Technical Report No. SNU-EE-TR-1997-5

VHDL Developer's Toolkit 2.6
User's Guide & Reference

Sanghun Park
Kiyoung Choi

September 1997

School of Electrical Engineering
Seoul National University

Haedong Digital Library

Room 312, Building 301, Seoul National University

Shinlim-dong, Kwanak-gu, Seoul 151-742, Korea

Tel: (02)880-1787 Fax: (02)882-4656

http://haedong.snu.ac.kr

Abstract

This report presents VDT (VHDL Developer's Toolkit) which has been developed to support fast and easy development and integration of VHDL application tools. VDT is a software package which is developed using object-oriented programming language. The toolkit provides a library of versatile routines and several utilities. As basic utilities, it provides a VHDL analyzer and a VHDL generator. The VHDL analyzer parses the given VHDL code and constructs the intermediate form. The VHDL generator regenerates the corresponding VHDL code from the given intermediate form. The toolkit also provides a procedural interface through which application tools can efficiently manipulate VHDL intermediate forms. Procedural interface is a library of versatile routines which are used to handle the intermediate form. The toolkit is based on a data model which has been designed to represent the basic structure of the VHDL intermediate form. The data model supports the full set of IEEE Std 1076-1987.

Table of Contents

1 Introduction

Since 1987, when VHDL (VHSIC Hardware Description Language) became the formal international standard[1], the language has been gaining its popularity due to the widespread top-down design methodology and the increasing need for ASICs (Application Specific Integrated Circuits) [2,3,4]. Although much effort has been put for the development of VHDL libraries and tools [5,6,7], the increasing design size and complexity are still pushing the tool developers for more and better libraries and tools. However, it is not an easy task to develop tools that support the broad range of VHDL's descriptive capability while maintaining good performance. The development takes a long time and much effort because every tool needs to be able to analyze VHDL descriptions, deal with complex intermediate data, and/or give much consideration to obtaining efficient data access.

The development of VDT (VHDL Developer's Toolkit) aims at providing a set of tools and an integrated environment to tool-developers so that they can speed up the development and integration of VHDL-related tools. The basic concept of VDT is similar to that of other existing systems such as the IBM VHDL Design System [8] and CLSI's VTIP(VHDL Tool Integration Platform) [9]. However, the internal structure of the existing systems are unclear and complex, and therefore, the systems are difficult to use. VDT has been built on a data model which is relatively simple but still maintains the full VHDL descriptive power. VDT also provides a clear procedural interface between applications and the intermediate form.

This report is structured as follows. The next section presents the structural overview of VDT. Section 3 explains the design of the data model that forms the basis for VDT. Section 4 explains the implementation aspect on data structure. Section 5 explains the intermediate form representation for various VHDL constructs. Section 6 explains the procedural interface in detail. Section 7 presents the method by which VDT utilities are developed and integrated into VDT.

2 Structure of VDT

VDT is a software package which is developed using the C++ programming language. This package includes a library of versatile routines and several basic tools which enable easy and quick development of VHDL application tools. The core module of the toolkit is PI (Procedural Interface) which is a library through which application tools can efficiently manipulate VHDL intermediate forms. As basic tools provided by the toolkit, there is the VHDL analyzer, VHDL generator, library browser, and so on. The toolkit is based on a data model which has been designed to represent the basic structure of the VHDL intermediate form. The data model supports the full set of IEEE Std 1076-1987.

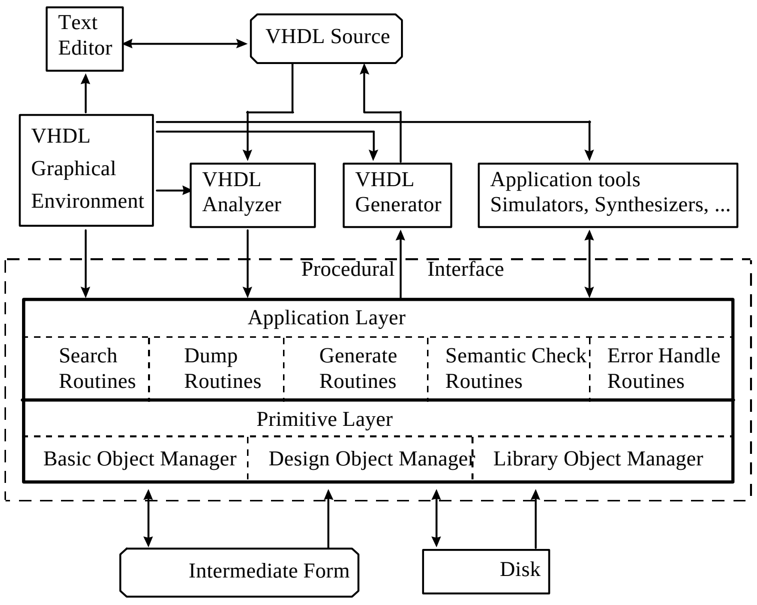


Figure 1. Structure of VHDL Developer's Toolkit

Figure 1 shows how the basic tools and utilities are configured together with other application tools, such as simulators and synthesizers. PI consists of two stacked layers. The primitive layer contains routines for primitive operations on library objects, design objects, and basic objects. These three kinds of objects form a hierarchy in the intermediate form. The application tools call on these routines to access the intermediate form. The VHDL analyzer, for example, analyzes a VHDL source and construct the intermediate form by calling on various routines in PI. Then it calls for the semantic check routines to see if the VHDL source has correct semantics. If it passes the semantic checking, the intermediate form is stored onto the disk in the form of well-arranged binary data. Because the basic routines, such as building and storing an intermediate form, are provided by PI, the analyzer could be developed quickly and easily. With this toolkit, developing application tools such as simulators are easy because all the routines for manipulating the intermediate form and even the analyzer are already there ready to be used.

Because the intermediate form preserves all the semantics implied by the original VHDL code, any application tool can use the same intermediate form as its inputs. VHDL code can be regenerated from a given intermediate form by the VHDL generator. The generated VHDL code may look different from the original code, but must have the same semantics.

3 Data Model

Because of the wide range of descriptive capability of VHDL, high complexity of the intermediate form is unavoidable. Accordingly, the manipulation of the VHDL intermediate form is a very complicated and error-prone process. However, we can alleviate the problem by designing and utilizing a data model which is a notation for representing the structure of intermediate forms. The data model can be used by both the PI developers and the application tool developers. The PI developers write procedures such that the procedures can manipulate the intermediate forms according to the data model. The application tool developers use the data model to understand how the intermediate form is structured and how it is accessed through PI's procedures.

In designing the data model, the following aspects are considered:

- The data model must be designed such that the full VHDL descriptive capability is kept intact in the intermediate form representation. No loss of information is allowed during the analysis process.
- The data model partially supports the VHDL semantics. Full support would cause too much overhead. Therefore, it is possible to build an intermediate form that is semantically incorrect but conforms to the data model. Detecting such an error is the job of the Semantic Checker module of PI.
- Efficiency (memory space and speed) is one of the most important factors that must be considered in designing the data model. The efficiency in manipulating intermediate forms affects the performance of every application tool and must be considered in the data model design phase.
- Application tool developers are expected to use the data model as a 'map' for accessing the intermediate form. Therefore, the data model must be as simple as possible so that the user can understand the structure of the intermediate form without too much difficulty.

3.1 Abstract VDT data model

The top-most VHDL construct is a design library. A design library is a host dependent storage facility for the intermediate forms of the analyzed design units. The second highest construct is a design unit. A design unit can be analyzed independently, and an intermediate form is constructed. The intermediate form can be stored in a design library. Usually, a design library is implemented as a directory of file system, whereas a design unit is implemented as a file stored in the directory. A design unit is composed of a library unit and a context clause. There are five kinds of VHDL constructs called library units: entity declaration, package declaration, configuration declaration, architecture body, and package body. The context clause proves the context in which a library unit is analyzed. As the lowest VHDL constructs, there are signal declaration, process statement, case statement, configuration specification, and so on. A design unit is composed of these constructs.

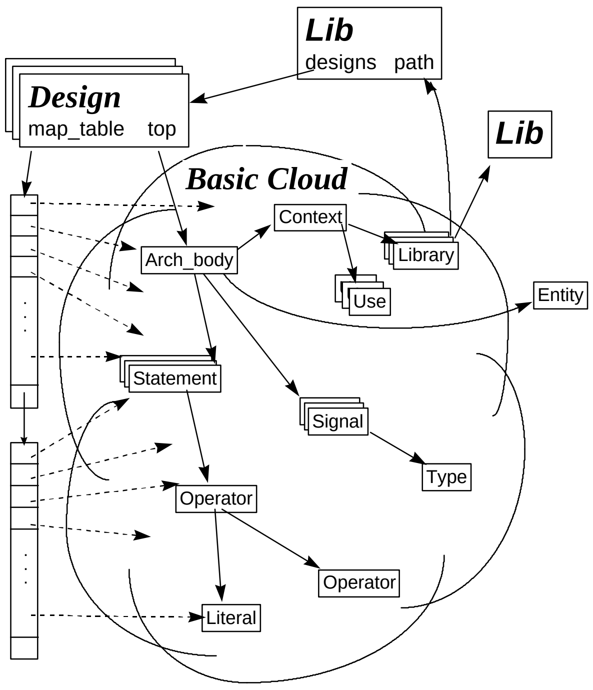


Figure 2. Abstract VDT data model.

Figure 2 shows an abstraction of VDT data model. VDT data model defines the general structure of intermediate forms. An intermediate form is composed of many kinds of objects which we call VDT objects. All VHDL constructs are described by VDT objects in an intermediate form representation. For each VDT object, there is a corresponding VHDL construct. An intermediate form can be modeled as a directed graph, where vertices represent VDT objects and edges represent the relationships.

VDT objects can be classified into three kinds; lib object, design object, and basic object. Lib object represents a design library, design object represents a design unit, and basic objects represents the rest of VHDL constructs. For instance, in an intermediate form representation, an architecture body is represented by an Arch_body object and a type declaration is described by a Type object. The Arch_body objects and Type objects are basic objects. A basic cloud is composed of only basic objects and relationships among these objects. Actually, the behavior of a circuit is described by a basic cloud. On the other hand, lib objects and design objects contain only the information on the environment. In the programmer’s view, the design object can be regarded as a container of basic objects. Similarly, a lib object can be regarded as a container of design object.

3.2 Relationships among basic objects

A relationship associates an object with other objects. For example, an object corresponding to a signal of bit type has a relationship to the bit type object. A relationship from an object can be one-to-one or one-to-many depending on how many objects are associated with the object.

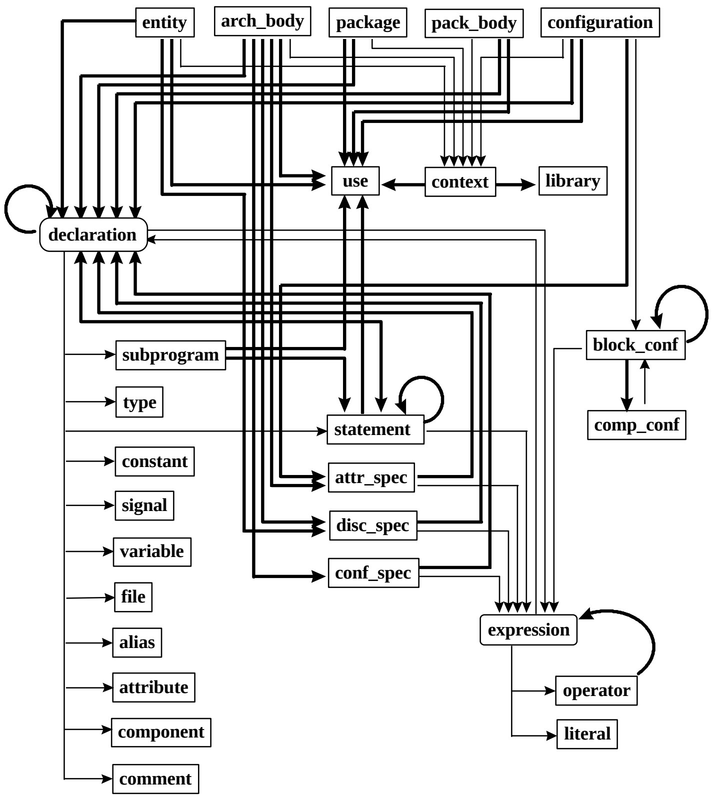


Figure 3. Relationships among VDT basic objects.

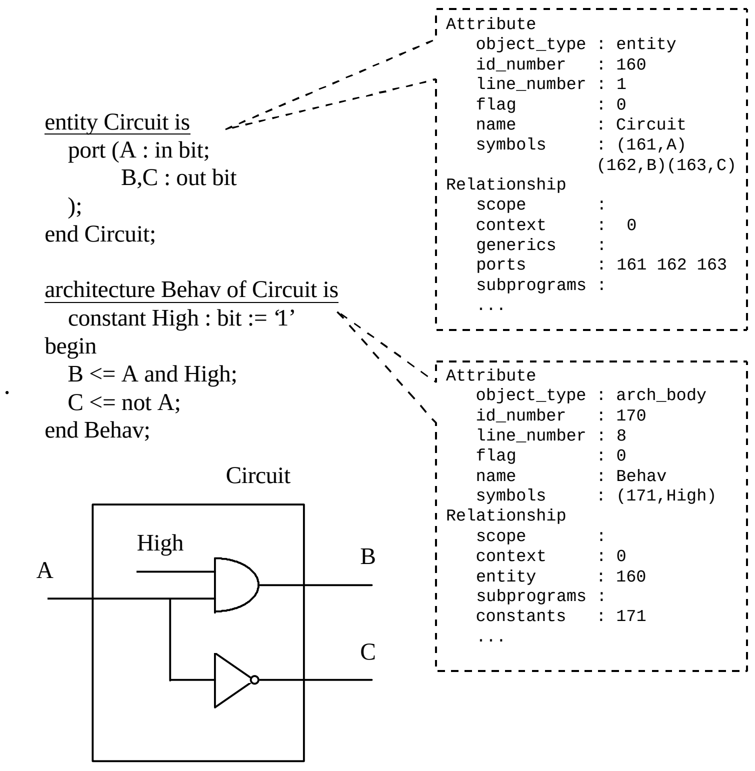
Figure 3 shows the detailed data model diagram of VDT basic objects and relationships among these objects. In these figures, a rectangular box represents an object type whereas an arc-box represents a class. Thin arrows represent one-to-one relationships whereas thick arrows represent one-to-many relationships.

Relationships across design units, those which are associations among Basic objects that are in different Basic clouds, are called external references. In Figure 2, the relationship from the Arch_body object to the Entity object is an external reference. When design object is stored onto the disk as a file, all basic objects contained within the corresponding basic cloud are also stored. However, the external references are broken during the storing of a design unit. The external references are restored when the design object and the basic cloud are loaded. For this purpose, the information on the external references are kept in the basic cloud.

3.2 Data structure of VDT objects

Given the data model, memory usage and performance are the key criteria in designing the data structure. Each object type in the data model is implemented by the class construct of the C++ programming language. Each attribute and relationship is implemented by a private member of the class. The access routines of these fields are implemented by member functions.

The data structure of VDT objects has many storage fields to capture a given VHDL description. These storage fields can be classified into two categories: attribute fields and relationship fields. The relationship fields store the pointer of a related object, which describes the relationship with the object. For instance, Signal object has a relationship field "type" to point a Type object that corresponds to a type of the signal. On the other hand, the attribute fields store the properties of object itself and corresponding VHDL construct, such as object sequence number, code line number, identifier, and so on.



```
entity Circuit is
   port (A : in bit;
          B,C : out bit
   );
end Circuit;

architecture Behav of Circuit is
   constant High : bit := '1'
begin
   B <= A and High;
   C <= not A;
end Behav;
```

```
Attribute
   object_type : entity
   id_number   : 160
   line_number : 1
   flag        : 0
   name        : Circuit
   symbols     : (161,A)
                (162,B)(163,C)
Relationship
   scope       :
   context     :  0
   generics    :
   ports       : 161 162 163
   subprograms :
   ...
```

```
Attribute
   object_type : arch_body
   id_number   : 170
   line_number : 8
   flag        : 0
   name        : Behav
   symbols     : (171,High)
Relationship
   scope       :
   context     : 0
   entity      : 160
   subprograms :
   constants   : 171
   ...
```

Figure 4. Data structure of VDT object.

Figure 4 shows the contents of two objects corresponding to entity declaration and architecture body. The architecture body has an identifier "Behav". This identifier is captured by the attribute field "name" of Arch_body object. The architecture body is associated with an entity declaration called "Circuit". This relationship is captured by the relationship field "entity".

Actually, the attribute values of VDT objects must be of type, such as string, integer, and user defined enumeration. All kinds of basic objects have some attribute fields in common, such as "object_type", "id_number", "line_number", and so on. The field "object_type" represents the kind of basic object. The field "id_number" represents the sequence number of basic object and identifies a unique basic object within a design unit. The field "line_number" represents the line number on which the corresponding VHDL construct appears in VHDL file.

There are two kinds of relationship, one-to-one relationship and one-to-many relationship. The one-to-one relationship field can point only at one object. The one-to-many relationship field can point at many objects. The naming convention distinguishes the two names of the one-to-one relationship field and one-to-many relationship field. If the name of a relationship field is an identifier with the ending 's', then the field is for one-to-many relationship. Otherwise, the field is for one-to-one relationship. The values of relationship fields must be the pointer of VDT object. When the contents of VDT objects are stored on disk or are printed on screen, the pointer value is replaced by the sequence number "id_number" of the pointed object.

VDT naming rule makes the convention on abbreviated names of VDT objects and the storage fields of these objects. For instances, Arch_body represents an object corresponding to architecture body, the relationship field "constants" of Arch_body object represents the list of constants declared within the architecture body.

3.3 Classification of basic objects

For each basic objects, there is a unique data structure. Basic objects can be distinguished by many kinds, as many as the kinds of VHDL constructs. For instances, a Signal object describes a signal declaration, a Conf_spec object describes a configuration specification in an intermediate form representation.

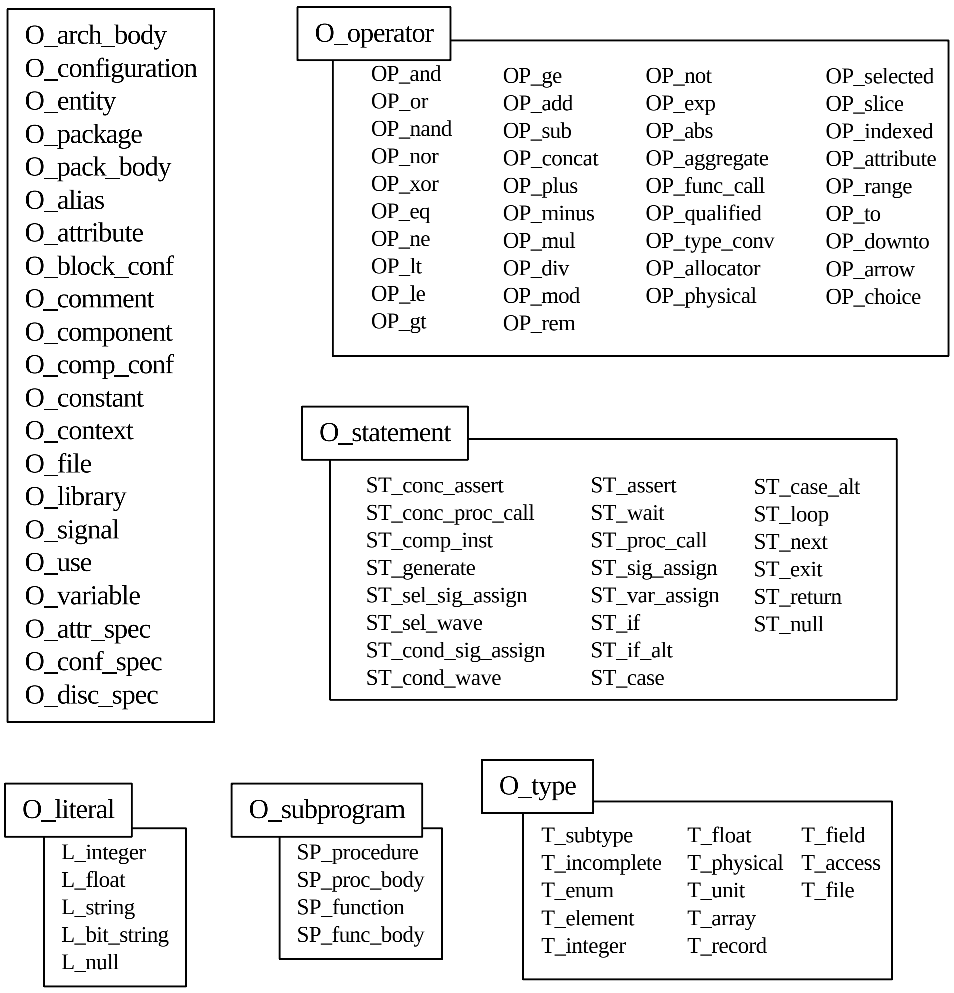


Figure 5. Basic object kinds.

Figure 5 shows the kinds of basic objects. Basic objects are identified by attribute field "object_type" whose type is of predefined enumeration type. The enumeration type named object_type has many elements shown in the above figure. O_arch_body is the object_type value of Arch_body object which is a kind of Basic object corresponding to architecture body. Simply, Arch_body object is called O_arch_body without confusion.

Several basic objects are divided more precisely. The related objects are O_literal, O_operator, O_statement, O_subprogram, and O_type. O_literal is classified in more detail into five kinds: L_integer, L_float, L_string, L_bit_string, and L_null. L_string is a kind of Literal objects, and corresponds to string_literal. Simply, a Literal object having "L_string" kind is called L_string without confusion.

3.4 Relationship checking

An individual intermediate form is constructed by analysis of a VHDL description that corresponds to a design unit. VHDL description is captured by VDT objects and the relationships among these objects in an intermediate form representation.

An intermediate form is composed of VDT objects and the relationships among these objects. The intermediate form representation can be regarded as a directed graph, where vertexes represent VDT objects, and edges represent the relationships among these objects.

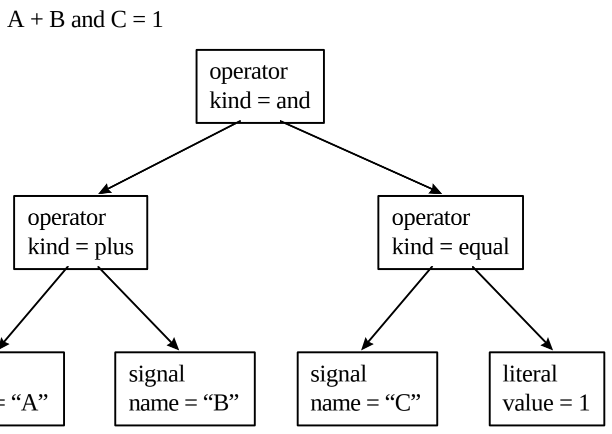


Figure 6. Representation of intermediate form

Figure 6 shows an example of intermediate form representation which corresponds to a VHDL expression. The intermediate form is composed of seven VDT objects and six relationships. The "and" operator is represented by OP_and, and "+", "=" operators are by OP_plus, OP_equal, respectively. The operands "A", "B", "C" are represented by O_signal, and the operand "1" is represented by L_integer.

The binary operators have two relationships that indicate the two operands respectively. In the above figure, the two relationships are represented by two directed arrows from an operator to the operands.

An intermediate form constructed by analysis of a design unit contains all syntax and semantics of a given VHDL description. If a given VHDL code has illegal syntax and semantics, the constructed intermediate form may contain these illegal syntax and semantics. For instance, a signal declaration creates a signal of the specified type. The corresponding O_signal has a relationship with O_type corresponding to the specified type. The object allowed to be linked with O_signal for this relationship are only O_type or Basic object regarded to a type.

The relationship checking does check if the relationship being made is legal by definition of VDT data model. Namely, the relationships defined by VDT data model restrict illegal intermediate form containing illegal syntax and semantics. But all illegal syntax and semantics cannot be restricted by these relationship definitions. So, an intermediate form may contain some illegal syntax and semantics. These remained illegalities can be removed by the analyzer and semantic checker.

3.5 Object-oriented Programming

VDT is developed by using C++ programming language[10], which is an object-oriented programming language. Object-oriented technique has three powerful features: data abstraction, inheritance, and polymorphism. VDT data model and PI library are implemented by fully using these features.

Data abstraction is provided by a *class* construct in C++ language. Each VDT object is implemented to a unique *class*. A *class* consists of a data structure and member functions that handle the data structure. Only the member functions of a class can access the associated data structure. A class construct provides protection mechanism that protects the accessing of a corresponding data structure from outside of a class.

Inheritance is accomplished by class hierarchy in C++ language. All classes corresponding to Basic objects are derived from one base class. All derived classes are cast to this base class, and inherit the storage fields and member functions of the base class. This inheritance mechanism enables code sharing among different classes. The code sharing is accomplished by defining a class with common storage fields and access routines, and by inheriting this class.

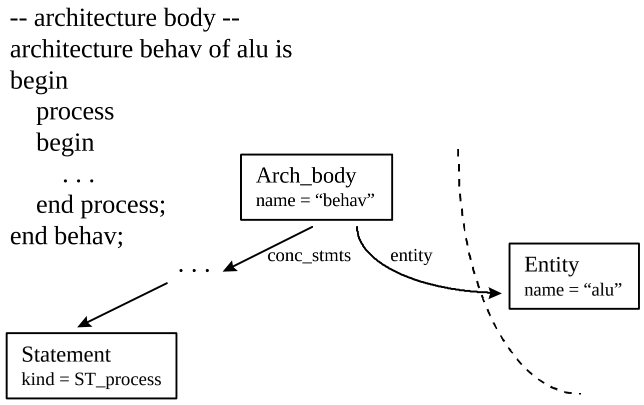
Polymorphism is accomplished by virtual function in C++ language. Each Basic object is implemented to a unique class that may have some common members to other class by inheriting from the same classes. A given Basic object may not have some member functions. A calling of these member functions for this object will cause error.

4 Intermediate Form Representation

In the programmer's view, there are containment relation among VHDL constructs. Design library is a VHDL construct larger than design unit and contains many design units. Design unit is a construct larger than entity declaration and contains many of these constructs. Entity declaration is a construct larger than signal declaration, constant declaration, etc.

4.1 Architecture body

Figure 7 shows the intermediate form representation corresponding to an architecture body. A process statement appears in architecture body. The architecture body and process statement are represented by Arch_body object named "behav" and Statement object, respectively. O_arch_body is Arch_body object, and ST_process is a Statement object attributed with ST_process kind. O_arch_body has a connection with ST_process, which represents the relation between architecture body and process statement.



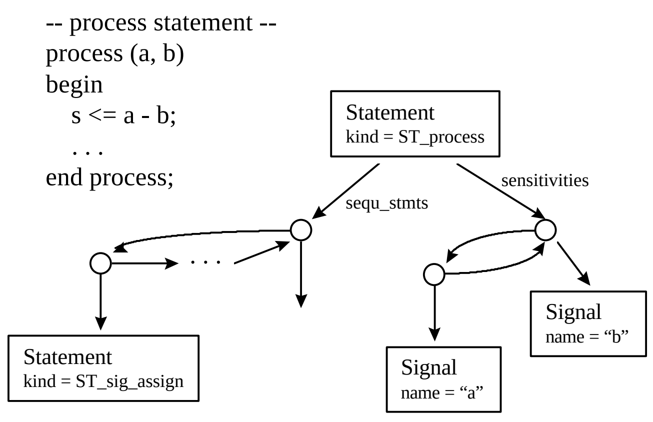
```
-- architecture body --
architecture behav of alu is
begin
   process
   begin
      . . .
   end process;
end behav;
```

Figure 7. Intermediate form representation for architecture body.

The data structure of Arch_body object has a attribute field named "name" which is of a string type and is written as "behav" in this example. And, it has a relationship field named "entity" which is of a pointer type and is written as the address of Entity object. It also has a relationship field named "conc_stmts" which is of a pointer array type. In this example, one ST_process is pointed by the "conc_stmts" field of Arch_body object. In addition, the data structure of Arch_body object has many fields.

4.2 Process Statement

In process statement, sensitivity list can appear. Sensitivity list indicates the signals that activate the process statement during simulation. ST_process has one-to-many relationship with these signals, and the objects related to "sensitivities" relationship of ST_process can be O_signal or objects that can be regarded to signal such as a signal attribute and a slice name of a signal. If it tries to make a relation with no signal object, error message will be reported by relationship checking routine of PI.



```
-- process statement --
process (a, b)
begin
    s <= a - b;
    . . .
end process;
```

Figure 8. Intermediate form representation of process statement.

Figure 8 shows the intermediate form representation corresponding to a process statement which contains many sensitivities and many sequential statements. The process statement is represented by ST_process. ST_process has sequ_stmts and sensitivities fields to describe the relations with sequential statements and sensitivities, respectively.

The containment relation is implemented to the directed pointer. The ST_process has the storage field named "sequ_stmts" and "sensitivities" of a pointer array type. The "sequ_stmts" field points the objects corresponding to sequential statements contained in this process statement. Actually, the pointer array is a single-linked list which enables easy append, deletion, and sequential searching.

4.3 Signal Assignment Statement

Figure 9 shows the intermediate form representation corresponding to signal assignment statement. A signal assignment statement is described by a ST_sig_assign in intermediate form. ST_sig_assign has "target" and "waveforms" relationships. The "target" relationship is 1:1 relationship and relates to an object corresponding to target. The "waveforms" relationship is 1:m relationship and relates to objects corresponding to waveform element list.

```
-- signal assignment statement --
s <= a + b after 3 ms;
```

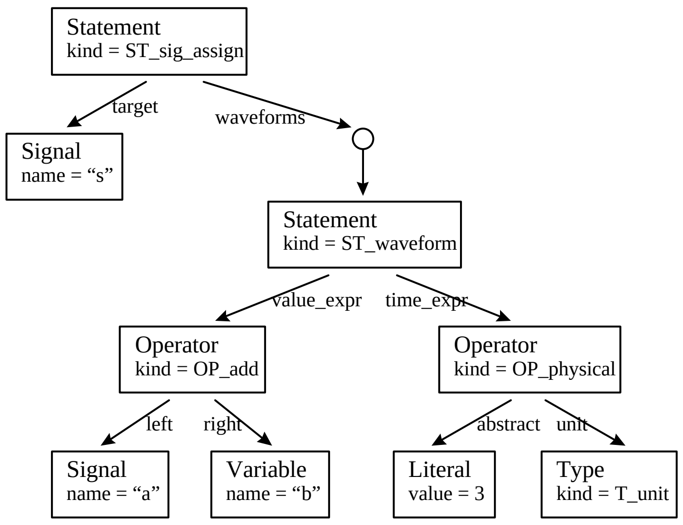


Figure 9. Intermediate form representation of signal assignment statement.

The syntax of waveform is "value_expression AFTER time_expression". ST_waveform is the Basic object corresponding to the waveform. ST_waveform has two relationships: "value_expr" and "time_expr" relationship. The "value_expr" relationship describes the relation to the value expression and points an object corresponding to an expression. The "time_expr" relationship describes its relation to the time expression, and points an object corresponding to an expression.

5 Procedural Interface

The core module of this toolkit is PI, which is a library of versatile routines that handle the intermediate form. The intermediate form is protected to be accessed by a pointer. The user can handle the intermediate form only through the routines provided by PI. PI library can be divided to three modules according to the related VDT object, design library manager, design unit manager, and library unit manager. According to library hierarchy, PI can be divided into two layers: primitive layer and application layer. The routines within primitive layer access directly the data structure of VDT object. On the other hand, the routines within application layer can not access the internal data of VDT object, but do more complex processing by using the routines within primitive layer.

The data structures are defined for each VDT object. And the access routines of an object are defined mutually for each field. The routines included in PI range up to several hundreds. It is hard to remember all the names of routines. The name of an access routine is a combined one with the name of a related storage field. This naming convention of routines enables a fast and easy recognition of the function of a routine.

PI library can be divided to three modules according to the related VDT object: library object manager, design object manager, and basic object manager. The module of library object management handles lib object which corresponds to the largest VHDL construct called design library. Design library is the storing location of design units and actually associated with a directory of file system. This module defines the data structure of lib object, and contains a few routines handling this object. The module of design object management handles design object which corresponds to the second largest VHDL construct called design unit. Design unit is an atomic unit which can be analyzed independently to other units. An intermediate form constructed by analysis of a design unit is stored as a file in a given directory. This module defines the data structure of design object, and contains a few routines handling this object. The module of basic object management handles basic object. All VHDL constructs except design unit and design library are represented by basic object in an intermediate form. Basic object is classified into many kinds. This module defines the data structures of each Basic object, and contains numerous routines handling these objects. The size of this module is very much larger than other modules, and most PI libraries are occupied by this module.

All routines in PI library can be classified into two categories: access routine and processing routine. The access routines read and write directly the data structure of VDT object. On the other hand, the processing routines just process the information of VDT object by using the access routines. The access routines are located in the primitive layer of PI, and the processing routines are located in a higher layer called the application layer of PI. The access routines can be classified into three categories according to related storage field of the data structure: attribute getting routine, 1:1 relationship making routine, and 1:m relationship making routine. The attribute getting routine reads a given storage field which is regarded as an attribute field. The 1:1 relationship making routine gets the object linked to a related 1:1 relationship field, and links a given object to that field. The 1:m relationship making routine gets the object list linked to a related 1:m relationship field, and links a given object to that field.

5.1 Library Object Management

This module has a few access routines and processing routines. As access routines, there are "open/close" routines, "add/sub/gen" routines, and "get" routine. As processing routines, there are "search" and "dump" routines.

The creation of a new design library is accomplished by just assigning the logical name of design library to a directory path-name of UNIX file system. The configuration is written to a special file named "$VDT/etc/.VDTRC". Figure 10 shows the contents of this configuration file. This file configures basically the design library named "STD" and "WORK", and includes other configuration files named "~/.vdtrc" and ".vdtrc".

```
-- contents of start-up file --
Library  STD      $VDT/etc/std
Library  WORK  .
Include  ~/.vdtrc
Include  ./.vdtrc
```

static SLList<Lib> gLib::_libs;*

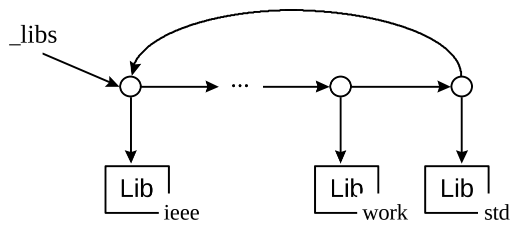


Figure 10. Create a design library.

All tools using VDT initially read the configuration file and automatically build the list of Lib objects. This list exists while a program is running and can be increased and decreased. If there is any error in a configuration file, a program cannot be run. The above is the syntax of design library configuration.

- Synopsis of new design library creation.

  LIBRARY   str1   str2

  char* str1; -- logical design library name.

  char* str2; -- path-name of a directory.

In default, there are three configuration files which have the precedence. If there are one more configuration for the same design library with different path-name, then the last configuration overrides.

5.1.1 Open/Close Routine

A Lib object represents a design library and captures the information of a directory corresponding to the design library such as path-name of the directory, the count of design units within the design library, and so on. Figure 11 shows the C++ code of open routine and the result of that running for "ieee" design library.

```
typedef Lib* lid;
gLib gl;
lid lib = gl.open("ieee");
```

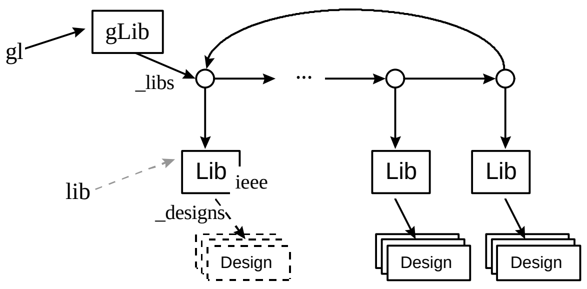


Figure 11. Open design library.

The type of "gLib" is predefined by PI library as a list of Lib objects. The type of "lid" is also predefined as a pointer of Lib object. The open routine is accomplished to a variable "gl" of "gLib" type with logical design library name "ieee". As a result, the variable "lib" gets the pointer of Lib object named "ieee" which is found in the list of Lib objects. During the processing of open routine, the files within a directory which are of VDT intermediate forms are collected and built to Design objects.

- Synopsis of open routine.

```
#include "libraryMan.h"
lid gl.open(str)
gLib gl;    -- variable of gLib type.
char* str; -- logical design library name.
```

Description

a. Find a Lib object for a given logical library name in the list of Lib objects.

b. Open a directory corresponding to the Lib object.

c. Collect files which are the saved images of VDT intermediate form.

d. Create a Design object for each file.

e. Append each created Design object to Lib object.

f. Close the directory.

g. Return the pointer of the found Lib object. If not found, return Null pointer.

cf) If a Lib object is already opened, the repeated running of open routine for the same Lib object will not accomplished. It saves the computing time and guarantees only one existence of the same Lib object in the system.

- Synopsis of close routine.

```
#include "libraryMan.h"
lid    gl.close(str)
gLib gl;    -- variable of gLib type.
char* str;  -- logical design library name.
```

Description

a. Find the Lib object for a given logical design library name.

b. Save each Design object to a file in the form of well-arranged binary data.

c. Locate the file in the directory corresponding to the Lib object.

d. Return the pointer of Lib object. If error occurs, return Null pointer.

cf) After the running of close routine, all data structures are resident in system until a program finishes.

5.1.2 Add/Sub/Gen Routine

A Lib object can contain one or more than one Design objects, and holds these objects as a list. This list represents the 1:m relationship between design library and design unit. A Design object can be added to, subtracted from, and referred to the list freely using PI routines.

```
typedef  Design*  did;
typedef  SLList<Design*>  gdid;
int st = lib->add_designs(dsn);
int st = lib->sub_designs(dsn);
gdid gd = lib->gen_designs();
```

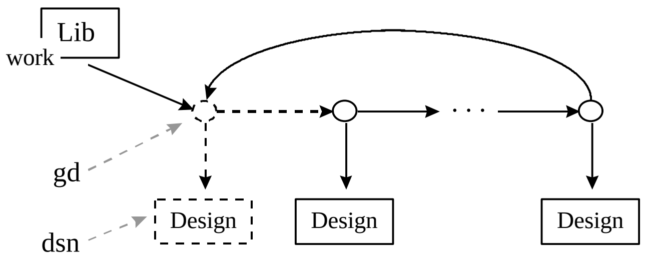


Figure 12. Add, Subtract, and refer design units.

Figure 12 shows the C++ code for handling Design object and the result of that running. The type "did" is predefined to a pointer of Design object by PI library. And, the type "gdid" is also predefined to a pointer of a single linked list of Design objects. For the Lib object named "work", add routine adds the given Design object to the list within Lib object, sub routine subtracts the given Design object from the list, and gen routine returns the list of all Design objects contained in Lib object.

- Synopsis of add routine

  #include "libraryMan.h"

  #include "designMan.h"

  int lib->add_designs(dsn)

  lid lib; -- pointer of a Lib object.

  did dsn; -- pointer of a Design object.

Description

a. Append a Design object to the list which is a storage field of a given Lib object.

b. Return 1 if finished successfully, otherwise return 0.

cf) If there is already a Design object which has the same name, it is an erroneous condition.

- Synopsis of sub routine

  #include "libraryMan.h"

  #include "designMan.h"

  int   lib->sub_designs(dsn)

  lid lib; -- pointer of a Lib object.

  did dsn; -- pointer of a Design object.

Description

a. Subtract a Design object from the list of a given Lib object.

b. Return 1 if finished successfully, otherwise return 0.

- Synopsis of gen routine

  #include "libraryMan.h"

  #include "designMan.h"

  gdid   lib->gen_designs()

  lid lib; -- pointer of a Lib object.

Description

a. Collect the Design objects which is connected to a given Lib object as a list.

b. Return the pointers of Design objects as a single linked list.

5.1.3 Get Routine

The type of attribute fields is not a pointer type but integer, string, enumeration, and so on. The attribute value of a VDT object is determined when the object is created, and can be changed. So, there is only access routine to refer the attribute fields. The get routine is the access routine of the attribute fields.

- Synopsis of get routine

  #include "libraryMan.h"

string   lib->get_***name***()

lid lib; -- pointer of a Lib object.

Description

a. Return the logical library name corresponding to a given Lib object.

cf) The above italic string "name" can be replaced to other strings in order to get the other attribute field.

5.1.4 Search Routine

As a design library is distinguished by a different logical name, a design unit is distinguished by a different design unit name. There are two methods to find a design unit contained in a design library. One is the sequential search of all Design objects, which can be obtained using gen routines. The other is by using the search routine which is defined as a member function of Lib object. Lib object internally has a symbol table for design units. The symbol table uses the name of design unit as a key. So, an easy and fast search can be accomplished only with the name of a design unit.

```
lid  lib  = gl.open(“work”);
did dsn = lib->search(“alu”, “behav”);
```

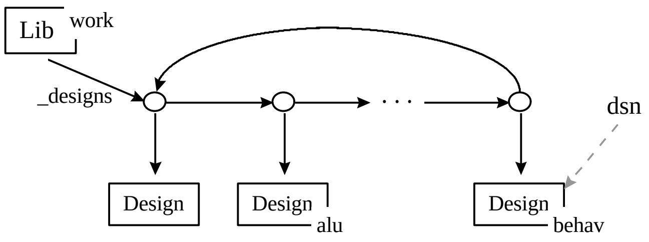


Figure 13. Search a design unit.

Figure 13 shows the C++ code of search routine and the result. The Lib object "work" can first be obtained using open routine. To search for a design unit, two names are needed as parameters of search routine. The first name "alu" is a primary unit name, the second name "behav" is a secondary unit name. Namely, the searching design unit may be a architecture body named "behav" corresponding to an entity named "alu". If a primary unit wish to be obtained, the second name should not be given. In a design library, only one design unit can exist with the same name. So, the search routine returns just one Design object.

- Synopsis of search routine

    #include "libraryMan.h"

    #include "designMan.h"

    did   lib->search(str1[, str2])

    lid lib;   -- pointer of a Lib object.

    char* str1; -- identifier of primary unit.

    char* str2; -- identifier of secondary unit.

Description

a. Make a symbol corresponding to given design unit names.

b. Find a symbol by searching the symbol table contained in a given Lib object.

c. Return the pointer of the found Design object. Return Null pointer if not found.

cf) To search for a design unit in a given design library. The corresponding Lib object should first be opened. If not opened, it will return Null pointer because the Lib object does not yet have a list of Design objects.

5.1.5 Dump Routine

VDT data model defines the Lib object and its data structure. The storage fields of the data structure can be classified into attribute fields and relationship fields as other VDT objects. The dump routine prints the contents of the data structure of a given object. This routine may help in the development of VDT applications by providing an easy and fast debugging mechanism.

```
int st =  lib->dump();

Design Library ...

        name        :  work
        path        :  ./LIB
        designs      :  (ALU_STATGE) (ALU_STAGE__BEHAVIOR)
        (ALU_STAGE__STRUCTURE) (TRAFFIC)(TLC) (TLC_SPEC)
```

Figure 14. Text output of the contents of Lib object.

Figure 14 shows the C++ code of dump routine and the result for a Lib object. The attribute field "name" describes the logical library name "work". The attribute field "path" describes the directory path-name "./LIB" corresponding to the design library. The field "designs" represents the 1:m relationship to design units, and describes the list of Design objects linked to the Lib object. In the textual output, the symbols of Design objects are printed instead of Design objects itself. The symbol of a Design object is made by using the name of corresponding design unit. Namely, the symbol of a secondary unit is made by "the associated primary unit name" + "__" + "the secondary unit name". The symbol of a primary unit is just its name. For instance, "TLC" may be an entity declaration. "TLC__SPEC" may be an architecture body named "SPEC", whose associated entity declaration is called "TLC".

- Synopsis of dump routine

```
#include "libraryMan.h"
int lib->dump()
lid lib; -- pointer of a Lib object.
```

Description

a. Print the contents of a given Lib object in textual form.

b. Return 1 if there is no error, otherwise return 0.

5.2 Design Unit Management

This module has few access routines and processing routines. As access routines, there are "load/save" routines, "create" routine, and "get" routine. As processing routines, there are "dump" routine and "check" routine.

A design unit is composed of a library unit and a context clause. A design unit certainly has an associated library unit. There are five kinds of VHDL construct called library unit. A library unit is either a primary unit or a secondary unit. VHDL constructs called primary unit include entity declaration, package declaration, and configuration declaration. Other constructs, architecture body and package body are called the secondary unit. A secondary unit is a separately analyzed body of a primary unit resulting from a previous analysis.

The design entity is the primary hardware abstraction in VHDL. It represents an individual hardware design that has well-defined inputs and outputs and performs a well-defined function.

A complete design is composed of one or more design entities, and is represented by a design hierarchy. A design entity may be described in terms of interconnected components. Each component of a design hierarchy may be bound to a lower level design entity. The bindings necessary to identify a design hierarchy can be specified in a configuration of the top level entity in the hierarchy.

A design hierarchy can be defined by a design entity or by configuration. A design entity is defined by an entity declaration together with a corresponding architecture body. There are associations between a primary unit and a corresponding secondary unit.

The name of a primary unit is given by the first identifier after the initial reserved word of that unit. As for the secondary units, only the architecture body is named. Each primary unit in a given design library must have a simple name that is unique within the given design library. Also, each secondary unit associated with a given primary unit must have a unique name, by which the name is assembled with the name of secondary unit and the name of associated primary unit.

	pname	sname	name
entity	entity identifier	architecture identifier	pname
arch_body	entity identifier	architecture identifier	pname + sname
package	package identifier		pname
pack_body	package identifier	package identifier	pname + sname
configuration	configuration identifier	entity identifier	pname

Figure 15. Naming of a design unit.

Figure 15 shows the naming rule of a design unit. The data structure of Design object has many attribute fields such as, "object_type", "pname", "sname", "name", and so on. These storage fields is for the identification of a design unit. The field "pname" stores the primary unit name. The field "sname" stores a secondary unit name. A design unit, which the associated library unit is a primary unit, has only "pname". If the associated library unit is a secondary unit, the design unit has "sname" and "pname", where the "pname" is the identifier of the associated primary unit.

The field "name" records a unique name for a design unit within a design library. The name is made by an assembly of primary unit name and secondary unit name. For instance, of a design unit associated with an architecture body, the name is an assembled string of "pname" and "sname" with separator "--", where "sname" is the identifier of the architecture body, and "pname" is the identifier of the associated entity declaration.

The design unit whose associated library unit is a primary unit does not have "sname". But the field "sname" is used for special purpose, a design unit associated with entity declaration uses the "sname" to record the identifier of an architecture body. This recording indicates the most recently analyzed architecture body associated to the entity declaration. This pair becomes a default design entity. Also, a design unit associated with configuration declaration uses the "sname" to record the identifier of associated entity declaration.

5.2.1 Load/Save Routine

Design object can be regarded as a container of Basic objects. A stored intermediate form is composed of one Design object and many Basic objects. The behavior of circuits are represented wholly by Basic objects and links among these objects which is called Basic cloud. The Basic cloud is a directed graph which contains only one Basic object corresponding to library units. This object is a top vertex of the graph, and Design object points to the top object using storage field "top".

```
did dsm = dsn->load();
did dsm = dsn->save();
```

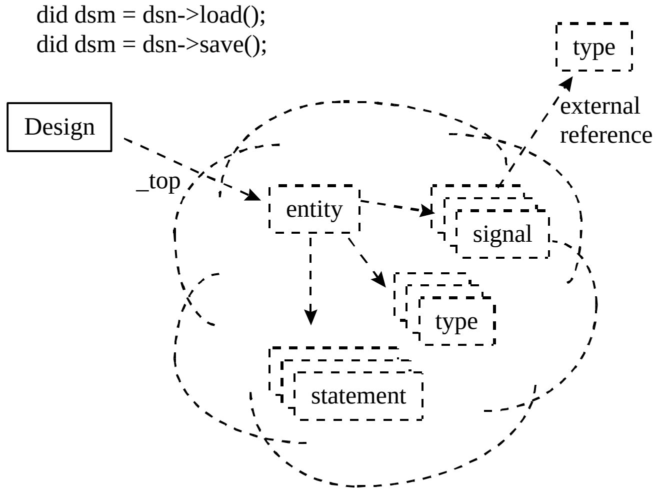


Figure 16. Load and save a design unit.

Figure 16 shows the C++ code of load/save routines and the results. A created or opened Design object just contains no Basic objects, and the field "top" has Null pointer. If there is the intermediate form file corresponding to a design unit, the "load" routine reads the file, constructs an intermediate form, and records the field "top" to the pointer of a top object. The "save" routine stores a given Design object and the related Basic cloud to a intermediate form file. The contents of Design object are located at the header part of a file, and the contents of Basic cloud is appended to the file. The "open" routine for a design unit just reads the header part of the corresponding intermediate form file, and creates a Design object with that contents. In the future, the "load" routine will read the remainder of the file.

An intermediate form corresponding to a given design unit may have any external reference like that from O_signal to O_type, where the O_type is not contained in the given design unit. The "save" routine breaks the external link during storing of a file, and the "load" routine restores the external links during reading.

- Synopsis of load routine

```
#include "designMan.h"
did    dsn->load()
did dsn; -- pointer of a Design object.
```

Description

a. Get a path-name of the intermediate form file from a given Design object.

b. Check that the file is not corrupted by reading the contents of the header part.

c. Reconstruct an intermediate form at the same time as when reading the remainder part of the file.

d. Return the pointer of a given Design object if there is no error, otherwise return Null pointer.

cf) During the reading of a given intermediate form file, there is any link to a Basic object which is contained in other Design object. The related design unit will be loaded automatically.

- Synopsis of save routine

  #include "designMan.h"

  did   dsn->save()

  did dsn; -- pointer of a Design object.

Description

a. Get a path-name for the intermediate form file from a given Design object.

b. First save the contents of Design object in the header part of the file.

c. Save the contents of Basic cloud in the remainder part of the file.

d. Return the pointer of a given Design object if there is no error, otherwise return Null pointer.

5.2.2 Create Routine

Design object is a container of Basic objects. The data structure of Design object has a storage field for Basic object list. The "create" routine creates a Basic object, and appends to the list of a given Design object. Each created Basic object is numbered sequentially, where the sequence number is unique within in a Design object. Actually, the Basic object list is implemented on to a linear table which is automatically and discretely extended.

typedef Basic oid;*
oid obj = dsn->create_type(ln, “bit”, T_enum);

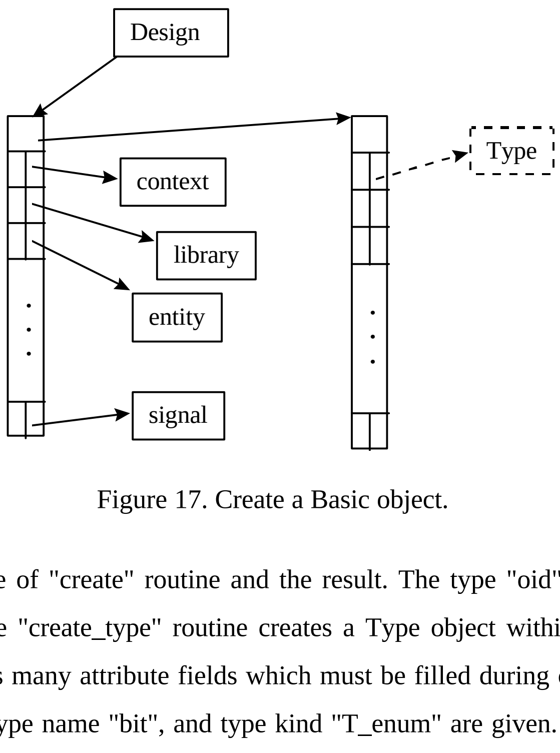


Figure 17. Create a Basic object.

Figure 17 shows the C++ code of "create" routine and the result. The type "oid" is predefined to a pointer of Basic object by PI library. The "create_type" routine creates a Type object within a Design object pointed by variable "dsn". Type object has many attribute fields which must be filled during creation. As parameters in the above code, line number "5", type name "bit", and type kind "T_enum" are given. Newly created Type object is appended to the last index of the linear table.

- Synopsis of create routine

  #include "designMan.h"

```
#include "basicMan.h"
oid    dsn->create_type(ln,str,tk)
did dsn;    -- pointer of a Design object.
int ln;       -- line number within VHDL file.
char* str;  -- identifier of type declaration.
type_kind tk; -- kind of type declaration.
```

Description

a. Create a Type object attributed by the given parameters.

b. Append the object to the Basic object list within a given Design object.

c. Return pointer of Type object if there is no error, otherwise return Null pointer.

cf) The above italic string "type" can be replaced into other strings to create other kinds of Basic object. All attribute fields of a created object must be given as parameters.

5.2.3 Get Routine

The type of attribute fields is not a pointer type but integer, string, enumeration, and so on. The attribute value of a VDT object is determined when the object is created, and can be changed. So, there is only access routine to refer the attribute fields. The get routine is the access routine of the attribute fields.

- Synopsis of get routine

```
#include "designMan.h"
string    dsn->get_pname()
did dsn; -- pointer of a Design object.
```

Description

a. Return the primary unit name corresponding to a given Design object.

cf) The above italic string "pname" can be replaced to other string for getting other attribute field.

5.2.4 Dump Routine

As other VDT objects, the data structure of Design object also has many attribute fields and a few relationship fields. The "dump" routine prints the contents of a given Design object in textual form. It will help to debug during the development of VDT applications.

```
int st = dsn->dump();

Design unit ...

     type        : entity
     pname       : ALU_STAGE
     sname       : STRUCTURE
     path        : ./LIB/ALU_STAGE.if
     spath       :
     count       : 171
     top         : 160
```

Figure 18. Textual output of the contents of a given Design object.

Figure 18 shows the C++ code of "dump" routine and the result. The attribute fields do not record the pointer value but integer or string value. The relationship fields record the pointers of related objects, where the related objects are only Basic objects. Instead of printing the pointer value of related object, the sequence number of that object is printed. This routine prints the contents of a Design unit pointed by variable "dsn", and return either success or fail.

- Synopsis of dump routine

  #include "designMan.h"

  int   dsn->dump()

  did dsn; -- pointer of a Design object.

Description

a. Print the contents of a given Design object in textual form.

b. Return 1 if no occur, otherwise return 0.

5.2.5 Generate Routine

A design unit is composed of a context clause and a library unit. The "generate" routine for a Design object makes VHDL code corresponding to each Basic object contained in this object. If the given object does not have enough information, the generated VHDL code may contain illegal syntax and semantics.

- Synopsis of generate routine

  #include "designMan.h"

  void dsn->generate()

  did dsn; -- pointer of Design object.

Description

a. Generate the VHDL code from a given Design object "dsn".

b. Print the textual code to the screen by default.

5.2.6 Check Routine

The access routines located in primitive layer of PI library can be classified into two categories: attribute getting routines and relationship making routines. The relationship making routines check if the relationship for the given objects is legal to make. But only a few illegal syntax and semantics can be detected by relationship making routines.

All lexical and syntactic errors are detected by analyzer during parsing of VHDL code and constructing of an intermediate form. And, all semantic errors are detected by this "check" routine. The "check" routine is located in application layer of PI library, and does the semantic checking for each Basic object contained within the given Design object.

● Synopsis of check routine

```
int dsn->check()
did dsn; -- pointer of a Design object.
```

Description

a. Get Basic objects contained within the given Design object.

b. Check semantics for each Basic object.

c. Print error message for each semantic error.

d. Return 1 if there is no error, otherwise return 0.

5.3 Library Unit Management

This module has numerous access routines and processing routines. As access routines, there are "set/get" routine, "add/sub/gen" routine, and "get" routine. As processing routines, there are "search", "dump", "generate", and "check" routine.

All VHDL constructs, except for design unit and design library, are represented by Basic objects in an intermediate form. Basic object is classified into many kinds, and identified by the value of storage field "object type". And some kind of objects are more classified by many kinds: "operator kind", "literal kind", and so on.

A declarative region is a portion of the text of the VHDL description. A declarative region is formed by the text of each of the several VHDL constructs, such as entity declaration, process statement, block configuration, and so on. The declarative region is said to be associated with the corresponding declaration or statement.

A declarative region is represented by the symbol table, which is located within a Basic object associated with the declaration region. The symbol table of a Basic object enrolls All VHDL constructs having names and appearing within the corresponding VHDL construct. There is hierarchy among declarative regions. For instance, a declarative region associated with entity declaration is larger than the region associated with process statement. Likewise, there is hierarchy among Basic objects having symbol table.

The scope of a declaration is a portion of the text in which the declaration may be visible. A named entity can be visible only within its own scope. A declaration appears with a declaration region associated with a certain VHDL construct. Namely, the scope of a named entity may be the declarative region appearing the declaration. The Basic object corresponding to a named entity is enrolled in to the symbol table, in which the declaration of the named entity appears within the declarative region corresponding to the symbol table.

5.3.1 Set/Get Routine

The storage fields for representing 1:1 relationship can record only one pointer of Basic object. The records of these fields can be modified by "set" routine, and can be referred by "get" routine.

```
oid os = dsn->create_signal(5, “clk”, M_in, S_null);
int  st = os->set_type(ot);
oid ot = os->get_type();
```

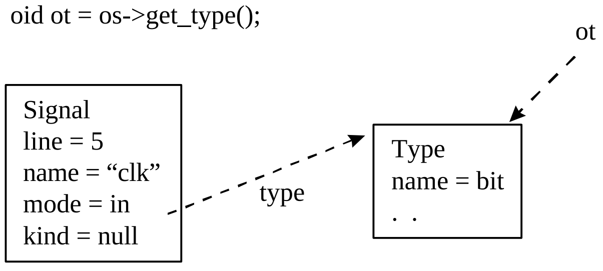


Figure 19. Set and refer a Basic object.

Figure 19 shows the C++ code of "set/get" routine and the results. The "create_signal" routine creates a Signal object named "clk", and returns the pointer "os" of the created object. For the object "os", the "set_type" routine makes a 1:1 relationship with a given Type object "ot". The "get_type" routine returns the Type object related to "os" object.

- Synopsis of set routine

    #include "basicMan.h"

    int obj->set_***type***(obt)

    oid obj; -- pointer of Basic object.

    oid obt; -- pointer of Basic object.

Description

a. Link the given object "obt" to the storage field "type" of "obj" object.

b. Print error message if "obj" object does not have the "type" relationship.

c. Print error message if the given "obt" object is illegal as Type object.

c. Return 1 if there is no error, otherwise return 0.

cf) The above italic string "type" can be replaced into other relationship names to make a 1:1 relationship.

- Synopsis of get routine

    #include "basicMan.h"

    oid obj->get_***type***()

    oid obj; -- pointer of Basic object.

Description

a. Get the object linked to the storage field "type" of a given object "obj".

b. Report error if the object "obj" does not have a relationship named "type".

c. Return the linked object if there is no error, otherwise return Null pointer.

cf) The above italic string "type" can be replaced into other relationship names to get the 1:1 relationship field.

5.3.2 Add/Sub/Gen Routine

The storage fields for representing 1:m relationship can link one more Basic object. The records of these fields can be modified by "add/sub" routine, and can be referred by "gen" routine.

```
int st = obj->add_signals(os);
int st = obj->sub_signals(os);
gid gob = obj->gen_signals();
```

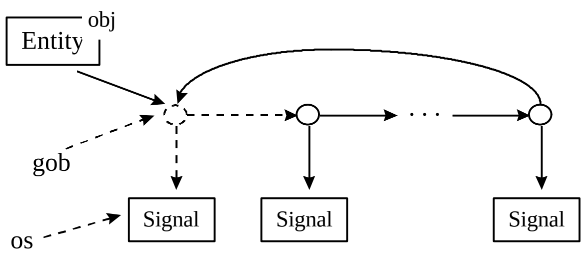


Figure 20. Append, detach, and refer Basic objects.

Figure 20 shows the C++ code of "add/sub/gen" routine and the results. The "add_signals" routine appends the given Signal object "os" to the list of Entity object "obj". The "sub_signals" routine detaches the given Signal object "os" from this list. The "gen_signals" routine returns the copy "gob" of this list.

- Synopsis of add routine

  #include "basicMan.h"

  int obj->add_***signals***(obs)

  oid obj; -- pointer of Basic object.

  oid obs; -- pointer of Basic object.

Description

a. Append the given object "obs" to the signal list of object "obj".

b. Report error if the object "obj" does not have a 1:m relationship for signal list.

c. Report error if the given object "obs" cannot be regarded as Signal object.

d. Return 1 if there is no error, otherwise return 0;

cf) The above italic string "signals" can be replaced into other relationship names to make the 1:m relationship.

- Synopsis of sub routine

  #include "basicMan.h"

  int obj->sub_***types***(obt)

  oid obj; -- pointer of Basic object.

  oid obt; -- pointer of Basic object.

Description

a. Detach the given object "obt" from the type list of object "obj".

b. Report error if the object "obj" does not have a 1:m relationship for type list.

c. Return 1 if there is no error, otherwise return 0.

cf) The above italic string "types" can be replaced into other relationship names to remove the 1:m relationship.

- Synopsis of gen routine

```
        #include "basicMan.h"
        gid obj->gen_signals()
        oid obj; -- pointer of Basic object.
```

Description

a. Copy this signal list of object "obj".

b. Report error if the object "obj" does not have a 1:m relationship for signal list.

c. Return the signal list if there is no error, otherwise return empty list.

cf) The above italic string "signals" can be replaced into other relationship names to refer the 1:m relationship.

5.3.3 Get routine

The storage fields for representing attribute can record one value which should be of a certain type. The records of these fields can be accomplished during the creation time of Basic object. Afterward, these records cannot be modified and can only be referred by "get" routine.

- Synopsis of get routine

```
        #include "basicMan.h"
        char* obj->get_name()
        oid obj; -- pointer of Basic object.
```

Description

a. Get the string value of attribute field "name".

b. Report error if the object "obj" does not have a attribute field named "name".

c. Return the value if there is no error, otherwise return Null string.

cf) The above italic string "name" can be replaced into other attribute names to get the attribute field. The type of returned value is related to the attribute field.

5.3.4 Search Routine

The "search" routine can easily find a named object by searching the symbol table of a given object. This routine may search globally with option "Global" by traversing the hierarchy and searching each symbol table.

```
typedef  SLList<oid*>  gid;
oid obj = dsn->get_top();
gid gob = obj->search("bit", Global);
```

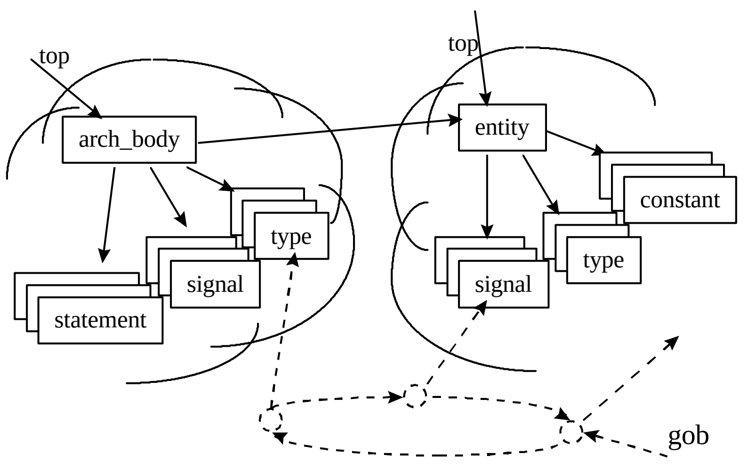


Figure 21. Search a named object.

Figure 21 shows the C++ code of "search" routine and the result. The "search" routine searches globally an object named "bit" with option "Global" by traversing the scope hierarchy. The traversing may be over to other design unit. This routine collects all objects named "bit" and return the object list in order, in which the object appearing in lowest region of scope hierarchy is first in the list.

- Synopsis of search routine

```
#include "basicMan.h"
gid obj->search(str, opt)
char* str; -- name of Basic object.
int opt;    -- option of searching.
```

Description

a. Search a given named object contained within a Basic object "obj".

b. Traverse to the upper region of object "obj" if with global option.

c. Make the list of found objects by searched order.

d. Return the list if there is no error, otherwise return empty list.

cf) This routine is defined only for the objects corresponding to VHDL constructs that have a declarative part. In other words, this routine can be invoked for a given object that is included in Region group.

5.3.5 Dump Routine

The data structure of each Basic object has many attribute fields and many relationship fields. The "dump" routine prints the contents of a given Basic object in textual form. It will help in the development of VDT applications.

```
int st = obj->dump();

Attribute
        object_type   : entity
        id_number     : 160
        line_number : 1
        flag          : 0
        name          : ALU_STAGE
        symbols       : (163,S1) (162,S2) (166,B1) (161,S3)
        (168,M)(167,C1) (164,S0) (165,A1) (169,C2) (170,F1)
Relationship
        scope         :
        context       :  0
        generics      :
        ports         : 161 162 163 164 165 166 167 168 169 170
        subprograms :
        types         :
        constants     :
        signals       :
        files         :
        aliases       :
        attributes    :
        attr_specs    :
        disc_specs    :
        uses          :
```

Figure 22. Textual output of the contents of Entity object.

Figure 22 shows the C++ code of "dump" routine and the result. The "dump" routine prints the contents of a given Basic object, Entity object in this figure. Entity object has a symbol table because the corresponding entity declaration is associated with a declaration region.

- Synopsis of dump routine

```
#include "basicMan.h"
void obj->dump()
oid obj; -- pointer of Basic object.
```

Description

a. Print the contents of a given Basic object in textual form.

b. Return 1 if there is no error, otherwise return 0.

5.3.6 Generate Routine

All kinds of Basic objects each have a corresponding VHDL construct. The "generate" routine for a Basic object makes the corresponding VHDL code from the information contained in the object. If the given object does not have enough information, the generated VHDL code may contain illegal syntax and semantics.

- Synopsis of generate routine

```
#include "basicMan.h"
void obj->generate()
```

```
oid obj; -- pointer of Basic object.
```

Description

a. Generate the VHDL code from a given Basic object "obj".

b. Print the textual code on the screen by default.

5.3.7 Check Routine

All kinds of Basic objects each have a corresponding VHDL construct. The "check" routine for a Basic object checks the all semantics of corresponding VHDL construct.

● Synopsis of check routine

```
#include "basicMan.h"
void obj->check()
oid obj; -- pointer of a Basic object.
```

Description

a. Check the semantics for a given Basic object "obj".

b. Print proper message immediately when error is detected.

c. Return no value.

5.3.8 Characterize Routine

Basic objects are classified into many kinds according to various properties of VHDL constructs. For instance, a named entity -- such as, signal, type, variable, and so on. -- can be referred by other portion of VHDL construct. These Basic objects can be grouped and called named object. Another example, a typed entity -- such as signal, variable, subprogram, and so on. -- has a relationship to Type object. These objects can be grouped and called typed object. The typed object can only appear in an expression.

There are numerous routines that get the properties from a given Basic object. The VDT user may get the properties by digging all the attribute fields and relationship fields for a given object. But it is easy to use these routines to get various properties.

● Synopsis of characterize routine

```
#include "basicMan.h"
int obj->hasScope()
int obj->isSequStmt()
#include "glossary.h"
int isStaticExpr(obj)
oid getExprType(obj)
oid obj; -- pointer of Basic object.
```

Description

a. The "has" routine returns if a given Basic object has "scope" field.

b. The "is" routine returns if a given object belongs to sequential statement group.

c. The third "is" routine returns if a given object belongs to static expression group.

d. The "get" routine returns the Type object related to a given expression.

cf) The above italic strings can be replaced into other strings to get other properties.

6 Characterize Routines

Lib object and Design object cannot be divided any further. But Basic object is classified into many kinds. Each object corresponds to a unique VHDL construct, and also has the properties which the corresponding construct has. For instance, an object has a name, an enclosing region, a specified type, and so on. In a detail, an object belongs to the group of expression, sequential statement, scalar type, and so on. Basic object can be classified not only by object kinds, but also by these properties.

All kinds of Basic objects are implemented to classes derived from a same base class named "Basic". These objects are viewed to user just as the same object. In many cases, the user wants to know the kind of a given Basic object and the properties of that object. The user can find these information by digging the storage fields of a given object. But it is easy to use the "characterize" routines that find any properties from a given object. These "characterize" routines are classified into three kinds: class identifying routines, group identifying routines, and property getting routines.

Class identifying routines identify that a given object belongs to a specified class. There are many intermediate classes defined just for code sharing. These routines are accomplished just by checking the class hierarchy. Group identifying routines identify that a given object belongs to a specified group. Basic objects having the same properties are grouped together, and are said to belong to any group. This can be accomplished by digging only attribute fields of a given object. Property getting routines get any specified properties from a given object by digging into the relationship fields of that object.

All attribute fields of VDT object are determined at the time of creation, and cannot be changed afterwards. So first and second routines always guarantee a correct result and errorless completion. These routines are used for the relationship checking that is accomplished within relationship making routines.

On the other hand, the relationship fields are changed freely by using the relationship making routines. This meas that property getting routines cannot guarantee correct results. For instance, if a given object does not have enough information represented by relationships, then these routines cannot obtain any properties.

6.1 Class identifying routines

The inheritance mechanism of object-oriented technique enables the code sharing. The common storage fields and member functions can be defined as an individual class. All classes derived from a class inherit all storage fields and member functions of that class.

In addition, the inheritance mechanism enables the grouping of objects that have some common properties. All objects corresponding to classes derived from a class are said to belong to that class.

For a given object, it usually needs to know if the corresponding class inherit a specified class, in other words, belongs to a specified class. Through the following "has" routine, a given object has some specified storage fields and member functions, in other words, belongs to a specified class.

- Synopsis of "has" routines.

  int obj->has***ClassName***();

  oid obj; **--** pointer of a Basic object.

Description

a. These routines return 1 if a given object belongs to the specified class, otherwise return 0.

cf) The italic string can be replaced into other class names to identify other classes.

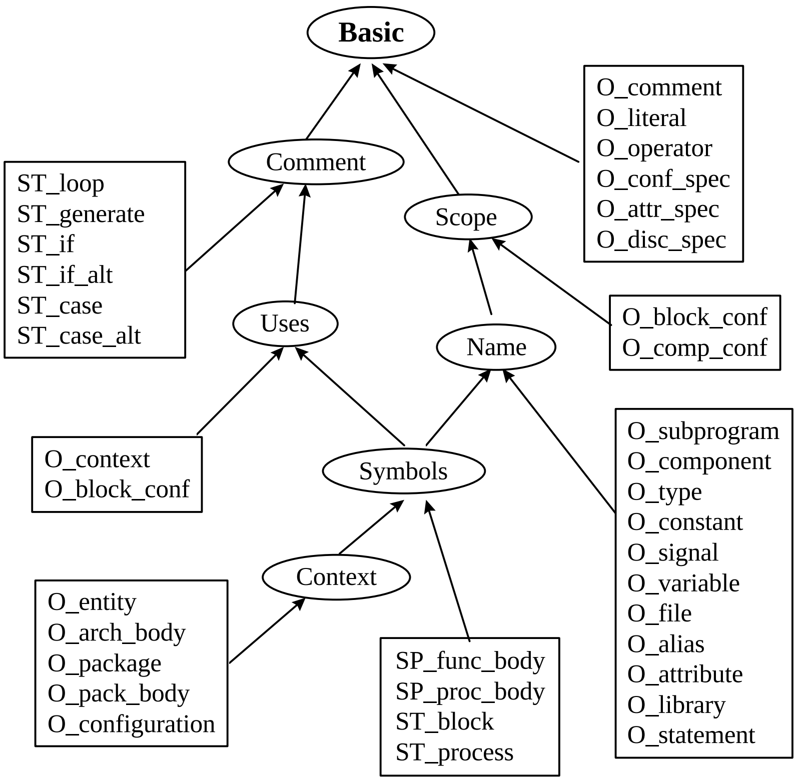


Figure 23. Class hierarchy of Basic objects.

Figure 23 shows the class hierarchy of Basic object. All classes are derived from the same class named "Basic". All terminal classes corresponds to some kind of Basic objects. The intermediate classes are defined for code sharing. Namely, these classes have common storage fields and member functions that are shared by multiple classes. The following list are all intermediate classes defined by the above class hierarchy.

- Context

This class contains a relationship field named "context" and several member functions. This field makes the relationship to an object that corresponds to context clause. All classes corresponding to library units inherit this class.

- Symbols

This class contains a relationship field named "symbols" and several member functions. This field makes a relationship to object list that corresponds to named entities. All classes corresponding to VHDL constructs that contain a declarative part inherit this class.

- Uses

This class contains a relationship field named "uses" and several member functions. This field makes a relationship to object list that corresponds to use clauses. All classes corresponding to VHDL constructs that may contain one or more use clauses inherit this class.

- Comments

This class contains a relationship field named "comments" and several member functions. This field makes a relationship to object list that corresponds to comments. All classes corresponding to VHDL constructs that may contain one or more comments inherits this class.

- Scope

This class contains a relationship field named "scope" and several member functions. This field makes a relationship to an object that corresponds to enclosing region. All classes corresponding to VHDL constructs that have an enclosing region inherit this class.

- Name

This class contains a attribute field named "name" and member functions. This field records the name of an object. All classes corresponding to VHDL constructs that may have a name inherit this class.

- Type

This class contains a relationship field named "type" and several member functions. This field makes a relationship to an object that corresponds to a type declaration. All classes corresponding to VHDL constructs that have a specified type inherit this class.

6.2 Group identifying routines

Basic objects having the same properties are grouped together, and are said to belong to the same group. For instance, the constraint in VHDL takes a role of constraining the range of the specified type. The constraints are described by several objects according to the text of constraint. These objects are grouped and are said to belong to a Constraint group.

As another example, an expression in VHDL is composed of operators and operands. Each operator and operand has a value and a type. The objects corresponding to operators or operands are grouped and are called Expression object. As the objects corresponding to operands of an operator, the objects included in Expression group are only allowed.

It usually needs to know if a given object belongs to any specified group. The following "is" routine provides this function.

- Synopsis of "is" routines

  int obj->is***GroupName***();

  oid obj; -- pointer of Basic object.

Description

a. These routines return 1 if a given object belongs to the specified group, otherwise return 0.

cf) The italic string can be replaced into other group names to identify other groups.

In special, the following routines are defined to identify that a given object is the object having specified kind. These routines are defined as virtual functions with a different argument.

virtual int isRight(const object_type) const;

virtual int isRight(const literal_kind) const;

virtual int isRight(const operator_kind) const;

virtual int isRight(const stmt_kind) const;

virtual int isRight(const subprog_kind) const;

virtual int isRight(const type_kind) const;

Figure 24 shows the groups that are classified Basic objects according to attribute fields only. All defined groups are listed as follows.

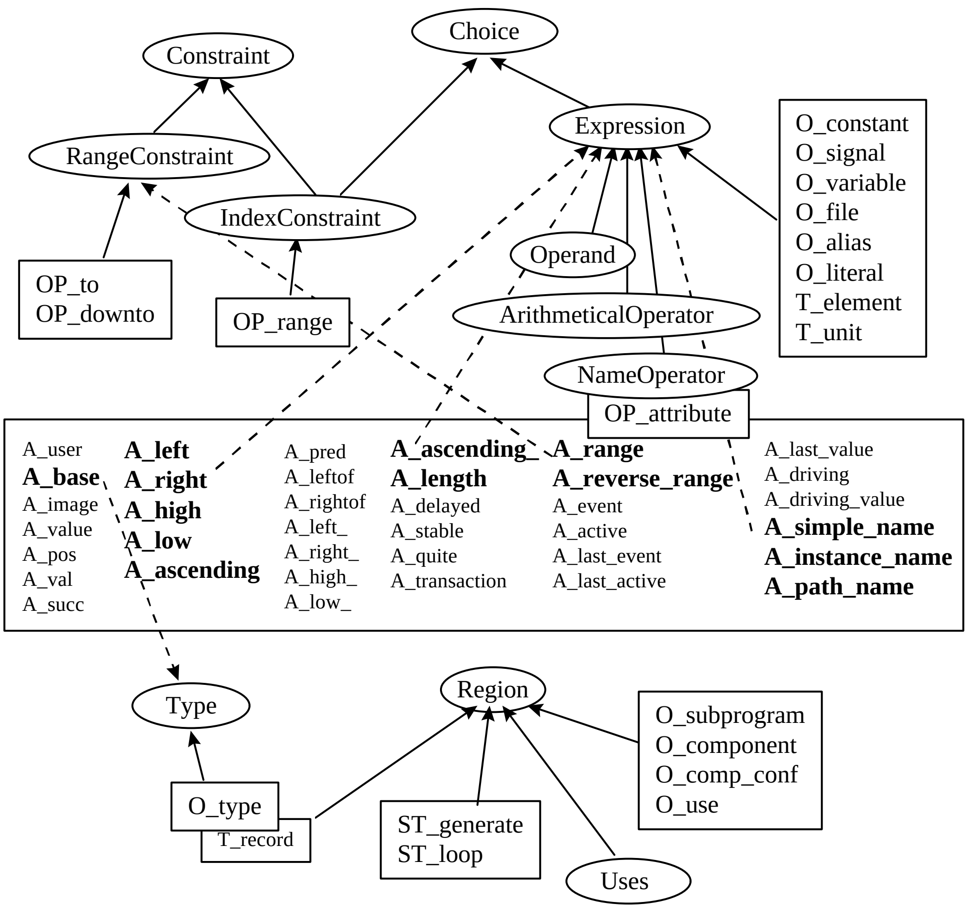


Figure 24. Classifying Basic objects by multiple groups.

- Choice

This group includes the objects corresponding to choice in VHDL. A choice constraint can be a simple expression or a discrete range.

- Constraint

This group includes the objects corresponding to constraint in VHDL. A constraint can be a range constraint or an index constraint.

- RangeConstraint

This group includes the objects corresponding to range constraint in VHDL. An index constraint can be a To/Downto range or an range attribute. The range attributes imply attribute names that result in a range.

- IndexConstraint

This group includes the objects corresponding to index constraint in VHDL. An index constraint is represented by OP_range which describes a discrete range.

- Expression

This group includes the objects corresponding to an expression in VHDL. An expression is composed of operators, operands, and names. The names denote objects, values, and attributes. The only attribute names allowed as expression are ones that result in a value.

- Type

This group includes the objects corresponding to a type in VHDL. All objects included in an expression group have a value and a type. The types imply the declarations, or attribute names that result in a type.

- Region

This group includes the objects corresponding to VHDL constructs that have a declarative part. Only these objects have the "search" member function.

Figure 25, 26 show the groups that are classified as Operator, Statement, Subprogram, and Type objects according to attribute fields only. All defined groups are listed as follows.

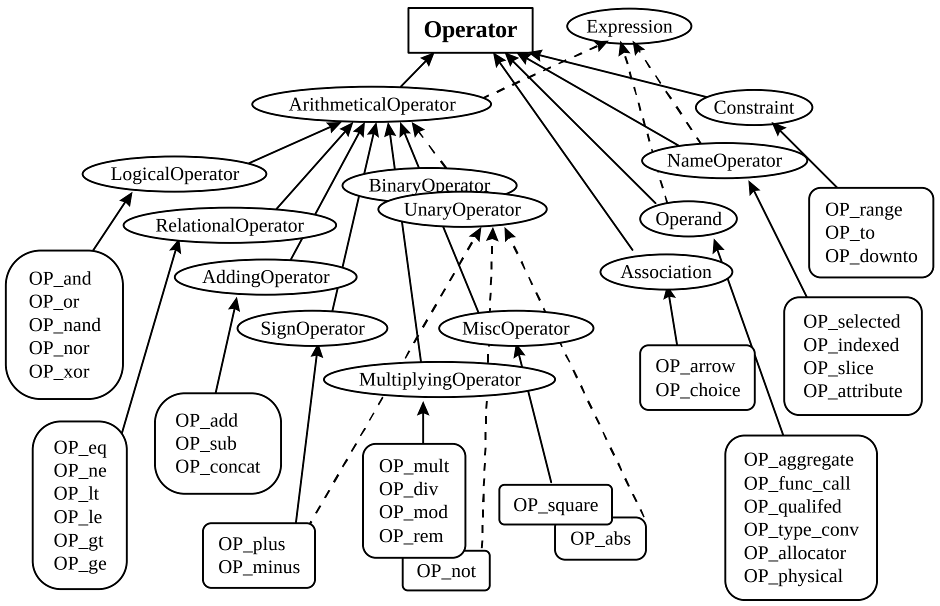


Figure 25. Classifying Operator objects by multiple groups.

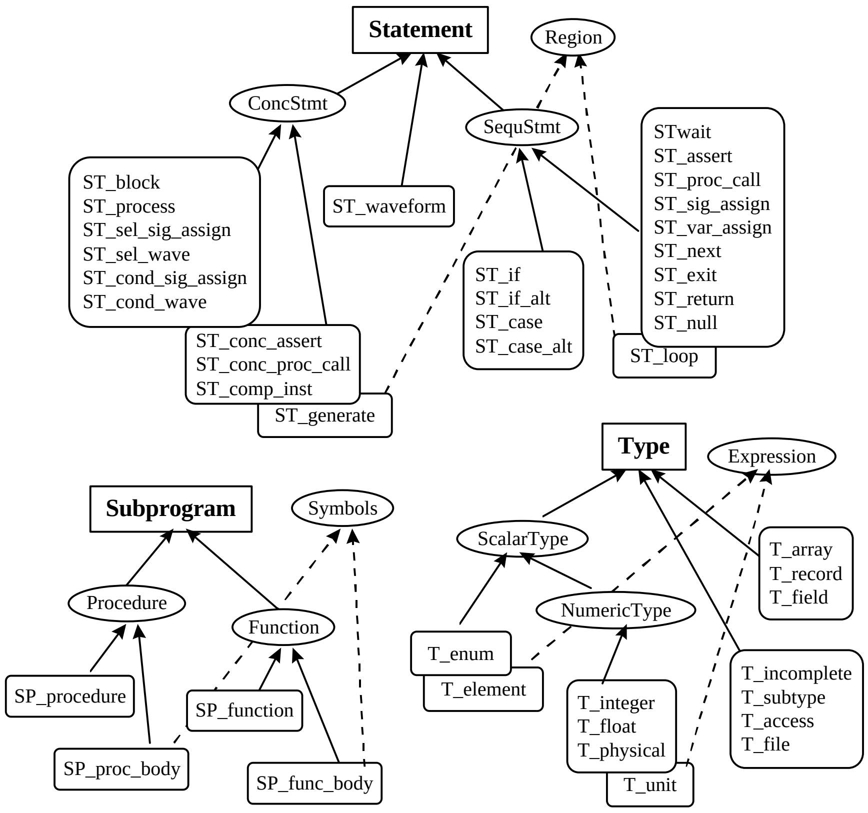


Figure 26. Classifying Statement, Subprogram, and Type objects by multiple groups.

● ArithmeticalOperator

This group includes the objects corresponding to operators in VHDL. The operators are classified according to the precedence level of seven groups: LogicalOperator, RelationalOperator, ShiftOperator, AddingOperator, SignOperator, MultiplyingOperator, MiscellaneousOperator. According to existence of left operand, the operator are classified by two groups: UnaryOperator, BinaryOperator.

● Operand

This group includes the objects corresponding to operands in VHDL. The operand is classified into aggregates, function calls, qualified expression, type conversion, allocators, and physical literal.

● NameOperator

This group includes the object corresponding to name in VHDL. The name is classified into selected name, indexed name, slice name, and attribute name.

● Association

This group includes the objects corresponding to association constructs in VHDL. A OP_arrow describes an association of a generic, a port, or a parameter. A OP_choice describes an element association that appears in an aggregate construct.

- SequStmt

This group includes the object corresponding to sequential statements. Sequential statements are used to define algorithms for the execution of a subprogram or process. They execute in the order in which they appear.

- ConcStmt

This group includes the objects corresponding to concurrent statements. Concurrent statements are used to define interconnected blocks and processes that jointly describe the overall behavior or structure of a design. They execute asynchronously with respect to each other.

- Function

This group includes the objects corresponding to function. These objects are SP_function, ST_func_body.

- Procedure

This group includes the objects corresponding to procedure. These objects are SP_procedure, SP_proc_body.

- ScalarType

This group includes the objects corresponding to scalar types in VHDL. Scalar types consist of enumeration types, integer types, physical types, and floating point types.

- NumericType

This group includes the objects corresponding to numeric types in VHDL. Numeric types consist of integer types, floating point types, and physical types.

6.3 Property getting routines

Each object corresponds to a unique VHDL construct, and also has the properties which the corresponding construct has. For instance, an object has a name, an enclosing region, a specified type, and so on. As another instance, an object belongs to the group of expression, sequential statement, scalar type, and so on. Basic object cannot be classified not only by object kinds, but also by these properties.

All kinds of Basic objects are implemented to classes derived from a same base class named "Basic". These objects are viewed to user just as the same object. In many cases, the user wants to know the kind of a given Basic object and the properties for that object. The user can find these information by digging into the storage fields of a given object. But it is easy to use the "characterize" routines that find any properties from a given object.

- Synopsis of "is" and "get" routines

  int is***Property***(obj);

  oid get***Property***(obj);

  oid obj; **--** pointer of Basic object

Description

a. The "is" routines return 1 if the answer is yes, otherwise return 0.

a. The "get" routines return an object corresponding to the result if it is successful, otherwise return Null pointer.

cf) The above italic string can be replaced into other strings to get other properties.

- Expression

The following routines get the properties of expression construct in VHDL. As a parameter, the objects included in Expression group are only allowed.

oid	getExprIndex(oid obj, int index);
oid	getExprLength(oid obj);
oid	getExprType(oid obj);
oid	getExprValue(oid obj);

- Static

The following routines get the properties of static semantics in VHDL. As a parameter, all kinds of Basic object are allowed.

int	isStaticChoice(gid choices, int flag);
int	isStaticChoice(oid choice, int flag);
int	isStaticExpr(oid obj, int flag);
int	isStaticRange(oid obj, int flag);
int	isStaticName(gid gob, int flag);
int	isStaticName(oid obj, int flag);
int	isStaticType(oid obj, int flag);

- Discrete

The following routines get the properties of discrete semantics in VHDL. As a parameter, all kinds of Basic object are allowed.

int	isDiscreteArray(oid obj);
int	isDiscreteRange(oid obj);
int	isDiscreteType(oid obj);

- Attribute

The following routines get the property of attribute clause. As a parameter, OP_attribute is only allowed.

```
oid         getResultType(oid obj);
oid         getParameterType(oid obj);
int         isFunctionKind(oid obj);
int         isRangeKind(oid obj);
int         isSignalKind(oid obj);
int         isTypeKind(oid obj);
int         isValueKind(oid obj);
```

- Type

The following routines get the properties of type in VHDL. As a parameter, the objects included in Type group or string denoting type name are only allowed.

```
oid         getBaseType(oid obj);
oid         getStandardType(char* nm);
int         isCharacterType(oid obj);
int         isDiscreteType(oid obj);
int         isIncompleteType(oid obj);
int         isNumericType(oid obj);
int         isScalarType(oid obj);
int         isStringType(oid obj);
int         isUnconstrainedType(oid obj);
```

- Miscellaneous

The following routines get the properties of miscellaneous constructs in VHDL. As a parameter, all kinds of Basic objects are allowed.

```
oid         getFormalDesignator(oid);
oid         getActualDesignator(oid);
oid         getRangeBound(oid obj, int flag);
int         isGuardedSignal(oid obj);
int         isGuardedSignal(gid gob);
int         isGuardedTarget(oid obj);
int         isPassiveStatement(oid obj);
int         isResolvedSignal(oid obj);
int         isSignalName(oid obj);
int         isSignalName(gid gob);
int         isSimpleName(oid obj);
int         isVariableName(oid obj);
```

7 VDT Utilities

To use the VDT basic tools and PI library, the following system environment should be set .

- Set the environment variable VDT to the directory location installed VDT.

```
ex) setenv VDT /usr/local/cad/vdt-2.5
```

- Append the directory located executable programs to the path variable.

```
ex) set path = ($path $VDT/bin)
```

- Create the design libraries just by configuring .vdtrc file.

```
ex) Library    IEEE    ./LIB
```

VDT is developed by using C++ language. PI library is an archive file compiled with GNU C++ compiler. VDT application programs should be compiled with PI header file and linked with PI archive file. The following shell command is an example of the way VDT application programs can be made.

```
% g++ program.cc -I $VDT/include -L $VDT/lib -lpi
```

Where "program.cc" is the source filename written by user, and "g++" is GNU C++ compiler.

7.1 VHDL Analyzer

This tool parses a given VHDL file, analyzes each design units contained in the design file, and constructs an intermediate forms for each design unit. Lexical and syntactic checking is accomplished automatically. But semantic checking is accomplished according to the option for each intermediate form. Finally, each intermediate form is stored as a file to the assigned working library.

- Synopsis of program running

```
% van [options] file ...
options:
        -w LIB     : Assign working design library. [default WORK]
        -s         : Do not check semantics. [not default]
        -h         : Print help messages.
```

where "file ... " represents the VHDL file list.

7.2 VHDL Code Generator

This tool generates VHDL code from a given intermediate form corresponding to a design unit. The given design unit is searched in the assigned working design library. The saved intermediate form contains full syntax and semantics of original VHDL code. The generated VHDL code may not have same textual form, but must have the same semantics.

- Synopsis of program running

```
% vgen [options] primary [secondary]
```

options :

```
        -w LIB     : Assign working design library. [default WORK]
        -o file    : Output VHDL code to a file. [default STDOUT]
        -h         : Print help messages.
```

where "primary" represents a primary unit name, "secondary" is optional and represents the secondary unit name associated with the previous primary unit.

7.3 VHDL Static Semantic Checker

This tool checks the static semantics of a given intermediate form corresponding to a design unit. The given design unit is searched in the assigned working design library. The semantic checking is accomplished to each object contained within a given design unit. For each semantic error, a proper message is reported.

- Synopsis of program running

```
% vsem [options] primary [secondary]
```

options:

```
        -w LIB     : Assign working design library. [default WORK]
        -o file    : Report error messages to a file. [default STDOUT]
        -h         : Print help messages.
```

where "primary" represents a primary unit name, "secondary" is optional and represents the secondary unit name associated with the previous primary unit.

7.4 Design Library Browser

This tool browses the design units contained within a given design library. An argument is a logical design library name which is configured by ".vdtrc" files. As for the reported contents, there are design unit type, design unit name, and Basic object count contained in this design unit.

- Synopsis of program running

```
% browse [options] library
```

options :

```
        -h         : Print help messages.
```

where "library" represents a logical design library name.

7.5 Intermediate Form Dumper

This tool dumps the contents of a given intermediate form corresponding to a design unit. An argument is a design unit name which is composed of primary unit name and associated secondary unit name. The given design unit is searched in the assigned working design library. The contents dumping is accomplished for in the case of each object contained within a given design unit.

- Synopsis of program running

% ifdump [options] primary [secondary]

options :

-w LIB	: Assign working design library. [default WORK]
-o file	: Output to a file. [default STDOUT]
-h	: Print help messages.

where "primary" represents a primary unit name, "secondary" is optional and represents the secondary unit name associated with the previous primary unit.

7.6 IF to BLIF Translator

This tool translates from a given intermediate form corresponding to a design entity to BLIF(Berkeley Logic Interchange Format) form. A design entity which is a pair of entity declaration and associated architecture body. The input is a design unit name which is composed of primary unit name and associated secondary unit name. The primary unit must correspond to an entity declaration. The given design unit is searched in the assigned working design library. The translated code represents a finite state machine. So, the given VHDL description must be a description of finite state machine.

- Synopsis of program running

% if2blif [options] primary [secondary]

options :

-w LIB	: Assign working design library. [default WORK]
-o file	: Output to a file. [default STDOUT]
-h	: Print help messages.

where "primary" represents a primary unit name, and "secondary" is optional and represents the secondary unit name associated with the previous primary unit.

7.7 BDNET to VHDL Translator

This tool is a format translator which converts from BDNET form to VHDL structural description. The input is BDNET file which is a net list of logic gates, such as nand, nor, flip flop and so on. In VHDL structural description, the type of net signal can be changed into a given type according to a given option, and the component library for logic gates can be assigned to any design library according to a given option.

- Synopsis of program running

%bd2vhd [options] file

options :

- LIB	: Assign working design library located components. [default WORK]

- type	: Assign the type of net signal. [default BIT]
- file	: Output to a file. [default STDOUT]
-	: Print help messages.

where "file" represents a BDNET format file.

Acknowledgment

The authors would like to acknowledge the contributions of Seung-Pil Shim, Won-Hoeh Song, and Wha-Yong Kim to this work. They helped to test VDT by using validation suite and implement some part of VHDL semantic check routines.

References

[1] The Institute of Electrical and Electronics Engineers Inc., New York, New York, *IEEE Standard VHDL anguage Reference Manual*, *IEEE Std 1076-1987,* 1988.

[2] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design,* Norwell, Kluwer Academic Publishers, 1989.

[3] S. S. Leung and M. A. Shanblatt, *ASIC System Design with VHDL: A Paradigm,* Norwell, Kluwer Academic Publishers, 1989.

[4] J. P. Huber and M. W. Rosneck, *Successful ASIC Design the First Time Through,* New York, Van Nostrand Reinhold, 1991.

[5] B. Harding, "HDLs: A high-powered way to look at complex designs," *Computer Design,* vol. 29, pp. 74-84, March 1990.

[6] Viewlogic Systems Inc., Marlboro, Massachusette, *VHDL User's Guide,* 1989.

[7] Synopsys Inc., Mountain View, California, *Design Compiler Reference Manual*, 1991.

[8] L. F. Saunders, "The IBM VHDL design system," in Proceeding of 24th ACM/IEEE Design Automation Conference, pp. 484-490, June 1987.

[9] E. Marshner, "A VHDL design environment," VLSI Systems Design, September 1988.

[10] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

8 Appendix A. VDT Object Definition

VHDL description is described by VDT objects in an intermediate form representation. Each VDT object corresponds to a VHDL construct. These VDT objects are divided into three kinds, Lib object, Design object, and Basic object. Lib object corresponds to a design library, and Design object corresponds to a design unit. Basic object corresponds to the lest of VHDL constructs.

All data structures and access routines of PI library are implemented by object-oriented programming using C++ language. Basic object is implemented to a class named Basic. Design object is to a class named Design, Lib object is to a class named Lib.

The description on VDT object definition is divided into three parts: syntax part, attribute part, and relationship part. The following shows the consistent format of description.

```
Syntax
    -- syntax of VHDL construct --
...
Attribute
    -- attribute fields --
...
Relationship
    -- relationship fields --
```

The syntax part shows the syntax of the corresponding VHDL construct. The second part enumerates the attribute fields of its data structure. The third part enumerates the relationship fields of its data structure. In syntax part, the words in capital letter represents VHDL reserved word. The words in small letter represents other syntax rules or the meaning of the word itself.

There are two kinds of relationship, 1:1 relationship and 1:m relationship. The 1:1 relationship field can point only one object. The 1:m relationship field can point many objects. The naming convention distinguishes the two names of the 1:1 relationship field and 1:m relationship field. If the name of a relationship field is an identifier with ending 's', then the field is for 1:m relationship. Otherwise, the field is for 1:1 relationship.

For each 1:1 relationship field, there are two access routines that is named "get_***relationship***()" and "set_***relationship***()". The italic string "***relationship***" represents the name of 1:1 relationship field. This name can be replaced into other names to access other 1:1 relationship fields.

For each 1:m relationship field, there are two access routines that is named "add_***relationship***()", "sub_***relationship***()", and "gen_***relationship***()". The italic string "***relationship***" represents the name of 1:m relationship field. This name can be replaced into other names to access other 1:m relationship fields.

For each attribute field, there is one access routine that is named "get_***attribute***()". The italic string "***attribute***" represents the name of attribute field. This name can be replaced into other names to access other attribute fields.

1. Basic object

Each kind of Basic objects is implemented onto a different class. These objects are viewed to user as a same Basic object, because all classes are derived from the same Basic class.

Basic class defines several attribute fields such as, "object_type", "line_number", "id_number", and "flag". All classes derived from Basic class inherit these storage fields and the member functions accessing these fields. In the following definitions of each kind of Basic object, the attribute part contains implicitly these storage fields.

object_type	: field of object_type type for recording kind of Basic object.
id_number	: field of integer type for recording sequence number.
line_number	: field of integer type for recording line number.
flag	: field of void* type to give some freedom to user.

All kinds of Basic object are distinguished and identified by the value of a storage field "object_type" which is of an enumeration type "object_type". Basic object is classified into 26 kinds, and the type named "object_type" is defined as follows.

```
enum object_type    {
        O_invalid            = 0,     O_alias              = 1,     O_arch_body          = 2,
        O_attribute          = 3,     O_attr_spec          = 4,     O_block_conf         = 5,
        O_commnet            = 6;     O_component          = 7,     O_comp_conf          = 8,
        O_configuration      = 9,     O_conf_spec          = 10,    O_constant           = 11,
        O_context            = 12,    O_disc_spec          = 13,    O_entity             = 14,
        O_file               = 15,    O_library            = 16,    O_literal            = 17,
        O_operator           = 18,    O_package            = 19,    O_pack_body          = 20,
        O_signal             = 21,    O_statement          = 22,    O_subprogram         = 23,
        O_type               = 24,    O_use                = 25,    O_variable                    = 26
};
```

1) Alias object

This corresponds to an alias declaration which declares an alternate name for an existing named entity. The name must be a static name that denotes a named entity. The type of aliased object must be the same as the type mark, which may not be a multi-dimensional array.

Syntax

alias_declaration ::= ALIAS identifier : type_mark [constraint] IS name;

Attribute

name	: field of string type for recording an identifier.

Relationship

scope	: indicate an object corresponding to enclosing region. related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_boyd, SP_proc_body
type	: indicate an object corresponding to type mark. related objects => O_type
ranges	: indicate object list corresponding to constraint. related objects => Constraint group
aliased_object	: indicate an object corresponding to name. related objects => O_constant, O_signal, O_variable, Name_operator group

2) Arch_body object

This corresponds to an architecture body which defines the body of a design entity. An architecture body is associated with an entity declaration.

The entity name identifies the name of the entity declaration that defines the interface of this design entity. For a given design entity, both the entity declaration and the associated architecture body must reside in the same library.

A single declarative region is formed by an architecture body with an associated entity declaration. The architecture declarative part contains many declarations. The architecture statement part contains many concurrent statements.

Syntax

```
architecture_body ::=
        [context_clause]
        ARCHITECTURE identifier OF entity_name IS
          architecture_declarative_part
        BEGIN
          architecture_statement_part
        END [architecture_simple_name];
```

Attribute

name	: field of string type for recording an identifier.

Relationship

symbols	: indicate named object list declared in this region.
	related objects => Name group
scope	: indicate an object corresponding to enclosing region.
	related objects => O_library
context	: indicate an object corresponding to context clause.
	related objects => O_context
entity	: indicate an object corresponding to entity name.
	related objects => O_entity
subprograms	: indicate object list corresponding to subprograms.
	related objects => O_subprogram
types	: indicate object list corresponding to type declarations.
	related objects => O_type

constants	: indicate object list corresponding to constant declarations. related objects => O_constant
signals	: indicate object list corresponding to signal declarations. related objects => O_signal
files	: indicate object list corresponding to file declarations. related objects => O_file
aliases	: indicate object list corresponding to alias declarations. related objects => O_alias
attributes	: indicate object list corresponding to attribute declarations. related objects => O_attribute
attr_specs	: indicate object list corresponding to attribute specifications. related objects => O_attr_spec
components	: indicate object list corresponding to component declarations. related objects => O_component
conf_specs	: indicate object list corresponding to configuration specifications. related objects => O_conf_spec
disc_specs	: indicate object list corresponding to disconnection specifications. related objects => O_disc_spec
uses	: indicate object list corresponding to use clause. related objects => O_use
conc_stmts	: indicate object list corresponding to concurrent statements. related objects => ConcStmt group

3) Attribute object
This corresponds to an attribute declaration which creates a user defined attribute. There are two categories of attributes: predefined attributes and user defined attributes. The type mark must denote a subtype that is neither an access type nor a file type.

Syntax

```
attribute_declaration ::= ATTRIBUTE identifier : type_mark;
```

Attribute

name	: field of string type for recording an identifier.

Relationship

scope	: indicate an object corresponding to enclosing region. related objects => O_entity, O_arch_body, O_package, ST_block, ST_process, SP_func_body, SP_proc_body
type	: indicate an object corresponding to type mark. related objects => O_type

4) Attr_spec object

This corresponds to an attribute specification which associates a user defined attribute with one or more named entities. The attribute name must denote a user defined attribute, and the entity name list identifies those named entities. The type of the specified expression must be the same as the type mark in the corresponding attribute declaration.

Syntax

```
        attribute_specification ::=
                ATTRIBUTE attribute_name OF entity_name_list : entity_class IS expression;

        entity_name_list ::=
                entity_designator, entity_designator
                | OTHERS
                | ALL
```

Attribute

Relationship

attribute : indicate an object corresponding to attribute name.
related objects => O_attribute

attr_objects: indicate object list corresponding to entity name list.
related objects => Name group

expression : indicate an object corresponding to expression.
related objects => Expression group

5) Block_conf object

This corresponds to a block configuration which defines the configuration of a block. The block denoted by block specification may be either an architecture body, a block statement, or a generate statement.

If a block configuration appears immediately within a configuration declaration, then the block specification of that block configuration must be an architecture name.

If a block configuration appears immediately within a component configuration, then the corresponding components must be fully bound, and the block specification of that block configuration must be an architecture name.

If a block configuration appears immediately within another block configuration, then the block specification of the contained block configuration must be a block statement label or generate statement label.

It is an error if, in a given block configuration, more than one configuration item is defined for the same block or component instance.

Syntax

```
        block_configuration ::=
                FOR block_specification
                  use_clause
                  block_configuration | component_configuration
                END FOR;

        block_specification ::=
                architecture_name
                | block_statement_label
                | generate_statement_label[(index_specification)]

        index_specification ::=
                discrete_range | static_expression
```

Attribute

Relationship

scope	: indicate an object corresponding to enclosing region.
	related objects => O_configuration, O_block_conf, O_comp_conf

block_spec : indicate an object corresponding to block specification.

related objects => Name group

index_spec : indicate an object corresponding to index specification.

related objects => Constraint group, Expression group

uses : indicate object list corresponding to use clauses.

related objects => O_use

block_confs: indicate object list corresponding to block configurations.

related objects => O_block_conf

comp_confs : indicate object list corresponding to component configurations.

related objects => O_comp_conf

6) Comment object

This object corresponds to a VHDL comment which starts with "--" text and extends up to the end of the line. VHDL comments starting with "--vdt" text are only captured by this object, and other comments are abandoned.

Syntax

commnet ::= **--**vdt text_of_comment ...

Attribute

string	: field of string type for recording text of comment.

Relationship

7) Component object

This corresponds to a component declaration which declares a virtual design entity interface. This virtual interface may be used in a component instantiation statement. A component configuration or a configuration specification can be used to associate a component instance with a design entity that resides in a design library.

Syntax

```
component_declaration ::=
        COMPONENT identifier
          [GENERIC (local_generic_list);]
          [PORT (local_port_list);]
        END COMPONENT [component_simple_name];
```

Attribute

name	: field of string type for recording an identifier.

Relationship

scope	: indicate an object corresponding to enclosing region.
	related objects => O_arch_body, O_package, ST_block
generics	: indicate object list corresponding to local generic list.
	related objects => O_constant
ports	: indicate object list corresponding to local port list.
	related objects => O_signal

8) Comp_conf object

This corresponds to a component configuration which defines the configuration of one or more component instances in a corresponding block. The instantiation list identifies the component instances to which this component configuration applies.

A component configuration can appear only within a block configuration. A component configuration that appears immediately within a given block configuration applies to component instances that appear immediately within the corresponding block. It is an error if two component configuration apply to the same component instance.

Syntax

```
    component_configuration ::=
        FOR instantiation_list : component_name  [USE entity_aspect
          [GENERIC MAP (generic_association_list)]
          [PORT MAP (port_association_list)] ;]
          [block_configuration]
        END FOR;

    entity_aspect ::=
        ENTITY entity_name[(architecture_identifier)]
        | CONFIGURATION configuration_name
        | OPEN
```

Attribute

Relationship

scope	: indicate an object corresponding to enclosing region. related objects => O_block_conf
instances	: indicate object list corresponding to instantiation list. related objects => ST_comp_inst
component	: indicate an object corresponding to component name. related objects => O_component
entity_aspect	: indicate an object corresponding to entity aspect. related objects => O_configuration, O_entity, O_arch_body
generic_maps	: indicate object list corresponding to generic association list. related objects => OP_arrow
port_maps	: indicate object list corresponding to port association list.

	related objects => OP_arrow
block_conf	: indicate an object corresponding to block configuration.
	related objects => O_block_conf

9) Configuration object

This corresponds to a configuration declaration which performs the binding of component instances to design entities. A configuration of the top level entity in a design hierarchy specifies the bindings necessary to identify the hierarchy.

The entity name identifies the name of the entity declaration that defines the design entity at the apex of the design hierarchy. For a configuration of a given design entity, both the configuration declaration and the corresponding entity declaration must reside in the same library.

Syntax

```
configuration_declaration ::=
        [context_clause]
        CONFIGURATION identifier OF entity_name IS
          configuration_declarative_part
          block_configuration
        END [configuration_simple_name];
```

Attribute

name	: field of string type for recording an identifier.

Relationship

symbols	: indicate named object list declared in this region.
	related objects => Name group
scope	: indicate an object corresponding to enclosing region.
	related objects => O_library
context	: indicate an object corresponding to context clause.
	related objects => O_context
entity	: indicate an object corresponding to entity name.
	related objects => O_entity
attr_specs	: indicate object list corresponding to attribute specifications.
	related objects => O_attr_spec
uses	: indicate object list corresponding to use clauses.
	related objects => O_use
block_conf	: indicate an object corresponding to block configuration.
	related objects => O_block_conf

10) Conf_spec object

This corresponds to a configuration specification which associates binding information with component instances. The instantiation list denotes a list of instantiation labels that represent component instances of a given component name. The entity aspect identifies a particular design entity to be associated with component instances.

Syntax

```
configuration_specification ::=
        FOR instantiation_list : component_name USE entity_aspect
          [GENERIC MAP (generic_association_list)]
          [PORT MAP (port_association_list)];

entity_aspect ::=
        ENTITY entity_name[(architecture_identifier)]
        | CONFIGURATION configuration_name
        | OPEN
```

Attribute

Relationship

instances	: indicate object list corresponding to instantiation list.
	related objects => L_string
component	: indicate an object corresponding to component name.
	related objects => O_component
entity_aspect	: indicate an object corresponding to entity aspect.
	related objects => O_configuration, O_entity, O_arch_body
generic_maps	: indicate object list corresponding to generic association list.
	related objects => OP_arrow
port_maps	: indicate object list corresponding to port association list.
	related objects => OP_arrow

11) Constant object

This corresponds to a constant declaration, an interface constant declaration, and a parameter specification. A constant declaration and an interface constant declaration declares a constant and an interface constant of the specified type, respectively. The identifier list denotes the names of [interface] constants. For each identifier on this list, one object is created. The type of [interface] constant must not be a file type or an access type.

If a constant declaration contains an expression, the expression specifies the value of the constant. If an expression is not present, the declaration declares a deferred constant that can only appear in a package declaration. The corresponding full constant declaration must appear in the body of the package.

An interface constant declaration can appear as generics of a design entity, a component, or a block, or as constant parameters of subprograms. If an interface declaration contains an expression, the expression specifies the default value of the interface object.

A parameter specification declares the loop parameter for a loop statement with a FOR iteration scheme, or declares the generate parameter for a generate statement with a FOR generation scheme. The identifier in parameter specification implies the name of a declared parameter. A loop parameter and a generate parameter is a constant within the corresponding statement.

The mode implies the direction of information flow through the port or parameter. To describe the mode of a constant, the enumeration type "mode_kind" is defined as follows:

enum mode_kind { M_invalid, M_null, M_in, M_out, M_inout, M_buffer, M_linkage };

A mode_kind field of this object identifies one of the three kinds of constants. The objects having M_in imply interface constants. If an object having M_null has a scope that is a loop statement or generate statement, then this object implies a loop parameter or a generate parameter, otherwise it implies a constant.

Syntax

```
constant_declaration ::=
        CONSTANT identifier_list : type_mark [constraint] [:= expression];

interface_constant_declaration ::=
        [CONSTANT] identifier_list : [IN] type_mark [constraint] [:= static_expression];

parameter_specification ::=
        FOR identifier IN discrete_range
```

Attribute

name	: field of string type for recording an identifier.
mode_kind	: field of mode_kind type for recording the kind of mode.

Relationship

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_package, O_pack_body,
	O_subprogram, ST_block, ST_loop, ST_generate
type	: indicate an object corresponding to type mark.
	related objects => O_type
ranges	: indicate object list corresponding to constraint.
	related objects => Constraint group
expression	: indicate an object corresponding to expression.
	related objects => Expression group

12) Context object

This corresponds to a context clause which provides the initial name environment in which a design unit is analyzed.

A library clause defines library logical names that may be referenced in the design unit. A use clause makes certain declarations directly visible within the design unit.

Syntax

context_clause ::= library_clause | use_clause

Attribute

Relationship

libraries	: indicate object list corresponding to library clauses.
	related objects => O_library
uses	: indicate object list corresponding to use clauses.
	related objects => O_use

13) Disc_spec object

This corresponds to a disconnection specification which defines the time delay to be used in the implicit disconnection of drivers of a guarded signal within a guarded signal assignment.

Each signal name in a guarded signal list must be a locally static name that denotes a guard signal. If the guarded signal is a declared signal or a slice thereof, the type of the signal must be the same as the type mark. Each signal must be declared in the declarative part enclosing the disconnection specification.

The time expression in a disconnection specification must be static and must be evaluated to a non-negative value. It is an error if more than one disconnection specification applies to drivers of the same signal.

Syntax

```
        disconnection_specification ::=
                DISCONNECT guarded_signal_list : type_mark AFTER time_expression;
```

Attribute

Relationship

guarded_signals	: indicate object list corresponding to guarded signal list.
	related objects => O_signal
type	: indicate an object corresponding to type mark.
	related objects => O_type
time_expr	: indicate an object corresponding to time expression.
	related objects => Expression group

14) Entity object

This corresponds to an entity declaration which defines the interface between a given design entity and the environment. A given entity declaration may be shared by many design entities, each of which has a different architecture. The entity declarative part of a given entity declaration declares items that are common to all design entities. The entity statement part, if present, consists of concurrent statements that are present in each such design entity.

A single declarative region is formed by an entity declaration with an associated architecture body. Names declared in the entity declarative part are visible within the bodies of corresponding design entities, as well as within certain portions of a corresponding configuration declaration.

Syntax

```
entity_declaration ::=
        [context_clause]
        ENTITY identifier IS
          [GENERIC (generic_list);]
          [PORT (port_list);]
          entity_declarative_part
        [BEGIN
          entity_statement_part]
        END [entity_simple_name];
```

Attribute

name	: field of string type for recording an identifier.

Relationship

symbols	: indicate named object list declared in this region.
	related objects => Name group
scope	: indicate an object corresponding to enclosing region.
	related objects => O_library
context	: indicate an object corresponding to context clause.
	related objects => O_context
generics	: indicate object list corresponding to generic list.
	related objects => O_constant
ports	: indicate object list corresponding to port list.
	related objects => O_signal
subprograms	: indicate object list corresponding to subprograms.

	related objects => O_subprogram
types	: indicate object list corresponding to type declarations.
	related objects => O_type
constants	: indicate object list corresponding to constant declarations.
	related objects => O_constant
signals	: indicate object list corresponding to signal declarations.
	related objects => O_signal
files	: indicate object list corresponding to file declarations.
	related objects => O_file
aliases	: indicate object list corresponding to alias declarations.
	related objects => O_alias
attributes	: indicate object list corresponding to attribute declarations.
	related objects => O_attribute
attr_specs	: indicate object list corresponding to attribute specifications.
	related objects => O_attr_spec
disc_specs	: indicate object list corresponding to disconnection specifications.
	related objects => O_disc_spec
uses	: indicate object list corresponding to use clauses.
	related objects => O_use
conc_stmts	: indicate object list corresponding to concurrent statements.
	related objects => ConcStmt group

15) File object

This corresponds to a file declaration which declares a file of a specified type. The type mark must be a file type. The file logical name must be a string expression. The file logical name identifies an external file in the host file system.

The mode implies the direction of information flow through the port or parameter. The mode of a file can be M_in or M_out. If not specified, the mode is M_in by default. To describe the mode of a file, the enumeration type "mode_kind" is defined as follows.

enum mode_kind { M_invalid, M_null, M_in, M_out, M_inout, M_buffer, M_linkage };

If a formal parameter of subprogram is a file object, it must be associated with an actual that also is a file object.

Syntax

```
file_declaration ::= FILE identifier : type_mark IS [mode] file_logical_name;
```

Attribute

name	: field of string type for recording an identifier.
mode_kind	: field of mode_kind type for recording kind of mode.

Relationship

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_block, O_package,
	O_pack_body, O_process, SP_proc_body, SP_func_body
type	: indicate an object corresponding to type mark.
	related objects => T_file
expression	: indicate an object corresponding to file logical name.
	related objects => L_string

16) Library object

This corresponds to a library clause which defines a logical name for a design library in the host environment. A design library is a storage facility for the analyzed design units.

A library clause appears as part of a context clause at the beginning of a design unit. The scope of a library clause starts immediately after the library clause, and extends to the end of the declarative region associated with the design unit.

There are two classes of design libraries: working libraries and resource libraries. A working library is the library into which the analyzed design unit is placed. A resource library is a library containing design units that are referenced within the design unit being analyzed. Only one library may be the working library during the analysis of any given design unit.

A secondary unit corresponding to a given primary unit may be put into the same design library in which the primary unit is placed.

Syntax

```
library_clause ::= LIBRARY identifier, identifier;
```

Attribute

name	: field of string type for recording an identifier

Relationship

lib	: indicate a Lib object corresponding to a design library.
	related objects => Lib object

17) Literal object

This corresponds to a literal. A literal is either an abstract literal, a physical literal, an enumeration literal, a string literal, a bit string literal, or literal NULL.

Abstract literals are literals of universal integer type or universal real type. String and bit string literals are representations of one dimensional arrays of characters. The literal NULL represents the null access value for any access type. The kind of a Literal object is identified by enumeration type "literal_kind" defined as follows.

enum literal_kind { L_invalid, L_integer, L_float, L_string, L_bit_string, L_null };

Enumeration literals are literals of enumeration types. A enumeration literal is described by a Type object whose kind is "T_enum". Physical literals are literals of physical types. A physical literal is described by Operator objects whose kind is "OP_physical".

Syntax

```
literal ::= abstract_literal
          | string_literal
          | bit_string_literal
          | NULL
```

Attribute

literal_kind: field of literal_kind type for recording the kind of literal.
string : field of string type for recording literal text.

-- only for abstract_literal --

abstract_value: field of double type for recording an evaluated value.

-- only for string_literal, bit_string_literal --

string_value : field of string type for recording an evaluated value.

Relationship

18) Operator object

This corresponds to an operator, an operand, or other VHDL construct. This object describes VHDL constructs that belong to neither operators nor operands. As VHDL constructs corresponding to these objects, there are association element, element association, index constraint, and range constraint. The kind of Operator object is identified by enumeration type "operator_kind" defined as follows.

```
enum operator_kind {
        OP_invalid, // arithmetical operator
        OP_and, OP_or, OP_nand, OP_nor, OP_xnor, OP_xor,
        OP_eq, OP_ne, OP_lt, OP_le, OP_gt, OP_ge,
        OP_sll, OP_srl, OP_sla, OP_sra, OP_rol, OP_ror
        OP_add, OP_sub, OP_concat, OP_plus, OP_minus,
        OP_mul, OP_div, OP_mod, OP_rem, OP_exp, OP_abs, OP_not,
        OP_aggregate, OP_func_call, OP_qualified,   // operand
        OP_type_conv, OP_allocator, OP_physical,
        OP_selected, OP_indexed, OP_slice, OP_attribute,
        OP_arrow, OP_choice, // association
        OP_range, OP_to, OP_downto // constraint
};
```

Literals which belong to operands are described by the Literal object. The object corresponding to an attribute name is further classified by enumeration type "attr_kind" defined as follows.

```
enum attr_kind {
        A_invalid, A_user, A_base,
        A_left, A_right, A_high, A_low, A_ascending, A_image,
        A_value, A_pos, A_val, A_succ, A_pred, A_leftof, A_rightof,
        A_left_, A_right_, A_high_, A_low_, A_ascending_,
        A_range, A_reverse_range, A_length,  A_delayed, A_stable,
        A_quiet, A_transaction, A_event, A_active, A_last_evant,
        A_last_active, A_last_value, A_driving, A_driving_value,
        A_simple_name, A_instance_name, A_path_name
};
```

An attribute name denotes a value, function, type, range, signal, or constant associated with a named entity. If the attribute simple name denotes a user defined attribute, the corresponding Operator object is an OP_attribute that has A_user as attribute kind. Other attribute kinds are for representing any predefined attributes.

Syntax

```
-- arithmetical_operator --
logical_operator      ::= AND | OR | NAND | NOR | XOR | XNOR
relational_operator ::=    =    |    /=    |    <    |    <=    |    >    |    >=
shift_operator         ::= SLL | SRL | SLA | SRA | ROL | ROR
adding_operator      ::=    +    |    -    |    &
sign                        ::=    +    |    -
multiplying_operator    ::=    *    |    /    |    MOD | REM
miscellaneous_operator ::=    **    |    ABS | NOT

-- operand --
physical_literal ::= [abstract_literal] unit_name
aggregate ::= (element_association_list)
function_call ::= function_name[(association_element_list)]
qualified_expression ::= type_mark'(expression) | type_mark'aggregate
type_conversion ::= type_mark(expression)
allocator ::= NEW type_mark'(expression) | NEW type_mark'aggregate
                  | NEW type_mark [constraint]
name
literal

-- name_operator --
selected_name ::= prefix.suffix
indexed_name ::= prefix(expression_list)
slice_name ::= prefix(discrete_range)
attribute_name ::= prefix'attribute_simple_name[(static_expression)]

-- association --
association_element ::= [formal_part =>] actual_part
element_association ::= [choices =>] expression

-- constraint --
index_constraint ::= (discrete_range, discrete_range)
range_constraint ::= RANGE range
discrete_range ::= type_mark [RANGE range] | range
range ::= prefix'attribute_simple_name[(static_expression)]
                | simple_expression TO/DOWNTO simple_expression
```

Attribute

operator_kind: field of operator_kind type for recording kind of operator.

-- only for attribute_name --

attr_kind : field of attr_kind type for recording kind of attribute name.

Relationship

-- only for arithmetical_operator --

function : indicate an object corresponding to predefined operator.
related objects => SP_function, SP_func_body

left : indicate an object corresponding to left operand.
related objects => Expression group

right : indicate an object corresponding to right operand.
related objects => Expression group

-- only for aggregate --

elements : indicate object list corresponding to element association list.
related objects => OP_choice

-- only for function_call --

function : indicate an object corresponding to function name.
related objects => SP_function, SP_func_body

parameter_maps: indicate object list corresponding to parameter association list.
related objects => OP_arrow, Expression group

-- only for qualified_expression --

type : indicate an object corresponding to type mark.
related objects => O_type

expression : indicate an object corresponding to expression.
related objects => Expression group

-- only for type_conversion --

type : indicate an object corresponding to type mark.
related objects => O_type

expression : indicate an object corresponding to expression.
related objects => Expression group

-- only for allocator --

type	: indicate an object corresponding to type mark.
	related objects => O_type
expression	: indicate an object corresponding to expression.
	related objects => Expression group
indexs	: indicate object list corresponding to constraint.
	related objects => Constraint group

-- only for physical_literal --

abstract	: indicate an object corresponding to abstract literal.
	related objects => L_integer, L_float
unit	: indicate an object corresponding to unit name.
	related objects => T_unit

-- only for selected_name --

prefix	: indicate an object corresponding to prefix.
	related objects => Name group, Name_operator group, OP_func_call
suffix	: indicate an object corresponding to suffix.
	related objects => Name group, L_character, L_string

-- only for indexed_name --

prefix	: indicate an object corresponding to prefix.
	related objects => Name group, Name_operator group, OP_func_call
indexes	: indicate object list corresponding to expression list.
	related objects => Expression group

-- only for slice_name --

prefix	: indicate an object corresponding to prefix.
	related objects => Name group, Name_operator group, OP_func_call
range	: indicate an object corresponding to discrete range.
	related objects => OP_range

-- only for attribute_name --

prefix	: indicate an object corresponding to prefix.
	related objects => Name group, Name_operator group, OP_func_call
attribute	: indicate an object corresponding to attribute simple name.
	related objects => O_attribute

expression	: indicate an object corresponding to static expression.
	related objects => Expression group

-- only for association_element --

formal	: indicate an object corresponding to the formal part.
	related objects => O_constant, O_signal, O_variable, OP_func_call
actual	: indicate an object corresponding to the actual part.
	related objects => Expression group

-- only for element_association --

choices	: indicate object list corresponding to choices.
	related objects => Constraint group, Expression group
expression	: indicate an object corresponding to expression.
	related objects => Expression group

-- only for discrete_range --

type	: indicate an object corresponding to type mark.
	related objects => O_type
range	: indicate an object corresponding to range.
	related objects => Constraint group

-- only for range --

left	: indicate an object corresponding to the left bound.
	related objects => Expression group
right	: indicate an object corresponding to the right bound.
	related objects => Expression group

19) Package object

This corresponds to a package declaration which defines the visible contents of a package. Packages provide a means of defining resources in a way that allows different design units to share the same declarations.

Items declared immediately within a package declaration become visible by selection within a given design unit wherever the name of that package is visible in the given unit. Such items may also be made directly visible by an appropriate use clause.

Syntax

```
package_declaration ::=
        [context_clause]
        PACKAGE identifier IS
          package_declarative_part
        END [package_simple_name];
```

Attribute

name	: field of string type for recording an identifier.

Relationship

symbols	: indicate named object list declared in this region. related objects => Name group
scope	: indicate an object corresponding to enclosing region. related objects => O_library
context	: indicate an object corresponding to context clause. related objects => O_context
subprograms	: indicate object list corresponding to subprogram. related objects => O_subprogram
types	: indicate object list corresponding to type declarations. related objects => O_type
constants	: indicate object list corresponding to constant declarations. related objects => O_constant
signals	: indicate object list corresponding to signal declarations. related objects => O_signal
files	: indicate object list corresponding to file declarations. related objects => O_file
aliases	: indicate object list corresponding to alias declarations. related objects => O_alias

components	: indicate object list corresponding to component declarations.
	related objects => O_component
attributes	: indicate object list corresponding to attribute declarations.
	related objects => O_attribute
attr_specs	: indicate object list corresponding to attribute specifications.
	related objects => O_attr_spec
disc_specs	: indicate object list corresponding to disconnection specifications.
	related objects => O_disc_spec
uses	: indicate object list corresponding to use clauses.
	related objects => O_use

20) Pack_body object

This corresponds to a package body which defines the bodies of subprograms and the values of deferred constants declared in the associated package declaration.

If a given package declaration contains a deferred constant declaration, then a constant declaration with the same identifier must appear as a declarative item in the corresponding package body.

In addition to subprogram body and constant declarative items, a package body may contain other certain declarative items to facilitate the definition of subprogram bodies. Items declared in package body cannot be made visible outside the body.

Syntax

```
package_body ::=
        [context_clause]
        PACKAGE BODY identifier IS
          package_body_declarative_part
        END [package_simple_name];
```

Attribute

name	: field of string type for recording an identifier.

Relationship

symbols	: indicate named object list declared in this region.
	related objects => Name group
scope	: indicate an object corresponding to enclosing region.
	related objects => O_library
context	: indicate an object corresponding to context clause.
	related objects => O_context
package	: indicate an object corresponding to associated package declaration.
	related objects => O_package
subprograms	: indicate object list corresponding to subprograms.
	related objects => O_subprogram
types	: indicate object list corresponding to type declarations.
	related objects => O_type
constants	: indicate object list corresponding to constant declarations.
	related objects => O_constant
files	: indicate object list corresponding to file declarations.

	related objects => O_file
aliases	: indicate object list corresponding to alias declarations.
	related objects => O_alias
uses	: indicate object list corresponding to use clauses.
	related objects => O_use

21) Signal object

This corresponds to a signal declaration or an interface signal declaration which respectively declares a signal or an interface signal of the specified type. The identifier list denotes the names of [interface] signals. For each identifier on this list, one object is created. The type of [interface] signal must not be of a file type or an access type.

An interface signal declaration can appear as ports of a design entity, a component, or a block, or as signal parameters of subprograms.

If a resolution function appears in a declaration, then the declared signals are called resolved signals. If a signal kind appears in a declaration, then the declared signals are guarded signals of that kind. To identify the signal kind, an enumeration type "signal_kind" is defined as follows.

enum signal_kind { S_invalid, S_null, S_register, S_bus };

For a signal that does not have a signal kind, the corresponding object has the signal kind "S_null". It is also an error if a guarded signal of a scalar type is neither a resolved signal nor a subelement of a resolved signal.

The mode implies the direction of information flow through the port or parameter. To describe the mode of a signal, the enumeration type "mode_kind" is defined as follows.

enum mode_kind { M_invalid, M_null, M_in, M_out, M_inout, M_buffer, M_linkage };

A mode_kind field of this object identifies one of two kind signals. The objects having M_null imply signals. The objects having other mode kinds correspond to interface signals.

Syntax

```
signal_declaration ::=
        SIGNAL identifier_list : [resolution_function] type_mark [constraint] [signal_kind] [:=
        expression];

interface_signal_declaration ::=
        [SIGNAL] identifier_list : [mode] [resolution_function] type_mark [constraint] [signal_kind]
        [:= static_expression]
```

Attribute

name	: field of string type for recording an identifier.
mode_kind	: field of mode_kind type for recording the kind of mode.
signal_kind	: field of signal_kind type for recording the kind of signal.

Relationship

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_block, O_package
res_func	: indicate an object corresponding to resolution function.
	related objects => SP_function, SP_func_body
type	: indicate an object corresponding to type mark.
	related objects => O_type
ranges	: indicate object list corresponding to constraint.
	related objects => Constraint group
expression	: indicate an object corresponding to initial expression.
	related objects => Expression group

22) Statement object

This corresponds to a concurrent statement or a sequential statement. There are many kinds of statements. These statements are identified by statement kind that is an enumeration type defined as follows.

```
enum stmt_kind {
        ST_invalid, ST_waveform
        // concurrent_statements
        ST_block, ST_process, ST_conc_proc_call, ST_conc_assert,
        ST_cond_sig_assign, ST_cond_wave, ST_sel_sig_assign,
        ST_sel_wave, ST_comp_inst, ST_generate,
        // sequential_statements
        ST_wait, ST_assert, ST_sig_assign, ST_var_assign,
        ST_proc_call, ST_if, ST_if_alt, ST_case, ST_case_alt,
        ST_loop, ST_next, ST_exit, ST_return, ST_null,
};
```

The concurrent statements can appear in entity declaration, architecture body, and block statement. The sequential statements can appear in process statement and subprogram body.

The signal assignment statement may contain one option of a delay mechanism. The concurrent signal assignment statement may contain one or both of the two options GUARDED and a delay mechanism. These options are described by an enumeration type "assign_option" defined as follows.

```
enum assign_option { AO_invalid, AO_inertial, AO_transport,
AO_guarded_inertial, AO_guarded_transport };
```

The option GUARDED specifies that the signal assignment statement is executed when a signal GUARD changes. The signal GUARD may be one of the implicitly declared GUARD signals associated with block statements that have guard expressions.

Syntax

```
-- concurrent_statement --
block_statement ::=
        block_label: BLOCK[(guard_expression)]
          [GENERIC (generic_list); [GENERIC MAP (generic_association_list);]]
          [PORT (port_list); [PORT MAP (port_association_list);]]
          block_declarative_part
        BEGIN
          block_statement_part
        END BLOCK [block_label];
process_statement ::=
        [process_label :] PROCESS [(sensitivity_list)]
          process_declarative_part
        BEGIN
          process_statement_part
        END PROCESS [process_label];
concurrent_assertion_statement ::=
        [label:] ASSERT condition [REPORT expression] [SEVERITY expression];
concurrent_procedure_call ::=
        [label:] procedure_name[(parameter_association_list)];
conditional_signal_assignment_statement ::=
        [label:] target <= options conditional_waveforms;
conditional_waveforms ::=
        waveform_list WHEN condition ELSE waveform_list
selected_signal_assignment_statement ::=
        [label:] WITH expression SELECT target <= options selected_waveforms;
selected_waveforms ::=
        waveform_list WHEN choices, waveform_list WHEN choices
component_instantiation_statement ::=
        label : component_name [GENERIC MAP (generic_association_list)]
        [PORT MAP (port_association_list)];
generate_statement ::=
        label : generation_scheme GENERATE
        concurrent_statement END GENERATE [label];

-- sequential_statement --
wait_statement ::=
        WAIT [ON sensitivity_list] [UNTIL condition] [FOR time_expression];
assertion_statement ::=
        ASSERT condition [REPORT expression] [SEVERITY expression];
procedure_call_statement ::=
        procedure_name[(parameter_association_list)];
signal_assignment_statement ::=
        target <= [TRANSPORT] waveform_list;
waveform ::=
        expression [AFTER time_expression]
variable_assignment_statement ::=
        target := expression;
```

```
case_statement ::=
        CASE expression IS case_statement_alternative
        case_statement_alternative END CASE;
case_statement_alternative ::=
        WHEN choices => sequential_statements
if_statement ::=
        IF    if_statement_alternative    ELSEIF    if_statement_alternative    [ELSE
        if_statement_alternative]

if_statement_alternative ::=
        [condition THEN] sequential_statements
loop_statement ::=
        [loop_label :] iteration_scheme LOOP
        sequential_statements END LOOP [loop_label];
next_statement ::= NEXT [loop_label] [WHEN condition];
exit_statement ::= EXIT [loop_label] [WHEN condition];
return_statement ::= RETURN [expression];
null_statement ::= NULL;
```

Attribute

name	: field of string type for recording a label.
stmt_kind	: field of stmt_kind type for recording kind of statement.

-- only for signal_assignment_statement --

assign_option	: field of assign_option type for recording option of signal assignment.

Relationship

-- only for block_statement --

symbols	: indicate named object list declared in this region.
	related object => Name group
scope	: indicate an object corresponding to enclosing region.
	related objects => O_arch_body, O_block
condition	: indicate an object corresponding to guard expression.
	related objects => Expression group
generics	: indicate object list corresponding to generic list.
	related objects => O_constant
generic_maps	: indicate object list corresponding to generic association list.

	related objects => OP_arrow
ports	: indicate object list corresponding to port list.
	related objects => O_signal
port_maps	: indicate object list corresponding to port association list.
	related objects => OP_arrow
subprograms	: indicate object list corresponding to subprograms.
	related objects => O_subprogram
types	: indicate object list corresponding to type declarations.
	related objects => O_type
constants	: indicate object list corresponding to constant declarations.
	related objects => O_constant
signals	: indicate object list corresponding to signal declarations.
	related objects => O_signal
files	: indicate object list corresponding to file declarations.
	related objects => O_variable
aliases	: indicate object list corresponding to alias declarations.
	related objects => O_alias
attributes	: indicate object list corresponding to attribute declarations.
	related objects => O_attribute
attr_specs	: indicate object list corresponding to attribute specifications.
	related objects => O_attr_spec
components	: indicate object list corresponding to component declarations.
	related objects => O_component
conf_specs	: indicate object list corresponding to configuration specifications.
	related objects => O_conf_spec
disc_specs	: indicate object list corresponding to disconnection specifications.
	related objects => O_disc_spec
uses	: indicate object list corresponding to use clauses.
	related objects => O_use
conc_stmts	: indicate object list corresponding to concurrent statements.
	related objects => ConcStmt group

-- only for process_statement --

symbols	: indicates named object list declared in this region.
	related objects => Name group
scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, ST_block, ST_generate

sensitivities	: indicate object list corresponding to sensitivity list.
	related objects => O_signal, Name_operator group
subprograms	: indicate object list corresponding to subprograms.
	related objects => O_subprogram
types	: indicate object list corresponding to type declarations.
	related objects => O_type
constants	: indicate object list corresponding to constant declarations.
	related objects => O_constant
variables	: indicate object list corresponding to variable declarations.
	related objects => O_variable
files	: indicate object list corresponding to file declarations.
	related objects => O_file
aliases	: indicate object list corresponding to alias declarations.
	related objects => O_alias
attributes	: indicate object list corresponding to attribute declarations.
	related objects => O_attribute
attr_specs	: indicate object list corresponding to attribute specifications.
	related objects => O_attr_spec
uses	: indicate object list corresponding to use clauses.
	related objects => O_use
sequ_stmts	: indicate object list corresponding to sequential statements.
	related objects => SequStmt group

-- only for concurrent_assertion_statement --

condition	: indicate an object corresponding to assert condition.
	related objects => Expression group
report	: indicate an object corresponding to report expression.
	related objects => L_string
severity	: indicate an object corresponding to severity expression.
	related objects => T_element

-- only for concurrent_procedure_call_statement --

procedure	: indicate an object corresponding to procedure name.
	related objects => SP_procedure, SP_proc_body
parameter_maps	: indicate object list corresponding to parameter association list.
	related objects => OP_arrow

-- only for conditional_signal_assignment_statement --

target	: indicate an object corresponding to target.
	related objects => O_signal, Name_operator group, OP_aggregate
cond_waves	: indicate object list corresponding to conditional waveforms.
	related objects => ST_conc_wave

-- only for conditional_waveform --

waveforms	: indicate object list corresponding to waveform list.
	related objects => ST_waveform
condition	: indicate an object corresponding to condition.
	related objects => Expression group

-- only for selected_signal_assignment_statement --

expression	: indicate an object corresponding to select expression.
	related objects => Expression group
target	: indicate an object corresponding to target.
	related objects => O_signal, Name_operator group, OP_aggregate
sel_waves	: indicate object list corresponding to selected waveforms.
	related objects => ST_sel_wave

-- only for selected_waveform --

waveforms	: indicate object list corresponding to waveform list.
	related objects => ST_waveform
choices	: indicate object list corresponding to choices.
	related objects => Choice group

-- only for component_instantiation_statement --

component	: indicate an object corresponding to component name.
	related objects => O_component
generic_maps	: indicate object list corresponding to generic association list.
	related objects => OP_arrow
port_maps	: indicate object list corresponding to port association list.
	related objects => OP_arrow

-- only for generate_statement --

scheme	: indicate an object corresponding to generation scheme.
	related objects => Expression group

conc_stmts	: indicate object list corresponding to concurrent statements. related objects => ConcStmt group

-- only for wait_statement --

sensitivities	: indicate object list corresponding to sensitivity list. related objects => O_signal, Name_operator group
condition	: indicate an object corresponding to condition. related objects => Expression group
time_expr	: indicate an object corresponding to time expression. related objects => Expression group

-- only for assertion_statement --

condition	: indicate an object corresponding to assert condition. related objects => Expression group
report	: indicate an object corresponding to report expression. related objects => L_string
severity	: indicate an object corresponding to severity expression. related objects => T_element

-- only for procedure_call_statement --

procedure	: indicate an object corresponding to procedure name. related objects => SP_procedure, SP_proc_body
parameter_maps	: indicate object list corresponding to parameter association list. related objects => OP_arrow

-- only for signal_assignment_statement --

target	: indicate an object corresponding to target. related objects => O_signal, Name_operator group, OP_aggregate
waveforms	: indicate object list corresponding to waveform list. related objects => ST_waveform

-- only for waveform --

value_expr	: indicate an object corresponding to value expression. related objects => Expression group
time_expr	: indicate an object corresponding to time expression. related objects => Expression group

-- only for variable_assignment_statement --

target	: indicate an object corresponding to target.
	related objects => O_variable, Name_operator group, OP_aggregate
expression	: indicate an object corresponding to value expression.
	related objects => Expression group

-- only for case_statement --

expression	: indicate an object corresponding to case expression.
	related objects => Expression group
case_alts	: indicate object list corresponding to case statement alternatives.
	related objects => ST_case_alt

-- only for case_statement_alternative --

choices	: indicate object list corresponding to choices.
	related objects => Choice group
sequ_stmts	: indicate object list corresponding to sequential statements.
	related objects => SequStmt group

-- only for if_statement --

if_alts	: indicate object list corresponding to if statement alternatives.
	related objects => ST_if_alt

-- only for if_statement_alternative --

condition	: indicate an object corresponding to condition.
	related objects => Expression group
sequ_stmts	: indicate object list corresponding to sequential statements.
	related objects => SequStmt group

-- only for loop_statement --

scheme	: indicate an object corresponding to iteration scheme.
	related objects => Expression group
sequ_stmts	: indicate object list corresponding to sequential statements.
	related objects => SequStmt group

-- only for next_statement --

loop	: indicate an object corresponding to loop label.
	related objects => ST_loop

condition	: indicate an object corresponding to next condition. related objects => Expression group

-- only for exit_statement --

loop	: indicate an object corresponding to loop label. related objects => ST_loop
condition	: indicate an object corresponding to exit condition. related objects => Expression group

-- only for return_statement --

expression	: indicate an object corresponding to expression. related objects => Expression group

23) Subprogram object

This corresponds to a subprogram declaration or a subprogram body. The definition of a subprogram can be given in two parts: a subprogram declaration defining its calling conventions, and a subprogram body defining its execution. There are two forms of subprograms: procedures and functions. These subprogram kinds are described by an enumeration type "subprog_kind" defined as follows.

enum subprog_kind { SP_invalid, SP_procedure, SP_proc_body, SP_function, SP_func_body };

A procedure designator is always an identifier. A function designator is either an identifier or an operator symbol. A procedure call is a statement. A function call is an expression that returns a value.

Syntax

```
subprogram_declaration ::=
        subprogram_specification;

subprogram_body ::=
        subprogram_specification IS
          subprogram_declarative_part
        BEGIN
          subprogram_statement_part
        END [designator];

subprogram_specification ::=
        FUNCTION designator[(formal_parameter_list)] RETURN type_mark
        | PROCEDURE designator[(formal_parameter_list)]
```

Attribute

name	: field of string type for recording an designator.
subprog_kind	: field of subprog_kind type for recording kind of subprogram.

Relationship

-- only for procedure_declaration, function_declaration --

scope	: indicate an object corresponding to enclosing region. related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_proc_body, SP_func_body
parameters	: indicate object list corresponding to formal parameter list. related objects => O_constant, O_signal, O_variable

-- only for procedure_body, function_body --

symbols	: indicate named object list declared in this region. related objects => Name group
subprograms	: indicate object list corresponding to subprograms. related objects => O_subprogram
types	: indicate object list corresponding to type declarations. related objects => O_type
constants	: indicate object list corresponding to constant declarations. related objects => O_constant
variables	: indicate object list corresponding to variable declarations. related objects => O_variable
files	: indicate object list corresponding to file declarations. related objects => O_file
aliases	: indicate object list corresponding to alias declarations. related objects => O_alias
attributes	: indicate object list corresponding to attribute declarations. related objects => O_attribute
attr_specs	: indicate object list corresponding to attribute specifications. related objects => O_attr_spec
uses	: indicate object list corresponding to use clauses. related objects => O_use
sequ_stmts	: indicate object list corresponding to sequential statements. related objects => SequStmt group

-- only for function_declaration, function_body --

return_type	: indicate an object corresponding to type mark. related objects => O_type

24) Type object

This corresponds to a type declaration or a subtype declaration which declare a type and a subtype, respectively. The types declared by type declarations are classified into many kinds. An object corresponding to a type or a subtype is identified by an enumeration type "type_kind" defined as follows:

enum type_kind { T_invalid, T_subtype, T_incomplete, T_access, T_file, T_enum, T_element, T_integer, T_float, T_physical, T_unit, T_array, T_record, T_field };

There are four classes of types. Scalar types are integer types, floating point types, physical types, and types defined by an enumeration of their values. Composite types are array and record types. Access types provide access objects of a given type. File types provide access to objects that contain a sequence of values of a given type.

A subtype is a type together with a constraint. A value is said to belong to a subtype of a given type if it belongs to the type and satisfies the constraint. The given type is called the base type of the subtype.

A type mark in a subtype declaration denotes a type or a subtype. If the type mark denotes an access type or a file type, the subtype declaration may not contain a resolution function. If a subtype declaration does not include a constraint, the subtype is the same as that denoted by the type mark.

Syntax

```
subtype_declaration ::=
        SUBTYPE identifier IS [resolution_function_name] type_mark [constraint];
incomplete_type_declaration ::=
        TYPE identifier;
access_type_declaration ::=
        TYPE identifier IS ACCESS type_mark [constraint];
file_type_declaration ::=
        TYPE identifier IS FILE OF type_mark

-- scalar type --
enumeration_type_declaration ::=
        TYPE identifier IS (enumeration_literal_list);
integer_type_declaration ::=
        TYPE identifier IS RANGE range;
floating_type_declaration ::=
        TYPE identifier IS RANGE range;
physical_type_declaration ::=
        TYPE identifier IS RANGE range UNITS
            identifier;
        END UNITS;
unit_declaration ::=
        identifier = [abstract_literal] unit_name;

-- composite type --
array_type_declaration ::=
        TYPE identifier IS ARRAY (discrete_range_list) OF type_mark [constraint];
        | TYPE identifier IS ARRAY (type_mark RANGE <>
        , type_mark RANGE <>) OF type_mark [constraint];
record_type_declaration ::=
        TYPE identifier IS RECORD
            element_declaration    element_declaration
        END RECORD;
element_declaration ::=
        identifier_list : [resolution_function] type_mark [constraint];
```

Attribute

name	: field of string type for recording an identifier.
type_kind	: field of type_kind type for recording kind of type.

Relationship

-- only for subtype_declaration --

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_package, O_pack_body,
	ST_block, ST_process, SP_func_body, SP_proc_body
res_func	: indicate an object corresponding to resolution function name.
	related objects => SP_function, SP_func_body
type	: indicate an object corresponding to type mark.
	related objects => O_type
ranges	: indicate object list corresponding to constraint.
	related objects => Constraint group

-- only for incomplete_type_declaration --

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_package, O_pack_body,

ST_block, ST_process, SP_func_body, SP_proc_body

-- only for access_type_declaration --

scope	: indicate an object corresponding to enclosing region. related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_body, SP_proc_body
type	: indicate an object corresponding to type mark. related objects => O_type
ranges	: indicate object list corresponding to constraint. related objects => Constraint group

-- only for file_type_declaration --

scope	: indicate an object corresponding to enclosing region. related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_body, SP_proc_body
type	: indicate an object corresponding to type mark. related objects => O_type

-- only for enumeration_type_declaration --

scope	: indicate an object corresponding to enclosing region. related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_body, SP_proc_body
elements	: indicate object list corresponding to enumeration literal list. related objects => T_element

-- only for element of enumeration_type --

scope	: indicate an object corresponding to enclosing region. related objects => T_enum

-- only for integer_type_declaration --

scope	: indicate an object corresponding to enclosing region. related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_body, SP_proc_body
range	: indicate an object corresponding to range. related objects => Range_constraint group

-- only for float_type_declaration --

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_body, SP_proc_body
range	: indicate an object corresponding to range.
	related objects => Range_constraint group

-- only for physical_type_declaration --

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_body, SP_proc_body
range	: indicate an object corresponding to range.
	related objects => Range_constraint group
units	: indicate object list corresponding to unit declarations.
	related objects => T_unit

-- only for unit_declaration of physical_type --

scope	: indicate an object corresponding to enclosing region.
	related objects => T_physical
literal	: indicate an object corresponding to abstract literal.
	related objects => L_integer, L_float
base_unit	: indicate an object corresponding to unit name.
	related objects => T_unit

-- only for array_type_declaration --

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_package, O_pack_body, ST_block, ST_process, SP_func_body, SP_proc_body
indexes	: indicate object list corresponding to discrete range list.
	related objects => OP_range
type	: indicate an object corresponding to type mark.
	related objects => O_type
ranges	: indicate object list corresponding to constraint.
	related objects => Constraint group

-- only for record_type_declaration --

scope	: indicate an object corresponding to enclosing region.
	related objects => O_entity, O_arch_body, O_package, O_pack_body,

	ST_block, ST_process, SP_func_body, SP_proc_body
fields	: indicate object list corresponding to element declarations.
	related objects => T_field

-- only for element_declaration of record_type --

scope	: indicate an object corresponding to enclosing region.
	related objects => T_record
res_func	: indicate an object corresponding to resolution function name.
	related objects => SP_function, SP_func_body
type	: indicate an object corresponding to type mark.
	related objects => O_type
ranges	: indicate object list corresponding to constraint.
	related objects => Constraint group

25) Use object

This corresponds to a use clause which makes certain declarations to be directly visible. For each expanded name, one Use object is created.

Each expanded name in a use clause identifies one or more declarations that will potentially become directly visible. If the suffix of the selected name is a simple name, character literal, or operator symbol, then the expanded name identifies only the declaration contained within the package or library denoted by the prefix. If the suffix is the reserved word ALL, then the selected name identifies all declarations that are contained within the package or library denoted by the prefix.

For each use clause, there is a certain region called the scope of the use clause. This region starts immediately after the use clause, and extends to the end of the declarative region that contains this use clause.

Syntax

```
use_clause ::= USE expanded_name, expanded_name;
```

Attribute

Relationship

scope	: indicate an object corresponding to enclosing region. related objects => O_block_conf, O_context, O_entity, O_arch_body, O_package, O_pack_body, O_configuration, ST_block, ST_process, SP_proc_body, SP_func_body
ext_ref	: indicate an object corresponding to expanded name. related objects => Name group

26) Variable object

This corresponds to a variable declaration or an interface variable declaration which declares a variable and an interface variable of the specified type respectively. The identifier list denotes the names of [interface] variables. For each identifier on this list, one object is created.

The mode implies the direction of information flow through the parameter. To describe the mode of a variable, the enumeration type "mode_kind" is defined as follows.

```
    enum mode_kind { M_invalid, M_null, M_in, M_out, M_inout, M_buffer, M_linkage };
```

A mode_kind field of this object identifies one of two kinds of variables. The objects having M_null imply variables. The objects having other mode kinds correspond to interface variables.

Syntax

```
variable_declaration ::= VARIABLE identifier_list : type_mark [constraint] [:= expression];

interface_variable_declaration ::= [VARIABLE] identifier_list : [mode] type_mark [constraint] [:=
                    static_expression];
```

Attribute

name	: field of string type for recording an identifier.
mode_kind	: field of mode_kind type for recording kind of mode.

Relationship

scope	: indicate an object corresponding to enclosing region. related objects => ST_process, O_subprogram
type	: indicate an object corresponding to type mark. related objects => O_type
ranges	: indicate object list corresponding to constraint. related objects => Constraint group
expression	: indicate an object corresponding to expression. related objects => Expression group

2. Design object

This corresponds to a design unit which is a construct that can be independently analyzed and individually stored in a design library. Design units in a design file are analyzed in the textual order of their appearance in the design file.

The text of a design unit is composed of a context clause and a library unit. A library unit may be an entity declaration, an architecture body, a configuration declaration, a package declaration, or a package body. A library unit may be either a primary unit or a secondary unit. A secondary unit is a separately analyzed body of a primary unit resulting from a previous analysis.

Syntax

```
    design unit ::=
        context_clause library_unit
```

Attribute

type	: field of object_type type for recording kind of related Basic object.
pname	: field of string type for recording associated primary unit name.
sname	: field of string type for recording associated secondary unit name.
path	: field of string type for recording path-name of intermediate form file.
spath	: field of string type for recording path-name of an associated design file.
count	: field of integer type for recording count of contained Basic objects.
time	: field of time_t type for recording analyzed time.

Relationship

top	: indicate an Basic object corresponding to library unit. related objects => Context group
mapTable	: indicate Basic object list contained in this design unit. related objects => Basic group

3. Lib object

This corresponds to a design library which is a storage facility for intermediate form representations of analyzed design units. A design library can be regarded as a container of analyzed design units.

The design units contained in a same design library are identified by unique names. Each name of primary unit name must be unique within the same design library. The unique name of a secondary unit is constructed by the combination of its name and the name of associated primary unit, so the multiple secondary units having the same names can exist in the same design library.

A library logical name denotes a design library in the host environment. For a given library logical name, the actual name of the corresponding design library in the host environment is a directory path-name in UNIX file system.

Syntax

Attribute

name	: field of string type for recording library logical name.
path	: field of string type for recording a directory path-name.

Relationship

designs	: indicate Design object list contained in this design library.
	related objects => Design object

9 Appendix B. Naming Rule

VHDL terminology is used as the names of VDT objects and the storage fields of these objects. VDT naming rule makes the convention on abbreviated name of VDT objects and the storage fields of these objects. The abbreviated names are originated from the long terms of VHDL terminology. For too long term constructed by combination of two more terms, the abbreviated term is used for these names.

This naming rule enables the user to recognize the functions of access routines quickly and easily, and makes the name of access routines more short. The following is the abbreviated terms of VDT objects and their storage fields corresponding to each VHDL term.

- architecture --> arch
  arch_body (architecture_body)
- assertion --> assert
  assert_cond (assertion_condition)
- assignment --> assign
  assign_stmt (assignment_statement)
  assign_option (assignment option)
- attribute --> attr
  attr_spec (attribute_specification)
  attr_kind (attribute_kind)
- component --> comp
  comp_inst (component_instantiation)
- conditional --> cond
  cond_sig_assign (conditional_signal_assignment)
- concurrent --> conc
  conc_assert (concurrent_assertion)
  conc_proc_call (concurrent_procedure_call)
- configuration --> conf
  conf_spec (configuration_specification)
- declaration --> omission
  entity (entity_declaration)
  configuration (configuration_declaration)
- disconnect --> disc
  disc_spec (disconnection_specification)
- expression --> expr
  time_expr (time_expression)
  initial_expr (initial_expression)

- external --> ext
  ext_ref (external_reference)
- instantiation --> inst
  comp_inst (component_instantiation)
- package --> pack
  pack_body (package_body)
- reference --> ref
  ext_ref (external_reference)
- selected -->sel
  sel_sig_assign (selected_signal_assignment)
- sequential --> sequ
  sequ_stmt (sequential_statement)
- signal --> sig
  sig_assign (signal_assignment)
  sel_sig_assign (selected_signal_assignment)
- specification --> spec
  attr_spec (attribute_specification)
  disc_spec (disconnection_specification)
- statement --> stmt
  conc_stmt (concurrent_statement)
- subprogram --> subprog
  subprog_body (subprogram_body)
  subprog_kind   (subprogram kind)
- variable --> var
  var_assign (variable_assignment)

10 Appendix C. FSM Synthesis from VHDL Description

STG(State Transition Graph) can be extracted from VHDL behavioral code describing FSM(Finite State Machine). This STG later can be synthesized by SIS(Sequential Interactive System) that is a synthesis tool of sequential logic.

1) VHDL Subset

FSM circuit is described more easily by VHDL behavioral code than by state transition table. All VHDL constructs is not needed to describe FSM circuit, only a few VHDL constructs are allowed. VHDL description must follow the following guide and limit.

- Only one process statement can appear within an architecture body.

- The process statement must contain a wait statement representing a synchronous circuit.

```
process
begin
        wait until CLK = '0'; -- falling edge
        . . .
end process;
```

- The process statement must describe a resettable FSM circuit, as follows.

```
if RST = '1' then
        -- reset circuit
end if;
```

- The following sequential statement can appear within a process statement.

```
if statement
case statement
signal assignment statement
variable assignment statement
```

- The allowed operators within expression are (and, or, nand, nor, not, xor, =).

```
if (A = B) then
        S <= not (A and B);
end if;
```

- The allowed types of signal and variable are (Bit, Boolean, Bit_vector, enumeration).

```
type STATE is (RESET, IDLE, BUSY);
```

```
signal S1 : STATE;            -- right
signal S2 : INTEGER;        -- wrong
```

- The assigned value of signal and variable assignment must be a static value.

```
CurState <= IDLE;            -- right
CurState <= INPUT;          -- wrong
```

- The case statement must be of a similar form, as follows.

```
case CurState is
when IDLE =>
            . . .
when BUSY =>        -- all value of CurState must be appeared.
            . . .           -- "when others => " is not allowed.
end case;
```

2) FSM Synthesis

The extracted STG can be synthesized through SIS tool. SIS inputs BLIF(Berkeley Logic Interchange Form) form file. The if2blif tool extracts STG from a given intermediate form, and outputs BLIF form file. The output of SIS is BDNET form file which is a net list of synthesized circuit. Finally, the BDNET file is translated into a structural VHDL description.

a. Analyze a given VHDL code describing FSM circuit.

```
% van detector.vhd
```

b. Extract blif code from constructed intermediate form.

```
% if2blif -o detector.blif DETECTOR
```

c. Run SIS tool and read blif code.

```
% sis
> read_blif detector.blif
```

d. Run state assignment routine for the current circuit.

```
> state_assign
```

e. Run technology mapping routine with any component library.

```
> source script.oct
```

f. Output bdnet code for the synthesized circuit.

```
> write_bdnet detector.bdnet
```

g. Generate VHDL structural code from the bdnet file.

```
> quit
% bd2vhd -o detector.out detector.bdnet
```

h. Compare the operation of two VHDL codes, original behavioral code, and generated structural code by simulation.

3) MSU Library

The net list of synthesized circuit describes the connections among logic components. To get the structural VHDL code corresponding to a given net list, VHDL library is necessary. The VHDL library must contain all VHDL components corresponding to each logic component. The logic-level MSU(Mississippi State University) library is installed in SIS tool package. The corresponding VHDL library is developed and installed in VDT. The following shows the kind of VHDL components, and explains the function of each component.

invf101(A1: O)	--> Inverter
norf201(A1,B1: O)	--> 2-input Nor gate
norf301(A1,B1,C1: O)	--> 3-input Nor gate
norf401(A1,B1,C1,D1: O)	--> 4-input Nor gate
nanf201(A1,B1: O)	--> 2-input Nand gate
nanf301(A1,B1,C1: O)	--> 3-input Nand gate
nanf401(A1,B1,C1,D1: O)	--> 4-input Nand gate
nanf211(A1,B1: O)	--> 2-input And gate
nanf311(A1,B1,C1: O)	--> 3-input And gate
nanf411(A1,B1,C1,D1: O)	--> 4-input And gate
norf211(A1,B1: O)	--> 2-input Or gate
norf311(A1,B1,C1: O)	--> 3-input Or gate
orf401(A1,B1,C1,D1: O)	--> 4-input Or gate
xorf201(A1,B1: O)	--> 2-input Xor gate
xnorf201(A1,B1: O)	--> 2-input Xnor gate
muxf201(A1,B2,SEL3: O)	--> 2-input Multiplexor
larf310(DATA1,RST3,CLK2: Q)	--> Clocked latch with reset
dfnf311(DATA1,CLK2: Q)	--> Falling edge triggered D flip-flop
dfbf311(DATA1,SET4,RST3,CLK2: Q)	--> D flip-flop with synchronous set/reset

11 Appendix D. VDT Installation

The distributed package of VDT includes all source codes and documents of PI library and utility programs. This package is located on public domain. This package is located in the anonymous ftp site (poppy.snu.ac.kr) as a named file "/pub/vhdl/vdt-2.6.tar.gz". The running of this package have been verified in the platform of SUN4 compatible system under running UNIX (SUNOS 4.1.1 or later version). The compilation and installation of this package can be accomplished just by following this flow.

- The following programs are necessary to compile this package.
  ANSI C++ Compiler (g++ version 2.7.0 or later)
  LEX (flex version 2.4.7 or later)
  YACC (bison version 1.22 or later)

- Set the UNIX environment variable VDT and include the path where all tools are installed.

```
% setenv VDT ~cad/vdt
% set path = ($path $VDT/bin)
```

- Read the contents of the VDT tape, then the final directory structure will have the following directories.

$VDT/bin	- where the tools are installed.
$VDT/doc	- where the manuals are placed.
$VDT/etc/std	- where the VHDL standard packages are installed.
$VDT/lib	- where the PI libraries are installed.
$VDT/include	- where the PI header files are installed.
$VDT/src	- where all source files are installed.
$VDT/src/pi	- where the PI(primitive layer) source files are installed.
$VDT/src/util	- where the vgen, ifdump, browse source files are installed.
$VDT/src/van	- where the van source files are installed.
$VDT/src/vt	- where the PI(application layer) source files are installed.

- Now you are ready to install everything into the $VDT directory, just type;

```
% make install
```

- Please e-mail to the following address.
  inmind@poppy.snu.ac.kr

so you can be placed on the updates and bug fixes mailing list. Whenever we make updates or fix bugs, we will send out the updates and bug fixes to you.

- We are very interested in getting bug reports. Please mail your bug reports to
  e-mail: inmind@poppy.snu.ac.kr