PROPOSAL: A definition for unpacked array concatenation using { }

Relates to: SV Mantis item 1702
Applies to: IEEE 1800-2008 draft 4

Version 1, 18 Nov 2007

Version 1a, 19 Nov 2007:
fix some grammatical errors and improve clarity; changes in magenta

Version 2, 27 Nov 2007:
Clarify slice direction of array items in an unpacked array concatenation. Remove overlaps with Mantis 1447 so that the two proposals are largely orthogonal. Add rules about (lack of) persistence of references to queue members. Fix a few minor errors. Put LRM changes into correct order.

Version 3, 14 Dec 2007:
clarify reference-outdating behavior in 7.11.3; changes in magenta

Proposed change

Add a rigorous definition of unpacked array concatenation to support the examples shown in LRM **7.11.1** and elsewhere. Generalise this mechanism to be applicable to other forms of unpacked array. Review various areas of the LRM where description of queue behavior is unclear or inconsistent.

Discussion, and points to review

I believe that the new clause **10.10** is sufficient to define the intent without further elaboration. However, some of its effects are non-obvious and I have therefore added clauses **10.10.1** to **10.10.3** to work through the consequences in more detail. It could be argued that this material is in a tutorial style and doesn't belong in the LRM, but I believe that it's useful clarification and should stand. I am aware that the committees do not look favourably on large amounts of non-normative text, so I've left it normative.

The proposed text for **10.10** would legitimise the use of concatenation to create queue values in examples in clause **7.11.1** and elsewhere. Taken with the proposed new text in **7.6**, it also legitimises this otherwise incomprehensible use of concatenation to build a dynamic array's value, at the end of **7.6**:

```
string d[1:5] = '{ "a", "b", "c", "d", "e" };
string p[];
p = { d[1:3], "hello", d[4:5] };
```

Earlier versions of this proposal had various changes to clause **7,** attempting to make it clear that queues, dynamic arrays and fixed-size unpacked arrays are all assignment-compatible with one another provided they have elements of assignment-compatible type and the target's size can accommodate the source expression. These proposals have now been moved out into **Mantis item 1447** and do not appear here.

There is a potentially controversial change to persistence of reference to queue elements, documented in the new clause **7.11.3** and a corresponding change to **13.5.2**. This change has also had a minor effect on comments in the examples in **7.11**.

Jonathan Bromley, 14 Dec 2007

ADD new text after the second paragraph of 7.11:

Queues are declared using the same syntax as unpacked arrays, but specifying `$` as the array size. The maximum size of a queue can be limited by specifying its optional right bound (last index).

Queue values may be written using assignment patterns or unpacked array concatenations (see **10.9**, **10.10**).

ADD new text after the second paragraph of 7.11.1:

Queues ~~and dynamic arrays have the same assignment and argument passing semantics. Also, queues~~ shall support the same operations that can be performed on unpacked arrays. ~~and use the same operators and rules except as defined below:~~ In addition, queues shall support the following operations:

- A queue shall resize itself to accommodate any queue value that is written to it, except that its maximum size may be bounded as described in **7.11**.
- In a queue slice expression such as `Q[a:b]`, the slice bounds may be arbitrary integral expressions and, in particular, are not required to be constant expressions.
- Queues shall support methods as described in **7.11.2**.

DELETE statements of equivalence in 7.11.2.2, .3, .4, .5, .6, .7 :

~~`Q.insert(i, e)` is equivalent to: `Q = {Q[0:i-1], e, Q[i,$]}`~~
~~`Q.delete(i)` is equivalent to: `Q = {Q[0:i-1], Q[i+1,$]}`~~
~~`e = Q.pop_front()` is equivalent to: `e = Q[0]; Q = Q[1,$]`~~
~~`e = Q.pop_back()` is equivalent to: `e = Q[$]; Q = Q[0,$-1]`~~
~~`Q.push_front(e)` is equivalent to: `Q = {e, Q}`~~
~~`Q.push_back(e)` is equivalent to: `Q = {Q, e}`~~

ADD new clause 7.11.3:

7.11.3 Persistence of references to elements of a queue

As described in **13.5.2**, it is possible for an element of a queue to be passed by reference to a task that continues to hold the reference while other operations are performed on the queue. Some operations on the queue shall cause any such reference to become outdated (as defined in **13.5.2**). This clause **7.11.3** defines the situations in which a reference to a queue element shall become outdated.

When any of the queue methods described in **7.11.2** updates a queue, a reference to any existing element that is not deleted by the method shall not become outdated. All elements that are removed from the queue by the method shall become outdated references.

When the target of an ~~any form of~~ assignment is an entire ~~updates a~~ queue, references to any element of the original queue shall become outdated.

As a consequence of this clause, inserting elements in a queue using unpacked array concatenation syntax, as illustrated in the examples in **7.11.1**, will cause all references to any element of the existing queue to become outdated. Use of the `delete`, `pop_front` and `pop_back` methods will outdate any reference to the popped or deleted element, but will leave references to all other elements of the queue unaffected. By contrast, use of the `insert`, `push_back` and `push_front` methods on a queue can never give rise to outdated references (except that `insert` or `push_front` on a bounded queue would cause the highest-numbered element of the queue to be deleted if the new size of the queue were to exceed the queue's bound).

ADD new clause 10.10… and renumber existing 10.10 (Net aliasing) to be 10.11

10.10 Unpacked array concatenation

Unpacked array concatenation provides a flexible way to compose an unpacked array value from a collection of elements and arrays. An unpacked array concatenation may appear as the source expression in an assignment-like context, and shall not appear in any other context. The target of such assignment-like context shall be an array whose slowest-varying dimension is an unpacked fixed-size, queue or dynamic dimension. A target of any other type (including associative array) shall be illegal.

An unpacked array concatenation shall be written as a comma-separated list, enclosed in braces, of zero or more items. If the list has zero items, then the concatenation shall denote an array value with no elements. Otherwise, each item shall represent one or more element of the resulting array value, interpreted as follows:

- an item whose self-determined type is assignment-compatible with the element type of the target array shall represent a single element;
- an item whose self-determined type is an array whose slowest-varying dimension's element type is assignment compatible with the element type of the target array shall represent as many elements as exist in that item, arranged in the same left-to-right order as they would appear in the array item itself;
- an item of any other type, or an item that has no self-determined type, shall be illegal except that the literal value **`null`** shall be legal if the target array's elements are of class type.

The elements thus represented shall be arranged in left-to-right order to form the resulting array. It shall be an error if the size of the resulting array differs from the number of elements in a fixed-size target. If the size exceeds the maximum number of elements of a bounded queue, then elements beyond the upper bound of the target shall be ignored and a warning shall be issued.

10.10.1 Unpacked array concatenations compared with array assignment patterns

Array assignment patterns have the advantage that they can be used to create assignment pattern expressions of self-determined type by prefixing the pattern with a type name. Furthermore, items in an assignment pattern can be replicated using syntax such as `'{ n{element} }`, and can be defaulted using the **`default:`** syntax. However, every element item in an array assignment pattern must be of the same type as the element type of the target array. By contrast, unpacked array concatenations forbid replication, defaulting and explicit typing, but they offer the additional flexibility of composing an array value from an arbitrary mix of elements and arrays. In some simple cases both forms can have the same effect, as in the following example:

```
int A3[1:3];

A3 = {1, 2, 3};   // unpacked array concatenation: A3[1]=1, A3[2]=2, A3[3]=3

A3 = '{1, 2, 3};  // array assignment pattern: A3[1]=1, A3[2]=2, A3[3]=3
```

The next examples illustrate some differences between the two forms:

```
typedef int AI3[1:3];

AI3 A3;

int A9[1:9];

A3 = '{1, 2, 3};


A9 = '{3{A3}};  // illegal, A3 is wrong element type

A9 = '{A3, 4, 5, 6, 7, 8, 9};  // illegal, A3 is wrong element type

A9 = {A3, 4, 5, A3, 6};  // legal, gives A9='{1,2,3,4,5,1,2,3,6}
```

```
A9 = '{9{1}};  // legal, gives A9='{1,1,1,1,1,1,1,1,1}
A9 = {9{1}};  // illegal, no replication in unpacked array concatenation

A9 = {A3, {4,5,6,7,8,9} };  // illegal, {...} is not self-determined here
A9 = {A3, '{4,5,6,7,8,9} };  // illegal, '{...} is not self-determined
A9 = {A3, 4, AI3'{5, 6, 7}, 8, 9};  // legal, A9='{1,2,3,4,5,6,7,8,9}
```

Unpacked array concatenation is especially useful for writing values of queue type, as shown in the examples in **7.11.1**.

10.10.2 Relationship with other constructs that use concatenation syntax

Concatenation syntax with braces can be used in other SystemVerilog constructs, including vector concatenation and string concatenation. These forms of concatenation are expressions of self-determined type, unlike unpacked array concatenation which does not have a self-determined type and which must appear as the source expression in an assignment-like context. If concatenation braces appear in an assignment-like context with an unpacked array target, they unambiguously act as unpacked array concatenation and must conform to the rules given in **10.10** above. Otherwise, they form a vector or string concatenation according to the rules given in **11.4.12**. The following examples illustrate how the same expression can have different meanings in different contexts without ambiguity.

```
string S, hello;
string SA[2];
byte B;
byte BA[2];

hello = "hello";

S = {hello, " world"};   // string concatenation: "hello world"
SA = {hello, " world"};  // array concatenation:
                         //SA[0]="hello", SA[1]=" world"

B = {4'h6, 4'hf};   // vector concatenation: B=8'h6f
BA = {4'h6, 4'hf};  // array concatenation: BA[0]=8'h06, BA[1]=8'h0f
```

10.10.3 Unpacked array concatenations and nested array expressions

Each item of an unpacked array concatenation is required to have self-determined type (see **10.10**), but a complete unpacked array concatenation has no self-determined type. Consequently it shall be illegal for an unpacked array concatenation to appear as an item in another unpacked array concatenation. This rule makes it possible for a vector or string concatenation to appear as an item in an unpacked array concatenation without ambiguity, as illustrated in the following example.

```
string S1, S2;
typedef string T_SQ[$];
T_SQ SQ;
S1 = "S1";
```

```
S2 = "S2";
SQ = '{"element 0", "element 1"};  // assignment pattern, two strings
SQ = {S1, SQ, {"element 3 is ", S2} };
```

In the last line of the example above, the outer pair of braces encloses an unpacked array concatenation whereas the inner pair of braces encloses a string concatenation, so that the resulting queue of strings is

```
'{"S1", "element 0", "element 1", "element 3 is S2"}
```

Alternatively the third item in the unpacked array concatenation could instead represent an array of strings, if it were written as an assignment pattern expression. The unpacked array concatenation would still be valid in this case, but now it would treat its third item as an array of two strings, each forming one element of the resulting array:

```
SQ = {S1, SQ, T_SQ'{"element 3 is ", S2} };
     // result:  '{"S1", "element 0", "element 1", "element 3 is ", "S2"}
```

With the exception of **`default:`** items, each item of an assignment pattern or an assignment pattern expression is in an assignment-like context (see **10.9**). Consequently an unpacked array concatenation may appear as a non-default item in an assignment pattern. The following example uses a two-dimensional queue to build a jagged array of arrays of `int`, using both an assignment pattern expression and unpacked array concatenations to represent the subarrays:

```
typedef int T_QI[$];
T_QI jagged_array[$] = '{ {1}, T_QI'{2,3,4}, {5,6} };
   // jagged_array[0][0] = 1  -- jagged_array[0] is a queue of 1 int

   // jagged_array[1][0] = 2  -- jagged_array[1] is a queue of 3 ints
   // jagged_array[1][1] = 3
   // jagged_array[1][2] = 4

   // jagged_array[2][0] = 5  -- jagged_array[2] is a queue of 2 ints
   // jagged_array[2][1] = 6
```

REPLACE last-but-one item in bullet-list in 13.5.2:

— The element of an associative array being referenced is deleted with the `delete()` method.

~~— A queue is assigned with an array aggregate expression that does not explicitly contain the element being referenced.~~

— The queue or dynamic array containing the referenced element is updated by assignment.

— The element of a queue being referenced is deleted by a queue method.