VHDL-200x Data Types and Abstractions
White Paper 2

Associative Arrays

Peter Ashenden, Ashenden Designs
peter@ashenden.com.au

Version 1, 21 April 2004

Abstract

This white paper proposes a package for associative arrays requested as a language feature for testbenches and verification. The package uses the extended generics feaures proposed in WP-001

Revision History

Version 1. 21-Apr-04, Peter Ashenden. Initial version based on type generics in WP-001-v2.

1 Requirements

Proposal TBV2 from the Test Bench and Verification team identifies a requirement for an associative array data type. Such a data type is essentially a partial mapping from an index type to an element type. The requirements included in TBV2 are

- The number of extant mappings (the "size" of the array) can be determined.
- For a given index value, the mapping to an element may or may not exist. Existence can be queried, and a mapping can be added, changed and deleted.
- The index type is ordered.
- Provision be made for iteration over the extant mappings.
- Provision be made to load an associative array from a file and to dump the contents of an associative array to a file.
- Provision be made for assignment of the contents of one associative array to an object of the same associative array type.
- Provision be made for passing associative arrays as parameters to subprograms.

2 Alternative Implementations of Associative Arrays

The requirements for associative arrays can be met with a generic package that defines an abstract data type (ADT). The ADT defines a type for associative arrays and a number of operations, provided as subprograms, for working with ADT values.

There are numerous tradeoffs that can be made when choosing and implementation for the ADT, including space vs performance tradeoffs. In particular, the choice of data structure used will affect storage space and runtime performance. Since the number of extant mappings in an associative array is not statically known, a dynamically-allocated data structure could be used. In the context of VHDL, however, a dynamically allocated structure cannot be used as the value of a signal. If that were required, a statically allocated data structure could be used, albeit at the cost of imposing a limit on the number of extant mappings in a given associative array value.

3 A Binary Tree Implementation

One dynamically allocated data structure that can be used is a binary tree, in which each node stores an index value and the element mapped from that element. The left subtree of a node contains all mappings for lesser indices, and the right subtree contains all mappings for greater indices.

3.1 A Binary Tree Package Declaration

A package declaration using the binary tree data structure is

```
package associative_arrays is
  generic ( type index_type;
            type element_type;
            function "<"( L, R : index_type ) return boolean is <> );

  type associative_array;

  -- tree_record and structure of associative_array are private
  type tree_record is record
    index : index_type;
    element : element_type;
    left_subtree, right_subtree : associative_array;
  end record tree_record;
  type associative_array is access tree_record;

  function size ( a : in associative_array ) return natural;

  function exists ( a : in associative_array;
                    i : in index_type ) return boolean;

  function get ( a : in associative_array;
                 i : in index_type ) return element_type;

  procedure set ( a : inout associative_array;
                  i : in index_type;
                  e : in element_type );

  procedure delete ( a : inout associative_array;
                     i : in index_type );

  procedure delete_all ( a : inout associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate ( a : in associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate_reverse ( a : in associative_array );

  procedure copy ( a1 : in associative_array;
                   a2 : out associative_array );

  generic ( procedure read ( file f : std.textio.text;
                             i : out index_type;
                             e : out element_type ) )
  procedure load ( file f : std.textio.text;
                   a : inout associative_array );

  generic ( procedure write ( file f : std.textio.text;
                              i : in index_type;
```

```
                              e : in element_type ) )
  procedure dump ( file f : std.textio.text;
                   a : in associative_array );

end package associative_arrays;
```

The package has three generics. The `index_type` and `element_type` generics are used as the types of indices and elements, respectively. Since the index type is ordered, there should be an ordering predicate that can be expressed as a “less than” operator. That operator function is the third generic of the package. The default value is whatever "<" operator for the index type is visible at the point of instantiating the package.

The type associative_array denotes the ADT defined by the package. The fact that it is implemented as a pointer to a binary tree is private to the package, but VHDL does not provide a mechanism for enforcing that.

The remainder of the package declaration is a collection of operations on ADT values. The iteration operations are expressed as generic procedures that have an action procedure to apply to each element. The iterators can be instantiated with different actions procedures to achieve different effects. The copy operation is needed to perform elementwise copy, as the built-in assignment operation would simply provide pointer aliasing. The load and dump procedures are also expressed as generic procedures, with generic subprograms to read an index and element value from a file and to write an index and element to a file, respectively.

3.2 A Binary Tree Package Body

A package body for the binary tree data structure is

```
package body associative_arrays is

  function size ( a : in associative_array ) return natural is
  begin
    if a = null then
      return 0;
    else
      return 1 + size(a.left_subtree) + size(a.right_subtree);
    end if;
  end function size;

  function exists ( a : in associative_array;
                    i : in index_type ) return boolean is
  begin
    if a = null then
      return false;
    elsif i = a.index then
      return true;
    elsif i < a.index then
      exists(a.left_subtree, i);
    else
      exists(a.right_subtree, i);
    end if;
  end function exists;

  function get ( a : in associative_array;
                 i : in index_type ) return element_type is
  begin
    if a = null then
      report “associative_arrays.get: no element at given index”
        severity failure;
    elsif i = a.index then
      return a.element;
    elsif i < a.index then
      return get(a.left_subtree, i);
```

```
  else
    return get(a.right_subtree, i);
  end if;
end function get;

procedure set ( a : inout associative_array;
                i : in index_type;
                e : in element_type ) is
begin
  if a = null then
    a := new tree_record'(i, e, null, null);
  elsif i = a.index then
    a.element := e;
  elsif i < a.index then
    set(a.left_subtree, i, e);
  else
    set(a.right_subtree, i, e);
  end if;
end procedure set;

procedure delete ( a : inout associative_array;
                   i : in index_type ) is

  variable node_to_delete : associative_array;

  procedure remove_least_node ( a : inout associative_array;
                                n : out associative_array ) is
  begin
    if a.left_subtree = null then
      n := a;
      a := a.right_subtree;
    else
      remove_least_node(a.left_subtree, n);
    end if;
  end procedure remove_least_node;

begin
  if a = null then
    return;
  elsif i = a.index then
    node_to_delete := a;
    if a.right_subtree = null then
      a := a.left_subtree;
    else
      remove_least_node(node_to_delete.right_subtree, a);
      a.left_subtree := node_to_delete.left_subtree;
      a.right_subtree := node_to_delete.right_subtree;
    end if;
    deallocate(node_to_delete);
  elsif i < a.index then
    delete(a.left_subtree, i);
  else
    delete(a.right_subtree, i);
  end if;
end procedure delete;

procedure delete_all ( a : inout associative_array ) is
begin
```

```
  if a = null then
    return;
  else
    delete_all(a.left_subtree);
    delete_all(a.right_subtree);
    deallocate(a);
  end if;
end procedure delate_all;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate ( a : in associative_array ) is

  procedure recursive_iterate ( a : in associative_array ) is
  begin
    if a = null then
      return;
    else
      recursive_iterate(a.left_subtree);
      action(a.index, a.element);
      recursive_iterate(a.right_subtree);
    end if;
  end procedure recursive_iterate;

begin
  recursive_iterate(a);
end procedure iterate;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate_reverse ( a : in associative_array );

  procedure recursive_iterate_reverse ( a : in associative_array ) is
  begin
    if a = null then
      return;
    else
      recursive_iterate_reverse(a.right_subtree);
      action(a.index, a.element);
      recursive_iterate_reverse(a.left_subtree);
    end if;
  end procedure recursive_iterate;

begin
  recursive_iterate_reverse(a);
end procedure iterate_reverse;

procedure copy ( a1 : in associative_array;
                 a2 : out associative_array ) is
begin
  if a1 = null then
    a2 := null;
  else
    a2 := new tree_record'(a1.index, a1.element, null, null);
    copy(a1.left_subtree, a2.left_subtree);
    copy(a1.right_subtree, a2.right_subtree);
  end if;
end procedure copy;

generic ( procedure read ( file f : std.textio.text;
```

```
                              i : out index_type;
                              e : out element_type ) )
    procedure load ( file f : std.textio.text;
                     a : inout associative_array ) is
      variable i : index_type;
      variable e : element_type;
    begin
      delete_all(a);
      while not endfile(f) loop
        read(f, i, e);
        set(a, i, e);
      end loop;
    end procedure load;

    generic ( procedure write ( file f : std.textio.text;
                                i : in index_type;
                                e : in element_type ) )
    procedure dump ( file f : std.textio.text;
                     a : in associative_array ) is

      procedure dump_pair ( i : in index_type; e : in element_type ) is
      begin
        write(f, i, e);
      end procedure dump_pair;

      procedure iterate_dump_pair is new iterate
        generic map ( action => dump_pair );

    begin
      iterate_dump_pair(a);
    end procedure dump;

  end package associative_arrays;
```

The implementation of the ADT operations is expressed using recursive subprograms. The size operation, for example, tests whether the tree is empty, and returns 0 if so. Otherwise, it returns the 1 (for the root node) plus the sum of the sizes of the two subtrees.

The exists operation tests whether there is a mapping for a given index value. If the tree is empty, there is no mapping, so the function returns false. If the sought index value is the same as the index value stored at the root node, the function returns true. Otherwise, the sought index value must be either less than or greater than that stored at the root node. The function thus recursively tests for existence in the left or right subtree, depending on the relationship between the sought index and the root-node index. It uses the "<" function provided as a generic subprogram to perform the comparison.

The get operation accesses the element mapped from a given index value. It is similar to the exists operation, but instead of returning true when a match is found, it returns the element value at the matching node. If no match is found, the function reports an assertion violation with severity failure.

The set operation updates the mapping from a given index value if such a mapping exists, or adds the mapping otherwise. If the tree is empty, the mapping does not exists, so it is added, creating a singleton tree. Creation of the tree updates the null pointer passed in as the procedure argument. If the root node index is the argument index, the mapping does exist, and is updated. Otherwise, the mapping is sought and updated in either the left or right subtree, depending on the relationship between the sought index and the root-node index.

The delete operation removes a mapping from a given index if such a mapping exists, otherwise it has no effect. The operation is more complex than those previously described, due to the need to rearrange the tree around the deleted node. The procedure first checks for an empty tree. In that case, there is no mapping from the given index, so the procedure simply returns.

In the case of the given index matching the root node index, the root node is the one that must be deleted. The procedure keeps a pointer to that node in the variable node_to_delete. The plan is then to move the node that is next in order of index up to the place of the deleted node. However, if the right subtree of the node to be deleted is empty, there is no such element within the tree rooted at the node to be deleted. So, instead, the entire left subtree, is moved up in place of the deleted node.

If there is a non-empty right subtree for the deleted node, the node with next greatest index value is found by traversing down the leftmost branch of that subtree until a node is found with no left subtree. This is done by the procedure remove_least_node, applied to the right subtree of the deleted node. The procedure returns a pointer to the least node in the parameter n, and updates the pointer to the least node to point to the least node's right subtree. This has the effect of removing the least node from the tree and splicing its subtree onto the least node's parent node.

The actual parameter to the call to remove_last_node is the pointer to the deleted node in the tree. Since the remove_last_node procedure updates that actual parameter with the pointer to the least node, the effect is to move the least node up to where the deleted node was in the tree. The subsequent two assignments then reattach the deleted node's subtrees to the repositioned node. Once that is done, storage for the deleted node is deallocated.

The final part of the delete procedure deals with the case of the deleted index not matching the root index. The mapping for the given index is deleted from the left or right subtree, as appropriate.

The delete_all operation deletes all mappings. For an empty tree, there is nothing to do. Otherwise, the procedure deletes all mappings in the left and right subtrees of the root node, then deallocates storage for the root node itself.

The iterate operation calls an action procedure for each mapping, in order. Note that the iterate procedure itself is a generic procedure, and so cannot be called recursively. Hence, the iterate procedure declares a local recursive procedure, recursive_iterate, to do the work. For an empty tree, there is nothing to do. Otherwise, the procedure recursively applies the iteration to the left subtree (all of whose nodes have lesser index values), then calls the action procedure for the index and element values in the root node, and finally applies the iteration to the right subtree (all of whose nodes have greater index values).

The iterate_reverse operation is similar to the iterate operation, except that it recursively applies the iteration to the right subtree before the root node, and to the left subtree after the root node. The efect is to iterate over mappings in order of greatest to least index value.

The copy operation is a "deep copy," which creates a new tree identical to an original tree. While it would be possible to implement this with an iteration over the original tree, using an action procedure to set a mapping in the new tree, the effect would be to create a "vine" structured tree, with consequent performance degradation. The approach adopted here is simply to replicate the original tree's structure. If the original tree is empty, the new tree is made empty. Otherwise, the root node is replicated, and recursive calls are made to copy the original left and right subtrees to the new left and right subtrees, respectively.

The load operation reads the contents of a text file to load an associative array. The generic procedure read is supplied as the action to read an index value and an element value from a textfile. The load procedure first deletes all extant mappings from the associative array. Then, so long as the end of the file has not been reached, the procedure reads the next index and element values from the file and adds the mapping to the associative array.

The dump operation writes the contents of an associative array to a file. Similarly to the load procedure, there is a generic procedure write to write an index value and an element value to a text file. The dump procedure makes use of the iterate generic procedure to write mappings in order. To do this, it defines an action procedure, dump_pair, to dump an index/element pair to the file, and instantiates the iterate procedure with this action. The dump procedure simply invokes the instantiated iterate procedure.

4 A Vector Implementation

A statically allocated data structure that can be used is a vector of index/element pairs. Mappings are stored in order of increasing index, with the size of the vector determining the maximum number of mappings that can be stored. In order to be able to track the number of extant mappings, the vector and a count of mappings is kept in a record data structure that represents an associative array.

4.1 A Binary Tree Package Declaration

A package declaration using the vector data structure is

```
package associative_arrays is
  generic ( type index_type;
            type element_type;
            max_size : positive;
            function "<"( L, R : index_type ) return boolean is <> );

  -- map_record and structure of associative_array are private
  type map_record is record
    index : index_type;
    element : element_type;
  end record map_record;
  type map_vector is array ( natural range <> ) of map_record;
  type associative_array is record
    size : natural range 0 to max_size;
    maps : map_vector(1 to max_size);
  end record;

  function size ( a : in associative_array ) return natural;

  function exists ( a : in associative_array;
                    i : in index_type ) return boolean;

  function get ( a : in associative_array;
                 i : in index_type ) return element_type;

  procedure set ( a : inout associative_array;
                  i : in index_type;
                  e : in element_type );

  procedure delete ( a : inout associative_array;
                     i : in index_type );

  procedure delete_all ( a : inout associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate ( a : in associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate_reverse ( a : in associative_array );

  procedure copy ( a1 : in associative_array;
                   a2 : out associative_array );

  generic ( procedure read ( file f : std.textio.text;
                             i : out index_type;
                             e : out element_type ) )
  procedure load ( file f : std.textio.text;
                   a : inout associative_array );

  generic ( procedure write ( file f : std.textio.text;
                              i : in index_type;
                              e : in element_type ) )
  procedure dump ( file f : std.textio.text;
                   a : in associative_array );

end package associative_arrays;
```

The package is almost identical to that for the tree data structure. One difference is the extra generic constant, max_size, that specifies the maximum number of mappings that can be stored in a given associative array value. Should different associative arrays be needed with different capacities, the package could instantiated multiple times with different values for the max_size generic.

The other difference is the concrete type used to represent an associative array. It is a record containing an element, size, that indicates how many mappings are extant in the array, and an element, maps, that is a vector of index/element records. The extant mappings are stored in vector elements starting from 1 and proceeding in order of index up to the value of the size element of the associative array record.

The remainder of the package declaration is the same collection of operations on ADT values that were declared in the binary tree version of the package. A model using associative arrays could change implementation simply by changing package instantiations. Application of operations is independent of the underlying implementation, except in contexts where dynamically allocated data structures are not allowed.

4.2 A Vector Package Body

A package body for the vector data structure is

```
package body associative_arrays is

  function size ( a : in associative_array ) return natural is
  begin
    return a.size;
  end function size;

  function exists ( a : in associative_array;
                    i : in index_type ) return boolean is
  begin
    for j in 1 to a.size loop
      if i = a.maps(j).index then
        return true;
      end if;
    end loop;
    return false;
  end function exists;

  function get ( a : in associative_array;
                 i : in index_type ) return element_type is
  begin
    for j in 1 to a.size loop
      if i = a.maps(j).index then
        return a.maps(j).element;
      end if;
    end loop;
    report "associative_arrays.get: no element at given index"
      severity failure;
  end function get;

  procedure set ( a : inout associative_array;
                  i : in index_type;
                  e : in element_type ) is
    variable j : positive range 1 to max_size+1;
  begin
    j := 1;
    while j <= a.size and a.maps(j).index < i loop
      j := j + 1;
    end loop;
    if j <= a.size and i = a.maps(j).index then
      a.maps(j).element = e;
```

```
  elsif a.size = max_size then
    report “associative_arrays.set: no space to insert new element”
      severity failure;
  else
    a.maps(j+1 to a.size+1) := a.maps(j to a.size);
    a.maps(j).index := i;
    a.maps(j).element := e;
    a.size := a.size + 1;
  end if;
end procedure set;

procedure delete ( a : inout associative_array;
                   i : in index_type ) is
  variable j : positive range 1 to max_size+1;
begin
  j := 1;
  while j <= a.size and a.maps(j).index < i loop
    j := j + 1;
  end loop;
  if j > a.size or a.maps(j) /= i then
    return;
  else
    a.maps(j to a.size-1) := a.maps(j+1 to a.size);
    a.size := a.size - 1;
  end if;
end procedure delete;

procedure delete_all ( a : inout associative_array ) is
begin
  a.size := 0;
end procedure delate_all;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate ( a : in associative_array ) is
begin
  for j in 1 to a.size loop
    action(a.maps(j).index, a.maps(j).element);
  end loop;
end procedure iterate;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate_reverse ( a : in associative_array );
begin
  for j in a.size downto 1 loop
    action(a.maps(j).index, a.maps(j).element);
  end loop;
end procedure iterate_reverse;

procedure copy ( a1 : in associative_array;
                 a2 : out associative_array ) is
begin
  a2 := a1;
end procedure copy;

generic ( procedure read ( file f : std.textio.text;
                           i : out index_type;
                           e : out element_type ) )
procedure load ( file f : std.textio.text;
```

```
                    a : inout associative_array ) is
      variable i : index_type;
      variable e : element_type;
    begin
      delete_all(a);
      while not endfile(f) loop
        read(f, i, e);
        set(a, i, e);
      end loop;
    end procedure load;

    generic ( procedure write ( file f : std.textio.text;
                                i : in index_type;
                                e : in element_type ) )
    procedure dump ( file f : std.textio.text;
                    a : in associative_array ) is

      procedure dump_pair ( i : in index_type; e : in element_type ) is
      begin
        write(f, i, e);
      end procedure dump_pair;

      procedure iterate_dump_pair is new iterate
        generic map ( action => dump_pair );

    begin
      iterate_dump_pair(a);
    end procedure dump;

  end package associative_arrays;
```

The implementation of the ADT operations is, in most cases, simpler for the vector data structure than for the binary tree, albeit at the cost of storage space consumed by partially populated vectors. The size operation, for example, simply returns the value of the size element of the associative array record.

The exists operation scans the maps vector starting from element 1 up to the last populated map. If it finds map with index value equal to the sought index, the function returns true. If the loop completes without finding such a map, the function returns false.

The get operation is similar to the exists operation, but instead of returning true when a match is found, it returns the element value from the matching map. If no match is found, the function reports an assertion violation with severity failure.

The set operation is somewhat more complex than its binary-tree counterpart. It starts by scanning the maps vector until it reaches the end or finds a map whose index is greater than or equal to the sought index. The position of the scan in the vector is maintained in the variable j. If, on completion of the scan, j refers to an extant map whose index equals the sought map, that map is simply updated. Otherwise, a new map needs to be inserted into the vector. If the associative array is already at full capacity, the procedure reports an error message. If there is room for a further map, maps from j upwards are moved along one position and the map at position j is updated with the new map index and element. Finally, the size of the associative array is incremented.

The delete operation, on the other hand, is a lot simpler than its binary-tree counterpart. The procedure starts by scanning the maps vector until it reaches the end or finds a map whose index is greater than or equal to the index to be deleted. On completion of the scan, if the position variable, j, is past the end of the extant mappings or the map at the scanned position is not equal to the index to be deleted, there is no mapping with the index to be deleted, so the procedure simply returns. Otherwise, the mapping to be deleted is extant and is at the position given by j. Maps after j in the vector are moved down one position, overwriting the map to be deleted. Finally, the size of the associative array is decremented.

The next four operations are all very simple. The delete_all operation deletes all mappings simply by setting the size of the associative array to 0. The iterate and iterate_reverse operations consist of for loops that scan the extant mappings, calling the action procedure for each one. The copy operation uses the built-in variable assignment operation to perform the copy.

The load and dump procedures are exactly the same as their binary-tree counterparts. This is because they are implemented using the other ADT operations defined in the package. They do not rely on the concrete data type used to implement the associative array type.

5 Examples of Package Usage

Suppose a model requires an associative array of bit-vector test patterns that use time values as the index type. If we are using the binary-tree implementation, the package may be instantiated as shown below. Since the predefined function "<"operating on time values is visible at the point of instantiation, it is used as the actual function for the formal function "<".

```
type test_pattern is bit_vector(0 to input_size-1);

package test_patterns is new work.associative_arrays
   generic map ( index_type => time,
                 element_type => test_pattern );
```

If we were to use the vector implementation and needed to allow for 1000 test pattens, we would instantiate the package as

```
package test_patterns is new work.associative_arrays
   generic map ( index_type => time,
                 element_type => test_pattern;
                 max_size => 1000 );
```

We can create a variable in which to store test patterns and a procedure to load test patterns from a file as follows:

```
variable patterns_to_apply : test_patterns.associative_array;

procedure read_pattern ( file f : std.textio.text;
                         t : out delay_length;
                         p : test_pattern ) is
  use std.textio.all;
  variable L : line;
begin
  readline(f, L);
  read(L, t);
  read(L, p);
end procedure read_pattern;

procedure load_patterns is new test_patterns.load
  generic map ( action => read_pattern );
```

We can also instantiate the iterate procedure to apply test patterns to a signal as follows:

```
signal test_input_signal : test_pattern;

procedure apply_pattern ( t : in delay_length;
                          p : in test_pattern ) is
begin
  wait for t - now;
  test_input_signal <= p;
end procedure apply_pattern;

procedure apply_patterns is new test_patterns.iterate
  generic map ( action => apply_pattern );
```

A process in a testbench incorporating these definitions can now call the instantiated procedures to load and apply the test patterns:

```
load_patterns(pattern_file, patterns_to_apply);
apply_patterns(patterns_to_apply);
```

6 Conclusions

In this white paper, we have illustrated how the proposed genericity extensions for VHDL can be used to describe an abstract data type (ADT) for use in a testbench and verification context. Different implementations of the ADT can be developed to meet various requirements on storage, performance and application context.

The associative array feature is one of several requested in the VHDL-200x Test Bench and Verification area. These requests could be met with a library of packages providing abstract data types with alternative implementations. Such a library would be similar in nature to standard libraries provided with programming languages, such as the C++ Standrad Template Library, and the collections classes in the Java class library.