SYSTEMATIC PROGRAM DEVELOPMENT FROM ABSTRACT DATA TYPE DECOMPOSITION. A CASE STUDY: THE TOPOLOGICAL SORT ON ORIENTED GRAPHS.

Francis Losavio, Alfredo Matteo
Centro de Ingeniería de Software Y Sistemas ISYS, Facultad de Ciencias, Universidad Central de Venezuela.
Ap. 47002, Los Chaguaramos 1041-A, Caracas, Venezuela.

Françoise Schlienger
Laboratoire de Recherche en Informatique L.R.I., Université Paris-Sud,
Bât.490, Orsay cedex 91405, Orsay, Francia

ABSTRACT

A methodology is presented for automatic construction of algorithms, involving decomposition of an Abstract Data Type (ADT) which is formally specified as an algebraic ADT. As a case study illustrating the proposed methodology, we have selected to construct a program corresponding to the topological sort problem, for an oriented graph data type, which is algebraically specified using a formal specification language. An Ada program is generated following our methodolgy. Early handling of exceptional situations is used as a major programming style. The *program unit* implementing the ADT, a *decomposition scheme* for representing common algorithms on the data type structure and an appropriated *construction strategy* depending on the nature of the problem, are the main elements used in this program construction process.

Our major goal is to show the systematic development of classical algorithms on commonly used structured data types (array, tree, graph, ...) and the derivation of the corresponding program, within an assisted program construction context.

1. INTRODUCTION.

This work presents a methodology [Gre 84], [Los 85], [Los] for automatic construction of algorithms involving decomposition of an Abstract Data Type (ADT) which is formally specified as an algebraic ADT. A case study will illustrate the proposed methodology; for

this purpose we have selected to solve the topological sort classical problem. An oriented graph data type is algebraically specified using the PLUSS [Cap 87], [Bid 89] specification language. Our methodology is independent from the language chosen for implementing the algorithm, provided the language holds ADT and exception handling features. Since Ada provides these features, we have chosen it as our target language. An Ada program is then generated following the methodolgy. Early handling of exceptional situations is used as a major programming style. The language *program unit* implementing the ADT, a *decomposition scheme* for repre ing common algorithms on the data type structure and an appropriated *construction strategy* depending on the nature of the problem, are the main elements used in this program construction process.

Our major goal is to show the systematic development of classical algorithms on commonly used structured data types (array, tree, graph, ...) and the derivation of the corresponding program implementing them, within an assisted program construction context.

This paper is structured in two main parts. The program construction methodology is presented in the first place, focousing on its context, antecedents, actual development, notations and terminology used. The second part is devoted to the description of the case study: the problem definition and the corresponding program obtained applying the methodology. An Appendix contains the oriented graph algebraic specification and the corresponding Ada package implementing the oriented graph ADT.

2. METHODOLOGY.

Within the field of automatic programming, the assisted program construction approach (where program generation is done with the user intervention) is the context of our approach. Richard Waters [R&W 88] describes several techniques employed in problem-solving, one of these being the so called *inspection methods*

based on *clichés*. It is of general agreement that human programmers think seldom in terms of primitive constructs such as assignments and tests, but rather in terms of combinations of elements (a cliché) of familiar concepts. The central feature of inspection methods is the codification and use of clichés, which is constitued of three parts: - a *skeleton* that is present in every occurrence of the cliché, - some *roles* whose contents vary from one occurrence to the next, - some *constraints* on what can fill the roles. There may be different kinds of clichés; we are particularly interested in *algorithmic clichés*, which belong to a specialized programming area. The "knowledge" of these clichés is represented by common algorithms (such as counting, searching and sorting) and by common data structures (array, sequences, etc.). When used in the assisted construction approach, inspection methods have the important advantage of constituing a shared vocabulary between the system and the user, which seems a natural and attractive medium of communication .

Waters'idea of algorithmic clichés is shared by our proposed assisted program construction method. Using the above terminology, "the knowledge" of our clichés is composed by:

1. The decomposition scheme , which corresponds to a particular way of processing the data structure. The intuitive notion of decomposition of a data type means to consider its data structure from an algorithmic point of view, separated into different parts or blocks, each one of these being constitued by a set of operations defined in the ADT signature. These parts must contain selector operations and at least one decomposition operation, that is to say, a destructor operation for the obtention of the objects of the structure. The algorithm is reflected in the way in which the operations, constituing the different blocks, are placed in order to obtain these objects. There will be two kinds of blocks: BLOCK_SEL, including a selector operation and BLOCK_DES, including a destructor operation. For example, in order to algorithmically deal with a sequence, we may require to separate its first element and process it accordingly, then recursively,

we apply the same reasoning for the rest of the sequence. Consequently, we consider this data structure splitted into two objects: the first element and the sequence constitued by all the elements, excepting for the first one.

An *abstract decomposition scheme* (*ADS*) [Los] is a general formulation for the above decomposition notion, allowing the expression of the algorithmic structure of a data type, by means of the operations on the type, which is algebraically specified.

The schemes we are dealing with are *concrete schemes*, in the sense that they are ADS, applied on a particular data type; they are also *generic schemes*, that is to say, parameterized by the same parameter as the corresponding generic abstract data type.

2. The construction strategy, which is a program scheme or algorithm depending on the nature of the problem (for example counting or searching) we are dealing with. The construction strategy relates the problem we are trying to build, which we will call the *Problem of Interest* (*PI*) and the decomposition scheme, by appropriate combinations of the input and output variables of PI and the output variables of the involved decomposition scheme. The construction strategies also greatly contribute to the structuring of the resulting algorithm, organizing the different blocks constituing the decomposition schemes. Several strategies are present in our methodology: COMBINE, TEST, TEST-COMBINE, ESCAPE [Los] . Each strategy is in general oriented to a particular kind of problem: COMBINE for counting, TEST and ESCAPE for searching, TEST-COMBINE for sorting. The COMBINE strategy, which produces an algorithm in which all the blocks are sequentially organized, will be discussed in details.

In the next section we will detail the program development methodology.

2.1 The program construction process.

The proposed program construction process follows the assisted approach. If we place ourselves in a user-system context, the general idea of the process is the following: first, a problem (PI) is proposed to the system as a function header, expressed in the

target language. The kind of PI and the data structure must be identified by the user. A convenient decomposition scheme is then selected by the user and the system applies (translates) it to the data structure, producing a program skeleton, "instanciated" with the PI variables. This skeleton contains various continuation points which have to be filled with the user assistance, in order to solve the constraints imposed by the selection of an appropiate construction strategy (It is clear that the user has to "master" the construction method, in the sense that he must have some knowlege on the strategy to be used for a particular PI). These constraints are the renaming of new variables and functions and the solving of the possible new sub-problems (some of the introduced new functions). In this way, a dialog system-user is established, until the program is completed. The program construction process is quasi-independent from the target language used (the "quasi" stay for the organization of the exception handling). We have selected Ada as our target language because of the similarity of the Ada generic package with the PLUSS generic SPEC construct, besides the characteristics of modularity and exception handling.

The resulting executable program systematically constructed following this methodology may need minor optimizations, such us eliminating unnecessary intermediate variables or unused exception handlers.

In order to assistedly construct a program corresponding to a given problem PI, the following information is required:

a. A **Specification Library** containing commonly used abstract data types, expressed in a formal specification language (the PLUSS specification language in our case). We will be dealing with *generic abstract data types*. The algebraic specification of the oriented graph data type, witten in PLUSS is shown in the Appendix.

b. A **Program Library** containing the program units implementing the algebraically specified ADT, written in a target language holding abstraction and exception handling features (the Ada language in our case; the Program Library will be called the Ada

Library. The program units contained will be basically Ada generic packages and application programs). The generic package OR_GRAPH, implementing the PLUSS generic specification SPEC OR_GRAPH, may be seen in the Appendix.

c. A **Decomposition Scheme Library** containing the Generic Concrete Decomposition Schemes (GCDS) for the abstract data types present in the Specification Library.

As an example of a GCDS, the oriented graph data type generic concrete decomposition scheme, will be shown below:

```
SCHEME SH1-GRAPH [NODE]
(g : graph --> n : node; g1 : graph)
begin
  n <-1- obtain_node(g);
  [1]
exception
   1 ---> no_node
end
begin
  g1 <-2- rest_of_graph(n,g);
  [2]
end
END SH1-GRAPH
```

Notice that the formal parameter is NODE, and that the exception case is *no_node*, which may be raised by the operation *obtain_node* applied on an empty graph; *rest_of_graph* is a total operation, hence it doesn't raise an exception. The numbers in [ ] indicate a continuation point, where the construction strategy has to be applied.

The GCDS is structered in a header, containing the so called input variable (g:graph) and output variables (n:node, g1:graph) of the scheme and the different blocks, one for each decomposition scheme output variable.

2.2 The construction strategies.

This section describes the application of the construction strategy to a decomposition scheme. We have selected the COMBINE strategy, as an example of this point. Similar description of the other strategies may be seen in [Los].

Application of the COMBINE strategy.

The application of this strategy to a decomposition scheme allows the computations of intermediate results, which will be denoted by R_r, $1 \le r \le k$, k number of blocks of the GCDS, from the set of output variables of the scheme. Each R_r depends entirely from the block where the result is computed, including the results computed in the exception handlers associated to the block. Each R_r is computed by a function F_r, whose arguments are the output variables of the scheme and the input variables of PI. In particular, for each F_r, the arguments are constitued by the variable e_i, $1 \le i \le m$ or S_j, $1 \le j \le n$, and all the output variables of the scheme. When variable S_j is present among the arguments of F_r is a recursive call to problem PI. If a function F_r raises exception EX_F_r, its handler is added to the already existing exception handler at the end of the block. At the end of the program body, a new function, called OP-COMBINE will compute the combination of the R_r intermediate results. If this operation raises an exception its handler is added at the end of the program body. The COMBINE strategy is well suited for solving counting type problems. The algorithm shown below is obtained:

The algorithm.

```
function PI (S : TS; I_1,...,I_q) return T is
-- local variable declarations
R_1 : T,... , R_m : T, R_m+1 : TS,...,R_K : TS; X_1 : T_1,...,X_k : T_k; S_1,...,S_n : TS;
e_1,...,e_m : TE;
begin -- body of PI
      begin                              -- block BLOCK_SEL_1
      X_1 <- OP_PREL_1
      e_1 <- SEL_1(S,Z_1)
      R_1 := F_1(e_1,I_1,...,I_q);
      exception
       when EX_OP_PREL_1 => return F_11(e_1,I_1,...,I_q);
       when EX_SEL_1 => return F_12(e_1,I_1,...,I_q);
       when EX_F_1 => return F_13(e_1,I_1,...,I_q);
      end;                               -- block BLOCK_SEL_1
      ... ... ...
```

```
begin                                   -- block BLOCK_SEL_m
I_m <- OP_PREL_m
e_m <- SEL_m(S,Z_m)
R_m := F_m(e_m,I_1,...,I_q);
exception
 when EX_OP_PREL_m => return F_m1(e_m,I_1,...,I_q);
 when EX_SEL_m => return F_m2(e_m,I_1,...,I_q);
 when EX_F_m => return F_m3(e_m,I_1,...,I_q);
end;                                    -- block BLOCK_SEL_m
```

```
begin                                   -- block  BLOCK_DES_1
I_m+1 <- OP_PREL_m+1
S_1 <- DES_1(S,Z_m+1)
R_m+1 := PI(S_1,I_1,...,I_q);
exception
 when EX_OP_PREL_m+1 => return F_(m+1)1(S_1,I_1,...,I_q);
 when EX_DES_1 => return F_(m+1)2(S_1,I_1,...,I_q);
end;                                    -- block  BLOCK_DES_1
```

. . . . . .

```
begin                                   -- block BLOCK_DES_n
I_k <- OP_PREL_k
S_n <- DES_k(S,Z_k)
R_k := PI(S_n,I_1,...,I_q);
exception
 when EX_OP_PREL_k => return F_k1(S_n,I_1,...,I_q);
 when EX_DES_n => return F_k2(S_n,I_1,...,I_q);
end;                                    -- block BLOCK_DES_n
return OP-COMBINE(R_1,...,R_k);
exception
  when EX_OP_COMBINE =>  ... ... ... [1]
end PI;
```

Notice that [1] is a continuation point that must be filled by the user with an adequate action. Notice also that the algorithm resulting from the application of the COMBINE strategy is written in an Ada like notation. For simplification, OP_PREL and SEL denote sets of operations; EX_OP_PREL and EX_SEL denote the corresponding sets of exceptions; the symbol <- means that an operation (respectively each operation in case of sets) assign a value to a local variable. In the next section this strategy will be used in the construction of the topological sort program.

3. CASE STUDY: THE TOPOLOGICAL SORT.

Our Problem of Interest, PI, is the following: build a sequence s containing all the nodes of an oriented graph g, in such a way that no node appears before one of its predecessors (PI is called a topological sort). The pre-condition for PI is the requirement that g must be without circuits. The post-condition is a partial ordering on the nodes of g.

The Ada program shown below has been produced accordingly to our methodology:

```
... ... ...

function topol_sort(g:graph) return seq is
n : integer; g1 : graph;
s : seq;
begin
 g1 := g; -- necessary because rest_of_graph modifies graph g
 begin   -- block SEL1
   n := obtain_node(g1);
   exception
     when no_node => return empty;
 end;
 begin   -- block DES1
   rest_of_graph (n,g1);
   s := topol_sort (g1);
 end;
 return add(n,s); -- OP_COMBINE
end topol_sort;

... ... ...
```

Notice that scheme SH1-GRAPH has been applied, and the COMBINE strategy is responsible for constructing the resulting sequence, adding the nodes with negative degree 0, obtained in each iteration as a partial result, by the operation add(n,s), placed at the end of function topol_sort. The operations on the graph data type are implemented in the ORIENTED_GRAPH package shown in the Appendix. The add operation belongs to the package implementing the SEQUENCE data type, which may be seen in [Los]. The algorithm stops when the graph is empty, raising exception no_node by operation obtain_node. We have used here COMBINE, which is a strategy adapted for solving counting problems, instead of TEST-COMBINE, suited for solving sorting problems, because the test on the node having negative degree 0 is implicit in the

obtain_node operation. We point out that this program and all the Ada code [DOD 83] shown in this paper, have been tested on the Alsys Ada Compiler of the L.R.I., Université de Paris-Sud, Orsay, France.

The M-APEX protoype Ada programming environment [Los 89], built by the ISYS research center at the Faculty of Science, U.C.V., Caracas, Venezuela, contains the APA (Automatic Programming in Ada) tool [G&V 91], [A&H 91], which implements the program construction methodology presented in this paper.

3. CONCLUSION.

The major goal of this paper is to introduce a systematic program construction methodology. It is clear that the algorithms automatically produced may not be optimal ones, in the sense that they may need further optimizations; also other representations implementing the ADT may be selected, in order to obtain more performant algorithms. Our present research focuses on how to "capture" the users experience on problem solving, in order to leave to the system the decison on the selection of the construction strategy, according to a given type of problem. We think also that the construction strategies may be further enriched by several variants. This research will drive us in a near future, towards an expert system, within an assisted program construction context, following this methodology.

4. ACKNOWLEDGMENT.

The authors are gratley indebted to Michel Bidoit, who has followed very closely this research from the beginning, for his pertinent advice and always frutiful remarks and they also thank Oscar Ordaz for his useful comments on the oriented graph ADT.

5. REFERENCES.

[G&V 91] ANES L., HERNANDEZ J.
"Un sistema de construccion automatica de programas Ada: impelemntación de los moduios de construcción.

Trabajo especial de Grado para la Licenciatura en Computación, UCV. Marzo 91.

[Bid 89] BIDOIT M.
Pluss, un langage pour le développement de specifications algebriques modulaires", Thèse d'Etat, Université de Paris-Sud, Orsay, May 1989.

[Cap 87] CAPY F.
ASSPEGIQUE: un environnement d'exceptions ... .
Thèse de 3ème cycle, Université Paris-Sud, Orsay 1987.

[DOD 83] DEPARTMENT OF DEFENSE OF THE USA.
Reference Manual for the Ada Programming Language
ANSI/MIL-STD 1815-A, January 1983

[Gre 84] GRESSE C.
"Contribution à la programmation automatique. CATY : un système de construction assistée de programmes". Thèse d' Etat, Université de Paris-Sud, Orsay, March 1984.

[G&V 91] GONZALEZ M., VELOZ, C.
"Un sistema de construcción automática de programas Ada: impelemntación de los módulos de interfaz.
Trabajo especial de Grado para la Licenciatura en Computación, UCV. Marzo 91

[Los] LOSAVIO F.
"Dérivation systématique des programmes Ada comportant des traitements d'exception à partir des spécifications algébriques de types de données"
Manuscript for Doctoral Dissertation, Université Paris-Sud, Orsay.

[Los 89] LOSAVIO F.
"Towards the Construction of Interactive Working Environments: A prototype Ada Environment"
Proceedings of the ICNTSSD'89, Caracas, November 1989.

[Los 85] LOSAVIO F.
"Construction assistée de programmes Ada fondée sur la prise en compte d'exceptions".
Thèse de 3ème cycle, Université de Paris-Sud, Orsay, November 1985.

[R&W 88] RICH C, WATERS R.
Automatic Programming. Myths and Prospects.
IEEE Computer 21(8)40-51 August 88.

APPENDIX.

Algebraic specification of the graph data type.

Many different real-life problems or situations (communication networks, tasks scheduling, hierarchy of specification modules ...) may be modeled by a graph. In order

to give a better description of the graph algebraic specification, we will introduce some notations and definitions.

An *oriented graph* g is a pair ‹S,A›, where S is a finite set of elements called *nodes* and A is a finite set of ordered pairs called *arcs*. A *non oriented graph* g is a pair ‹S,A›, where S is a finite set of nodes and A is a finite set of pairs of nodes called *edges*. In an oriented (or non oriented) graph g, a *path* (or *chain*) of length l is a sequence of l+1 nodes $(n_0,n_1,...,n_l)$, such that for all i, $0 \le i \le l-1$, $n_i \rightarrow n_{i+1}$ is an arc (or edge) of g. In an oriented (or non oriented) graph g, a path (or chain) $(n_0, n_1, ..., n_l)$, where all the arcs (or edges) are pairwise different, and such that the initial and terminal nodes of its path (or chain) are equal, is called a *circuit* (or *cycle*). An oriented (or non oriented) graph is called *simple*, when it has no loops (i.e. an arc (edge) with the same extremities) nor multiple arcs (or edges) (i.e. more than one arc with the same orientation (or edge), between two nodes).

In what follows we will consider graphs that are *simple* and without circuits.

We will present here an algebraic specification for oriented graphs, **SPEC** OR_GRAPH, parameterized by the NODE specification, which is not shown here for abbreviating our text. The specified operations are the followings :

The generators are *empty* to initialize a graph, *add_node* to construct a graph from one node and a graph and *add_arc* to construct a graph from an arc and a graph.

The observers are *empty_graph*, which tests if the graph is empty, *exist_node* to test if a node is present in the graph, *exist_arc* to test if an arc is present in the graph, d^- which returns the negative degree (the number of incident nodes) of a node, *nb_nodes* which returns the number of nodes of a graph and *predi* which, given a node and an integer i, returns the i-th predecessor of the node. We adopt the convention that the predecessors of one node are numbered in an increasing order.

The selector is *obtain_node*, to return a node such that $d^-(n,g) = 0$. Note that this is always possible because, in every graph g without circuits, there is always a node n

such that $d^{\circ}(n,g)=0$. The destructor is *rest_of_graph*, to return the graph without a node.

The exception cases are : *no_node*: corresponds to the search for a node, when the graph is empty. *no_graph*: corresponds to the addition of an arc to non existing nodes. *parallel_arc*: corresponds to the addition of an already existing arc. *loop*: is the adition of an arc with the same initial and terminal nodes. *existing_node*: is the addition of an already existing node. *no_pred*: a predecessor of a node cannot be greater than the number of incidents nodes or 0 cannot be the predecessor of a node *no_d°*: it occurs when the graph is empty.

SPEC : OR_GRAPH [NODE]
USE : BOOL, INTEGER
SORTS : graph
GENERATED BY :

empty :	-> graph
add_node :	node graph -> graph
add_arc :	node node graph -> graph
OPERATIONS :	
empty_graph :	graph -> bool
exist_node:	node graph -> bool
exist_arc :	node node graph -> bool
obtain_node :	graph -> node
rest_of_graph :	node graph -> graph
d° :	node graph -> integer
nb_nodes :	graph -> integer
predi :	node integer graph -> node
pred_aux :	node integer graph -> node
EXCEPTION CASES :	
no_node :	empty_graph(g) = true => obtain_node(g)
no_graph :	or(not exist_node(n1,g),not exist_node(n2,g)) => add_arc(n1,n2,g)
parallel_arc :	exist_arc(n1,n2,g) = true => add_arc(n1,n2,g)
loop :	n1 = n2 => add_arc(n1,n2,g)
existing_node :	exist_node(n,g) = true => add_node(n,g)
no_pred :	or(or(i=0,(i>d°(n,g)),not exist_node(n,g))=>pred_aux(n,i,g)
no_d°:	exist_node(n,g) = false => d°(n,g)
OK-AXIOMS :	
empty_graph1 :	empty_graph(empty) = true
empty_graph2 :	empty_graph(add_node(n,g)) = false
empty_graph3 :	empty_graph(add_arc(x,n,g)) = false
exist_node1 :	exist_node(n,empty) = false
exist_node2 :	n1 = n2 => exist_node(n1,add_node(n2,g)) = true

exist_node3 : n1≠n2 => exist_node(n1,add_node(n2,g)) = exist_node(n1,g)

exist_node4 : or(n1=n2, n1=n3) => exist_node(n1,add_arc(n2,n3,g)) = true

exist_node5 : and(n1 ≠ n2, n1 ≠ n3) => exist_node(n1,add_arc(n2,n3,g)) = exist_node(n1,g)

exist_arc1 : exist_arc(n1,n2,empty) = false

exist_arc2 : exist_arc(n1,n2,add_node(n3,g)) = exist_arc(n1,n2,g)

exist_arc3 : and(n3 = n1, n4 = n2) => exist_arc(n1,n2,add_arc(n3,n4,g)) = true

exist_arc4 : or(n3 ≠ n1, n4 ≠ n2) => exist_arc(n1,n2,add_arc(n3,n4,g)) = exist_arc(n1,n2,g)

rest_of_graph1 : rest_of_graph(empty) = empty

rest_of_graph2 : n1 = n2 => rest_of_graph(n1,add_node(n2,g)) = g

rest_of_graph3 : n1 ≠ n2 => rest_of_graph(n1,add_node(n2,g)) = add_node(n2,rest_of_graph(n1,g))

rest_of_graph4 : or(n1 = n2, n1 = n3) => rest_of_graph(n1,add_arc(n2,n3,g)) = rest_of_graph(n1,g)

rest_of_graph5 : and(n1 ≠ n2, n1 ≠ n3) => rest_of_graph(n1,add_arc(n2,n3,g)) = add_arc(n2,n3,rest_of_graph(n1,g))

obtain_node : $\overset{\cdot}{d}^{-}$(obtain_node(g))= 0

$\overset{\cdot}{d}^{-}$1 : n1 = n2 => $\overset{\cdot}{d}^{-}$(n1,add_node(n2,g)) = 0

$\overset{\cdot}{d}^{-}$2 : n1≠ n2 => $\overset{\cdot}{d}^{-}$(n1,add_node(n2,g)) = $\overset{\cdot}{d}^{-}$(n1,g)

$\overset{\cdot}{d}^{-}$3 : n1 = n2 => $\overset{\cdot}{d}^{-}$(n1,add_arc(n2,n3,g)) = $\overset{\cdot}{d}^{-}$(n1,g)

$\overset{\cdot}{d}^{-}$4 : n1 = n3 => $\overset{\cdot}{d}^{-}$(n1,add_arc(n2,n3,g)) = $\overset{\cdot}{d}^{-}$(n1,g) +1

$\overset{\cdot}{d}^{-}$5 : and(n1 ≠ n2, n1 ≠ n3) => $\overset{\cdot}{d}^{-}$(n1,add_arc(n2,n3,g)) = $\overset{\cdot}{d}^{-}$(n1,g)

nb_nodes1 : nb_nodes(empty) = 0

nb_nodes2 : nb_nodes(add_node(n,g)) = nb_nodes(g) + 1

nb_nodes3 : nb_nodes(add_arc(n1,n2,g)) = nb_nodes(g)

pred_aux1 : n1 ≠ n2 => pred_aux(n1,k,add_node(n2,g)) = pred_aux(n1,k,g)

pred_aux2 : and(n1 = n3, k=1) => pred_aux(n1,k,add_arc(n2,n3,g)=n2

pred_aux3 : and(n1 = n3, 1<k≤$\overset{\cdot}{d}^{-}$(n1,g) = > pred_aux(n1,k,add_arc(n2,n3,g)) = pred_aux(n1,pred(k),g)

pred_aux4 : n1 ≠ n3 = > pred_aux(n1,k,add_arc(n2,n3,g)) = pred_aux(n1,k,g)

predi : predi(n,i,g) = pred_aux(n,$\overset{\cdot}{d}^{-}$(n,g)-i+1,g)

WHERE :

n,n1,n2,n3,n4 : node

i,k : integer

g : graph

END OR_GRAPH

Note that this specification is not complete; the implementation of the operation *obtain_node* is left to the specifier's choice.

In order to obtain the specification of a non oriented graph, it is sufficient to change in the OR_GRAPH specification *arc* by *edge* and d^- by d° ($d^\circ(x,g) = d^+(x,g) + d^-(x,g)$, with x, y nodes of g). Moreover, for every nodes x, y of g, we have : *exist_edge(x,y,g)* => *exist_edge(y,x,g)*.

The graph package.

The implementation given corresponds to the oriented graph data type. The representation is the usual adjacency list. The graph is implemented as an array of pointers to the list of adjacent nodes. We use the *p_array* type of the *sp_array* package, not shown here (see [Los]), for storing the pointers to the adjacency list of the nodes.

```
-- generic package for the graph data type
--
with text_io; use text_io;

with sp_array;

generic
 type node is private;

package oriented_graph is

type inter is range 0..20;

no_node, no_graph, no_arc, existing_node, parallel_arc, loop_ing,
no_pred, no_deg, too_big : exception;

type graph is private;

function empty return graph;
function empty_graph (g : graph) return boolean;
function exist_node (n : node; g : graph) return boolean;
function exist_arc (x, n : node; g : graph) return boolean;
function nb_nodes (g : graph) return integer;
procedure add_node (n:node; g : in out graph);
procedure add_arc (x, n : node; g : in out graph);
procedure rest_of_graph (n : node; g : in out graph);
function neg_degree (n : node; g : graph) return integer;
function obtain_node (g : graph) return node;
function pred (n : node; pi : integer; g : graph) return node;
```

```
private

type struct;

type ptr is access struct;
 type struct is
     record
       no : node;
       next : ptr;
     end record;

package tab_ptr is new sp_array (index => inter, elem => ptr);
use tab_ptr;

-- the graph is implemented as an array of pointers to the list
-- of adjacent nodes

package ind_io is new integer_io(inter);
use ind_io;

package int_io is new integer_io(integer);
use int_io;

type gra is new tab_ptr.p_array;

type graph is
    record
      occ : inter;
      graph_tab : gra;
    end record;

package tab_int is new sp_array (index => inter, elem => integer);

-- the table of negative degrees is implemented as an array of integers

type tab_degree is new tab_int.p_array;

end oriented_graph;

package body oriented_graph is

function empty return graph is
g : graph;

begin
 g.occ := 0;
 g.graph_tab := init(0,20);
 return g;
end empty;

function empty_graph (g:graph) return boolean is

begin
 if g.occ = 0  then return true;
```

```
        else return false;
 end if;
end empty_graph;

function exist_node (n:node; g:graph) return boolean is
i : inter;

begin
   if not empty_graph(g) then
   for i in lwb(g.graph_tab)+1..g.occ loop
    if n = acces(g.graph_tab,i).no then return true; end if;
   end loop;
   end if;
   return false;
end exist_node;

function exist_arc (x, n:node; g:graph) return boolean is
v : ptr;
i : inter;

begin
 if empty_graph(g) then return false; end if;

   for i in lwb(g.graph_tab)+1..g.occ loop
    if x = acces(g.graph_tab,i).no then
      begin
       v := acces(g.graph_tab,i);
       while v /= null loop
        if n = v.no then return true;
              else v := v.next;
        end if;
       end loop;
       return false;
      end;
    end if;
   end loop;

end exist_arc;

function nb_nodes (g:graph) return integer is

begin
 return inter'pos(g.occ);
end nb_nodes;

procedure add_node (n:node; g:in out graph) is
ub : inter;
v : ptr;
-- the node is added at the end of the array

begin
   if exist_node(n,g) then raise existing_node; end if;
   g.occ := g.occ+1;
   v := new struct;
```

```
    v.no := n;
    v.next := null;
    assign(g.graph_tab,g.occ,v);
  end add_node;

  procedure add_arc (x,n:node; g:in out graph) is
  i : inter;
  v, v1 : ptr;

  begin
    if empty_graph(g) then raise no_graph; end if;
    if x = n then raise loop_ing; end if;
    if exist_arc(x,n,g) then raise parallel_arc; end if;
    if not exist_node(x,g) then raise no_arc; end if;
    if not exist_node(n,g) then raise no_arc; end if;
    for i in lwb(g.graph_tab)+1..g.occ loop
     if x = acces(g.graph_tab,i).no then
      begin
       v := acces(g.graph_tab,i);
       while v.next /= null loop
        v := v.next;
       end loop;
       v1 := new struct;
       v1.no := n;
       v1.next := null;
       v.next:=v1;
       -- the successor is added at the end of the list
      end;
     end if;
    end loop;
  end add_arc;

  procedure rest_of_graph (n:node; g:in out graph) is
  i,k,j : inter;
  v : ptr;

  begin
    if not empty_graph(g) then
     if exist_node(n,g) then
      begin
       for i in lwb(g.graph_tab)+1..g.occ loop
        if n = acces(g.graph_tab,i).no then
         begin
          k := i;
          -- k contains the index of n in g
          v := acces(g.graph_tab,i).next; -- v points on the successor list
          if v /= null then
           begin
            while v /= null loop
             if v.no /= n then
                        v := v.next;
                 else
                  begin
                   put("v.no ");new_line;
```

```
                        if v.next.next /= null then
                                v.next := v.next.next;
                        -- the element is deleted
                                else
                        v := v.next;
                        end if;
                        exception
                         when constraint_error => v.next := null;
                       end;
              end if;
             end loop;
            end;
           end if;
          end;
         end if;
       end loop;
       -- actualization of the array
       g.occ := g.occ-1;
       for j in k..g.occ loop
        assign(g.graph_tab,j,acces(g.graph_tab,j+1));
       end loop;
       sub_array(g.graph_tab,lwb(g.graph_tab),g.occ);
      end;
    end if;
   end if;
   exception
    when illegal_sub_array => raise no_graph;
end rest_of_graph;


function neg_degree(n:node; g:graph) return integer is
n_deg : tab_degree;
cpt : integer;
v : ptr;
i : inter;

begin
 if empty_graph(g) then raise no_deg; end if;

 cpt := 0;

 for i in lwb(g.graph_tab)+1..g.occ loop
  v := acces(g.graph_tab,i);
  v := v.next;
  while v /= null loop
   if v.no = n then cpt := cpt +1;
   end if;
   v := v.next;
  end loop;
 end loop;
 return cpt;

end neg_degree;
```

```
function obtain_node (g:graph) return node is

-- the first element of the array whose negative degree is equal 0
-- is taken

begin
 if empty_graph(g) then raise no_node; end if;

 for i in lwb(g.graph_tab)+1..g.occ loop
  if neg_degree(acces(g.graph_tab,i).no,g) = 0 then return acces(g.graph_tab,i).no;
  end if;
 end loop;

end obtain_node;

function pred(n:node; pi:integer; g:graph) return node is
i : inter;
np : integer;
v : ptr;

begin
 if not empty_graph(g) then
  if exist_node(n,g) then
    begin
     if pi > neg_degree(n,g) or pi = 0 then raise no_pred; end if;
     np := 0;
     for i in lwb(g.graph_tab)+1..g.occ loop
      v := acces(g.graph_tab,i);
      v := v.next;
      while v /= null
          loop
           if v.no = n then
              begin
               np := np+1;
               if pi = np then return acces(g.graph_tab,i).no; end if;
              end;
           end if;
           v := v.next;
          end loop;
     end loop;
    end;
  end if;
 else raise no_pred;
 end if;
end pred;

end oriented_graph;
```