VHDL-200x Data Types and Abstractions White Paper 1

Type Genericity

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

Version 2, 19 April 2004

Abstract

This white paper proposes a design for type-genericity extensions to VHDL. The extensions are based on a design developed as part of the SUAVE project by Peter Ashenden while at the University of Adelaide and Phil Wilsey at the University of Cincinnati. That design, in turn, was strongly influenced by the type-genericity features of the Ada programming language.

Revision History

Version 1. 28-Sep-03, Peter Ashenden. Initial version based on SUAVE Language Description.

Version 2. 19-Apr-03, Peter Ashenden. Simplified version, using formal incomplete type definitions.

1 Introduction

Reuse of a design unit can be improved by making it applicable in a wider set of contexts, for example, by making it more generic. VHDL currently includes a mechanism, generic constants, that allows components and entities to be parameterized with formal constants. Actual generic constants are specified when components are instantiated and when entities are bound. The generic constant mechanism is widely used to specify timing parameters and array port bounds, among other things.

In this proposal we extend the generic mechanism of VHDL to improve support for reuse. There are two main aspects to the extension. The first is to allow subprograms and packages to have generic interface clauses. The second is to allow formal types in a generic interface clause, making the generic item reusable for a variety of different types. Formal subprograms and formal packages are also allowed as a corollary to allowing formal types.

A further extension that would be required if process declaration were included in the language would be the inclusion of formal processes in generic clauses. This would parallel formal subprograms, allowing specification of action processes for instances of generic units.

In this white paper, we present the syntax and static semantics of generic units. In BNF syntax rules, we underline those rules or parts of rules that are extensions to existing VHDL rules. We start by describing the extended forms of package and subprogram declarations that include formal generic lists. We also describe the way in which generic packages and subprograms may be instantiated with a generic map aspect providing actuals for the formal generics. We then present a detailed description of the various forms of formal generic declaration, illustrating them with examples.

Note that this white paper focuses on proposed extensions to the generics mechanism in relative isolation. The SUAVE proposal makes further extensions. Some of those extensions are intended to make generic units more usable. An example is inclusions of package declarations and package instances in any declarative part, not just as design units. Other extensions arise from interactions with additional language features proposed in SUAVE. An example is formal generics supporting derived types. We do not present any of those extensions in this white paper, preferring to focus just on what is essential for extended generics.

2 Generic Packages

We extend the declaration of packages to allow inclusion of a formal generic clause. The extended syntax rule for a package declaration is shown is:

package_declaration ::=
package identifier **is**
[*formal* generic_clause]
package_declarative_part
end [**package**] [*package*_simple_name] ;

A package that includes a formal generic clause is a generic package. A generic package is a template for an ordinary package, and does not provide declarations itself. It must be instantiated as described below. Examples of packages with formal generic clauses are shown in subsequent sections.

2.1 Instantiating a Generic Package

In order to make use of a generic package, it must be instantiated to associate actuals with the formal generics. The syntax rule is:

generic_package_instantiation ::=
package identifier **is new** *generic_package*_name
[generic_map_aspect] ;

A generic package may be instantiated as a design unit. The extended syntax rule is:

primary_unit ::=
. . .
| generic_package_instantiation

An instance of a generic package instantiated as a design unit is semantically equivalent to a normal package as currently defined in VHDL.

In order to allow use of generic packages with locally declared type and subprograms, we allow generic packages to be instantiated in declarative parts where types or subprograms may be declared. The extended syntax rules are:

block_declarative_item ::=
. . .
| generic_package_instantiation

entity_declarative_item ::=
. . .
| generic_package_instantiation

package_declarative_item ::=
. . .
| generic_package_instantiation

package_body_declarative_item ::=
. . .
| generic_package_instantiation

process_declarative_item ::=
. . .
| generic_package_instantiation

protected_type_body_body_declarative_item ::=
. . .
| generic_package_instantiation

subprogram_declarative_item ::=
. . .
| generic_package_instantiation

An instance of a generic package instantiated in a declarative part is elaborated in order as part of elaboration of the declarative part.

Examples of generic package instantiation are shown in subsequent sections.

3 Generic Subprograms

We also extend the declaration of subprograms to allow inclusion of a formal generic clause. The extended syntax rule for a subprogram declaration is:

subprogram_declaration ::=
[**generic** (generic_list)] subprogram_specification ;

and for a subprogram body is:

subprogram_body ::=
[**generic** (generic_list)] subprogram_specification **is**
subprogram_declarative_part
begin
subprogram_statement_part
end [subprogram_kind] [designator] ;

A subprogram declaration or body that includes a formal generic clause in its specification is a generic subprogram. A generic subprogram is a template for an ordinary subprogram. It cannot be called, but must be instantiated as described below. Examples of subprograms with formal generic clauses are shown in subsequent sections.

If a generic subprogram is declared as a separate subprogram declaration and subprogram body, the subprogram body may include the formal generic clause, in which case it must conform with the formal generic clause in the subprogram declaration.

3.1 Instantiating a Generic Subprogram

A generic subprogram may be instantiated as a subprogram. The syntax rule is:

generic_subprogram_instantiation ::=
subprogram_kind designator **is new** *generic_subprogram_*name
[generic_map_aspect] ;

Generic subprograms may be instantiated in any declarative part in which subprograms may be declared (except protected type declarations). Examples of generic subprogram instantiation are shown in subsequent sections. The extended syntax rules are:

block_declarative_item ::=
. . .
| generic_subprogram_instantiation

package_declarative_item ::=
. . .
| generic_subprogram_instantiation

package_body_declarative_item ::=
. . .
| generic_subprogram_instantiation

process_declarative_item ::=
. . .
| generic_subprogram_instantiation

protected_type_body_declarative_item ::=
. . .
| generic_subprogram_instantiation

subprogram_declarative_item ::=
. . .
| generic_subprogram_instantiation

4 Extended Generic Maps

A generic map aspect is used to associate actual generics with formal generics upon instantiation of a generic subprogram, a generic package, a component or an entity. A generic map is also used to associate actual generics with formal generics in a block statement.

The extended syntax rule for an actual designator, allowing specification of actual generics for formal type and subprogram generics, is:

actual_designator ::=
. . .
| type_mark
| *subprogram*_name

For a formal generic type, the associated actual type is designated by a type mark. For a formal generic subprogram, the associated actual subprogram is designated by a subprogram name. Examples of extended generic maps are shown in subsequent sections.

5 Extended Generic Clauses

We extend the kinds of formal generics that may be specified in a formal generic clause to include formal types and formal subprograms. These can be included in generic clauses of package declarations, subprogram specifications, block statements, entity declarations and component declarations. The revised syntax rule is:

interface_declaration ::=
. . .
| interface_type_declaration
| interface_subprogram_declaration

Interface type and subprogram declarations may only appear in formal generic clauses. A formal generic clause may only include interface constant, type and subprogram declarations.

The rule (LRM ¶4.3.2.1) that prohibits use of an item declared in an interface list within the declaration of other items in the interface list is relaxed in the case of generic interface lists. Items declared in a generic interface list may be used in the declaration of items declared subsequently in the interface list.

A further change is a relaxation of the rule (LRM ¶??.??) that prohibits interface constants from being of access types. We allow constant parameters of subprograms and impure functions to be of access types, provided ... (contitions to be determined...).

6 Formal Generic Types

An interface type declaration defines a formal generic type that can be used to pass a particular type when the generic unit is instantiated. The syntax rules are:

interface_type_declaration ::=
type identifier

An interface type declaration defines a formal generic type that can denote any type. The generic unit can only assume that operations available for all types are applicable, namely, variable assignment, equality and inequality operations.

Example

A package defining an ADT for sets of elements can be made reusable by making it generic with respect to element type, as shown below. The formal type generic element_type represents the element type, and the actual associated with it can be any type.

```
package sets is
    generic ( type element_type );
    type set;
    -- element_node and structure of set are private
    type element_node is record
        next_element : set;
```

```
        value : element_type;
    end record element_node;
    type set is access element_node;
    constant empty_set : set;
    procedure copy ( from : in set; to : out set );
    function "+"( R : element_type ) return set; -- singleton set
    impure function "+"( L : set; R : element_type ) return set; -- add to set
    impure function "+"( L : element_type; R : set ) return set; -- add to set
    impure function "+"( L, R : set ) return set; -- set union
    . . .
end package sets;
package body sets is
    constant empty_set :set := null;
    . . .
end package body sets;
```

Given a type thingy, an ADT for sets of elements of this type may instantiated as follows:

```
package thingy_sets is new work.sets
    generic map ( element_type => thingy );
```

———

7 Formal Generic Subprograms

An interface subprogram declaration defines a formal generic subprogram that can be used to pass a particular subprogram when the generic unit is instantiated. The syntax rule is:

interface_subprogram_declaration ::=
    subprogram_specification [**is** subprogram_default]

subprogram_default ::= name | <>

The subprogram default specifies the subprogram to use if no actual generic subprogram is provided on instantiation. If a name is specified as the subprogram default, it must denote a callable subprogram with the same signature as that of the subprogram specification. If a box (<>) is specified as the subprogram default, it indicates that the actual generic subprogram should be a subprogram that is directly visible at the point of instantiation and that has the same name and signature as those of the subprogram specification.

Example

The following package defines an ADT for lookup tables. A table contains elements that are each identified by a key value. The formal function key_of determines the key for a given element. No default function is provided, so the user must supply an actual function on instantiation of the package. The formal function "<" is used to compare key values. The default function is specified using the "<>" notation, so if an appropriate function named "<" is visible at the point of instantiation, no actual need be specified. The generic procedure traverse is parameterized by an action procedure. An instance of traverse applies the actual action procedure to each element in the table.

```
package lookup_tables is
    generic ( type element_type;
                type key_type;
                function key_of ( E : element_type ) return key_type;
                function "<"( L, R : key_type ) return boolean is <> );
    type lookup_table;
    -- tree_record and structure of lookup_table are private
    type tree_record is record
        left_subtree, right_subtree : lookup_table;
        element : element_type;
    end record tree_record;
    type lookup_table is access tree_record;
```

```
    procedure lookup ( table : in lookup_table; lookup_key : in key_type;
                       element : out element_type; found : out boolean );
    procedure search_and_insert ( table : in lookup_table; element : in element_type;
                                  already_present : out boolean );
    generic ( procedure action ( element : in element_type ) )
    procedure traverse ( table : in lookup_table );
end package lookup_tables;
```

The package body is shown below. The formal functions **key_of** and "<" are invoked using the formal name.

```
package body lookup_tables is
    procedure lookup ( table : in lookup_table; lookup_key : in key_type;
                       element : out element_type; found : out boolean ) is
        variable current_subtree : lookup_table := table;
    begin
        found := false;
        while current_subtree /= null loop
            if lookup_key < key_of( current_subtree.element ) then
                lookup ( current_subtree.left_subtree, lookup_key, element, found );
            elsif key_of( current_subtree.element ) < lookup_key then
                lookup ( current_subtree.right_subtree, lookup_key, element, found );
            else
                found := true;
                element := current_subtree.element;
                return;
            end if;
        end loop;
    end procedure lookup;
    procedure search_and_insert ( table : in lookup_table; element : in element_type;
                                  already_present : out boolean ) is ...
    procedure traverse ( table : in lookup_table ) is
    begin
        if table = null then
            return;
        end if;
        traverse ( table.left_subtree );
        action ( table.element );
        traverse ( table.right_subtree );
    end procedure traverse;
end package body lookup_tables;
```

Suppose a model requires a lookup table of test patterns that use character strings as keys. Such a table may be instantiated as shown below. Since the predefined function "<"operating on strings is visible at the point of instantiation, it is used as the actual function for the formal function "<".

```
type test_pattern_type is . . .
function test_id_of ( test_pattern : in test_pattern_type ) return string;
package test_pattern_tables is new work.lookup_tables
    generic map ( element_type => test_pattern_type,
                  key_type => string,
                  key_of => test_id_of );
```

The traversal procedure can be used to count the number of elements in the table by instantiating it as follows:

```
use test_pattern_tables.all;
variable count : natural := 0;
procedure count_a_test_pattern ( test_pattern : in test_pattern_type ) is
begin
```

```
        count := count + 1;
    end procedure count_a_test_pattern;
    procedure count_test_patterns is new traverse
        generic map ( action => count_a_test_pattern );
```

The instantiated traversal function can be called with a test pattern lookup table as a parameter, as follows:

```
    variable patterns_to_apply : lookup_table;
    . . .
    count_test_patterns ( patterns_to_apply );
```

———