International Institute for
Applied Systems Analysis
IIASA www.iiasa.ac.at

Dilos Reference Manual: Part I

Briabrin, V.M.

IIASA Research Memorandum
July 1976

Briabrin, V.M. (1976) Dilos Reference Manual: Part I. IIASA Research Memorandum. IIASA, Laxenburg, Austria, RM-76-052 Copyright © July 1976 by the author(s). http://pure.iiasa.ac.at/632/ All rights reserved. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage. All copies must bear this notice and the full citation on the first page. For other purposes, to republish, to post on servers or to redistribute to lists, permission must be sought by contacting repository@iiasa.ac.at

RM-76-52

DILOS REFERENCE MANUAL: PART I

V.M. Briabrin

July 1976

Research Memoranda are interim reports on research being conducted by the International Institute for Applied Systems Analysis, and as such receive only limited scientific review. Views or opinions contained herein do not necessarily represent those of the Institute or of the National Member Organizations supporting the Institute.

ABSTRACT

This paper decribes the special programming system intended for the Data Base Management. The system is written in the LISP language, and provides manipulation of objects that represent specific knowledge on applied problem areas. Three basic functions are provided by the system:

1) Information storage and retrieval, using an associative search mechanism;
2) Initiating application programs and extracting (filling) of application data sets;
3) Organization of logical control for particular problem solving processes.

The system will be used as an intelligent interprocessor between the end-users, and the application programs and data sets. A linguistic processor and deductive inference mechanism will extend the sytem's and the end users' capabilities, providing natural language access, automatic planning, the realization of problem-solving algorithms, etc.

PREFACE

From the outsider's point of view, the first years of IIASA's activities in most research areas were devoted to theoretical studies, to the establishment of international contacts, and to the search for sources of appropriate information. Some mathematical models were developed and implemented in the form of computer programs. However, until now these models and their implementation remained separate entities unable to communicate with one another except by being passed physically.

A new period seems to have arrived, when an appeal for integrating different models and data sets becomes essential. This integration should appear both in the mental activities of IIASA scientists as well as in theoretical and programming products developed by them.

These considerations, together with the general trend in systems analysis and computer science, led the Director of IIASA and the Computer Science Project to initiate a new task: the installation of a Data Base Management System (DBMS) on IIASA's PDP 11/45 computer. As a first step, the Computer Science Project organized in June 1975 the Workshop on the Question-Answering Systems. The Workshop discussed several proposals for IIASA-DBMS. Since then, members of the Computer Science Project carried out a study of available DBMS and the practical development of intelligent DBMS was started in IIASA's National Member Organizations (NMOs). The participants at the Workshop agreed that the system should be developed by some NMOs, for example, the USSR, the GDR, the USA, and Italy, with the possibility of eventually extending it toward a sophisticated, artificial intelligence programming complex.

This paper describes a part of the Dialog Information Logical System (DILOS) being developed at the Computing Center of the Academy of Sciences in Moscow. It was programmed in LISP language for BESM-6, the basic Soviet scientific computer. The transfer of LISP programs from one computer to another is a technical task; therefore, the implementation of DILOS on the

PDP-11/45 was done with relatively little manpower: the author accomplished this during a two-month visit to IIASA.

Part I of the DILOS Reference Manual contains instructional information relating to the use of DILOS in a UNIX environment for the PDP-11/45.

It is planned to transfer the remainder of the system during 1976-1977, and to produce additional publications concerning:

1) Recommendations to systems analysts for creating problem-oriented data bases;
2) Information about system structure and maintenance;
3) Description of a linguistic processor, to be attached as a front-end part of DILOS.

I would like to express my gratitude to my colleagues in Moscow who have participated in this system's development, in particular, Dimitrii Pospelov, Grigorii Senin, Vladimir Abramov, Evgenii Veselov, Ludmila Litvinzeva, and Vladimir Masurik. I am also grateful to those at IIASA who helped me with my work in the computer section and in the preparation of this manual; in particular, Jim Curry, Bernhard Schweeger, and Mark Pearson. A series of discussions with many other IIASA scientists in the Computer Science, the Water Resources and the Energy Projects influenced and encouraged my work at IIASA and contributed to the successful installation of the system.

DILOS REFERENCE MANUAL: PART I

I. HOW TO GET IN AND OUT

DILOS is a LISP-program running under the UNIX operating system for PDP 11/45 computers. All UNIX facilities are available to the user outside DILOS. Later, we shall see that it is easy to "freeze" DILOS almost at any point, and to "dive" into the UNIX-SHELL for purposes of some extra editing, fortran-compiling, etc., and then to bubble back to DILOS, restoring its current state.

To better understand the system, the reader is recommended to study LISP, a very powerful and elegant programming language now being widely used for artificial intelligence research. Language descriptions are given in Weissman (1967), and Teitelman et al. (1974); a manual for running LISP on the PDP 11 is presented in Howard (1975).

The following sequence of commands brings the user inside DILOS at the initial stage; the system's prompts and replies have been omitted in the following examples:

Example 1.1

:login: victor	Login with the account "victor"; after this action the user is under UNIX control.
@ L110	Call L110 which is Harvard-LISP implementation for PDP 11 (see Howard, 1975).
AM110 LISP 5/25/75	
⇒ (load 'dilos) nil	The file with DILOS programs is opened, and all function definitions are evaluated.
⇒ (start) (dilos is ready)	This function performs some preparatory actions, after which the user can start normal interaction with DILOS.

Function "start" can have an argument, handles as a *calling pattern* for some application area.

Example 1.2

(start "development game")	Besides normal (start) operation, this call initiates a special function associated with the "development game" pattern. An error message will be returned to the user's terminal if a wrong calling pattern is used.

After starting, a long session of user's conversation with DILOS can take place until the user decides to quit. There are two functions available for this purpose: (stop), and (exit).

Example 1.3

(stop)	This function closes all opened files, preserving the changes that were made to data base objects during the session. The protocol of the session is automatically written to the file "L110 protocol" in the current directory. Control is returned to UNIX.
(exit)	This function does not make file closing and protocol preservation, but returns control to UNIX. If you want to preserve the protocol then call (unprotocol) before (exit).

In Examples 1.1 to 1.3, the user's and some of the system's typing on the terminal are given in lower case letters as in actual terminal sessions. In other examples given in this manual, capital letters are used to represent actual texts to be typed or stored in the data base, whereas lower case letters represent metavariables. After coming back to UNIX, you can do many other things, as for example, listing the contents of your files updated by DILOS. For final quit, type "bye" command and check that UNIX prompts with :login: for another user.

II. RECOVERY FROM ERRORS

List of recovery actions:

DEL	delete one character from unfinished line;
↑U	delete current line;
]	compensate all "(" in the current message;
(reset)	Restore from L110 error message;
↑O	suppress output;
↑O	suppress DILOS-LISP execution.

If one made a mistake in typing but noticed it before pushing "cr" (carriage return), then "DEL" and "↑U" (CTRL - U) are used as elsewhere in UNIX for character and line deletion.

If one typed in several lines, but there is no reaction from DILOS, then the general explanation is that DILOS considers the message unfinished, for example, because there are not enough ")" to compensate "(" in your text. The simplest way to obtain an explanation is to type "]", which compensates for all still unbalanced "(". It can work for the best but an error could also emerge because of an inconsistency somewhere in the message.

An error message from L110 will give some hint as to the reason for the mistake. It is followed by the prompt : > . The easiest way to recover is to call (reset), after which should be repeated the enquiry more carefully.

"↑0" and "↑c suppress output and DILOS-LISP executions, respectively. It could be used when there is a suspicion of an endless loop in some function, but it is unlikely that such a loop can be created if a "structured programming" style, encouraged by LISP, is used for function definitions.

III. UNIX FILES AND MDB-OBJECTS

All information related to DILOS functioning as well as to problem-oriented data base contents is stored in the ordinary UNIX files, which reside in one or several directories.

DILOS is oriented toward the manipulation of Model Data Bases (MDB) which could represent complicated hierarchical structures; it is obvious that a special representation should be chosen for MDB-objects.

The general structure of MDB-objects is:

```
(objname (p1  v1  ...  pk  vk))
```

where *objname* atom (identifier, number or any double quoted set of characters) represents the name of the object in some file. All objects in the same file have different names.

The pairs {p1 v1}, {p2 v2}, ...,{pk vk} represent *properties*, where p1,...,pk are *indicators* and v1,...,vk are *property values*, which could be "terminal", or in their turn, have some structure.

<u>Example 3.1</u>

```
("CYBER-74" ( * COMPUTER
              OS "SCOPE 3.4.3"
             SOFTWARE  (LANGUAGES (, ALGOL  FORTRAN  COBOL
                       BASIC)  "APPL-PACKAGES"  (,  MATH-LIB
                       CERN-LIB  BMD  EISPACK))
             HARDWARE  (MEM "98K"
                       WSIZE "60B"
                       TAPES ("9TR"  4  "7TR"  1)
                       PAP-TAPE  1)
               ACCESS  (BATCH  "REM-JOB-ENTRY"
                       INTER "12 TERMINALS")))  .
```

In this example, object "CYBER-74" at the topmost level has five properties with indicators: *; OS; SOFTWARE; HARDWARE; and ACCESS. The values of * and OS properties are *atoms*, whereas other properties have *structured values* represented by lists. For example, the property HARDWARE has four *subproperties* with indicators: MEM; WSIZE; TAPES; and PAP-TAPE. The property TAPES in turn, have two sub-properties: "9TR"; and "7TR". Thus an arbitrary hierarchy of sub...subproperties could be represented in this way.

Property values could be represented as follows:

atom: COMPUTER; "SCOPE 3.4.3;" "98K;" 4; "12 TERMINALS";

set: (, ALGOL FORTRAN COBOL BASIC); or

list: to be considered as a sub-structure or as a terminal value.

The *set-value* (represented by a list with a comma-sign as a first element) contains one or more *partial values* and it is subject to set operations: union; intersection; difference.

The *list-value* could be handled in the same way as an atomic terminal value, if it is not considered a substructure. Interpretation of the list-value depends on the external program (or on the user) manipulating the given object.

Example 3.2

```
("MATRIX 1"      (1  (95  120   0)
                  2  (70    0  790)
                  3  (0    35    1)))  .
```

In this example, object "MATRIX 1" has three properties with numerical indicators 1, 2, 3, and list-values containing numbers.

Two additional property values proved to be useful in different applications:

range-value;

executable-value.

Example 3.3

```
(CITY (POPUL (:  0.2  20.0)
       AREA (:   10  1000)
    DENSITY (+  (QUOTIENT  POPUL  AREA))  ))   .
```

In this example, object CITY has properties POPUL and AREA, with the corresponding range-values: (POPUL min = 0.2 , POPUL max = 20.0); and (AREA min = 10 , AREA max = 1000). Thus a property with a *range-value* is supposed to have an *actual numerical value* varying from the given minimum to the given maximum.

Property DENSITY in our example has an executable-value (+ (QUOTIENT POPUL AREA)) implying execution of function QUOTIENT with the arguments, POPUL and AREA.

Both types of property values are represented by the lists:

(: min max); and

(+ function-call).

All MDB-objects are stored in UNIX files in textual form, even if they have numerical property values. It provides ease of file (and MDB contents), amendment, independence from the hardware, availability of symbol manipulation procedures, etc.

If an application program requires binary representation of data, then a special transformation procedure should be invoked after the extraction of textual data and before passing it to application program.

IV. INTERPRETATION OF MDB-OBJECTS

There are three basic interpretations of MDB-objects that could be considered:

A. Containers of numerical or textual data, to be manipulated by the user or by applied programs;

B. Descriptors of other objects allocated elsewhere in the data base;

C. Process descriptors (theorems) defining the sequences of actions to be performed by the system when certain circumstances occur.

We shall refer to these categories as A-objects, B-objects and C-objects and, in the following sections, consider their representation in MDB as well as the basic functions applicable to their manipulation.

Our first concern is with A-objects, considered as carriers of applied data and processed by DILOS functions analogous to those available in conventional DBMSs. The basic functions are:

FIND - for searching and extracting objects and/or their properties;

ADD - for putting new objects or adding (changing) their properties;

DEL - for deleting objects or their properties.

The examples given below illustrate usage of the function, and are followed by formalized syntax description; however, for simplicity, I do not pretend to give exact syntax definitions. Each of the examples begins with a verbal expression representing the meaning of the user's or the applied program's inquiry. It should be noted here that another part of the DILOS system (now in the debugging stage) will perform a linguistic translation of the user's natural language expressions into the formal expressions decribed on the following page.

V. FUNCTION OPEN

After the user has accomplished the initial actions (illustrated in Examples 1.1 and 1.2), he comes into contact with the *current MDB division* which usually corresponds to the same UNIX file. If the "start" function was called without an argument (Example 1.1), then the current division is called ROOT. Otherwise, the system could open another division implied by the processing of the calling pattern (Example 1.2). Changing the current division name is performed by the function: (OPEN 'divname). In most cases, this is implicitly evaluated by the system, although the user could make an explicit call of this function at any moment. Besides establishing the current division name, this function transfers the contents of the UNIX file into the LISP-core memory, if not done so earlier.

Occasionally, and on the execution of the "stop" function (Example 1.3), the system closes opened files, which means transferring their core images onto the disk. File transference in both directions is somewhat time consuming and can create unexpected effects (for example, lack of core space).

The experience and the means to overcome the negative effects should be derived from a wider use of this system.

VI. FUNCTION FIND

We now consider a series of examples which illustrate searching in MDB and extracting objects and/or their properties. The functions could be called directly by the user typing on the terminal, or indirectly, through the execution of other functions, as we shall see later.

Example 6.1

"Find all objects in COMPUTER division"

(a) (FIND *COMPUTER)

(b) (FIND *COMPUTER =X) .

The system opens COMPUTER division and prints out *all* the objects that have the property {* COMPUTER} as the one given in Example 3.1.

In the case of Example 6.1 (b), the system also creates a *pattern variable* X which is bound to a list of object names. Any identifier can be used instead of X.

Example 6.2

"Find all objects in the current division"

(a) (FIND ALL)

(b) (FIND =X) .

These functions have the same effect as in Examples 6.1 (a) and 6.1 (b), except that no opening takes place and the action is performed over the current division.

The value of the pattern variable X could be used later in another FIND, ADD or DEL function by typing +X ("value of X"). It could also be used in any regular LISP expression as a normal variable name.

Example 6.3

"Find an object PDP-11/45 in COMPUTER division"

(FIND *COMPUTER PDP-11/45) .

Only one object is looked for and printed out. This object should have an internal representation:

(PDP-11/45 (* COMPUTER other properties)) .

Example 6.4

"Find an object PDP-11/45 in the current division"

(FIND PDP-11/45) .

The result of this action is obvious.

Example 6.5

"Find all objects in COMPUTER division and extract their SOFTWARE and HARDWARE properties"

(a) (FIND *COMPUTER : SOFTWARE ; HARDWARE)

(b) (FIND *COMPUTER =X : SOFTWARE =Y ; HARDWARE =Z) .

The system acts as in Example 6.1, but only the values of SOFTWARE and HARDWARE properties are extracted and printed out. Pattern variables X, Y, Z in Example 6.1(b) are bound to

corresponding lists: X becomes a list of object names,
Y -- a list of SOFTWARE values; Z -- a list of HARDWARE values.

Example 6.6

"Find an object PDP-11/45 in the current division and extract the value for SOFTWARE/LANGUAGES property"

(a) (FIND PDP-11/45 : (SOFTWARE LANGUAGES))

(b) (FIND PDP-11/45 : (SOFTWARE LANGUAGES) =X) .

In this case, the division name is omitted, but it could be used as in Example 6.3.

The compound property name is represented by a list (SOFTWARE LANGUAGES) that could have more elements if a deeper subproperty structure is assumed.

Example 6.7

"Find all objects in the current division which have ALGOL as a value of SOFTWARE/LANGUAGES property, and extract the values of ACCESS and HARDWARE properties".

(a) (FIND ALL : (SOFTWARE LANGUAGES) ALGOL ; ACCESS ; HARDWARE)

(b) (FIND =X : (SCFTWARE LANGUAGES) ALGOL ; ACCESS =Y ; HARDWARE)

This enquiry implies a complicated search -- only those objects that contain ALGOL in its SOFTWARE/LANGUAGES property value are considered, and from those objects ACCESS and HARDWARE property values are extracted and printed out.

An object in Example 3.1 could satisfy the given search criteria because it has a set-value (, ALGOL FORTRAN COBOL BASIC) under the property SOFTWARE/LANGUAGES. Now, a general syntax of FIND expression could be presented:

<find expr> :: = (FIND <pattern>)
<pattern> :: = <heading> | <heading> : <body>
<heading> :: = ALL | <divreference> <objreference>
<divreference> :: = *<divname> | <empty>
<objreference> :: = <objname> | =<pattvar> | +<pattvar> | <empty>
<body> :: = <propertyreference> | <propertyreference> ; <body>
<propertyreference> :: = <indicator> | <indicator> =<pattvar>
| <indicator> +<pattvar> | <indicator>
value>
<indicator> :: = <atom> | <listofatoms> .

Few comments could be made about this syntax.

A <pattern> consists of a <heading>, or a <heading> and a <body>, separated by a column (compare Examples 6.1 to 6.4 with 6.5 to 6.7).

A <heading> could be a special atom ALL implying searching for all objects in the current division. It could also consist of <divreference> and <objreference>, each of which could be empty. Besides atomic <objname>, and <objreference> could be represented by =<pattvar> which has the same effect on the search as an empty <objname>. +<pattvar> is the same as an <objname> = value [<pattvar>] .

A <body>, if present, consists of one or more <property-references> separated by semicolons. Each of the <property-reference> contains an <indicator> and possibly a =<pattvar>, or +<pattvar>, or <value>; <divname> and <objname> are atoms; <pattvar> could be any identifier; <value> is an atom or a list with appropriate interpretation (Examples 3.1 to 3.3).

VII. FUNCTIONS ADD AND DEL

The syntax of the ADD and DEL expressions is almost the same as that of the FIND expression except for some natural restrictions.

Example 7.1

"Add new object SOLAR with the properties PROD 370 , CONSUMP 120 into the EN-SRS division"

(ADD *EN-SRS SOLAR : PROD 370 ; CONSUMP 120) .

A new object appears in the EN-SRS division if it did not exist earlier and if it has the following representation:

(SOLAR (*EN-SRS PROD 370 CONSUMP 120)) .

Example 7.2

"Put the value 930 under the property RESOURCE/EXPLORED in the object GAS of the current division"

(ADD GAS : (RESOURCE EXPLORED) 930) .

The possible result of this action consists of adding the new property to the existing object GAS so that the latter will have

the following structure:

```
(GAS ( ...  RESOURCE ( ... EXPLORED  930  ... ) ...))  .
```

Pattern variables of the form =X cannot be used in ADD expressions because it is meaningless to add something undefined to the data base. On the other hand, *variable values* of the form +X could be used more frequently here than in FIND expressions assuming that the added value is found in some previous action. *Executable* property values are also used more naturally in ADD expressions.

Example 7.3

"Find PROD property values in all objects of EN-SRS division; put the arithmetic sum of these values under the property PROD into the object TOTAL"

```
(FIND  *EN-SRS : PROD =X)

(ADD  TOTAL : PROD (+(SUM X))  .
```

The first expression provides searching in EN-SRS division for all objects containing PROD property. The list of extracted values becomes assigned to X variable. The second expression puts a new property value to the object TOTAL under the PROD indicator. This value is a *result of evaluation* of the LISP expression: (SUM X), which assumes applying PLUS operation to all numbers contained in the X list.

Example 7.4

"Delete COBOL from the property SOFTWARE/LANGUAGES of the object CYBER-74"

```
(DEL  CYBER-74 : (SOFTWARE LANGUAGES) COBOL)  .
```

If this expression is applied to the object CYBER-74 from Example 3.1, then its original set-value (, ALGOL FORTRAN COBOL BASIC) becomes (, ALGOL FORTRAN BASIC, i.e., COBOL value is deleted from the set.

VIII. MDB-OBJECTS USED AS DESCRIPTORS OF OTHER OBJECTS

Functions FIND,ADD, and DEL are intended first of all for the manipulation of A-objects, although they are also applicable to B- and C-objects, (see Section IV). We now consider B-objects in more detail.

It is useful to distinguish between several kinds of B-objects. At present, we are interested in the following:

(1) Descriptors of MDB-object types;

(2) Descriptors of applied program modules;

(3) Descriptors of applied data modules.

Example 8.1

Consider the structure of an object from Example 3.1. It could be represented graphically as shown in Figure 1.

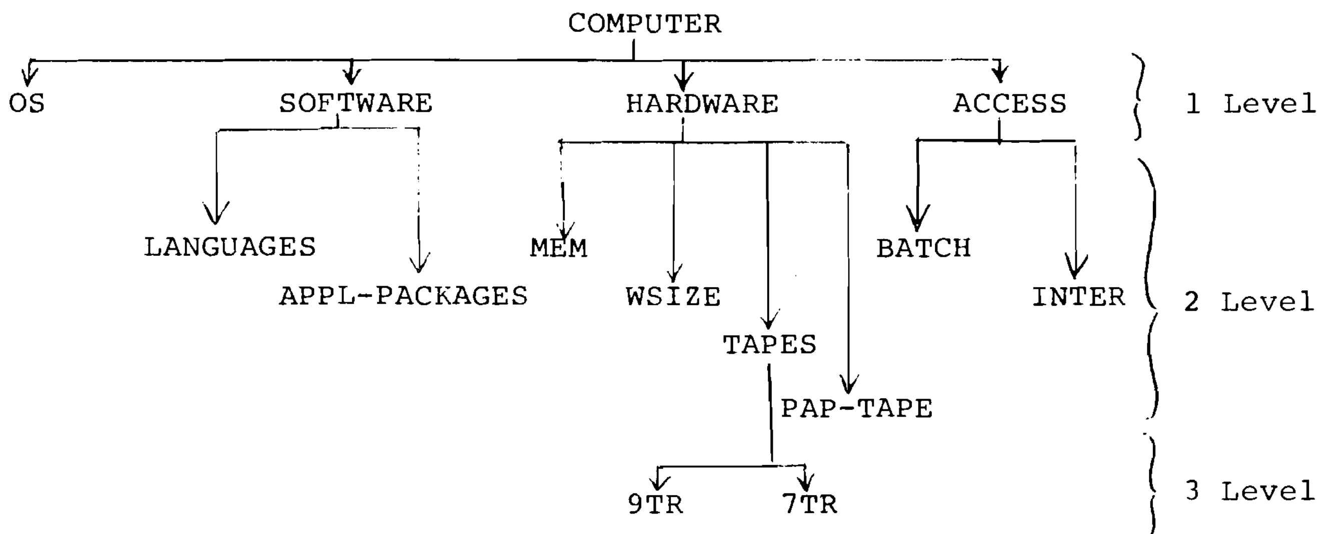

Figure 1.

The corresponding *type descriptor* has the following representation:

```
(COMPUTER  (IS   BC
            BP   (OS  ()
```

```
SOFTWARE (LANGUAGES  ()
          "APPL-PACKAGES" ())
HARDWARE (MEM  ()
          WSIZE  ()
          TAPES ("9TR"  ()
                 "7TR"  ())
          PAP-TAPE  ())
  ACCESS (BATCH  ()
          INTER  () ))))  .
```

The first property {IS BC} says that this object "is a basic concept", and the second property {BP (...)} contains, under the BP ("Basic Properties") indicator, the description of the whole structure. If such an object appears in MDB, then more concise representation could be chosen for the object from Example 3.1 as well as for other similar objects:

```
("CYBER 74" (*  COMPUTER
            VAL  ("SCOPE 3.4.3"
                  ((,  ALGOL  FORTRAN  COBOL  BASIC)
                   (,  MATH-LIB  CERN-LIB  BMD  EISPACK))
                   ("98K"  "60B"  (4  1)  1)
                   ("REM-JOB-ENTRY"  "12 TERMINALS"))))  .
```

Most of the text in this representation is concerned with property *values* and not with the *indicators*. Only brackets would be considered redundant characters, and they are needed to preserve the object *structure*. The first property, {* COMPUTER}, remains a pointer to the "superconcept", which allows normal usage of functions FIND, ADD and DEL as in Example 6.1 to 7.4.

Any object can actually be stored in a *normal* or in a *typified* representation (Examples 3.1 versus 8.1). The system tries to handle each object as a normal one and, if there is no success (no required properties), then the system switches to handling the object as a typified one.

Each of () in the type descriptor above could have been substituted by a *value-descriptor* which can be a value (for example, a specific number or text), or a predicate function that determines some restriction over the corresponding sub... sub-property value.

Example 8.2

Consider an object that will be used as a *program module descriptor:*

```
(DEV-GAME   (*  PM
             SYS (PDP-11/45  FORTRAN)
             LOC (VALASEK    GO       )
             ARG (PHASE-VAR  STEP-NUMB  CNTRL-VAR)
             RES (VALUES  (INT  2)  COMMENT  (CHA  22))
             INP  W1
             OUT  W2  ))  .
```

The first property {* PM} serves as an indicator that this is a "program module descriptor". Properties {SYS ...} and {LOC ...} indicate that the corresponding program is written in FORTRAN for the PDP-11/45 computer, and that it is located under the name GO in the VALASEK directory. Note that program name GO does not necessarily coincide with the pm-descriptor name DEV-GAME.

Property {ARG ...} introduces the argument variables PHASE-VAR, STEP-NUMB, CNTRL-VAR. Each of these variables has to be bound to appropriate values prior to program execution. The system will collect all the values in the input file W1, as indicated by the property {INP W1}.

During execution, the GO program will read the necessary values from the input file, and return results to the output file W2 from where they will be extracted by the system and assigned to result variables VALUES (2 integers) and COMMENT (22 characters).

Thus, the system handles the applied program module in three stages, as illustrated in Figure 2:

(1) Initialization, i.e. preparation of input file from argument variables;
(2) Execution, i.e. input/output transformation; and
(3) Finalization, i.e. separation of output file into result variables.

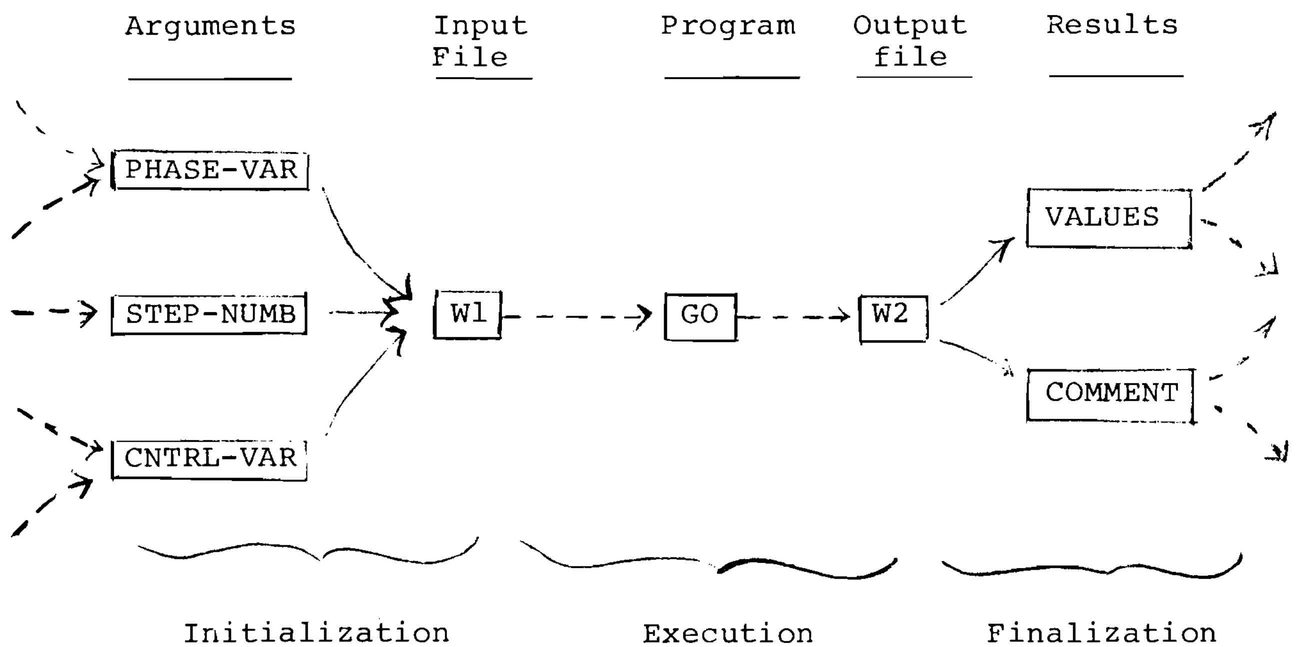

Figure 2.

Special functions in DILOS provide appropriate actions:

(INIT pm-descriptor)
(RUN pm-descriptor)
(FINISH pm-descriptor) .

In the next section, an appropriate example will be presented, providing more illustrations of the manipulation of arguments and results.

It should be noted that the process described here could be applied to the program residing in *any computer* connected to the PDP-11/45 through a network. The necessary provision includes *interface processors*--special programs on the PDP-11/45 and connected computers--which allow file transferring and program execution.

Example 8.3

The last type of B-object to be considered is a *data module descriptor*.

```
(HM-COEFF  (*  DM
            SYS (PDP-11/45 TEXT)
            LOC  (LEO  COEFF)))  .
```

The property {* DM} serves to indicate that this is a "data module descriptor". Properties {SYS ...} and {LOC ...} play the same role as in the previous example, except that the reserved identifier TEXT shows that the data are stored in *textual* form which is the normal representation for applied data. BIN says that it is in a binary form; in this case, DILOS does not have any responsibility for its compatibility with other data participating in the process.

Data modules are used to contain large amounts of unstructured data, for instance, matrixes of numerical coefficients. As opposed to A-objects contained in MDB, data modules cannot be recognized by associative search, i.e. functions FIND, ADD and DEL are inapplicable to data modules themselves, although applicable to dm-descriptors. In most cases, it is the responsibility of applied programs to separate data modules into elements, or to generate them by appropriate algorithms.

Since the main purpose of data modules is to pass to and from program modules, DILOS provides binding of argument and result variables to data modules contents. This is accomplished by means of special functions:

(EXTR dm-descriptor arg-variable) - extract contents of data module and make it a value of the argument-variable.

(FILL dm-descriptor res-variable) - take the value of the result-variable and pass it to data module.

Concatenation of data module contents and other amending operations should be performed outside DILOS by appropriate commands.

In conclusion, it is worth stressing again that the introduction of special descriptors in MDB allows the system analysts to construct the program and data complexes from the modules allocated on different computers. This approach leads to the idea of distinguishing between two parts of the data base: *model data base* (MDB), containing the structured knowledge of problem domains; and *conventional data base* (CDB), represented by normal file systems on different computers that contain the terminal data and calculating procedures.

The complete picture of interactions between the end-user, MDB and CDB in this case, is illustrated in Figure 3:

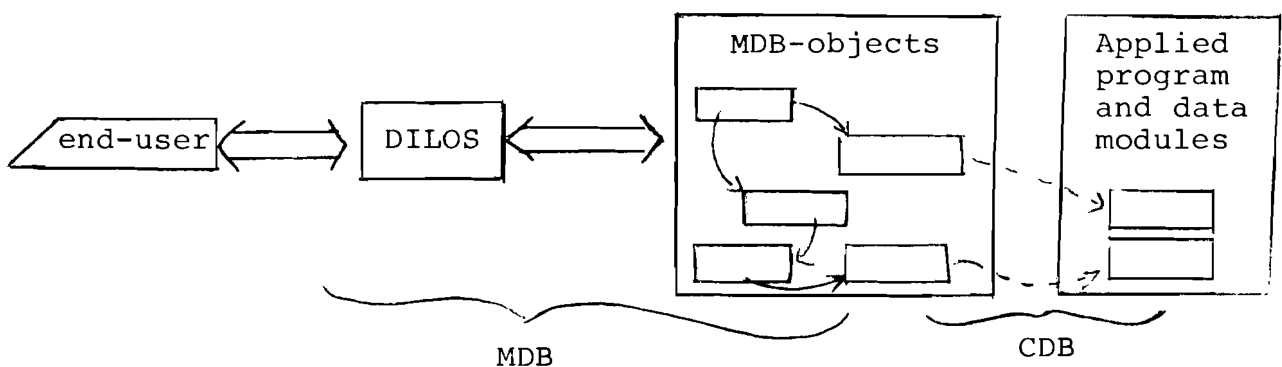

Figure 3.

IX. PROCESS DESCRIPTORS

So far, we have considered the representation and usage of A- and B-objects. Now it is timely to show the way of organizing the whole problem-oriented system. The essential role here is

played by *theorems*--special objects defining the sequences of actions to be performed under certain conditions.

In general, each theorem contains two basic parts: *pattern* and *function*. The theorem becomes initiated when a *calling pattern*, generated somewhere in the system, matches the theorem's pattern. Thus, the theorem's pattern and functional parts correspond to a logical implication rule : <pattern> $\supset$ <function>. The origin of this notation and interpretation could be found in the dissertation on PLANNER by Hewitt (1972) developed later by several researchers in artificial intelligence including Briabrin (1975a, 1975b).

We have already met a calling pattern when considering the START function (Example 1.2). Another function that can be used in different places for generating calling patterns is EX function: (EX calling-pattern).

Example 9.1

Consider a hypothetical "development game", played by the end-user against some program residing in the data base. The first theorem defines the general skeleton of the game:

```
(T1  (*  TH
     PATT  (,  "DEVELOPMENT GAME"  "DEV-GAME"  DG)
     FUN   (PROG  ()
                  (PRLINE  "DEVELOPMENT GAME IS STARTED")
                  (EX  "PRELUDE")
            LOOP  (EX  "MODEL ACTIVATION")
                  (ASK  "WANT TO REPEAT?"  =Q)
                  (COND  ((EQ  Q  'YES)  (GO  LOOP)))
                  (EX  "FINAL")
                  (PRLINE  "THE GAME IS OVER.  COME AGAIN")))) .
```

T1 is the theorem's name. The first property {* TH} serves as an indicator that this is a theorem. Two other properties {PATT ...}, and {FUN ...} contain the theorem's pattern and function parts as discussed earlier.

This theorem is initiated by one of the following calling patterns: "DEVELOPMENT GAME", "DEV-GAME", DG. Other desirable patterns could be added to this list.

The functional part of the theorem takes the form of a LISP program containing seven function calls. Functions PRLINE in the beginning and in the end of this program provide printing messages on the user's terminal. Functions EX with different calling patterns initiate other theorems. Function ASK prints the message "WANT TO REPEAT?" on the terminal and assigns the end-user's reply to the variable Q. This reply is compared with YES by the function EQ; the results of this comparison causes function COND to repeat the loop or to pass control toward the end of the game.

Thus, the main logical structure of the gaming process appears clear in this theorem. However, further work is needed to describe details of "prelude", "model activation", and "final" stages of the game. Appropriate theorems can do that.

```
T2  (*  TH
     PATT (, "PRELUDE" "MODEL PREPARATION")
     FUN  (PROG ()
     (FIND INITIAL-VAL : TOTAL-PROD =P1 ; CAPIT-COST =P2)
     (ASSIGN  (LIST P1 P2)  PHASE-VAR)
     (PRLINE  "PHASE VARIABLES ARE READY"))))  .
```

This theorem could be initiated by the "PRELUDE" or "MODEL PREPARATION" calling pattern, and this could be done during T1 evaluation or by external call : (EX PRELUDE), which is useful for debugging purposes.

Function FIND extracts the values of TOTAL-PROD and CAPIT-COST properties from the object INITIAL-VAL; this object presumably resides in the same MDB division where T1 and T2 are stored.

Extracted values are assigned to P1 and P2 pattern variables, and the next function ASSIGNS the list of P1 and P2 values to a new variable PHASE-VAR.

Printing out the message "PHASE VARIABLES ARE READY" completes the T2 evaluation.

```
(T3  (*  TH
     PATT  "MODEL ACTIVATION"
     FUN  (PROG  ()
                 (ASK "YOUR STEP NUMBER?" =STEP-NUMB)
                 (ASK "THE VALUE OF IMPORTED CAPITAL?" =C1)
                 (ASK "THE VALUE OF EXPORTED CAPITAL?" =C2)
                 (ASSIGN  (LIST C1 C2)  CNTRL-VAR)
                 (INIT  DEV-GAME)
                 (ASK "READY TO RUN?" =R)
                 (COND ((NEQ R 'YES) (RETURN 'WAITING)))
                 (RUN DEV-GAME)
                 (FINISH DEV-GAME)
                 (PRINT VALUES)
                 (PRINT COMMENT)
                 (ASSIGN VALUES PHASE-VAR)
                 (PRLINE "THE PHASE IS COMPLETE"))))  .
```

The functional part of this theorem is built on the assumption that there exists a program module descriptor DEV-GAME with the properties as given in Example 8.2. The four functions at the beginning provide the values of STEP-NUMB and the CNTRL-VAR arguments for the main program. After this is done, all three arguments have appropriate values, and (INIT DEV-GAME) prepares an input file for the main program.

The system then asks the end-user whether he is ready to run the main program. If he does not confirm this by typing "YES", then the system responds by "WAITING" and this ends T3 evaluation.

Otherwise, the system executes the main program, separates the output file into result variables, prints out the value of VALUES (2 integers) and the value of COMMENT (22 characters). Then, it ASSIGNS the value of VALUES to PHASE-VAR, and prints the message "THE PHASE IS COMPLETE".

Looking back at the T1 theorem, we can see that after the T3 evaluation, the system has a short conversation with the end-user and either repeats the T3 evaluation with the new value of PHASE-VAR, or proceeds to the last step--executing the "final" stage of the game. A corresponding theorem could be formulated for the "FINAL" calling pattern, but it will not give the reader new information about the system.

We would conclude this lengthy example by depicting the basic features of the illustrated approach.

Process descriptors, or theorems help the system analyst to:

(a) Detach the logical structure of the constructed process from the programmed implementation of internal calculation algorithms;

(b) Arrange a pleasant conversation between the end-user and systems logical controller;

(c) Organize the internal system's access to MDB contents by means of the same FIND, ADD and DEL functions available to the end-user when he is working from the terminal;

(d) Provide the possibility of deductive inference based on the similarity of the theorem to logical rules of implication.

The last feature becomes possible if the theorems' patterns contain not only constants, as in Example 9.1, but also *pattern variables* which could obtain values from the calling patterns and be used in functional parts of the same theorems.

In general, the DILOS approach to data base management could be outlined as follows:

(1) Structured data representing the system analyst's knowledge about specific problem domains are stored separately from the terminal nonstructured data, manipulated by applied programs.

(2) Applied programs are considered data base objects similar to applied data modules, and they are initiated through structured descriptors residing in MDB rather than directly by the user.

(3) Interaction between the users and the applied program and data complexes is organized by means of special process descriptors, allowing flexible logical control of the problem-solving processes.

X. LIST OF BASIC FUNCTIONS

(1) General functions for MDB-objects manipulation:

```
(OPEN   'divname)
(FIND   pattern)
(ADD    pattern)
(DEL    pattern).
```

(2) Functions for program modules initiation, execution and finalization:

```
(INIT   progr-module-descriptor)
(RUN    progr-module-descriptor)
(FINISH progr-module-descriptor).
```

(3) Functions for extracting and filling data modules:

```
(EXTR   data-module-descriptor  arg-variable)
(FILL   data-module-descriptor  res-variable).
```

(4) Functions for interaction with the end-users:

```
(ASK    message = variable)
(PRLINE message).
```

(5) Functions for initiating theorems:

```
(START  calling-pattern)
(EX     calling-pattern).
```

(6) General purpose LISP-functions:

```
(COND   (predicate1  expr11  expr12 ... )
        (predicate2  expr12  expr22 ... )
                       .
                       .
                       .                )
(GO     label)
(RETURN expr)
(SETQ   variable expr)
(PRINT  expr).
```

(7) Functions for manipulation of MDB-object properties:

```
(GETP   'objname  'propname)
```

```
(GETPV 'objname  'propname  'propvalue)
(DELP  'objname  'propname)
(DELPV 'objname  'propname  'propvalue)
(PUTPV 'objname  'propname  'propvalue).
```

References

Briabrin, V.M. (1975), Universal Semantic Memory, in *Symbol Inf. Processing*, 2, Computing Center of the Academy of Sciences, Moscow.

Briabrin, V.M., V.A. Serebryakov, and V.M. Yufa (1975), *LORD: LISP-Oriented Resolver and Data Base*, 4th Intern. Joint Conf. on Artificial Intelligence, Massachusetts Institute of Technology, Cambridge, Mass.

Hewitt, C. (1972), *Description and Theoretical Analysis of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot*, AI-TR-258, AI Lab., Massachusetts Institute of Technology, Cambridge, Mass.

Howard, F. (1975), *L110 Programmers Manual*, HRSTS Science Center, Lynnfield, Mass.

Teitelman, W. et al. (1974),*INTERLISP Reference Manual*, XEROX Palo Alto Research Center, Calif.

Weissman, C. (1967), *LISP 1.5 Primer*, Dickenson Publishing Co., Encino, Calif.