A Comparison of Recursive and Repetitive Models of Recursive Hardware Structures

Peter J. Ashenden

Department of Computer Science
The University of Adelaide, SA 5005
Australia

Abstract

This paper examines techniques for developing VHDL descriptions of recursive hardware structures, using repetitive and repetitive component instantiations. A fat tree interconnection network is examined as an example of a recursive hardware structure. The recursive description is easier to develop, and more clearly express the structure, making it easier to understand. The main difficulty in developing repetitive structures lies in devising a way of interconnecting the basic components comprising the structure. It is shown that the difficulties result from fundamental language design decisions made in VHDL, and that it is not appropriate to modify the language to avoid the difficulties. Hence, in general, the recursive style is preferred over the repetitive style for describing recursive hardware structures.

1. Introduction

Many of the hardware structures that are designed consist of a number of instances of a single basic component, interconnected in a regular pattern. For example, a register consists of a one-dimensional array of flip flop cells with inputs and outputs connected in parallel to buses that carry words of data. Similarly, a memory circuit consists of a two-dimensional array of storage cells, with parallel connections to data and select signals.

In order to describe such regular structures, most hardware description languages, such as VHDL [2, 3, 4, 5] and Verilog [9], include some form of repetitive generator, allowing us to repeatedly instantiate a basic cell and its connections. Most languages also include a conditional generator that can be used to express variations within the regular structure, such as often occur at the boundaries. For example, the end cells of a shift register are not connected to neighboring cells, but to signals in the enclosing environment.

The mechanism provided in VHDL for generating regular structures is the generate statement. The language includes both repetitive and conditional forms of the generate statement, which achieve their effect when a description is elaborated prior to simulation or synthesis. The repetitive form specifies a discrete range of values, and for each of these it generates an instance of the statements in the body of the generate statement. For each repetition, the value from the range is bound to an identifier, and can be used as an index to select elements of signal arrays for connection to components. The conditional form of generate statement includes a boolean expression, which governs whether the statements comprising the body of the generate statement are included.

There is, however, an important class of regular structures, namely recursive structures, that can be difficult to describe with just these forms of generators. A recursive structure is one which is parameterized with respect to its size, and which is described in terms of smaller instances of the same structure. For example, a tree structure of height $h$ can be described as root cell connected to some number of sub-trees of height $h-1$. While it is straightforward to describe a recursive structure using just repetitive generators if the size is fixed and known a priori, in general it is difficult if the size is a parameter whose actual value is not known. This is a significant problem, since good design practice dictates that we should parameterize aspects of a system to make it resilient to changes in design specifications, and reusable in as many contexts as possible.

In this paper, we examine techniques for modeling parameterized recursive structures using VHDL. In particular, we examine, prepare and compare recursive and repetitive constructions using the VHDL generate statement to model such structures. The techniques are illustrated by an example structure: a fat tree interconnection network. (A somewhat simpler example, a fanout tree of buffers, is described in [1]. However, space limitations prevent its inclusion in this paper.) Comparisons between the recursive and repetitive descriptions highlight the simplifications afforded by the recursive descriptions.
In the following section, we define what we mean by a parameterized recursive hardware structure, and outline the technique for modeling such structures with recursive VHDL models. Then, in Section 3 we present recursive and repetitive VHDL models for the fat tree interconnection network. Section 4 compares the two styles of model, and discusses the issues in language design that arise. We draw our conclusions in Section 5.

2. Parameterized recursive structures

A parameterized recursive structure can be defined as a hardware entity \( E_s = (s, p_1, \ldots, p_m, I, O) \), where \( s \) is a non-negative integer parameter that specifies the size of the structure, \( p_1, \ldots, p_m \) are additional parameters that determine the implementation of the structure, \( I \) is a set of input ports, and \( O \) is a set of output ports. The sizes of \( I \) and \( O \) may depend on any or all of the parameters \( s \) and \( p_1, \ldots, p_m \).

Furthermore, the implementation of \( E \), includes one or more instances of \( E_{s,t} \) (\( s > t \) for some \( t \)), and the implementation of \( E \), contains no instances of \( E \), for any \( i \). This last condition on \( E \), provides a means of terminating the recursion.

A recursive structure can be expressed in VHDL using an entity declaration, with \( s \) and \( p_1, \ldots, p_m \) being represented by generic constants, and \( I \) and \( O \) being represented by ports of the entity. If the ports are of some unconstrained array base type, then the values of the generic constants can be used to constrain the index ranges of the ports, thus making their size depend on the entity parameters. The implementation of the entity is described in an architecture body, which uses conditional generators including the value of \( s \) in their conditions. If \( s = t \), a structure that does not include instances of the entity is generated. If \( s > t \), a structure is generated that includes instances of the entity, with the generic constant representing the size set to \( s-1 \).

A more general definition of a recursive structure is an entity \( E = (s_1, \ldots, s_m, p_1, \ldots, p_m, I, O) \), in which the single parameter specifying the size of the structure is replaced by \( m \) distinct parameters. The implementation of \( E \) includes further instances of \( E \), with values for \( s_1, \ldots, s_m \) calculated as functions of the parameters of the enclosing entity. Recursive instantiation must eventually terminate in an instance of \( E \) whose parameter values are such that its implementation involves no further recursive instances of \( E \). This more general form of recursive structure can be expressed in VHDL in a similar way to the simpler form. Conditional generators include the values of \( s_1, \ldots, s_m \) in their conditions to determine whether recursively instantiate the entity. In order to simplify the discussion in this paper, we will restrict ourselves to an example of the simpler form, although our comments and conclusions apply equally to models of more general recursive structures.

If the original 1987 version of VHDL [3, 5] is used, the instantiation of the entity within its own architecture body must be done indirectly by instantiating a declared component, which is bound to the entity in a separate configuration declaration. Furthermore, any signals used to interconnect the generated component instances must be declared in the outer part of the architecture body, rather than locally to the generate statements. The value of the generic constant representing the size of the structure may need to be used to determine how many signals to declare.

The more recent 1993 version of VHDL [2, 4] offers some simplification, particularly if the flexibility of indirect instantiation is not needed. The entity may be directly instantiated in the generated structure, and the interconnecting signals may be declared locally to the generate statements. For most of the example models in this report, the (currently) more widely used VHDL-87 version of the language will be used. In some cases, where the restrictions in VHDL-87 would make the model cumbersome, VHDL-93 is used and the reasons for its use are given.

3. The fat tree interconnection network

An interesting example of a recursive structure is a fat tree interconnection network, described by Leiserson in [6], and used in multicomputers such as the Thinking Machines CM-5 [7]. A fat tree can be composed of switch elements that route messages between elements of a multicomputer. Each switch has \( c \) child connections and \( p \) parent connections. It can route messages between its children directly, and can forward messages for nephews via any of its parents. Switches are composed into a three dimensional structure, branching downwards across one plane, and upwards across an orthogonal plane. Figure 1 illustrates a number of fat trees with \( c = 3 \) and \( p = 2 \), and shows how they can be composed in a bottom up recursive manner. The same networks can also be composed in a top down recursive manner, as described in [1]. A fat tree with height \( h \) has \( c^h \) child connections for communication with child switches or the processor elements of the multicomputer, and \( p^h \) parent connections to communicate with parent switches in the interconnection network. The topmost parent connections are not used. The important feature of the fat tree topology is that the aggregate communication bandwidth increases in the higher levels of the network as it scales, thus avoiding the congestion that occurs in a planar tree network [6].

An entity declaration for the fat tree network is shown in Figure 2. The type connection_vector is an array of connection records, and is used to represent a collection of message connections between switches. The type acknowledge_vector is an array of acknowledgment signals used to synchronize the message passing protocol. The precise composition of these types is not directly relevant to a de-
Figure 1. Bottom up compositions of fat trees with \( c = 3 \) and \( p = 2 \): (a) a single switch component, comprising a fat tree with \( h = 1 \), and a symbolic representation of the fat tree; (b) a fat tree with \( h = 2 \), its decomposition into switch components and fat trees with \( h = 1 \), and a symbolic representation; (c) a fat tree with \( h = 3 \), composed of switch components and fat trees with \( h = 2 \).

The fat tree entity is parameterized by height \( h \), and by child and parent degrees \( c \) and \( p \) using generic constants. Because the number of connections between the tree and the child and parent components must be determined by these parameters, they are implemented as vectors. The number of connections to the child components is determined by \( c \) and \( h \), and the number of connections to parent components is determined by \( p \) and \( h \).

### 3.1. Recursive description of the fat tree

The entity `aux_fat_tree`, shown at the top of Figure 3, is an auxiliary entity used to implement the fat tree structure. The bottom up recursive description that will be developed in this section requires additional generic constants to control the routing algorithm of each switch instantiated in the structure. The details of these generic constants will not be discussed in this paper. The interested reader is referred to [1] for a full description.

A bottom up recursive architecture declaration for the auxiliary fat tree is shown in Figure 4. In a fat tree of any height \( h \) composed in the bottom up manner, there are \( p \) sub-trees of height \( h - 1 \). In this description, they are indexed from 0 to \( p - 1 \), as defined by the subtype `subtree_range`. The number of switches at the bottom level is \( c^{h-1} \). They are indexed from 0 to \( c^{h-1} - 1 \), as defined by the subtype `switch_range`. The signals declared in the architecture body are used to connect the switch instances with the subtree instances, and will be discussed in detail below. The component `switch` represents the basic switching element from which the fat tree is constructed. It has generic constants \( c \) and \( p \) representing the child and parent degrees, and \( k, i \), and \( j \) used in the routing algorithm. The component `aux_fat_tree` is the template that is to be instantiated as subtrees above the bottom level of switches. These instances will be bound to the auxiliary fat tree entity in a separate configuration declaration.

The architecture body includes two conditional generate statements. If the value of the generic constant \( h \) is one, a simple fat tree is generated, consisting of a single switch connected directly to the child and parent ports of the entity. This causes the recursion to terminate. If the value of the generic constant \( h \) is greater than one, the statement labelled `compound_tree` generates an array of switches, an array of subtrees of height \( h - 1 \), and an array of connections between them, as shown in Figure 5. The child ports of each switch are connected to a slice of length \( c \) of the child ports of the fat tree entity. The parent ports of each subtree are connected to a slice of length \( p^{h-1} \) of the parent ports of the fat tree entity. The recursion in the description is completed in a separate configuration declaration, which binds each instance of the fat tree component to the fat tree entity using the bottom up recursive architecture.

```vhdl
use work.fat_tree_types.all;
entity fat_tree is
generic ( h : natural; c, p : positive );
port ( receive_msg_from_child :
in connection_vector(0 to c**h - 1); send_ack_to_child :
out acknowledge_vector(0 to c**h - 1);
send_msg_to_child :
in connection_vector(0 to c**h - 1);
receive_ack_from_child :
in connection_vector(0 to c**h - 1);
send_msg_to_parent :
in connection_vector(0 to p**h - 1);
receive_ack_from_parent :
in acknowledge_vector(0 to p**h - 1) => (others => false);
send_msg_from_parent :
in connection_vector(0 to p**h - 1) => (others => default_connection);
send_ack_to_parent :
out acknowledge_vector(0 to p**h - 1));
end fat_tree;
```

Figure 2. Entity declaration for a parameterized fat tree interconnection network.
use work.fat_tree_types.all;

denity aux_fat_tree is
    generic ( h : natural; c, p : positive; k : positive; i, j : natural );
    port ( receive_msg_from_child : in connection_vector(0 to c**h - 1);
    send_ack_to_child : out acknowledge_vector(0 to c**h - 1);
    send_msg_to_child : out connection_vector(0 to c**h - 1);
    receive_ack_from_child : in acknowledge_vector(0 to c**h - 1);
    send_msg_to_parent : out connection_vector(0 to p**h - 1);
    receive_ack_from_parent : in acknowledge_vector(0 to p**h - 1);
    receive_msg_from_parent : in connection_vector(0 to p**h - 1);
    send_ack_to_parent : out acknowledge_vector(0 to p**h - 1));
end aux_fat_tree;

architecture aux of fat_tree is
    component aux_fat_tree
        generic ( h : natural; c, p : positive; k : positive; i, j : natural );
        port ( receive_msg_from_child : in connection_vector(0 to c**h - 1);
        send_ack_to_child : out acknowledge_vector(0 to c**h - 1);
        send_msg_to_child : out connection_vector(0 to c**h - 1);
        receive_ack_from_child : in acknowledge_vector(0 to c**h - 1);
        send_msg_to_parent : out connection_vector(0 to p**h - 1);
        receive_ack_from_parent : in acknowledge_vector(0 to p**h - 1);
        receive_msg_from_parent : in connection_vector(0 to p**h - 1);
        send_ack_to_parent : out acknowledge_vector(0 to p**h - 1));
    end component;

begin
    the_tree : aux_fat_tree
        generic map ( h, c, p, k => 1, i => 0, j => 0 )
        port map ( receive_msg_from_child, send_ack_to_child,
        send_msg_to_child, receive_ack_from_child,
        send_msg_to_parent, receive_ack_from_parent,
        receive_msg_from_parent, send_ack_to_parent );
end aux;

Figure 3. Top: entity declaration for the auxiliary fat tree, with additional generic constants needed in the routing algorithm of the switches in the tree. Bottom: architecture declaration for the fat tree, instantiating the auxiliary fat tree component.

Ideally, the connections between the switch array and the subtree array would be formed from rectangular matrices of signals of size $p \times c^{k-1}$. The child connections of subtree $i$ would be connected to row $i$ of the matrices, and the parent connections of switch $j$ would be connected to column $j$ of the matrices. However, VHDL does not allow a row or column of a matrix to be treated as a vector slice. For this reason, each of the matrices for sending and receiving messages, and for sending and receiving acknowledgments, are comprised of two separate arrays of vectors.

The four signals subtree.receive_msg_from_switch, subtree.send_ack_to_switch, subtree.send_msg_to_switch and subtree.receive_ack_from_switch are represented by the top surface of the interconnection layer shown in Figure 5. Each of these signals is an array of $c^{k-1}$ elements, and each element is a vector of length $p$. The vectors in each array are connected to the parent ports of the $c^{k-1}$ switches. The nested generate statements, labelled connect_subtree and connect_switch, create drivers that connect the vertically matching elements of corresponding pairs of signals.

The recursion in the auxiliary fat tree description is completed in a configuration declaration, which binds each instance of the fat tree component to the fat tree entity and the bottom up recursive architecture body. (The details are presented in [1].)

3.2. Repetitive description of the fat tree

A repetitive description of the fat tree structure is significantly more difficult to design and understand than the re-
architecture bottom up_recursive of aux_fat_tree is
    subtype subtree_range is natural range 0 to p − 1;
    subtype switch_range is natural range 0 to c**(h − 1) − 1;
    subtype subtree_connection_vector is connection_vector(switch_range);
    subtype subtree_acknowledge_vector is acknowledge_vector(switch_range);
    type subtree_connection_vector_array is array (subtree_range) of subtree_connection_vector;
    type subtree_acknowledge_vector_array is array (subtree_range) of subtree_acknowledge_vector;
    signal subtree收到_msg_from_child, subtree.send_msg_to_child, subtree.send_ack_to_child : subtree_connection_vector_array;
    signal subtree收到_ack_from_switch, subtree.send_ack_to_switch : subtree_acknowledge_vector_array;
    subtype switch_connection_vector is connection_vector(switch_range);
    subtype switch_acknowledge_vector is acknowledge_vector(switch_range);
    type switch_connection_vector_array is array (switch_range) of switch_connection_vector;
    type switch_acknowledge_vector_array is array (switch_range) of switch_acknowledge_vector;
    signal switch收到_msg_from_subtree, switch.send_msg_to_subtree, switch.send_ack_to_subtree : switch_connection_vector_array;
    signal switch收到_ack_from_subtree, switch.send_ack_to_subtree : switch_acknowledge_vector_array;

circuit switch
    generic ( c, p : positive; k : positive; i, j : natural );
    port ( receive_msg_from_child : in connection_vector(0 to c − 1);
           send_ack_to_child : out acknowledge_vector(0 to c − 1);
           send_msg_to_child : out connection_vector(0 to c − 1);
           receive_ack_from_child : in acknowledge_vector(0 to c − 1);
           send_msg_to_parent : out connection_vector(0 to p − 1);
           receive_msg_from_parent : in connection_vector(0 to p − 1);
           send_ack_to_parent : out acknowledge_vector(0 to p − 1) );

end component;

circuit aux_fat_tree
    generic ( h : natural; c, p : positive; k : positive; i, j : natural );
    port ( receive_msg_from_child : in connection_vector(0 to c**h − 1);
           send_ack_to_child : out acknowledge_vector(0 to c**h − 1);
           send_msg_to_child : out connection_vector(0 to c**h − 1);
           receive_ack_from_child : in acknowledge_vector(0 to c**h − 1);
           send_msg_to_parent : out connection_vector(0 to p**h − 1);
           receive_msg_from_parent : in connection_vector(0 to p**h − 1);
           send_ack_to_parent : out acknowledge_vector(0 to p**h − 1) );

end component;

circuit simple_tree
    begin
        simple_tree : if h = 1 generate
            the_switch : switch
                generic map ( c, p, k, i, j )
                port map ( receive_msg_from_child => receive_msg_from_child,
                      send_ack_to_child => send_ack_to_child,
                      send_msg_to_child => send_msg_to_child,
                      receive_ack_from_child => receive_ack_from_child,
                      send_msg_to_parent => send_msg_to_parent,
                      receive_ack_from_parent => receive_ack_from_parent,
                      receive_msg_from_parent => receive_msg_from_parent,
                      send_ack_to_parent => send_ack_to_parent );
        end generate simple_tree;
end begin;

Figure 4. Bottom up recursive architecture body for a fat tree.

ductive description. The main difficulty lies in the inter-
connections between the switch elements comprising the structure. As Figure 6 shows, the total number of interconnection pairs needed for sending and receiving messages and acknowledgments (including the parent and child ports of the fat tree) is:

\[
S_h = p^h + p^{h-1}c + \cdots + pc^k + c^h
\]

\[
= \begin{cases} 
\frac{p^{k+1}c^{h+1}}{p-c} & \text{if } p \neq c \\
(h+1)p^h = (h+1)c^h & \text{if } p = c 
\end{cases}
\]

and the total number of switches is \( S_{h,c} \). Ideally, one vector of length \( S_h \) would be used for each of the signal sets. The parent switches at each level would connect to slices of length \( c \), whereas the children switches would connect to slices of length \( p \), but with each element in the slice separated by some stride determined by \( c, p \) and the position of the switch in the structure. Since this is not possible in VHDL, a pairs of vectors is needed for each set of signals. One vector is of length \( S_{h,c} \), and is divided into slices of length \( p \) to connect to the parent ports of each of the switches. It threads the tree from back to front then left to
compound_tree : if \( h > 1 \) generate
switch array : for i in switch range generate
  the_switch : switch
    generic map (c, p, k, i, j)
    port map (receive_msg_from_child =>
      send_ack_to_child =>
      receive_ack_from_child =>
      send_msg_to_parent =>
      receive_msg_from_parent =>
      send_ack_to_parent =>
    )
  end generate switch array;
subtree_array : for i in subtree range generate
  the_subtree : aux_fat_tree
    generic map (h - 1, c, p, k + 1, outer_j * p + i, j)
    port map (receive_msg_from_child =>
      send_ack_to_child =>
      receive_ack_from_child =>
      send_msg_to_parent =>
      receive_msg_from_parent =>
      send_ack_to_parent =>
    )
  end generate subtree array;
connect_subtree : for i in subtree range generate
  connect_switch : for j in switch range generate
    switch receive_msg_from_subtree(0)(i) <= switch send_msg_to_switch(0);
    subtree receive_msg_from_switch(0)(i) <= switch send_msg_to_subtree(0);
    switch receive_ack_from_subtree(0)(i) <= switch send_ack_to_subtree(0);
    end generate connect_switch;
  end generate connect_subtree;
end generate compound_tree;
end bottom_up_recursive;

Figure 4. (continued)

\[ p \times c^{h-1} \] connections

Figure 5. Connections between the bottom row of switches and the subtrees of height \( h - 1 \).

right, starting at the top. The other vector is of length \( S_1 - p^k \),
and is divided into slices of length \( c \) to connect to the child
ports of each of the switches. It threads the tree from left
to right then back to front, starting at the bottom. In each
layer of the interconnection structure, drivers are created to
connect matching elements of the vectors together.

A repetitive architecture body for the fat tree entity is shown in Figure 7. The signals declared in the architecture
represent the interconnection vectors described above.
The outer generate statement, labelled switch_levels, generates the levels of switches, starting from level 1 at the bottom
and repeating upwards to level \( h \). The nested generate
statement, labelled rows, generate the horizontal rows of switches at the current level, indexed from 0 at the back to $p^{k-1} \times 1$ at the front. The innermost generate statement, labelled cols, generates the individual switches within each row, indexed from 0 on the left to $c^{h-k}$ at the right. The values of the generate statement parameters $k, i$ and $j$ effectively determine a three dimensional coordinate for each switch created. These coordinates are passed to each switch for use in the routing algorithm.

The complexity in this repetitive description lies in the mapping functions that determine which elements of the vectors each switch must connect to, and which elements of the parent and child connection vectors should be interconnected. The mapping functions are dependent on $c$, $p$, $h$ and the coordinates of a switch or connection in the structure.

Consider first a switch at position $(k, i, j)$ within the $p^{k-1} \times c^{h-k}$ array of switches at level $k$. This switch makes connections to the parent vectors in the layer above it, and to the child vectors in the layer below it. The parent ports of the switch are connected to slices of length $p$ of the parent vectors starting at element $M_p(k, i, j)$ in each vector, where:

$$M_p(k, i, j) = O_p(k) + j p^k + ip.$$ 

The function $O_p(k)$ is an offset function that gives the starting index in the child vectors for the parent connections of switches at level $k$. Its value is the number of child vector elements for switches at levels $h$ down to $k + 1$, as follows:

$$O_p(k) = p^k + p^{h-1}c + p^{h-2}c^2 + \cdots + p^1c^{h(k-1)}$$

$$= p^k + (p^h)^{k+1} + (p^h)^{k+1}c + \cdots + c^{h(k-1)}$$

$$= p^k + S_{p(k+1)}$$

(Note that $S_{i,j} = 0$.) Similarly, the child ports of the switch are connected to slices of length $c$ of the child connection vectors starting at element $M_c(k, i, j)$, where:

$$M_c(k, i, j) = O_c(k) + ic^{h(k-1)} + jc.$$ 

The function $O_c(k)$ is an offset function that gives the starting index in the child vectors for the child connections of switches at level $k$. Its value is the number of child vector elements for switches at levels 1 up to $k-1$, as follows:

$$O_c(k) = c^k + pc^{h-1} + p^2c^{h-2} + \cdots + pc^{h-1}c^{h(k-2)}$$

$$= c^{h(k-2)}(c^{h-2} + p(c^{h-2})^{k-1} + \cdots + p^{k-2})$$

$$= c^{h(k-2)}S_{c^{k-2}}$$

The derivation of these formulas is described in more detail in [1]. The formulas are represented in the architecture body by the functions $M_p$, $O_p$, $M_c$, $O_c$ and $S$. They are used in the component instantiation statement for the switch to determine the bounds of the slices of the vectors associated with the ports of the switch. The use of these functions in the expressions determining slice bounds for signals is not legal in VHDL-87, although it is legal in VHDL-93. The VHDL-93 form is used here to avoid the added complexity, necessitated in VHDL-87, of writing the
architecture repetitive of fat_tree is

    function S (h : integer) return natural is
    begin
        if p = c then
            return (p**(h + 1) - c**(h + 1)) / (p - c);
        elsif h >= 0 then
            return (h + 1) * p**h;
        else
            p = c and h = -1
            return 0;
        end if;
    end S;

    function Op (k : natural) return natural is
    begin
        return (p**(k + 1) * S(h - (k + 1));
    end Op;

    function Mp (k, i, j : natural) return natural is
    begin
        return Op(k) + j * p**k + i * p;
    end Mp;

    function Oc (k : natural) return natural is
    begin
        return c**(h - (k - 2)) * S(k - 2);
    end Oc;

    function Mc (k, i, j : natural) return natural is
    begin
        return Oc(k) + i * c**(h - (k - 1)) + j * c;
    end Mc;

    subtype parent_connection_vector is connection_vector(0 to S(h) - c**h - 1);
    subtype parent_acknowledge_vector is acknowledge_vector(0 to S(h) - c**h - 1);
    signal switch_receive_msg_from_parent, switch_send_msg_to_parent : parent_connection_vector;
    signal switch_receive_ack_from_parent, switch_send_ack_to_parent : parent_acknowledge_vector;

    subtype child_connection_vector is connection_vector(0 to S(h) - p**h - 1);
    subtype child_acknowledge_vector is acknowledge_vector(0 to S(h) - p**h - 1);
    signal switch_receive_msg_from_child, switch_send_msg_to_child : child_connection_vector;
    signal switch_receive_ack_from_child, switch_send_ack_to_child : child_acknowledge_vector;

    component switch
        generic (c, p : positive; k : positive; i,j : natural);
        port (receive_msg_from_child : in connection_vector(0 to c - 1);
              send_ack_to_child : out acknowledge_vector(0 to c - 1);
              send_msg_to_child : out connection_vector(0 to c - 1);
              receive_ack_from_child : in acknowledge_vector(0 to c - 1);
              receive_msg_from_parent : in connection_vector(0 to p - 1);
              send_ack_to_parent : out acknowledge_vector(0 to p - 1);
              send_msg_to_parent : out connection_vector(0 to p - 1);
        );
    end component;

Figure 7. Repetitive architecture body for the fat tree entity.

expressions “in line”. (The complete VHDL-87 version is included in [1].)

Now consider the interconnections between the parent signal vectors connected to switches at level \( k \) and the child signal vectors connected to switches at level \( k + 1 \). The elements of the parent vectors at position \((i, j)\) in the layer have index given by \( O_p(k) + j p^k + i \), and the elements of the child vectors at position \((i, j)\) in the layer have index given by \( O_c(k + 1) + j c^{k - 1} + j \), where \( O_p \) and \( O_c \) are as defined above. (These derivations are also described in detail in [1].) The interconnections between these elements are generated by the generate statement labelled connection_levels and its nested generate statements in the repetitive architecture body. The immediately preceding set of concurrent signal assignments interconnect the child ports of the fat tree with the bottom row of the child signal vectors, and the concurrent signal assignments after the generate statements interconnect the parent ports of the fat tree with the top column of the parent signal vectors. The restrictions in VHDL-87 on expressions used in signal slice bounds mentioned above also apply to expressions used as signal indices. Such expressions may not include function applications. For this reason, the VHDL-93 form, using the function \( Op \) and \( Oc \), is shown here for the concurrent signal assignments that interconnect the vector elements.

In order to complete the repetitive description, we need to write a configuration declaration to bind component instances to entities. It is somewhat simpler than that for
the recursive description, since it only needs to bind the instances of the switch components to the behavioral implementation of the switch entity. Furthermore, the complication of the auxiliary entity is obviated.

4. Comparison of recursive and repetitive descriptions

The fat tree example illustrates the advantages of recursive VHDL descriptions of recursive structures over repetitive descriptions. The main difference between the two styles of description lies in the interconnection of the basic components used to construct the structure. In a recursive description, signals are needed to interconnect a small set of components with a small set of recursive instances of the structure. These interconnections are organized into a regular pattern, such as a simple array or matrix. The complexity of the full set of interconnections in the entire structure comes about through the recursive instantiation of the structure, including sub-interconnections, with different parameter values for different instances.

In a repetitive description, on the other hand, all of the signals for the full set of interconnections for the entire structure must be statically declared in the architecture body. The signals must be declared in such a way that their number is dependent upon the parameters that characterize the structure. Since the values of these parameters cannot be known a priori, the signals must be declared as arrays, whose size depends on the parameters. A set of mapping functions must then be devised to map ports of instances of the basic components onto slices of these signal arrays. This step may require considerable ingenuity on the part of the designer, and its implementation significantly obscures the design.

Given this complication in a repetitive description, one must ask whether the language is lacking sufficient expressive power. The complication stems from the requirement that the signals be statically declared, and be global to the generate statements that create instances of the basic components. An alternative language design is to allow signals to be generated at elaboration time, in an analogous way to that in which component instances are generated. This would also require that the names of such dynamically generated signals be visible outside of the generate statements.
that create them, so that they can be referred to in the port association lists of the component instances within other generate statements. Such a name visibility scheme essentially amounts to dynamic scoping, similar to visibility schemes seen in early versions of Lisp and other programming languages. Experience with these languages indicates that there are major software engineering and reliability issues arising from dynamic scoping [8], which is one of the reasons that languages such as Ada and VHDL use a static scoping scheme. Thus to modify VHDL to incorporate these facilities to ease the implementation of repetitive descriptions of recursive structures would be contrary to the fundamental design requirements of the language.

5. Conclusions

In this paper, techniques for developing VHDL descriptions of recursive hardware structures have been examined. Both recursive and repetitive component instantiations were examined for modeling recursive structures. Recursive and repetitive models of an example structure, a fat tree interconnection network, were developed and compared. The recursive description was significantly easier to develop, and more clearly express the structure, making it easier to understand. The main difficulty in developing the repetitive model lay in devising a way of interconnecting the basic components comprising the structure.

In general, we conclude that a recursive model of a recursive hardware structure is more appropriate if the structure is more complex than a rectangular multidimensional array of elements, or if the interconnections pattern between the elements is more complex than a rectangular multidimensional array. For such structures, a recursive model simplifies the generation of elements and the declaration and interconnection of signals. A repetitive model, on the other hand, requires the designer to devise complex mapping functions to map ports of elements onto an array of signal elements. These mapping functions significantly obscure the intent of the model, making it more difficult to establish correctness, and adversely affecting maintainability of the model.

We have shown that the difficulties inherent in repetitive models result from fundamental language design decisions made in VHDL, and that it is not appropriate to modify the language to avoid the difficulties. Hence, in general, the recursive style is preferred over the repetitive style for describing recursive hardware structures.

Acknowledgements

This work was undertaken during the author’s study leave visit to the Computer Design Lab at the Department of Electrical and Computer Engineering, University of Cincinnati. The author would like to thank Dr. Philip Wilsey for support and the conducive environment that made the work possible and enjoyable. Thanks are also due to the referees for their helpful comments. The author also acknowledges the support of the University of Adelaide through its study leave grants scheme.

References


