PUC

Series: Monographs in Computers Science
and Computer Applications

Nº 9/71

A MODELLING TECHNIQUE IN PROGRAMMING

by

Carlos J. P. de Lucena
and
Luiz F. de Almeida Cunha

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

A MODELLING TECHNIQUE IN PROGRAMMING

Carlos J. P. de Lucena
Associate Professor

Luiz F. de Almeida Cunha
Assistant Professor

Computer Science Department
PUC/RJ

This paper will be published in the Proceding of The Fifth Asilomar Conference on Circuits and Systems (San Francisco - November 1971).

Series Editor: Prof. A. L. Furtado December/1971.

ABSTRACT

This paper presents an approach for the implementation of the semantics of information structures. It is hoped that the facility described will be useful in the process of incorporating the existing formalisms for the description of information structures in programming languages. The system here discussed in operational in the form of a package of sub-programs written in FORTRAN and assembler language which can, with very small effort, be adapted to any existing computer system.

1. BASIC CONCEPTS

In this work a set of conceptual entities are defined which can be represented within a digital computer memory, and that can be put together to represent information structures.

These entities, or objects can be constructed by means of appropriate operators, called creators[1] can have their properties and components modified by others called updaters[1] and can be deleted by destroyers[1]. The properties of these objects can be detected by means of discriminators, and there are selectors which allow for the access of individual items[1].

Each of these objects is created in an environment called structural space of a program[2], and there it remains for use. Within a structural space of a program each object is identified by its name, and there are not two objects bearing the same name in that space. These names are non-empty strings of characters chosen from a particular alphabet.

A modular structure[2,3] is a model constructed by means of modules and connections, and one of its modules is distinct from the others from the point of view of some application. It is called the origin of this application of the structure. There is a certain class of these objects to which non-empty strings of characters chosen from a particular alphabet can be associated. They enable the representation of information related to the objects. A module includes an information nucleus and an associated ring of connections. Each connection establishes a link between two (possibly not distinct) modules.

2. MODULES AND CONNECTIONS

The information nucleus associated to a module includes three components: the name of a module, a string of characters that belongs to a particular alphabet, and a format describing an interpretation of the string. We shall name these items ALFA, BETA and GAMA, respectively.

ALFA is the name of the origin of an application of a modular structure and can be either a string or an empty string. The structure pointed to by ALFA is said to be internal or to be in a lower level with respect to the module that contains ALFA. If ALFA is an empty string, the module to which it corresponds is said to be a terminal module. Otherwise it is said to be a non-terminal module.

If a module is non-terminal, then BETA and GAMA for that module are simultaneously represented by means of empty strings. BETA and GAMA for terminal modules are non-empty strings of characters from two alphabets (not necessarily distinct) and are used to ultimately describe information.

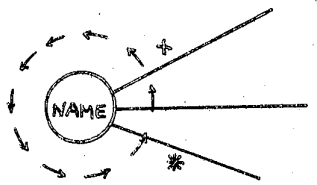
A connection establishes an interrelationship between two modules. It can be said that a connection has a module in each extreme and is individualized through its names. A connection may have one or all of the following characteristics: be oriented, be marked, have an associated modular structure. A connection is marked or has an associated modular structure if it possesses elements called MARK or FNAME respectively. FNAME is the name of a module. MARK is a feature for programming with the modular structure model.

A ring of connections is an ordered and non-empty set of connections in which one is chosen to be the first in the collection. The set has a wrap-around structure, in the sense that the n^{th} element preceeds the first-one.

The connections of a modular structure may belong to one or two rings of connections, a connection belongs to a single ring when it indicates a loop (a module linked to itself).

As an element of a ring of connections, a connection may have the following properties: occupy the first position in the ring and be ACTIVE. The first connection is called a HEAD connection. It is ACTIVE if it has an associated element called TAG. There is always one and only one connection which is active in a certain moment with respect to one ring of connections. It means that at any given moment there is only one connection that enables the access to another module of the structure. Connections can be ACTIVE or HEAD from the point of view of one or both of rings to which it is related.

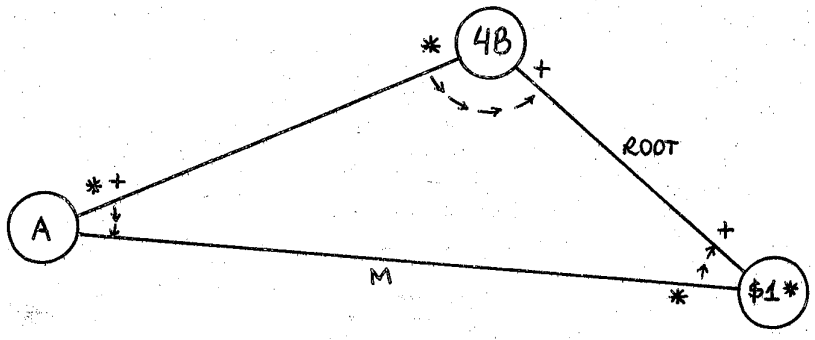
The first connection to a module is initialized with an activation TAG in the moment of its creation. TAGS can be moved forward and backward along the connections of the ring of connections of a module. These ideas can be resumed pictorially.



Picture 1

Picture 1 represents a module where * indicates the HEAD connection + the ACTIVE connection and NAME its name.

In Picture 2, the arrow indicates an oriented connection, the letter M the fact that the connection is marked and the name ROOT a modular structure associated with the connection. The elements ALFA, BETA, GAMA can be thought of as separate tables.



Picture 2

3. PROGRAMMING WITH THE MODULAR STRUCTURE

The concepts discussed before are encompassed in a package that can be utilized by means of the functions to be presented in sequel. These functions act as operators and discriminators that can be handled by the programmer for the construction of a programming system. As mentioned before, a modular structure is constructed by means of operators that are classified into three categories: creators updaters and destroyers.

Modules can be created in the system by means of the function DCLNCL (name 1) (declare nucleus) that declares a module of name

name 1. The functions STRINT (name 2, name 3) defines the component ALFA of the nucleus of the module name 2 as name 3. The non-application of STRINT to name 2 keeps its ALFA equal to an empty string.

The function DINFO (name 2, information, format) defines the components BETA and GAMA of name 2.

The organization of the modular structure in the structural space of a program is controled through a symbol table. In the case of the present system the symbol table that establishes the association between the name of a module and the pointer in the free storage list to the area occupied by the module is handled by means of a hashing[5] technique.

By their turn connections can be constructed by means of the function CONECT (name 7, name 8), that sets up a link between name 7 and name 8 disregarding a possible previous declaration of them.

The function DCLNCL provides for a module to be associated to any connections previously defined as linked to it. Attention is paid by the system to the order in which they were created. This order is kept in the ring of connections. The function EDGFNC (name 5, name 6, name 3) defines name 3 as the item FNAME of the connection linking name 5 and name 6. Function DIRECT (name 5, name 6) establishes an order in which name 5 and name 6 are to be considered with respect to the connection. Although this connection is said to go from name 5 to name 6 this constraint can be overridden while programming with the system.

We have been using until now the notation namei for the name of function parameters. The choice of this notation is related to the general properties that each of the function parameters have. These properties and their relation to the used notation can be summarized by the fol

lowing table.

	Appeared as argument of DCLNCL	Appeared as one of the arguments of CONECT
name 1	NO	YES / NO
name 2	YES	YES / NO
name 3	YES / NO	YES / NO
name 4	YES	YES

	One or both appeared as arguments of DCLNCL	Appeared together as the arguments of CONECT
name 5 and name 6	YES / NO	YES
name 7 and name 8	YES / NO	NO
name 9 and name 10	YES / NO	YES / NO
name 11 and name 12	YES	YES

Following the defined convention, we introduce now the update functions: ADVTAG (name 4) moves the TAG forward to the next connection in the ring of connections of name 4. If TAG is in the n^{th} connection it is moved to the first one in the ring: BCKTAG (name 4) produces the opposite

effect of ADVTAG; SETTAG (name 4) moves the TAG to the HEAD connection in the ring. When the module is created, this action in taken automati cally if there are connections already defined to link it to other modules in the structural space. If no connection was defined yet, this action is taken when the first one is created; MARK (name 4) associates a MARK to the ACTIVE connection of the ring of connections of name 4 . This MARK is not affected by the displacement of the TAG; UNMARK (name 4) removes the MARK of the ACTIVE connection.

Three updaters are defined that can be applied to connections, one of them reverses the order previously indicated for the module at the extremes of a connection, if any ordering had already been established. Another simply undoes the ordering, if any, and finally one substitutes the FNAME component of the connection, if any, by an empty string. The corresponding functions are respectively CHDIRE (name 5, name 6), UNDICT (name 5, name 6) and NOFUNC (name 5, name 6).

Complementing the set of constructors that act on modules, two destroyers were defined. Their names are RESET (name 2) and DELETE (name 2). RESET (name 2) deletes all the connections linking with name 2 from the structural space of a program. The rings of connections of the modules linked to name 2 will be rearranged automatically, and if a HEAD connection is destroyed, the next in the ring will take its pla ce. Besides, if one of the connections whose deletion was indicated is ACTIVE from the point of view of some module, an error condition oc - curs, and the action of RESET is not completed. DELETE (name 2)acts in much the same way as RESET, but includes the deletion of name 2.

Just one destroyer is defined for connections, whose name is DISCON (name 5, name 6). It eliminates the connection linking name 5 and name 6, from the structural space. The same sort of care taken

in RESET is applied here.

As it was mentioned before, the access to individual items in a structural space of a program is achieved by means of selectors. Selectors are defined on modules and connections. For modules, four of them were defined: INTSTR (name 2). They allow respectively for the access to the name of the origin of the modular structure internal to name 2, to the number of connections linking name 2, to the components BETA and GAMA of name 2, and to the module linked to name 4 by an ACTIVE connection.

For connections the only function defined is called EDGEF (name 5, name 6). It enables the access to the component FNAME of the connection.

To detect the properties of particular modules or connections in a structural space a set of discriminators or predicates are defined. There are five of them defined for modules and five for connections. The predicates for modules are the following: DECLED (name 3)which is TRUE if name 3 was previously declared; TERMIN (name 4) TRUE if the HEAD connection is terminal; HEADER (name 4) TRUE if the HEAD connection is ACTIVE; INCIDE (name 4) is TRUE if the ACTIVE connection is not directed or if in case it is directed name 4 is the second element of the ordered pair; MARKED (name 4) is TRUE if the ACTIVE connection has a MARK. For connections, these predicates are: CONNEX (name 9, name 10) is TRUE if there is a connection linking name 9 and name 10 ; TAGGED (name 11, name 12) is TRUE if the connection is ACTIVE from the point of view of one or both of the modules linked through it; HEADCN (name 11, name 12) same as the preceeding for a HEAD mark; ORIENT (name 5 , name 6) is TRUE is name 5 and name 6 are in this order; MARKEX (name 11, name 12) is TRUE if the connection has MARK.

4. COMMENTS

Experience with the system has shownthat is enables a very comfortable construction of information structure models. The order in which the information in the model is to be retrieved is immaterial during its construction phase and the manipulation of pointers is eliminated. Also, traversing modular structure models is a fairly easy task, as illustrated by the example it the annex. It was probably transparent from the description of the system that its greatest advantages are felt when complex structures are being modelled. It is in this case that it achieves its maximun efficiency as its obvious overheads are easily paid. Interpreters[6] of all kinds can be fairly simply built by means of the method discussed. It is due to the possibility of associating predicates of any complexity to connections (FNAME) generating in this form an interpreted graph.

The reason for designing the system as a set of functions was to make it useful to act as a low-level programming language for the semantics of existing syntatic definitions[7,8,9] of information structures. Languages for graph processing can also be based in the proposed mechanisms[10].

5. REFERENCES

(1). LASKI, J. "The Morphology of PREX - An Essay in Meta-Algorithms" in "Machine Intelligence 3" Edited by Donald Michie, American Elsevier - 1968.

(2) LUCENA, C. J. P "An Approach for Mapping Abstract Information Structures on Digital Computers Memories" IEEE International Conference

(3) ALMEIDA CUNHA, L. F. "A General Method for Representing Information Structures: Modular Structure" Dept. Informatica - PUC/RJ - April - 1971.

(4) ROSS, D. T. "Data Structure and Storage Management" The Newcastle University - IBM Seminar in Computer Science - December - 1968.

(5) MAURER, W. D. "Scatter Storage Techniques" Comm. ACM 11,1 - 1968.

(6) WEGNER, P. "Data Structure Models for Programming Languages" in "Data Structures in Programming Languages" Edited by Julius T. Tou and P. Wegner University of Florida - 1971.

(7) STANDISH, T. A. "A Data Definition Facility for Programming Languages" Carnegie Institute of Technology - May - 1967.

(8) van der MEULEN, S. G. and LINDSEY, C. H. "Informal Introduction to Algol 68" Mathematisch Centrum, Amsterdam - 1970.

(9) EARLEY, J. "Forward an Understanding of Data Structures"
Notes of course 200 A, Computer Science Dept. University of California - Berkeley - 1969.

(10) CRESPI-REGHIZZI, S. and MORPURGO, A. "A Language for Treating Graphs"
Comm. ACM 13, 5 - 1970.

APPENDIX

```
C----SAMPLE PRØGRAM
C----GIVEN A DIGRAPH REPRESENTING
C----ITS VERTICES BY SIMBØLS WITH
C----UP TO FIVE ALPHAMERIC CHARACTERS PRINTS
C----EVERY PATH BETWEEN TWØ GIVEN VERTICES
      LOGICAL INCIDE, HEADER, MARKED
      INTEGER ADJACT, A, B, P, C(10)
      DATA IBR/1H /
C----SETS UP LIST OF AVAILABLE SPACE.
     CALL LSPACE
C----READS THE NAMES OF TWO VERTICES.
     CALL READ (INIC, IEND)
     CALL DCLNCL(INIC)
     CALL DCLNCL(IEND)
C----READS THE PAIRS OF ADJACENT
C----VERTICES (ARRAØWS).
    1 CALL READ (I1, I2)
      IF (I1 .EQ. IBR) GØ TØ 2
      CALL CONECT (I1, I2)
      CALL DIRECT (I1, I2)
      GØ TØ 1
C----DECLARES THE MØDULES REPRESENTING
C----VERTICES.
    2 CALL READ(I1)
      IF(I1 .EQ. IBR) GØ TØ 3
      CALL DCLNCL(I1)
      GØ TØ 2
C----
C----TRAVERSES PATHS BETWEEN VERTICES
C----INIC AND IEND
```

```
 3 A = INIC
 4 IF(.NØT. INCIDE(A)) GØ TØ 6
 5 CALL ADVTAG(A)
   IF(.NOT. HEADER(A)) GØ TØ 4
   STØP
 6 CALL MARK(A)
   B = ADJACT(A)
   IF(B .EQ. IEND) GØ TØ 12
 7 IF(.NØT. INCIDE(B)) GØ TØ 11
 8 CALL ADVTAG(B)
   IF(.NØT. HEADER(B)) GØ TØ 7
 9 IF(MARKED(B)) GØ TØ 10
   CALL BCKTAG(B)
   GØ TØ 9
10 CALL UNMARK(B)
   P = B
   B = ADJACT(B)
   CALL SETTAG(P)
   IF(B .EQ. INIC) GØ TØ 5
   GØ TØ 8
11 IF(ADJACT(B) .EQ. IEND) GØ TØ 12
   IF(MARKED(ADJACT(B))) GØ TØ 8
   CALL MARK(B)
   B ADJACT(B)
   GØ TØ 7
12 P = INIC
   I = 0
13 I = I + 1
   C(I) = P
   IF(P .EQ. IEND) GØ TØ 14
   P = ADJACT(P)
   GØ TØ 13
14 CALL PRINT(C, I)
   GØ TØ 8
   END
```