size]` and `[position -: size]`. VeraLite does not support these operators.
Resolution: Remove this limitation. VeraLite will support the +: and -: slicing operators.

3.22 Signed vs. Unsigned

Conflict: VeraLite does not follow the SystemVerilog sign extension rule. Instead, it follows the Verilog-1995 rule: zero fills when converting a signed number (i.e. integer) to a number of higher precision (i.e., reg[64:0]). SystemVerilog requires that the sign be extended.
Resolution: Perform sign extension according to SystemVerilog.

4 Programming Overview

4.1 Program Block (2-3)

Clarification: A typical VeraLite test-bench contains type definitions, data declarations, subroutines, connections to the design via VeraLite-interfaces (not to be confused with SystemVerilog interfaces), and a program block. The **program** block serves two basic purposes:

1. It provides an entry point where the test-bench begins execution.
2. It creates a scope that encapsulates program-wide (global) data.

A SystemVerilog **module** provides both of these functions: it creates a new scope, and can include an **initial** block to serve as the test-bench entry point. Thus, a module is a natural choice for modeling the program block. However, such a *"test-bench module"* differs from a regular SystemVerilog module in several ways. First, the communication between the test-bench and the design takes place via special *ports* that in addition to type, direction, and size, can also specify a clocking scheme (see section 7.1). Second, VeraLite provides for execution using cycle semantics as well as event semantics. The program construct serves as a clear separator between the design and the test-bench, and, more importantly, it indicates the special nature of the *test-bench module,* which allows a compiler to enable the cycle semantics for all elements within the program. Finally, the program block contains a single implicit initial block, and no always blocks or other programs (unlike modules, programs can't be nested).

Simplification: In the donation, a program block is shown as containing no arguments. Unlike SystemVerilog, in which ports are explicitly declared in a module declaration, a program connects to the design through an implicit set of ports that are derived from a series of VeraLite interface declarations. Implicit port declarations can be confusing, error prone, and, in a fluid design environment, hard to maintain. Without loosing any generality, the connection between design and test-bench can be significantly simplified by using the same paradigm used by SystemVerilog to specify port connections. This simple change makes the program block much more consistent with SystemVerilog. The syntax for the program block then becomes:

```
program program_name ( list_of_ports );
    program_declarartions
    program_code
endprogram
```

For example:

```
program test (input clk, input [16:1] addr, inout [7:0] data);
    . . .
endprogram
```

The `list_of_ports` allowed by a program shall be the same as the one allowed for any SystemVerilog module. A more complete example is included in Section 7.

Clarification: Although the **program** construct is new to SystemVerilog, its inclusion does not represent a conflict, but a natural extension. The program construct should be considered the declaration of a special type of module (i.e., a module with a test-bench attribute). Once the program block has been declared, it can be instantiated in the proper hierarchical location (typically at the top level) and its ports can be connected in the same manner as any other module.

Limitation: Some VeraLite constructs and data-types cannot be used in declarative contexts such as module ports, gates, or continuos assignments. These constructs will need a formal definition (possibly via BNF) and additional semantic restrictions that limit their use within the procedural (test-bench) environment. Note that this limitation is not new to VeraLite, it simply extends the rules set forth by SystemVerilog, which disallows automatic variables from triggering event expressions or be written using non-blocking assignments.

4.1.1 Static Data Initialization

In SystemVerilog, setting the initial value of a static variable as part of the variable declaration requires that the initialization occurs before any **initial** or **always** blocks are started. Likewise, VeraLite allows static data (including static class members) to specify an initial value as part of their declaration, and, like SystemVerilog, VeraLite requires that all such data be initialized before the *program* (see section 4.1) starts executing. It is important to note that VeraLite initial values are not constrained to simple constants, but may include run-time expressions, including dynamic memory allocation. For example, a static class can be initialized via its **new** method, or a **Semaphore** may be initialized by calling its **alloc** function. While this does not represent a conflict with SystemVerilog, it may require a special pre-initial pass at run-time, which may need changes to the initial SystemVerilog simulation cycle.

4.1.2 Scope Rules

In the donation, the following VeraLite language constructs are always defined in global scope, sharing the global name space so no two of them can have the same name:

- Type declarations: Classes and Enumerated Types

    - Elements of each enumerated type
- Subroutines: Tasks and Functions
- VeraLite Interfaces (see Section 7.1)
- Program block
- Data declared in the outermost block of the program block
- Data declared outside any block, in the global scope

Conflict: VeraLite does not allow type declarations (class or enums), tasks, or functions inside the program block (i.e. the test-bench module). This is inconsistent with SystemVerilog scope rules.
Resolution: Allow VeraLite type declarations, tasks, and functions inside the program block (in the scope of the test-bench module). All such definitions can be encapsulated in a manner consistent with SystemVerilog scope rules. Data declared within the program will be local in scope (local to the program block) and will have static lifetime.
Global declarations (outside the program block or any other module) will reside in $root and have static lifetime.

Clarification: The following VeraLite constructs create a new scope:
- A class definition
- A task or function definition
- A block statement
    - A fork-join does not create a scope unless it contains a block statement.

Subroutines (tasks and functions) cannot be nested within themselves, but they can contain block statements that do create a scope. Block statements do not have to be named to create a new scope.

The declaration in the closest enclosing scope is matched: A scope nested inside another scope has visibility of (and may reference) all elements visible or declared in its parent scope. A name declared inside a scope hides all elements with the same name that are visible or declared in the parent scope. All these rules are consistent with SystemVerilog.

Limitation: VeraLite does not allow explicit references to global data, only to global data that is visible within a given scope. If a locally declared element shares the same name as a global element, the global element is not visible and cannot be referenced. SystemVerilog allows references to a global *name* via explicit use of **$root.***name*.
Resolution: Remove this limitation and allow explicit references to global data.

4.1.3 Multiple Programs

Clarification: The donation does not mention the possibility of having multiple test-bench programs, however, it is completely compatible with both SystemVerilog and VeraLite to have any arbitrary number of program definitions or instances. The programs can be fully independent (without inter-program communication), or cooperative. Users can control the degree of communication by choosing to share data via $root or making the data private by declaring it inside the corresponding program block.

Clarification: The VeraLite constructs simplify the creation and maintenance of test-benches. Furthermore, since modeling the environment can be a significant part of a test-bench, the same set of VeraLite constructs can be effectively used for writing models at a higher abstraction level than currently provided by SystemVerilog. The ability to instantiate and individually connect each instance of a program enables their use as generalized models.

4.2 Preprocessor Directives (2-4)

Conflict: VeraLite uses # for preprocessor directives (like C), but SystemVerilog uses `.
Resolution: VeraLite will use the same preprocessor directives as SystemVerilog.

4.3 Subroutines (2-5)

Conflict: VeraLite allows blocking functions, SystemVerilog does not.
Resolution: VeraLite will disallow blocking functions.
Also, functions will be restricted to disallow passing arguments of type event.

Conflict: In VeraLite the default lifetime for tasks and functions is **automatic**. In SystemVerilog the default lifetime is **static**.
Resolution: VeraLite will Adopt the SystemVerilog default.
In addition, we propose an optional module attribute that specifies the default lifetime of all tasks and functions declared within the module. The lifetime attribute can be set to **automatic** or **static.** The default is **static** for regular modules**,** and **automatic** for the program block. This enhancement allows existing VeraLite code to execute unchanged. Also, class methods are by default automatic, regardless of the lifetime attribute of the module in which they are declared.

4.4 Discarding Function Return Values (2-8)

Conflict: Both VeraLite and SystemVerilog have a **void** keyword. In SystemVerilog **void** is a type, whereas in VeraLite it is a special syntactical value that can used (1) to discard function values, and (2) in special operations such as void drives.
In VeraLite, void is strictly a compiler scheme. For example:

```
void = some_function();
```

instructs the compiler to ignore the return value from the function.

In SystemVerilog the same can be done using a cast:

```
void'(some_function());
```

Resolution: Adopt SystemVerilog's cast operator to discard return values. Propose using the VeraLite form '`void =`' as an alternative since it shows the intent more clearly than a cast, and doesn't force users to use an extra set of parenthesis.

4.5 Tasks (2-9)

Clarification: The section describes the syntax for declaring a *local task*, a task with file scope. In SystemVerilog subroutines can have global or module scope, not file scope, so a local task declaration is unnecessary, thus deprecated.

4.6 return Statement (2-10)

Conflict: SystemVerilog allows the return statement from a function to include an expression, whereas VeraLite doesn't.
Resolution: Allow the more general SystemVerilog form. This can be done by a simple translation:

```
function int foo() begin          function int foo() begin
   return expression;      =>        foo = expression;
end                                  return;
                                  end
```

4.7 External Declarations (2-13)

Conflict: The VeraLite **extern** keyword is used to support separate compilation, in a manner similar to C. This collides with SystemVerilog's use of **extern**, which is used by a SystemVerilog **interface** to declare tasks external to the interface**.**
Resolution: VeraLite **extern** declarations have no effect and can be ignored. The use of **extern** is deprecated by VeraLite.
Alternatively, the compiler could be instructed to allow the extern modifier, and ignore it.

5 Sequential Control

5.1 case Statements (3-3)

Limitation: VeraLite does not support SystemVerilog's **unique** and **priority** qualifiers for **if-else, case, casez** and **casex** statements.
Resolution: VeraLite will support the **unique** and **priority** qualifiers for **if-else, case, casez** and **casex** statements. This qualifiers do not represent a conflict.

5.2 for loops (3-6)

Limitation: VeraLite only allows data declarations at the start of a block, however, SystemVerilog allows a loop variable to be declared following the **for** keyword. For example:

```
for(integer j = 0; j < n; j++) begin $display(j); end
```

The scope of the variable is the loop itself.
Resolution: Remove this limitation. VeraLite will allow loop variables to be declared within the loop.

5.3 break and continue (3-8)

Clarifications: Both **break** and **continue,** although not in Verilog-1995, have the same semantics in both VeraLite and SystemVerilog.

6 Concurrency Control

6.1 fork and join (4-2)

VeraLite and SystemVerilog have similar functionality for starting dynamic processes: the ability to fork threads without a join that forces the parent process to block. VeraLite uses the **fork .. join none** construct, while SystemVerilog uses the **process** qualifier. In addition, VeraLite supports the existing **fork .. join** (same as **join all**)**,** and one more variant: **fork ... join any,** in which the parent process blocks until at least one of its children terminate.
Overlap: VeraLite's fork/join constructs do not represent a conflict, but **fork .. join any** does overlap with SystemVerilog's **process** qualifier.
Resolution: While both forms can coexist, they may lead to confusion. We propose deprecation of **process** and adoption of the **fork .. join any** construct, which is more natural to existing Verilog (and Vera) users.

6.2 wait_var() (4-8)

Correction: The donation incorrectly states the data-types allowed as arguments to the **wait_var()** system task. The document should state:

`variable_list`

> The `variable_list` consists of one or more variables (separated by commas) of type **integer, bit** (i.e.**, reg), bit[], enum,** or **string**. Each variable may be either a simple variable, or a **var** parameter (variable passed by reference), or a member of an array, associative-array, or object (class). Object handles are not allowed.

Clarification: Arguments to **wait_var()** can be an array subscript expressions, in which case the index expression is evaluated only once when **wait_var()** is executed.

Clarification: This is a system task and should use the appropriate syntax: **$wait_var()**.

6.3 terminate (4-9)

Clarification: SystemVerilog supports the **disable** construct, which will end a process when applied to the block being executed by the process. However, **terminate** differs from **disable** in that **terminate** considers the *dynamic* parent-child relationship of the processes, whereas **disable** uses the *static* syntactical information of the disabled block. Thus, **disable** will end all processes executing a particular block, whether the processes were forked by the calling thread or not, while **terminate** will end only those processes that were spawned by the calling thread.

Clarification: This is a system task and should use the appropriate syntax: **$terminate()**.

6.4 suspend_thread (4-10)

Clarification: Calling **suspend_thread()** is conceptually similar to a 0 delay statement:

```
#0 ;
```

Clarification: This is a system task and should use the appropriate syntax: **$suspend_thread()**.

6.5 Maximum Threads (4-11)

This feature is deprecated and is not supported by VeraLite. Please disregard.

6.6 Events (1-9)

Section 1-9 of the donation briefly describes VeraLite events, but the section describing event operations in detail is missing. The missing sections are attached below.

6.6.1 Synchronizing concurrent processes with event variables

Events are variables that synchronize concurrent processes. When a **sync** is called, a process blocks until another process sends a trigger to unblock it. Events act as the go-between for triggers and syncs.
This subsection includes:

- sync System Task
- trigger System Task
- event Variables

6.6.2 sync System Task or Function

The **sync()** system task synchronizes statement execution to one or more triggers. **sync** can be used as either a task or a function.
The syntax to call the **sync()** task is:

```
task sync(ALL | ANY | ORDER, event event_name1, ..., event_nameN);
function integer sync(CHECK, event event_name1, ..., event_nameN);
```

event_name:

The *event_name* is the event variable name on which the sync is activated.

Predefined Macros:

ALL

The ALL sync type suspends the process until all of the specified events are triggered. For example:

```
sync(ALL, event_a, event_b, event_c);
```

This example suspends the thread until each of the events are triggered. Once they are triggered, the statement immediately following the **sync()** call is executed.

ANY

The ANY sync type suspends the process until any of the specified events is triggered. For example:

```
sync(ANY, event_a, event_b, event_c);
```

This example suspends the thread until any of the specified events is triggered. Once one of the events is triggered, the statement immediately following the **sync()** call is executed.

ORDER

```
sync(ORDER, event_a, event_b, event_c);
```

This example suspends the thread until all of the specified events are triggered. As soon as an event is received out of order, the process unblocks and a simulation error occurs. Also, only the first event can be in the **ON** state when the sync is called. If both *event_a* and *event_b* are **ON** when the call is made, a simulation error occurs.
Note: Events set to **null** are treated as if they were received in the correct order.

CHECK

The **CHECK** sync type is called as a function. It does not suspend the thread. It returns a 1 if the trigger is **ON** and a 0 if it is not. This sync type can only be used with **ON** and **OFF** trigger types. For example:

```
if (sync(CHECK, event_a) )
      printf("The event is ON.\n");
```

If the event is on, this **sync()** call returns a 1 the message is printed. If the event is **OFF**, **sync()** returns a 0.

6.6.3 trigger System Task

Triggers are used to turn events ON or OFF. By default, all events are OFF.
The syntax to call a trigger is:

```
task trigger(ONE_SHOT | ONE_BLAST | HAND_SHAKE | ON | OFF,
     event event_name1 , ... , event_nameN);
```

event_name:

The *event_name* is the event variable name on which the sync is activated.

Predefined Macros:

ONE_SHOT

The **ONE_SHOT** trigger turns on the vent momentarily, causing all currently waiting syncs to unblock; subsequent syncs will not unblock. If there are no processes waiting for the trigger, the trigger is discarded. Note that in order for a trigger to activate a sync, the sync must execute before the trigger.

ONE_BLAST

ONE_BLAST triggers are similar to **ONE_SHOT** triggers with one exception: the ON condition persists until simulation time advances. Thus, ONE_BLAST triggers will unblock any sync that executes before or at the same simulation time as the trigger.

VeraLite does not yet support ONE_BLAST, but it's included here for completeness. Also, it easy to implement and it can be useful.

HAND_SHAKE

A **HAND_SHAKE** trigger unblocks only one sync, even if multiple syncs are waiting for triggers. If a sync has been called and is waiting for a trigger then the **HAND_SHAKE** trigger will unblock the sync. If no sync has been called when the trigger executes, the **HAND_SHAKE** trigger is stored. When a sync is eventually called, the sync is immediately unblocked and the trigger is removed. Vera uses LIFO scheduling for events, but that is inconsistent with all other synchronization primitives. VeraLite proposes using FIFO. We may choose to document that here or leave it open.

ON

The **ON** trigger turns the event on, causing all currently waiting as well as subsequent syncs on that event to unblock. This condition will persist until there is a trigger **(OF**F) call.

OFF

The **OFF** trigger turns an event OFF, causing subsequent syncs on that evebnnt to block.

6.6.4 Event Variables

Event variables serve as the link between triggers and syncs. They are a unique data type with several important properties.

Bidirectional Event Variables

Event variables are bidirectional variables when used as arguments in syncs and triggers. The same event variable can be used to pass and receive triggers. For example:

```
task T1 (event trigger_a)
```

```
{
  printf("\nT1 syncing" );
  sync(ALL, trigger_a);   // Blocked: proceed after receiving
                          //trigger
   printf("\nT1 event trigger_a received");
   repeat (7) @(posedge CLOCK);
   printf("\nT1 triggering trigger_a");
   trigger(ONE_SHOT, trigger_a);
}

program trigger_play
{
   event trigger1;

// top block code starts here

   fork
      T1(trigger1);       // start T1 and go on
   join none
   repeat(8) @(posedge CLOCK); // blocks waiting for trigger

   fork
   {
      printf("\nPROGRAM triggering trigger1");
      printf("\nPROGRAM This unblocks T1");
      trigger(ONE_SHOT, trigger1);     // unblock the waiting
   T1
   }
   {
      repeat (5) @(posedge CLOCK);
      printf("\nPROGRAM syncing\n\n");
      sync (ALL, trigger1);  // wait for T1 to unblock me
   }
   join
   wait_child();
   printf("\nTrigger play done!");
}
```

This example declares the task T1, which is called in the main program. Then T1 is called in a thread forked off from the main program. The program continues without waiting for the child thread to complete. Because T1 contains a sync within its definition, the child thread blocks, waiting for a trigger. Then another fork is used to fork off a trigger, which unblocks the suspended T1. A second thread in that fork then calls a sync. This sync occurs as T1 is unblocked. T1 then continues its execution, which includes the execution of the trigger that unblocks the final child thread.

6.6.5 Disabling Events

If an event variable is assigned a null value, the event is ignored in subsequent **sync()** calls that may be waiting for a trigger on the event variable. That is, when the event is set to null, any blocking sync becomes non-blocking. There is no way to make it block again.

For example:

```
event E1 = null;
sync(ALL, E1);
```

The sync is immediately satisfied because of the null value assigned to E1.

6.6.6 Merging Events

When an event is assigned to another event, the two are merged, which causes triggers on either one to affect both.

For example:

```
event E1, E2, E3;
E1 = E2;
trigger(ON, E3);
trigger(ON, E1); // this will trigger E2, as well
trigger(ON, E2); // this will trigger E1, as well
E1 = E3;
E2 = E1;
trigger(ON, E1); // this will trigger E2 and E3, as well
trigger(ON, E2); // this will trigger E1 and E3, as well
trigger(ON, E3); // this will trigger E1 and E2, as well
```

However, use caution when merging events. The assignment only affects subsequent triggers and syncs. For example, if a process is blocked waiting for event1 when you assign another event to event1, the sync will never unblock. For example:

```
fork
{
   while (1) {sync (ALL, E2);}
}
{
   while (1) {sync (ALL, E1);}
}
{
   E2 = E1;
   while (1) {trigger(ON, E2);}
}
join
```

This example forks off three concurrent threads. Each starts at the same time in the simulation. So, at the same time that threads 1 and 2 are blocked, thread 3 assigns the event E1 to E2. This means that thread 1 can never unblock, because the event E2 is now E1. To unblock both threads 1 and 2, the merger of E2 and E1 must take place before the fork.

Conflict: VeraLite's event data-type collides with SystemVerilog's named events. In SystemVerilog, named events are triggered via the -> operator, and processes can wait for events using an @ operation.
A SystemVerilog event is analogous to a VeraLite event that uses a ONE_SHOT trigger. However, VeraLite events are much more general than SystemVerilog events. The most salient semantic difference is that SystemVerilog named events do not have a value nor a duration, whereas VeraLite events have a value (ON, OFF) and a persistency that can be controlled via the trigger options. Also, VeraLite events are handles to synchronization objects, thus, they can be passed as arguments to tasks, and they can be dynamically allocated and reclaimed, whereas named events are static and cannot be passed as arguments. Rather than a basic data type, VeraLite events behave more like object handles; they can be assigned to one another, they can be assigned the value **null,** they can be arguments to tasks, and they are dynamically allocated and reclaimed.

Resolution: Extend SystemVerilog event to encompass the VeraLite functionality. The new event data-type will continue to support the old Verilog operations and semantics, which are completely backward compatible. Thus, an event that is triggered via the -> operator, and that is used in @ expression, will continue to behave exactly as it does now. The backward compatible declarative operations (@ and ->) will be restricted to events with static lifetime. In addition, the event data-type will be extended to support the new VeraLite operations and semantics.

Clarification: Events incorporate a set of system tasks and should use the appropriate syntax: **$sync()**, and **$trigger().**

6.7 Semaphores

6.7.1 Allocating a Semaphore (4-12)

Clarification: The prototype for the alloc() function is shown as:

```
function integer alloc(SEMAPHORE, integer semaphore_id,
          integer semaphore_count, integer key_count);
```

VeraLite imposes the following restrictions:

- semaphore_id must be 0.
- semaphore_count must be 1.

Proposal: This function should be simplified to:

```
function integer alloc(SEMAPHORE, integer key_count);
```

6.8 Mailboxes

6.8.1 Allocating a Mailbox (4-16)

Clarification: The prototype for the alloc() function is shown as:

```
function integer alloc(MAILBOX, integer mailbox_id,
          integer mailbox_count);
```

VeraLite imposes the following restrictions:

- mailbox_id must be 0.
- mailbox_count must be 1.

Proposal: This function should be simplified to:

```
function integer alloc(MAILBOX);
```

6.8.2 Returning Data: mailbox_get() (4-17)

Clarification: Mailboxes are type-less, that is, a single mailbox can send and receive any type of data. Thus, in addition to the data being transmitted (i.e., the message), mailbox implementations must maintain the message data type placed by mailbox_put. This is required in order to enable run-time type checking.

Clarification: Semaphore and Mailbox incorporate a set of system tasks and should use the appropriate syntax: **$alloc()**, **$mailbox_get(), $mailbox_put(),** etc**.**

6.9 Timeout Limit (4-20)

Clarification: VeraLite does not support timeout limits. Please disregard this section.

7 Interfacing to the Device Under Test

7.1 Interface Declaration

VeraLite interfaces specify how a VeraLite test-bench communicates with the device under test (i.e., the Verilog design). A test-bench may contain one or more interfaces. Each interface contains an arbitrary number signals (with designated size, and input, output, or inout direction), and *at most* one of the signals that is designated as the *clock*. A VeraLite interface serves two functions. First, it itemizes and describes the signals (nets) through which the test-bench will interact with the design being tested. Thus, the

signals that comprise the interfaces denote the boundary between the design and the test[7]. Second, and most importantly, an interface groups together signals that are synchronous to a particular clock, that is, it defines a clocking domain. The clocking domain is the core element in a cycle-based methodology. This methodology enables users to write test-benches at a higher level of abstraction. Rather than focusing on signals and transitions in time, the test can be defined in terms of cycles and transactions.

Conflict: VeraLite uses an **interface** to connect the test-bench to the device under test. The keyword **interface** collides with SystemVerilog's **interface**.
Resolution: As described above, the key function of a the VeraLite interface is to enable the clocking domain abstraction. Therefore, a more appropriate name for this construct is **clocking_domain.** The term **clocking_domain** not only eliminates the conflict with SystemVerilog, but also elucidates this construct's main function.
As proposed in Section 4.1, the signals through which the program interacts with the rest of the design are specified via the program ports, while the **clocking_domain** specifies the synchronization or communication paradigm used by those signals. Thus, using this simplified and more general approach, the **clocking_domain** applies only to a particular program and must be declared within the program, that is, a **clocking_domain** will have program scope.

Simplification: A VeraLite interface requires that each one of its elements specify not only the synchronization paradigm, but also its direction and size. Since the direction and size information and can be easily deduced from the synchronization constructs (PHOLD, , …) or the program's ports, this information becomes optional. Note that it may still be needed for cross-module references (see hdl-node in Section 7.4), but for regular ports, this information can be omitted.

For example:

```
program test   (   input phi1, input [15:0] data, output write,
                   input phi2, inout [8:1] cmd, input enable
               );

    clocking_domain cd1
    {
        phi1 CLOCK;
        data PSAMPLE #-1;
        write PHOLD #1;
        input [2:0] state PSAMPLE #-1 hdl_node "top.cpu.state";
    }

    clocking_domain cd2
    {
        phi2 CLOCK;
        cmd NSAMPLE #-2ps NHOLD;
        enable PSAMPLE #1;
    }
```

[7] In the newly proposed program declaration this boundary is specified in part by the program ports.

```
    // program begins here
        . . .
    // user can access cd1.data , cd2.cmd , etc…
endprogram
```

And, the test module can be instantiated and connected to the device under test.

```
    module top;
        logic phi1, phi2;

        main_test( phi1, data, write, phi2, cmd, enable );
        cpu( phi1, data, write );
        mem( phi2, cmd, enable );
    endmodule
```

Clarification: A **clocking_domain** encapsulates a set of signals that share a common clock, therefore, specifying a clocking domain using a SystemVerilog **interface** can significantly reduce the amount of code needed to connect the test-bench. Furthermore, since the signal directions in the clocking domain are with respect to the test-bench, and not the design under test, a **modport** declaration can appropriately describe the direction. Conceptually, one can envision a VeraLite program as being contained within a program module, and whose ports are interfaces that correspond to the signals declared in each clocking-domain. The interface's wires will have the same direction as specified in the clocking-domain when viewed from the test-bench side (i.e., **modport** test), and reversed when viewed from the device under test (i.e., **modport** dut).

For example, the previous example could be re-written as:

```
interface bus_A (input clk);
    wire [15:0] data;
    wire write;
    modport test (input data, output write);
    modport dut (output data, input write);
endinterface

interface bus_B (input clk);
    wire [8:1] cmd;
    wire enable;
    modport test (input enable);
    modport dut (output enable);
endinterface


program test( bus_A.test a, bus_B.test b );

    clocking_domain cd1
    {
        a.clk CLOCK;
```

```
        a.data PSAMPLE #-1;
        a.write PHOLD #1;
        input [2:0] state PSAMPLE #-1 hdl_node "top.cpu.state";
    }

    clocking_domain cd2
    {
        b.clk CLOCK;
        b.cmd NHOLD #-2ps NSAMPLE;
        b.enable PSAMPLE;
    }

    // program begins here
        . . .
    // user can access cd1.a.data , cd2.b.cmd , etc…
endprogram
```

And, the test module can be instantiated and connected to the device under test.

```
    module top;
        logic phi1, phi2;

        bus_A a(phi1);
        bus_B b(phi2);

        test( a, b );
        cpu( a );
        mem( b );
    endmodule
```

Clarification: The signals names in the clocking domain must match the names of the ports in the program block. Thus, a signal can appear only once in a particular clocking domain. However, the same port can be included in multiple clocking domains.

7.1.1 Signals in Multiple Clocking Domains

The proposed organization for the program block and clocking domains, does not disallow using the same port in more than one clocking domain. For input signals, the semantics is clear; each clocking domain samples the signal using a different clock. However, for output signals, there are two possibilities, the output port is either driven to a resolved value or to the latest value assigned (as a procedural assignment). Typically, this is not an issue since signals in different clocking domains truly are separate signals and each corresponds to a separate port. But, sometimes a signal may be driven by more than one clock edge, for example, dual-data-rate memories are driven on both the positive and negative edges of a clock. In those situations the procedural (last drive wins) is the more useful choice. Users can easily accomplish value resolution by using separate ports for the same net.

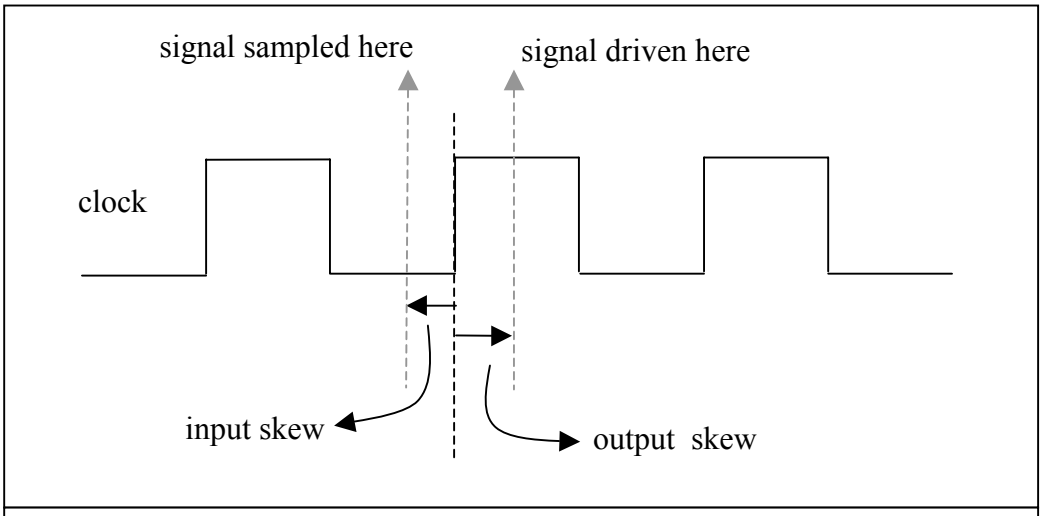
7.2 Interface Signal Declarations (5-2)

Correction: The donation includes the terms *input_signal_type* and *output_signal_type*, but does not define them.

- *input_signal_type:* An input signal type consists of one the words **NSAMPLE** or **PSAMPLE** plus an optional input skew, whose syntax is: ***#-skew_value.***
- *output_signal_type:* An output signal type consists of one the words **NHOLD** or **PHOLD** plus an optional output skew, whose syntax is: ***#skew_value.***

Note that the input skew contains a minus sign (-), whereas the output skew does not.

VeraLite samples input and inout signals at a specific edge (positive-edge for PSAMPLE and negative-edge for NSAMPLE) of their corresponding clock. If the optional input skew is specified in the signal declaration then the value of the signal sampled is the one at *skew* simulation time units *before* the clock edge.
Similarly, an output signal that specifies an output skew is driven *skew* simulation time units *after* the corresponding clock edge. The figure below depicts the sample and drive times for a signal with respect to the positive edge of a clock.



Sample and drive times for a signal with optional input and output skews

Limitation: SystemVerilog allows physical time units to be specified, whereas VeraLite accepts only time ticks to specify skews.
Resolution: Remove this limitation. VeraLite will accept both time ticks as wells as physical time units to specify skews.
For example:

```
input [16:0] data NSAMPLE -12ps;
```

Clarification: The input skew specification **#-0** has a special meaning that specifies the signal should be sampled an infinitesimal *delta* before the clock edge. That is, the value sampled is always the signal's last value before the clock edge.

Clarification: When signals in a clocking-domain do not specify a skew, VeraLite uses a skew of zero, that is, no skew. Signals with no input skew are sampled at the same time as their specified clock edge. Likewise, signals with no output skew are driven at the same time as their specified clock edge. Note that this type of zero-delay processing is a typical source of non-determinism that often results in races, however, VeraLite avoids both of these by means of two mechanisms. First, by constraining VeraLite processes to execute only at *synchronize time* (see Section 7.3), once all zero-delay transitions have propagated through the design and the system has reached a steady state. Second, by queuing all outgoing signal drives until the end of the VeraLite execution cycle, and then propagating all the drives as one event. This is described in more detail in Section 8.2. Supporting signals with zero (input or output) skew without races is an important feature of the test-bench environment. This is because test-benches with no timing information are quite common, particularly during the early phases of a design, when designers are mostly focused on functionality and not timing.

7.3 Cycle Behavior with SystemVerilog Event Queue

There are two major sources of nondeterminism in SystemVerilog. The first one is that active events can be taken off the queue and processed in any arbitrary order. The second one is that statements without time-control constructs in behavioral blocks do not execute as one event. However, from the test-bench perspective, these effects are all unimportant details. The primary task of a test-bench is to generate valid input stimulus for the design under test, and to verify that the device operates correctly. Furthermore, test-benches that use cycle abstractions are only concerned with the stable or steady state of the system for both checking the current outputs and for computing stimuli for the next cycle.
To avoid the nondeterminsm and races inherent in Verilog's event queue management, VeraLite executes test-bench processes only after the system has settled to its steady state. Until now, this was done at *synchronize time*, but that was mostly due to PLI limitations. Recently, this behavior has been modified to execute after non-blocking assignments, thus, treating all transitions towards the steady state in the same consistent manner. Accordingly, signals driven from the test bench with no delay are propagated into the design as one event immediately before *read-only synchronize time.* With this behavior, the correct cycle semantics can be modeled without races, and make the test-bench compatible with the various assertions mechanisms.
Proposal: In order to standardize the cycle behavior, the execution after non-blocking assignments described above should be added to SystemVerilog's event cycle. This is not a requirement unique to the test-bench. Many other subsystems such as monitors, checkers, waveform tools, and temporal assertions, have similar requirements. However, it is the test-bench that exacerbates this need because in addition to examining the current state it must also react and provide new stimuli for the next cycle, which is often driven with no delay.

7.3.1 Blocking Tasks in Cycle/Event mode

Calling functions in the program block from other modules and vice-versa is permitted and needs no special handling. Likewise, calling a blocking task in the program block

from outside the program block presents no problem and can be treated like a regular task call. The blocking task will simply block using cycle semantics. However, calling a blocking task outside the program block from inside the program (where cycle semantics are observed) requires explicit synchronization upon return from the task.

7.4 hdl_path (5-6)

Clarification: An hdl_path stands for a hierarchical name, or cross-module reference. VeraLite accepts any valid Verilog hierarchical expression.

Correction: Page 5-7 of the donation states that "the path always starts from the top level HDL module". This is incorrect. The path can specify any hierarchical path that can be reached from the point of instantiation of the program module (see Section 4.1) according to SystemVerilog's scope search rules.

7.5 Interface Signal of type CLOCK (5-8)

Clarification: The note in the donation states that ".. the same signal can be associated with multiple clocks via multiple interface definitions. However, despite multiple interfaces, a single signal cannot be driven to two values at the same time."
That statement is misleading. When each signal in an interface (i.e., clocking-domain) represents a Verilog wire, each output signal has a register associated with the signal that will hold the last value to which the signal is driven . If the same net is an output from more than one interface then that net has multiple drivers, and Verilog's resolution function will determine the net's final value. However, when the ***same*** output signal is driven (from within VeraLite) more than once at the same time then VeraLite checks for conflicting drives. When conflicting drives are detected, VeraLite issues a simulation error, and drives an X onto each conflicting bit.

8 Signal Operations (6-1)

The donation lists "The expect event" as one of the topics covered in the chapter. That section is not part of the donation. Please disregard.

8.1 Synchronization (6-2)

8.1.1 Interface_signal (6-2)

The donation states "… It can be any signal in an interface declaration or **CLOCK**. The interface signal can be any subfield of a signal as well. If **CLOCK** is specified, the synchronization operation is performed on SystemClock.

Clarification: SystemClock is not defined. SystemClock is a legacy concept based on designs that were either single clocked or had a discernable master clock (or system clock). Every VeraLite test-bench has an implicit clocking signal called SystemClock, which serves two functions:

1. It us used as a clock so that messages are issued in terms of cycle numbers, such as: "Error in program test1 (test1.vr, line 19, cycle 20)".
2. It allows use of **@(CLOCK)** as a shorthand for **@(***interface_name***.***clock_name***).**

In a true multi-clocking environment, *System Clock* can be confusing.

Resolution: Eliminate the need for SystemClock, and change the meaning of CLOCK in this context to represent the clock signal within a particular clocking-domain. This will allow users to specify **@(***interface_name***.CLOCK)** regardless of the name that was given to the clock signal.

For example:

```
interface myIfc
{
    input CLOCK clk;
    ...
}
```

Given the clocking-domain declaration above, users would be allowed to use `myIfc.clk` or `myIfc.CLOCK`. However, use of @(CLOCK) by itself is deprecated.

Error messages should be issued using simulation time. If it is deemed important to have cycle-based messages, users may be allowed to associate a clocking signal via a system task, for example **$set_system_cycle(** *clock_signal* **);**

8.1.2 Synchronization (6-3)

Correction: The second synchronization example uses @(CLOCK). This use should be deprecated. Instead, require explicit naming of the clocking-domain (see Section 8.1.1). as in: @(ram_bus.CLOCK).

Correction: The last paragraph of the Synchronization section that section states "At initialization, HDLs can create edges at time = 0 (for example, going from X to the initialized value). This means that synchronization conditions can be set before initialization of the signal."

That statement is incorrect. VeraLite delays execution of a program's initial statements until all such activity has settled (see Section 7.3).

Also, this means that any clock edges at time 0 will not trigger a new cycle. This avoids races between the various initial blocks and the test-bench, and is more consistent with SystemVerilog's initialization rules.

8.2 Blocking and Non-Blocking Drives (6-5)

Clarification: The donation implies that blocking and non-blocking drives in VeraLite are analogous to the Verilog operators by the same name. That is not true. In VeraLite all drives are queued, regardless of whether they are blocking or non-blocking. As long as VeraLite has processes ready to execute at the current time, these drives will remain in

the queue. Then, at the end of all (VeraLite) process execution, all drives are propagated into the design in one fell swoop. Thus, in VeraLite a non-blocking drive simply means that when the delay cycle count is not zero, the process that executes the drive continues to execute without blocking. It does not mean that those drives will be propagated as a Verilog non-blocking assignment (*NBA*)! Likewise, VeraLite blocking drives seem to imply that the driven value is propagated through the design immediately, perhaps unblocking other processes that are sensitive to the driven signal. As explained above, this is not the case: All drives are queued until the end of the VeraLite processing.

Conflict: VeraLite uses the same syntax as SystemVerilog for non-blocking drives, but their semantic is very different.
Resolution: Eliminate the confusion by simplifying VeraLite to support only one type of drive: the non-blocking drive (=). If users wish to emulate the current Vera non-blocking behavior, they can easily do so by means of an auxiliary thread:

```
ifc.sig <= value;    becomes    fork ifc.sig = value; join none;
```

If the non-blocking syntax is to be supported then it should behave as in SystemVerilog.

Clarification: The signal drive operator syntax may appear to be ambiguous with certain event control expressions in SystemVerilog. For example:

```
integer  j = 4;
@j a = b;
```

The last statement above has the same syntactical form as a signal drive. But, it has two different meanings: in SystemVerilog the process blocks until *j* changes value, whereas a signal-drive causes the process to block for *j* cycles.
Nevertheless, the compiler can easily resolve the ambiguity by examining the type of operand involved in the signal drive (a above). If the operand is defined in a clocking domain, the signal is synchronous and should be driven using cycle semantics via a signal drive. Otherwise, the statement is a regular event control assignment.

8.2.1 Drives (6-6)

Correction: The last line of the first paragraph states "Conflicting drives drive the signal to X and result in a simulation error". That description is inaccurate, it should state: "Conflicting drives drive the conflicting bits to X and result in a simulation error". Also see Section 8.2 for an explanation of blocking vs. non-blocking VeraLite drives.

8.3 Sampling a Signal (6-6)

Correction: The note states that "when sampling a signal in an expression, it is done *immediately* (i.e., asynchronously)." . That is incorrect. A synchronous VeraLite signal always evaluates to the *sampled* value, i.e., the synchronous value. That statement would imply that the following code segments might yield different results:

```
// expression                    // assignment
integer y = ifc.sig;             integer y;
                                 y = ifc.sig;
```

8.4 Implicit Synchronization (6-7)

Clarification: Please disregard the second example:
foobus.strobe_1 == 1'b1;
That operation is not part of the donation.

8.5 Asynchronous Signal Operations (6-8)

Limitation: The **async** modifier is not supported by VeraLite. If a signal is truly asynchronous, it should not be declared in a synchronous clocking-domain.
Resolution: To create an asynchronous clocking domain, simply omit the CLOCK signal from the corresponding clocking-domain. When a clocking domain has no clock specification, VeraLite will consider all signals in the domain to be asynchronous: the signals will not be sampled at any clock edge, instead their instantaneous value will be used.

8.6 Sub-Cycle Delays (6-9)

Limitation: SystemVerilog allows physical time units to be specified, but the VeraLite delay task accepts only time ticks to specify a time.
Resolution: Remove this limitation. The delay task will accept both time ticks as well as physical time units.

Clarification: Calling the delay task **delay(n)** is analogous to a **# n** declaration. The only difference is that VeraLite always executes after non-blocking assignments (see Section 7.3). whereas the # operator does not specify this behavior. Also, delay(0), is simply ignored, to accomplish a delay(0) **suspend_thread()** (see Section 6.4).

9 Class and Methods

9.1 Objects and Instance of Classes (7-3)

Clarification: The donation states that an instance of a class is created using the **new** keyword. That should state using the **new** task.

Correction: The example lists task1 as:

task task1 (integer a, (obj_example myexample = null))
That notation is incorrect. The use of parenthesis to create optional arguments is not supported by VeraLite.

9.2 Constructors (7-5)

Clarification: Event though the **new** operation is defined as a task, it is treated as a function, that is, the **new** task may not block. The VeraLite compiler will issue an error if the new task is determined to be blocking.

Clarification: The donation doesn't mention that every class has a default (built-in) new method. That method first calls its parent class (super.new) and then proceeds to initialize each member of the current object to its default value.

9.3 External Classes (7-11)

Resolution: This use of the **extern** keyword is deprecated. See discussion in section 4.7

9.4 Typedef (7-11)

Overlap: The **typedef** keyword is used by SystemVerilog to define any arbitrary user-defined type. For example: typedef bit [4:1] nibble.
VeraLite uses the **typedef** keyword only for forward-referencing of class declarations in order to satisfy the rule that a type must be declared before its use.
Resolution: Extend SystemVerilog's **typedef** to allow forward references for classes. The two uses do not represent a conflict and can coexist without problem.
Note that the **class** construct always creates a type, it does not require a **typedef** declaration as in **typedef class …** . This is the same as in C++.

9.5 Classes, Structs, and Unions

SystemVerilog includes structs and unions. VeraLite supports the object-oriented class construct. On the surface, it might appear that class and struct provide equivalent functionality, and only one of them is needed. However, that is not true; classes differ from structs in four fundamental ways:

1. SystemVerilog structs are strictly static objects; they are created either in a static memory location (global or module scope) or on the stack of an automatic task. Conversely, VeraLite objects (i.e., class instances) are exclusively dynamic, their declaration doesn't create the object; that is done by calling **new**.

2. SystemVerilog structs are type compatible so long as their bit sizes are the same, thus copying structs of different composition but equal sizes is allowed. In contrast, VeraLite objects are strictly strongly-typed. Copying an object of one type onto an object of another is not allowed.
3. VeraLite objects are implemented using handles, thereby providing C-like pointer functionality. But, VeraLite disallows casting handles onto other data types, thus, unlike C, VeraLite handles are guaranteed to be safe (no memory crashes).
4. VeraLite objects form the basis of an Object-Oriented framework that provides true polymorphism. Class inheritance, abstract classes, and dynamic casting are powerful mechanisms that go way beyond the mere encapsulation mechanism provided by structs.

9.6 Automatic Memory Management

The memory used by VeraLite objects and strings is allocated dynamically. When objects are created, VeraLite allocates more memory. When an object is not needed anymore, VeraLite automatically reclaims the memory, making available for re-use. The automatic memory management system is an integral part of VeraLite. One might be tempted to think that a manual memory management system, such as the one provided by C's malloc and free, might be sufficient. However, VeraLite's (and SystemVerilog's) multi-threaded, re-entrant environment create many opportunities for users to *shoot themselves in the foot.* For example, consider the following example:

```
myClass  obj = new;
fork
   task1( obj );
   task2( obj );
join none
```

In this example, the main process (the one that forks off the two tasks) doesn't know when the two processes might be done using the object *obj.* Similarly, neither task1 nor task2 knows when any of the other processes will no longer be using the object *obj*. It is evident from this simple example that no single process has enough information to determine when it is safe to free the object. The only two options available to the user are (1) play it safe and never reclaim the object, or (2) add some form of reference count that can be used to determine when it might be safe to reclaim the object. Adopting the first option will cause the system to quickly run out of memory. The second option places a large burden on the users, who in addition to managing their test-bench, must also manage the memory using less than ideal schemes. To avoid these shortcomings, VeraLite manages all dynamic memory automatically. Users no longer need to worry about dangling references, premature memory free's, or memory leaks. The system will automatically reclaim any object that is no longer being used. In the example above, all that users do is assign **null** to the handle *obj* when they no longer need it.

9.7 Inheritance

The donation briefly refers to Object-Orientated Programming, but the section describing class inheritance is missing. The missing sections are attached below.
Inheritance introduces several new keywords (these have already been accounted for in Section 3.2). Those keywords are:

- **extends**
- **local**
- **protected**
- **super**
- **virtual**

9.7.1 Subclasses and Inheritance

In the previous section we have defined a class called Packet. Assume we want to extend this class so the packets can be chained together into a list. One solution would be to create a new class called LinkedPacket that contains a variable of type Packet.
Whenever we refer to a property of Packet, we need to reference the variable packet.

```
class LinkedPacket
{
   Packet packet;
   LinkedPacket next;

   function LinkedPacket get_next()
   {
      get_next = next;
   }
}
```

Since LinkedPacket is a specialization of Packet, a more elegant solution is to extend the class creating a new subclass that *inherits* the members of the parent class. Thus, for example, we could have:

```
class LinkedPacket extends Packet
{
   LinkedPacket next;
   function LinkedPacket get_next()
   {
      get_next = next;
   }
}
```

Now, all of the methods and properties of Packet are part of LinkedPacket - as if they were defined in LinkedPacket – and LinkedPacket has additional properties and methods. We can also override the parent's methods, changing their definitions.

9.7.2 Overriden Members

Subclass objects are also legal representative objects of their parent classes. For example, every LinkedPacket object is a perfectly legal Packet object.
We can assign the handle of a LinkedPacket object to a Packet variable:

```
LinkedPacket lp = new;
Packet p = lp;
```

In this case, references to p access the methods and properties of the Packet class. So, for example, if you have overridden properties and methods in LinkedPacket, when you reference these overridden members through p you get the original members in the Packet class. From p, new and all overridden members in LinkedPacket are hidden from you.

```
class Packet
{
   integer i = 1;
   function integer get()
   {
      get = i;
   }
}
class LinkedPacket extends Packet
{
   integer i = 2;
   function integer get()
   {
      get = -i;
   }
}

LinkedPacket lp = new;
Packet p = lp;
j = p.i;            // j = 1, not 2
j = p.get();        // j = 1, not -1 or –2
```

Note that this is different from the semantics of, for example, Java. In Java, you would get the *original* properties, but you would get the overridden methods of the child class. In VERA, to get the overridden method, the parent method needs to be declared virtual (see below).

9.7.3 super

The **super** keyword is used from within a derived class to refer to properties of the parent class. It is necessary to use **super** when the property of the derived class has been overridden, and cannot be accessed directly.

```
class Packet                          //parent class
{
   integer value;
   function integer delay()
   {
      delay = value * value;
```

```
    }
}

class LinkedPacket extends Packet   //derived class
{
    integer value;
    function integer delay()
    {
        delay = super.delay()+value * super.value;
    }
}
super.value;
super.delay();
```

The property may be a member declared a level up or a member inherited by the class one level up. There is no way to reach higher (for example, super.super.count is not allowed).
Subclasses are classes that are extensions of the current class. Whereas super-classes are classes that the "current" class is extended from, beginning with the original base class.

Note:
When using the **super** within **new**, it must be the first statement in the constructor.

9.7.4 Casting

It is always legal to assign subclass variable to a variable of superclass higher in the inheritance tree. It is never legal to directly assign a superclass variable to a variable of one of its subclasses. However, it may be legal to place the contents of the superclass handle in a subclass variable.

To check if the assignment will be legal, use the **cast_assign()** function.

The basic syntax for **cast_assign()** is:

```
function integer cast_assign(var destination_handle
          source_handle);
```

This function checks the hierarchy tree (super and subclasses) of the source_handle to see if it contains the class destination_handle. If it does, **cast_assign()** does the assignment; if it is not, **cast_assign()** generates an error and terminates.

A second version of this function allows you to check the results without generating an error:

```
cast_assign(destination_handle, source_handle, CHECK);
```

does the assignment and returns a 1 if the assignment is valid. Otherwise, it sets the destination handle to null and returns a 0.

9.7.5 Chaining Constructors

When a subclass is instantiated, one of the system's first actions is to invoke the class method **new().** The first, implicit action **new()** takes is to invoke the **new()** method of its superclass, and so on up the inheritance hierarchy. Thus, all the constructors are called, in the proper order, beginning with the base class and ending with the current class.
If the initialization method of the super-class requires arguments, you have two choices. If you want to always supply the same arguments, you can specify them at the time you extend the class:

```
class EtherPacket extends Packet(5) {
```

This will pass 5 to the **new** routine associated with Packet.

A more general approach is to use the **super** keyword, to call the superclass constructor:

```
task new() {
    super.new(5);
```

If you use this approach then this must be the first executable statement in new.

9.7.6 Data Hiding and Encapsulation

So far, we have made all of our properties and methods available to the outside world without restriction. However, for most data (and most subroutines) we want to hide them away from the outside world, "seal them away in the capsule" of the class. This keeps other programmers from relying on your specific implementation - so you can safely modify it later - and it also protects against accidental modifications to properties that are internal to the class. When all data becomes hidden - being accessed only by public methods - testing and maintenance of the code becomes much easier.
Unlabeled properties and methods are public, available to anyone who has access to the object's name.
A member identified as **local** is available only to methods inside the class. Further, these local members are not visible even to subclasses and cannot be inherited. Of course, non-local methods that access local properties or methods can be inherited, and work properly as methods of the subclass.
A protected property or method has all of the characteristics of a **local** member, except that it can be inherited; it is visible to subclasses.
Note that within the class, we can reference a *local* method or property of our class, even if it is in a *different* instance. For example

```
class Packet
{
    local integer i;
    function integer compare (Packet other)
    {
        compare = (this.i == other.i);
    }
}
```

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

In summary:

- Wherever possible, use **local** members. Hide members that the outside world doesn't need to know about.
- Use **protected** members if the outside world doesn't have a need to know, but subclasses might.
- Public access should only be allowed when it is absolutely necessary, and the access should be limited as much as possible. Generally, don't provide direct access to properties but rather provide access methods - provide, for example, only read access if a variable should never be written. This provides an extra level of protection and preserves flexibility for future changes.

9.7.7 Abstract Classes and Virtual Methods

Often we create a set of classes that can be viewed as all derived from a common base class. For example, we might start with a common base class of type BasePacket that sets out the structure of packets but is incomplete; we would never want to instantiate it. From this base class, though, we might derive a number of useful subclasses: Ethernet packets, token ring packets, GPSS packets, satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they could vary significantly in terms of their internal details.
We start by creating the base class that sets out the prototype for these subclasses. Since we don't need to instantiate the base class, we declare it to be *abstract* by declaring the class to be **virtua**l:

```
virtual class BasePacket {
```

By themselves, abstract classes are not tremendously interesting, but abstract classes can also have *virtual* methods. Virtual methods provide prototypes for subroutines, all of the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type if it is needed. Later, when subclasses override virtual methods, they must follow the prototype exactly. Thus, all versions of the virtual method will look identical in all subclasses:

```
virtual class BasePacket
{
   virtual protected function integer send(bit[31:0] data);
}
class EtherPacket extends BasePacket
{
   protected function integer send(bit[31:0] data)
   {
      // body of the function
```

```
        ...
    }
```

EtherPacket is now a class we can instantiate. In general, if an abstract class has several virtual methods, all of the methods must be overridden for the subclass to be instantiated. If all of the methods are not overridden, the subclass needs to be abstract.
Methods of normal classes can also be declared virtual. In this case, the method must have a body. If the method does have a body, then the class can be instantiated, as can its subclasses. However, if the subclass overrides the virtual method, then the new method must exactly match the superclass's prototype.

9.7.8 Polymorphism: Dynamic Method Lookup

Polymorphism allows us to use superclass variables to hold subclass objects, and to reference the methods of those subclasses directly from the superclass variable. As an example, consider the base class for our packet objects, BasePacket. Assume that it defines, as virtual functions, all of the public methods that are to be generally used by its subclasses, methods such as send, receive, print, etc. Even though BasePacket is abstract, we can still use it to declare a variable:

```
BasePacket packets[100];
```

We can now create instances of various packet objects, and we can put these into the array we just created:

```
EtherPacket ep = new;
TokenPacket tp = new;
GPSSPacket gp = new;
packets[0] = ep; packets[1] = tp; packets[2] = gp;
```

If our data types were, for example, integers, bits and strings, we couldn't store all of these types into a single array, but with *polymorphism* we can with objects. In this example, since the methods were declared as virtual, we can access the appropriate subclass methods from the superclass variable even though the compiler didn't know - at compile time - what was going to be loaded into, for example, packets[1]:

```
packets[1].send();
```

will invoke the send method associated with the TokenPacket class. At run-time, the system correctly binds the method from the appropriate class.
This is a typical example of polymorphism at work, providing capabilities that are far more powerful than what is found in a non-object-oriented language.

10 Linked Lists (8-1)

Clarification: The donation doesn't mention this, but, the Linked List package is analogous to the C++ STL (*Standard Template Library*) List container, that is popular

with C++ programmers. Note, however, that VeraLite doesn't support C++ *templates*. Instead, the generic code is accomplished via macros.

10.1 List Macros (8-2)

Correction: The only necessary macro is **MakeVeraList(type).** The second macro mentioned in the donation, **ExternVeraList** is unnecessary. Also, unnecessary is the inclusion of any header file other than ListMacros.vrh. In particular, the file VeraListProgram.vrh is deprecated and unnecessary.