In 6.22.1 Matching types

Replace

f) Two array types match if they have the same number of unpacked dimensions and their slowest varying dimensions have matching types and the same left and right range bounds. The type of the slowest varying dimension of a multidimensional array type is itself an array type.

```
typedef byte MEM_BYTES [256];
typedef bit signed [7:0] MY_MEM_BYTES [256]; // MY_MEM_BYTES matches
                                             // MEM_BYTES
typedef logic [1:0] [3:0] NIBBLES;
typedef logic [7:0] MY_BYTE; // MY_BYTE and NIBBLES are not matching types
```

With

f) Two array types match if they ~~have the same number of unpacked dimensions and their slowest varying dimensions have matching types and~~ are both packed or both unpacked, are the same kind of array (fixed-size, dynamic, associative, or queue), have matching index types (for associative arrays), and have matching element types. Fixed-size arrays shall also have the same left and right range bounds. ~~The type of the slowest varying dimension of a multidimensional array type~~ Note that the element type of a multiple dimension array is itself an array type.

```
typedef byte MEM_BYTES [256];
typedef bit signed [7:0] MY_MEM_BYTES [256]; // MY_MEM_BYTES matches
                                             // MEM_BYTES
typedef logic [1:0] [3:0] NIBBLES;
typedef logic [7:0] MY_BYTE; // MY_BYTE and NIBBLES are not matching types

typedef logic MD_ARY [][2:0];
typedef logic MD_ARY_TOO [][0:2]; // Does not match MD_ARY
```

In 6.22.2 Equivalent types

Replace

d) Unpacked array types are equivalent by having equivalent element types and identical shape. Shape is defined as the number of dimensions and the number of elements in each dimension, not the actual range of the dimension.

```
bit [9:0] A [0:5];
bit [1:10] B [6];
typedef bit [10:1] uint10;
uint10 C [6:1]; // A, B and C have equivalent types
typedef int anint [0:0]; // anint is not type equivalent to int
```

With

d) Unpacked fixed-size array types are equivalent ~~by having~~ if they have equivalent element types and ~~identical shape. Shape is defined as the number of dimensions and the number of elements in each dimension, not the actual range of the dimension~~ equal size; the actual range bounds may differ. Note that the element type of a multiple dimension array is itself an array type.

```
bit [9:0] A [0:5];
```

```
bit [1:10] B [6];
typedef bit [10:1] uint10;
uint10 C [6:1]; // A, B and C have equivalent types
typedef int anint [0:0]; // anint is not type equivalent to int
```

d) Dynamic array, associative array and queue types are equivalent if they are the same kind of array (dynamic, associative, or queue), have equivalent index types (for associative arrays), and have equivalent element types.

In 7.4 Packed and unpacked arrays

Replace

Arrays can be *fixed* or *dynamic*. Fixed-size unpacked arrays can be multidimensional and have fixed storage allocated for all the elements of the array. Each dimension of an unpacked array can be declared as having a fixed or unfixed size. A dynamic array allocates storage for elements at run time along with the option of changing the size of one of its dimensions. An associative array allocates storage for elements individually as they are written. Associative arrays can be indexed using arbitrary data types. A queue type of array grows or shrinks to accommodate the number of elements written to the array at run time.

With

~~Arrays can be *fixed* or *dynamic*. Fixed-size unpacked arrays can be multidimensional and have fixed storage allocated for all the elements of the array. Each dimension of an unpacked array can be declared as having a fixed or unfixed size. A dynamic array allocates storage for elements at run time along with the option of changing the size of one of its dimensions. An associative array allocates storage for elements individually as they are written. Associative arrays can be indexed using arbitrary data types. A queue type of array grows or shrinks to accommodate the number of elements written to the array at run time.~~

Unpacked arrays may be fixed-size arrays (see 7.4.2), dynamic arrays (see 7.5), associative arrays (see 7.9), or queues (see 7.11). Unpacked arrays are formed from any data type, including other packed or unpacked arrays (see 7.4.5).

In 7.4.2 Unpacked arrays

Replace

Unpacked arrays can be made of any data type, and can be used to group elements of the declared element type into multidimensional data. Arrays shall be declared by specifying the element address range(s) after the declared identifier.

With

Unpacked arrays can be made of any data type~~, and can be used to group elements of the declared element type into multidimensional data~~. Arrays whose elements are themselves arrays are declared as multiple dimension arrays (see 7.4.5). Unpacked ~~A~~arrays shall be declared by specifying the ~~element address range(s)~~ dimension(s) after the declared identifier.

Replace

Each dimension shall be represented by an address range, such as `[1:1024]`, or a single positive number to specify the size of an unpacked array, like C. In other words, `[size]` becomes the same as `[0:size-1]`.

The following examples declare equivalent size two-dimensional arrays of **`int`** variables:

With

Each fixed-size dimension shall be represented by an address range, such as `[1:1024]`, or a single positive number to specify the size of a~~n~~ fixed-size unpacked array, like C. In other words, `[size]` becomes the same as `[0:size-1]`.

The following examples declare equivalent size two-dimensional fixed-size arrays of **`int`** variables:

Remove

~~An unpacked array element can be assigned a value in a single assignment. To assign a value to an element of an array, an index for every dimension shall be specified. The index can be an expression.~~

~~Each dimension of an unpacked array can be declared as having a fixed or unfixed size. Fixed-size unpacked arrays can be multidimensional and have fixed storage allocated for all the elements of the array~~. If an unpacked array has one or more dynamic, associative, or queue~~d~~ dimensions, it is considered a variable-size array.

In 7.4.5 Multiple dimension arrays

Replace

The dimensions preceding the identifier set the packed size. The dimensions following the identifier set the unpacked size.

With

A multiple dimension array is an array of arrays. Multiple dimension arrays can be declared by including multiple dimensions in a single declaration. The dimensions preceding the identifier set the packed ~~size~~ dimensions. The dimensions following the identifier set the unpacked ~~size~~ dimensions.

Replace

When the array is used with a smaller number of dimensions, these have to be the slowest varying ones.

```
bit [9:0] foo6;
foo6 = foo1[2]; // a 10-bit quantity.
```

With

A *subarray* is an array that is an element of another array. As in the C language, subarrays are referenced by omitting indices for one or more array dimensions, always omitting the ones that vary most rapidly. Omitting indices for all the dimensions references the entire array. ~~When the array is used with a smaller number of dimensions, these have to be the slowest varying ones.~~

```
~~bit [9:0] foo6;~~
~~foo6 = foo1[2]; // a 10-bit quantity.~~
int A[2][3][4], B[2][3][4], C[5][4];
…
A[0][2] = B[1][1];  // assign a subarray composed of four ints
```

```
A[1] = B[0];  // assign a subarray composed of three arrays of four ints
              // each
A = B;        // assign an entire array
A[0][1] = C[4];     // assign compatible subarray of four ints
```

In 7.5 Dynamic arrays

Replace

A dynamic array is any dimension of an unpacked array whose size can be set or changed at run time. The space for a dynamic array does not exist until the array is explicitly created at run time.

The syntax to declare a dynamic array is as follows:

data_type array_name [];

where data_type is the data type of the array elements. Dynamic arrays support the equivalent types as fixed-size arrays.

For example:

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

The **new**[] operator is used to set or change the size of the array.

The size() built-in method returns the current size of the array.

The delete() built-in method clears all the elements yielding an empty array (zero size).

With

A dynamic array is ~~any dimension of~~ an unpacked array whose size can be set or changed at run time. ~~The space for a dynamic array does not exist until the array is explicitly created at run time.~~ The default size of an uninitialized dynamic array is zero. The size of a dynamic array is set by the **new** constructor or array assignment, described in 7.5.1 and 7.6 respectively. Dynamic arrays support ~~the equivalent types as fixed-size arrays~~ all variable data types as element types, including arrays.

Dynamic array dimensions are denoted in the array declaration by **[]**. Any unpacked dimension in an array declaration may be a dynamic array dimension.

For example:

bit [3:0] nibble[]; // Dynamic array of 4-bit vectors
integer mem[2][]; // Fixed-size unpacked array ~~Dynamic~~ composed of 2 dynamic subarrays of integers

Note that in order for an identifier to represent a dynamic array, it must be declared with a dynamic array dimension as the leftmost unpacked dimension

The **new**[] ~~operator~~ constructor is used to set or change the size of the array and initialize its elements (see 7.5.1).

The size() built-in method returns the current size of the array (see 7.5.2).

The delete() built-in method clears all the elements yielding an empty array (zero size) (see 7.5.3).

In 7.5.1 New[]

Replace

The built-in function **new** allocates the storage and initializes the newly allocated array elements either to their default initial value or to the values provided by the optional argument.

The prototype of the **new** function is as follows:

<box with grammar productions>

Syntax 7-3—Declaration of dynamic array new (excerpt from Annex A)

[expression **]**:
The number of elements in the array. The type of this operand is **longint**. It shall be an error if the value of this operand is negative.

(expression **)**:
Optional. An array with which to initialize the new array. If it is not specified, the elements of the newly allocated array are initialized to their default value. This array identifier must be a dynamic array of a data type equivalent to the array on the left-hand side, but it need not have the same size. If the size of this array is less than the size of the new array, the extra elements shall be initialized to their default value. If the size of this array is greater than the size of the new array, the additional elements shall be ignored.

This argument is useful when growing or shrinking an existing array. In this situation, the value of **(** expression **)** is the same as the left-hand side; therefore, 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);
```

The **new** operator follows the SystemVerilog precedence rules. Because both the square brackets [] and the parenthesis () have the same precedence, the arguments to this operator are evaluated left to right: **[** expression **]** first, and **(** expression **)** second.

With

The ~~built-in function~~ **new** constructor sets the size of a dynamic array ~~allocates the storage~~ and initializes its elements. ~~initializes the newly allocated array elements either to their default initial value or to the values provided by the optional argument.~~ It may appear in place of the right-hand side expression of variable declaration assignments and blocking procedural assignments when the left-hand side indicates a dynamic array.

~~The prototype of the **new** function is as follows:~~

<Note to editor: retain syntax box, update as in A.6.2>

Syntax 5-1—~~Declaration~~ Syntax of dynamic array new (excerpt from Annex A)

[expression **]**:

The desired size of the dynamic array. ~~number of elements in the array.~~ . The type of this operand is **longint**. It shall be an error if the value of this operand is negative.

(expression **)**:

Optional. An array with which to initialize ~~the new~~ the dynamic array. ~~If it is not specified, the elements of the newly allocated array are initialized to their default value. This array identifier must be a dynamic array of a data type equivalent to the array on the left-hand side, but it need not have the same size. If the size of this array is less than the size of the new array, the extra elements shall be initialized to their default value. If the size of this array is greater than the size of the new array, the additional elements shall be ignored.~~

~~This argument is useful when growing or shrinking an existing array. In this situation, the value of **(** expression **)** is the same as the left-hand side; therefore, 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);~~
```

~~The **new** operator follows the SystemVerilog precedence rules.~~ Because both the square brackets [] and the parenthesis () have the same precedence, the arguments to this ~~operator~~ constructor are evaluated left to right: **[** expression **]** first, and **(** expression **)** second.

Comment [MJB1]: Mantis 2176 says to strike this sentence.

Dynamic array declarations may include a declaration assignment with the **new** constructor as the right-hand side:

```
int arr1 [][2][3] = new [4];  // arr1 sized to length 4; elements are fixed-size arrays and so do
                              // not require initializing.
int arr2 [][] = new [4];      // arr2 sized to length 4; dynamic subarrays remain unsized and
                              // uninitialized.
int arr3 [1][2][] = new [4];  // Error – arr3 is not a dynamic array, though it contains dynamic
                              // subarrays.
```

Dynamic arrays may be initialized in procedural contexts using the **new** constructor in blocking assignments:

```
int arr[2][][];
arr[0] = new [4];       // dynamic subarray arr[0] sized to length 4.
arr[0][0] = new [2];    // legal, arr[0][n] created above for n = 0..3
arr[1][0] = new [2];    // illegal, arr[1] not initialized so arr[1][0] does not exist
arr[0][]  = new [2];    // illegal, syntax error - dimension without subscript on left hand side
arr[0][1][1] = new[2]; // illegal, arr[0][1][1] is an int, not a dynamic array.
```

In either case, if the **new** constructor call does not specify an initialization expression, the elements are initialized to the default value for their type.

The optional initialization expression is used to initialize the dynamic array. When present, it shall be an array that is assignment-compatible with the left-hand-side dynamic array.

```
int idest[], isrc[3] = '{5, 6, 7};
idest = new [3] (isrc); // set size and array element data values (5, 6, 7)
```

The size argument need not match the size of the initialization array. When the initialization array's size is greater, it is truncated to match the size argument; when it is smaller, the initialized array is padded with default values to attain the specified size.

```
int src[], dest1[], dest2[];
src = new [3] ('{2, 3, 4});
dest1 = new[2] (src); // dest1's elements are {2, 3}.
dest2 = new[4] (src); // dest1's elements are {2, 3, 4, 0}.
```

This behavior provides a mechanism for resizing a dynamic array while preserving its contents. An existing dynamic array can be resized by using it both as the left-hand side term and the initialization expression.

```
integer addr[]; // Declare the dynamic array.
addr = new[100]; // Create a 100-element array.
...
// Double the array size, preserving previous values.
// Preexisting references to elements of addr are outdated.
addr = new[200](addr);
```

Resizing or reinitializing a previously-initialized dynamic array using **new** is destructive; no preexisting array data is preserved (unless reinitialized with its old contents – see above), and all preexisting references to array elements become outdated.

In **7.6 Array assignments**

Replace

A dynamic array can be assigned to a fixed-size array of an equivalent type if the size of the dynamic array dimension is the same as the length of the fixed-size array dimension. Unlike assigning with a fixed-size array, this operation requires a run-time check that can result in an error, in which case no operation shall be performed.

```
int A[100:1]; // fixed-size array of 100 elements
int B[] = new[100]; // dynamic array of 100 elements
int C[] = new[8]; // dynamic array of 8 elements

A = B; // OK. Compatible type and same size
A = C; // type check error: different sizes
```

A dynamic array or a one-dimensional fixed-size array can be assigned to a dynamic array of a compatible type. In this case, the assignment creates a new dynamic array with a size equal to the length of the fixed-size array. For example:

With

Associative arrays are only assignment compatible with associative arrays (see 7.10.9). Fixed-size unpacked arrays, dynamic arrays and queues are assignment compatible with each other under the following conditions:

- Their element types shall be assignment compatible. Note that if the element type is any variety of array, then this is a recursive condition.
- If the destination is fixed-size, then the source and destination sizes shall be equal.

Since the size of a dynamic array or queue can only be determined at run time, failure of the above condition shall result in a run time error and ~~A dynamic array can be assigned to a fixed-size array of an equivalent type if the size of the dynamic array dimension is the same as the length of the fixed-size array dimension. Unlike assigning with a fixed-size array, this operation requires a run-time check that can result in an error, in which case~~ no operation shall be performed. Example code showing assignment of a dynamic array to a fixed-size array follows.

```
int A[2][100:1]; // fixed-size array of 100 elements
int B[] = new[100]; // dynamic array of 100 elements
int C[] = new[8]; // dynamic array of 8 elements
int D [3][][]; // multiple dimension array with dynamic subarrays
D[2] = new [2]; // initialize one of D's dynamic subarrays.
D[2][0] = new [100];
A[1] = B; // OK. Compatible type and same size. Both are arrays of 100 ints.
A[1] = C; // type check error: different sizes (100 vs. 8 ints)
A = D[2]; // A[0:1][100:1] and subarray D[2][0:1][0:99] both comprise 2 subarrays of 100 ints.
```

~~A dynamic array or a one-dimensional fixed-size array can be assigned to a dynamic array of a compatible type. In this case, the assignment creates a new dynamic array with a size equal to the length of the fixedsize array. For example:~~ Examples showing assignment to a dynamic array are below. (See 7.5.1 for additional assignment examples involving the dynamic array **new** constructor).

In **7.7 Arrays as arguments to tasks or functions**

Replace

Arrays can be passed as arguments to tasks or functions. The rules that govern array argument passing by value are the same as for array assignment (see 7.6). When an array argument is passed by value, a copy of the array is passed to the called task or function. This is true for all array types: fixed-size, dynamic, or associative.

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

<snip…>

```
int b[3:1][4:1]; // error: incompatible size (3 vs. 4)
```

A subroutine that accepts a one-dimensional fixed-size array can also be passed a dynamic array of a with compatible type of the same size.

For example, the declaration

```
task bar( string arr[4:1] );
```

declares a task that accepts one argument, an array of 4 strings. This task can accept the following actual arguments:

```
string b[4:1]; // OK: same type and size
string b[5:2]; // OK: same type and size
string b[] = new[4]; // OK: same type and size,
```

A subroutine that accepts a dynamic array can be passed a dynamic array of a compatible type or a one-dimensional fixed-size array of a compatible type.

For example, the declaration

```
task foo( string arr[] );
```

declares a task that accepts one argument, a dynamic array of strings. This task can accept any one-dimensional array of strings or any dynamic array of strings.

An import DPI function that accepts a one-dimensional array can be passed a dynamic array of a compatible type and of any size if formal is unsized and of the same size if formal is sized. However, a dynamic array cannot be passed as an argument if formal is an unsized output.

With

Arrays can be passed as arguments to tasks or functions. The rules that govern array argument passing by value are the same as for array assignment (see 7.6). When an array argument is passed by value, a copy of the array is passed to the called task or function. This is true for all array types: fixed-size, dynamic, queue, or associative.

The rules that govern whether an array actual argument can be associated with a given formal argument are the same as the rules for whether a source array's values can be assigned to a destination array (see 7.6). If a dimension of a formal is unsized (unsized dimensions can occur in dynamic arrays, queues, and ~~in~~ formal arguments of import DPI functions), then it matches any size of the actual argument's corresponding dimension ~~of an actual is accepted~~.

<snip…>

```
int b[3:1][4:1]; // error: incompatible size (3 vs. 4)
```

A subroutine that accepts a ~~one-dimensional~~ fixed-size array can also be passed a dynamic array or queue ~~of a~~ with compatible element type and equal ~~of the same~~ size.

For example, the declaration

```
task bar( string arr[4:1] );
```

declares a task that accepts one argument, an array of 4 strings. This task can accept the following actual arguments:

```
string b[4:1]; // OK: same type and size
string b[5:2]; // OK: same type and size
string b[] = new[4]; // OK: same type,number of dimensions and dimension size,
```

A subroutine that accepts a dynamic array or queue can be passed a dynamic array, queue, ~~of a compatible type~~ or ~~a one-dimensional~~ fixed-size array of a compatible type.

For example, the declaration

task foo(**string** arr[]);

declares a task that accepts one argument, a dynamic array of strings. This task can accept any one-dimensional unpacked array of strings or any one-dimensional dynamic array or queue of strings.

~~An import DPI function that accepts a one-dimensional array can be passed a dynamic array of a compatible type and of any size if formal is unsized and of the same size if formal is sized.~~
The rules that govern dynamic array and queue formal arguments also govern the behavior of unpacked dimensions of DPI open array formal arguments (see 7.6). DPI open arrays can also have a solitary unsized, packed dimension (see 34.5.6.1). ~~However, a~~ A dynamic array or queue ~~cannot~~ shall not be passed as an actual argument if the DPI formal argument has ~~is an~~ unsized dimensions and an output direction mode.

In A.2.4 Declaration assignments

<Note to editor: Make the same changes in syntax box in 6.7>

Replace

variable_decl_assignment ::=
…
dynamic_array_variable_identifier [] [= dynamic_array_new]

With

variable_decl_assignment ::=
…
dynamic_array_variable_identifier ~~[]~~ unsized_dimension { variable_dimension } [=
dynamic_array_new]

In A.6.2 Procedural blocks and assignments

<Note to editor: Make the same changes in syntax box in 7.5.1 and 10.4.1>

Replace

blocking_assignment ::=
…
hierarchical_dynamic_array_variable_identifier = dynamic_array_new

With

blocking_assignment ::=
…
~~hierarchical_dynamic_array_variable_identifier~~ nonrange_variable_lvalue = dynamic_array_new

In A.8.4 Primaries

Add

select ::=
[{ . member_identifier bit_select } . member_identifier] bit_select [[part_select_range]]

nonrange_select ::=
[{ . member_identifier bit_select } . member_identifier] bit_select

In A.8.5 Expression left-side values

Add at end

nonrange_variable_lvalue ::=
[implicit_class_handle . | package_scope] hierarchical_variable_identifier nonrange_select