DB-Main Manual Series

VOYAGER 2 REFERENCE MANUAL

VERSION 8 RELEASE 0 - DECEMBER 2005

The University of Namur - LIBD
REVER s.a.

PREFACE

We believe that programs like emacs, AutoCAD[1], Word[2] and TeX[3] owe their success partially to the existence of a language[4] allowing the user to write macros or even programs. Indeed, such languages fill a gap between built-in functionalities and those expected by the user. This argument suffices to explain why we chose to define and to implement such a language for the DB-MAIN tool.

Because small steps are more secure than large ones, at the beginning - when Voyager did not exist yet - this language had to be a simple script facility for generating reports. Now, this language shares the characteristics of its big brothers and even has a name: Voyager 2. This last issue was the most difficult to settle!

This manual was written as a reference manual and therefore is quite concise in order to give the reader a maximum of details economically. We are conscious that the examples are rather scarce, especially regarding the use of the repository. For this reason, this document is only a first version of what will ultimately become a series of manuals: reference manual; users's guide; and tutorial.

I thank the DB-MAIN, DB-MAIN/01, INTER-DB, PROAGEC research groups and - last but not least - the Professor J.-L. Hainaut for their support in my work.

V. Englebert.

1. AutoCad and AutoLisp are trademarks of Autodesk.
2. Word and WordBasic are trademarks of Microsoft.
3. TeX is a sophisticated program designed to produce high-quality typesetting, especially for mathematical text. It was created by Donald Knuth.
4. Elisp, AutoLisp, WordBasic.

FOREWORDS

Foreword to Version 2 Release 1

The last edition of this manual was named release 1.0. We decided to split the version number and to name each part respectively *version* and *release*. A new version introduces important modifications or significative modifications although a new release means only minor changes.

The Version 2 release 1 introduces major changes like: lexical analysis facilities[1], new object types[2], textual properties, object removal[3], modification and meta-properties[4] in the repository.

One major change in the environment is the new console. It is no more possible to quit DB-MAIN by closing the Voyager 2's console. This console has one disadvantage: the display is quite slow. But be sure that your program is as fast as before.

In this release, the format of the .oxo files has changed. So it is nevessary to recompile your former programs with the new compiler. The compiler is backward compatible.

Foreword to Version 3 Release 0

Voyager 2 has now the same version number as DB-MAIN.

Several mistakes in the reference manual have been corrected[5]. I thank Jean-Marc Hick for his help to improve the quality of the "modular programming" part with his pertinent remarks. And, last but not least, I thank Richard Mairesse for his kindness - the page 42 would never have been printed without his knowledge of Postscript.

The architecture of the abstract machine has been improved. In previous releases, the abstract machine was unique and static. We can now have several abstract machines at the same time, and the number of abstract machines is not limited. This improvement allows us to call functions/procedures from other V2 programs (cfr. 17).

Voyager has now standard Windows dialog boxes (cfr. page 42).

The use of the compilers and DB-MAIN are now limited to people having an electronic key. This is the price we have to pay for being famous. Without the electronic key, it is impossible to run the compiler and DB-MAIN behaves in "demo" mode (the size of the repository is limited). A whole chapter explains these changes (cfr. 19).

The programs[6] listings have been removed from the appendices and the font has been reduced to get a smaller and more handy manual.

A β -version[7] existed during a while between the releases 2.1 and 3.0. This version allowed the use of one undocumented procedure (call_V2). This statement is now deprecated and should no more be used. (cfr. 17)

Foreword to Version 4 Release 0

The abstract machine and the compilers have been translated into 32-bit code. Hence, some limitations vanished. The repository has been improved in order to represent graphical properties (position, font, size, ...). The following bugs have been fixed:

- Concatenation of two empty lists.
- The documentation now describes the CharToStr function.

1. Chapter 10.
2. Sections 12.48 and 12.49.
3. Chapter 15.
4. Chapter 16.
5. I thank Philippe Thiran for his help.
6. The user can find these files in the DB-MAIN distribution.
7. This version also had some dialog boxes and was distributed to some partners.

- The SetFlag function is fixed.

The main change is the evolution of the repository in order to generalize it and to represent various kind of information like programs, process, procedures, etc. For this reason, we decided to attach the coll_et object-type to data_object via the DATA_COLET link and no more to the entity_type object-type. Unfortunately, Voyager 2 is not able to take this generalization into account and to preserve the existent programs.

So, a query expressed as

```
entity_type: ent;
...
ENTITY_TYPE[ent]@ENTITY_COLET:L
```

should be translated into

```
data_object: dta;
dta2ent(DATA_OBJECT[dta]@DATA_COLET:L with GetType(dta)=ENTITY_TYPE)
```

where dta2ent is a function that you can write yourself as:

```
function list dta2ent(list: L1)
  entity_type: ent;
  list: L2;
{ for ent in L1 do {
    AddLast(L2,ent);
  }
  return L2;
}
```

The compiler will now stop if the entity_colet constant is encountered.

Another modification is the introduction of the first concepts to represent "processes"[8] (see page 92).

Foreword to Version 5 Release 0

This version is endowed with a large number of new concepts. Processes and advanced graphical representations are now supported by DB-MAIN. The repository has thus considerably growed to such an extent that its representation does no more take up one sheet! Some functions have also been added (see section 8.8).

People have indicated some troubles when the path of the compiler contains one or several space character. This problem can be avoided with the "-quote" parameter in the command line. Unfortunately, this does not work with Windows NT.

They are now two switches in the electronic key. One to allow the programmer to port .oxo files to some other station with a precise electronic key; and another key to distribute them in an anonymous way (the -Kall option). The compiler displays the status of those switches.

The extension of the repository has introduced new keywords. The programmer should check if these new reserved words are not used as variables or function names in his programs. The posx, posy, and color attributes are now managed in a different way (see section 8.8).

Foreword to Version 6 Release 0

This version has no big changes. Several bugs have been fixed (mainly in the libraries). Amongst the bugs, we can cite:

- the parameters of a foreign procedure.
- the assignment between processes when a process does not exists.

8. In the large!

- the behaviour of some dialog boxes.
- the compiler now generates error messages that are Emacs-compliant.
- the size of the stack has been increased (3 X)
- the SetProperty has been fixed (carriage return).
- the concatenation of strings (empty strings).

Foreword to Version 6 Release 5

This version has two major changes. Several bugs has been also fixed. The changes are:

- The documentation describes two new function. The CallSync function allows a Windows or DOS program to be executed and forces the calling Voyager 2 program to wait for the execution of that Windows program to be finished before continuing its execution. The GetLambda function returns a lambda expression from a function/procedure name in a Voyager 2 program.
- This version manages a new concept. The note object (free text attached to any generic object) is now supported by Voyager 2.

Foreword to Version 7 Release 0

Version 7 of Voyager 2 contains various major changes, some of them making previous versions of Voyager 2 programs obsolete:

- The new procedure "transfo" allows a Voyager 2 program to execute an Advanced Global Transformation Script of DB-MAIN. This script are handled in strings, so they can be build dynamically.
- Several modifications to the repository of DB-MAIN which force changes to Voyager 2 programs:
  - Notes can now be connected to several objects in a schema.
  - Schemas can now be of various types. A new field has been added to the schema object type and new constants have been defined as values for this field.
  - The part of the repository concerning processing units, including their decomposition and the resources they use, has been completely redesigned.
  - An owner_of_proc_unit is no more a schema, a group or a data_object, but only a schema or an ent_rel_type.
- The compiler was previously made up of two parts: comp_v2.exe and comp_v1.exe. The second part is now a dll file in order to avoid problems for finding it by the first part when long filenames are used. So, now, the files are comp_v2.exe and comp_v1.dll.

This manual itself contains two main changes:

- A new chapter about global variables has been added.
- The chapter presenting the repository has been rewritten.

Foreword to Version 8 Release 0

Version 8 of Voyager 2 contains major repository changes, some of them making previous versions of Voyager 2 programs obsolete:

- The part of the repository concerning consumptions, including their roles with processing units and resources, has been completely redesigned.
- The sub_element has been renamed into rel_element.
- The objects consumption, rel_element, re_isa and environment have names and descriptions.

New blackboxes have been implemented and several bugs have been also fixed.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PART I

THE VOYAGER 2 LANGUAGE

Chapter 1

Preliminaries

Voyager 2 is an imperative language with original characteristics like the *list* primitive type with garbage collection and declarative requests on the predefined repository of the DB-MAIN tool. Other characteristics will be discussed further in this document. Because Voyager 2 is similar to traditional languages like C and Pascal, we will suppose in this reference manual that the reader has a good knowledge of them.

A Voyager 2 program is composed of three distinct sections:

global variables definitions

functions definitions

main body

The *global variables definitions* section contains the definition of all the global variables of the program. The scope of these variables is the whole program as well as the functions and the procedures. Constants can also be defined in this section. The *functions definitions* section will contain the definition of all the functions and all the procedures needed by the program. Functions will not be distinguished from procedures in this document unless it is explicitly mentioned. The scope of a function is the whole program[1]. The *main body* section is the main program, ie. a list of instructions enclosed between the two keywords "begin" and "end". Only the last section is mandatory in a Voyager 2 program. The main program is the equivalent fo the *main* function in C, Voyager 2 begins the execution there.

1. Here is a first difference with Pascal: a function f can call a function g defined afterwards.

Chapter 2

Lexical Elements

2.1 Comments

A comment in a Voyager 2 program begins with an occurrence of the two characters /*, which are not enclosed in a string constant, and ends with the first occurrence of the two characters */. Comments may contain any characters and may spread over several lines of the program. Comments do not have any effect on the meaning of the program.

A comment may also be any characters found after the two characters // in one line.

Example:

```
/* Add comments to
** your programs, please ! */
begin
  x:=x+1; // and comments must be pertinent !
end
```

2.2 Operators

The operator tokens are divided in several groups as shown in Table 2.1.

Token class	Tokens
expression operators	+ - / mod ++ ** or and xor not < > <= >= <> =
instruction operators	:= <- << >> +> <+
separators	. , ; ( ) [ ] { }

Table 2.1 - Operators and separators.

Expression operators are used to build new expressions from other ones, instruction operators are a convenient way to replace classical functions by infix operators.

2.3 Identifiers

An *identifier* is a sequence of lower-case and upper-case letters, digits and underscores. An identifier must begin with a letter, identifiers beginning with an underscore are reserved for keywords having a special meaning for the language. There is no restriction on the length of an identifier. Finally, an iden-

tifier must be distinct of any reserved keyword (cfr. section 2.4) and any predefined constant name (cfr. section 2.5).

Examples:

factorial	PI_31415	A_B__C_	are all valid identifiers.
_PI	314_PI	for	are all incorrect identifiers.
A_Einstein	A_EINSTEIN	a_einstein	are three distinct identifiers.

2.4 Reserved words

Some words (cfr. tables 2.2 and 2.3) are reserved for the language and can not be redefined by the user.

_GetFirst	do	IsNotNull	StrBuild
_GetNext	else	IsNoVoid	StrConcat
AddFirst	end	IsNull	StrFindChar
AddLast	Environment	IsVoid	StrFindSubStr
and	GetCurrentSchema	kill	StrGetChar
as	eof	label	StrGetSubStr
AscToChar	ExistFile	Length	StrItos
begin	export	member	StrLength
break	for	mod	StrSetChar
call	function	neof	StrStoi
callSync	get	OpenFile	StrToLower
case	GetAllProperties	or	StrToUpper
CharIsAlpha	GetCurrentObject	otherwise	switch
CharIsAlphaNum	GetFirst	print	TheFirst
CharIsDigit	GetFlag	printf	then
CharToAsc	GetLast	procedure	TheNext
CharToLower	GetProperty	read	to
CharToStr	GetType	readf	Transfo
CharToUpper	goto	rename	until
ClearScreen	halt	repeat	use
CloseFile	if	return	void
continue	in	SetFlag	Void
create	interrupt	SetPrintList	while
delete	IsActive	Setproperty	xor

Table 2.2 - Reserved keywords.

2.5 Constants

In Voyager 2, *constants* are predefined variables with constant expressions. The constant names are listed in tables 2.4, 2.5, 2.6, 2.8, and 2.7.

attribute	component
char	connection
cluster	constraint
co_attribute	data_object
coll_et	document
collection	ent_rel_type
complex_user_object	entity_type

Table 2.3 - Reserved keywords (types).

et_role	product
file	real_component
float	rel_type
generic_object	role
group	schema
integer	set_of_product
list	set_product_item
member_cst	si_attribute
meta_object	string
meta_property	sub_type
nn_note	system
note	user_object
owner_of_att	

Table 2.3 - Reserved keywords (types).

_char	_file
_float	_integer
_lambda	_list
_program	_string
ATTRIBUTE	CLUSTER
CO_ATTRIBUTE	COLL_ET
COLLECTION	COMPLEX_USER_OBJECT
COMPONENT	CONNECTION
CONSTRAINT	DATA_OBJECT
DOCUMENT	ENT_REL_TYPE
ENTITY_TYPE	ET_ROLE
GENERIC_OBJECT	GROUP
MEMBER_CST	META_OBJECT
META_PROPERTY	NN_NOTE
NOTE	OWNER_OF_ATT
PRODUCT	REAL_COMPONENT
REL_TYPE	ROLE
SCHEMA	SI_ATTRIBUTE
SUB_TYPE	SYSTEM
USER_OBJECT	

Table 2.4 - Constants denoting entity-types.

CLU_SUB	COLL_COLET
CONST_MEM	CONTAINS
DATA_GR	DOMAIN
ENTITY_COLET	ENTITY_CLU
ENTITY_ETR	ENTITY_SUB
GR_COMP	GO_NN
GR_MEM	IS_IN
MO_MP	NOTE_NN
OWNER_ATT	REAL_COMP
REL_RO	RO_ETR

Table 2.5 - Constants denoting links.

SCH_COLL	SCH_DATA
SYS_MO	SYSTEM_SCH

Table 2.5 - Constants denoting links.

_A	L_ROLE
_R	L_SNAME
_W	L_VERSION
ARRAY_CONTAINER	LIST_CONTAINER
ASS_GROUP	MARK1
BAG_CONTAINER	MARK2
BOOL_ATT	MARK3
CHAR_ATT	MARK4
COMP_GROUP	MARK5
CON_COPY	MAX_STRING
CON_DIC	N_CARD
CON_GEN	NUM_ATT
CON_INTEG	OBJECT_ATT
CON_XTR	OR_MEM_CST
DATE_ATT	PROP_CORRUPTED
EQ_CONSTRAINT	PROP_NOT_FOUND
ERA_SCHEMA	RTROUND
ETROUND	RTSHADOW
ETSHADOW	RTSQUARE
ETSQUARE	SELECT
FALSE	SEQ_ATT
FLOAT_ATT	SET_CONTAINER
HIDEPROD	TAR_MEM_CST
INC_CONSTRAINT	TRUE
SCHEMA_DOMAINS	UMLACTIVITY_DIAGRAM
INDEX_ATT	UMLCLASS_DIAGRAM
INT_MAX	UMLUSECASE_DIAGRAM
INT_MIN	UNIQUE_ARRAY_CONTAINER
L_CRITERION	UNIQUE_LIST_CONTAINER
L_DATE	USER_ATT
L_FREE	VAR_CHAR_ATT
L_NAME	

Table 2.6 - Miscellaneous Constants.

atleastone	key
coexistence	last_update
container	length
creation_date	mark_plan
criterion	max_con
decim	max_rep
disjoint	mem_role
exclusive	min_con
file_desc	min_rep
filename	multi

Table 2.7 - Field Constants.

flag	name
font_name	other
font_size	path
identifier	posx
posx2	text_font_size
posy	total
posy2	type
predefined	type_object
primary	type_of_file
recyclable	updatable
reduce	user_const
secondary	value
sem	version
short_name	view
stable	where
status	xgrid
tech	ygrid
text_font_name	zoom

Table 2.7 - Field Constants.

ERR_BAD_TRANSFO	ERR_FILE_CLOSE
ERR_CALL	ERR_FILE_OPEN
ERR_CANCEL	ERR_PATH_NOT_FOUND
ERR_DIV_BY_ZERO	ERR_PERMISSION_DENIED
ERR_ERROR	

Table 2.8 - Error Constants.

Chapter 3

Global Definitions

The *Global Variables Definitions* section contains the definition of all the global variables and all the constants of the program. This section is composed of definition lines which respect the following syntax:

⟨definition line⟩ ← ⟨type⟩ : ⟨var-const⟩, …, ⟨var-const⟩;

⟨var-const⟩ ← ⟨variable⟩ | ⟨constant⟩

⟨variable⟩ ← ⟨identifier⟩

⟨constant⟩ ← ⟨identifier⟩ = ⟨expression⟩

Types are defined in chapter 4. In a definition line, when an expression is associated with an identifier, this constant is considered being initialized by this expression. Each time this variable is used, its occurrence is replaced by the corresponding expression. This characteristics differs from the C[1] and Pascal languages since they evaluate the expression as soon as it is found. In the Voyager 2 language, the evaluation process is delayed until the variable is used. As a consequence, constant expression may contain identifiers and function names that are outside the scope of the expression. Unlike macros in C, constants have a type and the evaluation of the constant must match it.

Example:

Program 1.

```
integer: s=m+c2, age;
integer: lname=strlen(pname),m,c2;
string: pname="Einstein";
begin
m:=2; c2=3;
print(s*2);
m:= 4;
print(s*2);
print(lname);
end
```

will print the values "10" ((2+3)2), "14" ((4+3)2), and "8" (length of "Einstein"). Let us note that the evaluation of constants may return different values depending on the context.

1. The comparison does not hold neither with the macros of the C language nor with the const type specifier of the C++ language

Program 2.

```
integer: sum=a+b;
procedure foo(integer: a)
  integer: b;
{ b:=1;
  print(sum);
}
begin
  foo(2);
end
```

will print the value "3". The function definitions section will contain all the function/procedure definitions. The syntax of a function/procedure definition is fully explained in Chapter 9. Each function/procedure can be called from anywhere in the program: from a function, from a procedure or from the main body even if the call to the function/procedure is before its definition.

Chapter 4

Types

4.1 Integers

The integer type covers all the integer values from INT_MIN to INT_MAX. Integers are signed and the integer constant INT_MIN (resp. INT_MAX) is the smallest (resp. greatest) value of this type. Integer constants are signed[1] literals composed of digits 0,1,…,8,9. The integer type is named integer.

Examples:

1, 123, -458, -1021 are valid integer constants

+458, 3.1415, 3E+6 are not valid integer constants

4.2 Floats

The float type covers all the float values from 3.4E-38 to 3.4E+38. Float constants are signed literals composed of digits 0,1,…,8,9 and "." as decimal separator,. The float type is named float.

Examples:

3.1415, -1000000000000 are valid float constants

+458.25, 3E+6 are not valid float constants

4.3 Characters

The character type covers the whole ASCII character set from code 0 to 255. All the characters having a graphic representation have a corresponding constant in this type: the graphic representation itself enclosed between simple quotes. Otherwise characters can be represented by their ASCII value like '^val^'.

Examples:

char: a=’a’, Z=’Z’, plus=’+’; char: bell='^7^', strange='^236^';

Some interesting non-graphic characters have a special representation, as illustrated in table 4.1.

Character	Representation
backspace	’\b’

Table 4.1 - Conventions about special characters.

1. The unary operator + is not allowed.

form feed	’\f’
newline	’\n’
carriage return	’\r’
tab	’\t’
’	’’’

Table 4.1 - Conventions about special characters.

4.4 Strings

Strings are sequences of characters. Although the programmer must take care of details like the size of the memory block where the string is stored, in Pascal and C, these mechanisms are completely transparent in Voyager 2. In this manual, the sentence "the size of the string s" means the number of characters stored in the string s. String constants are sequence of characters between double quotes. The length of a string must be less than the value found in the constant MAX_STRING.

Example:

The statement

```
print("Albert Einstein")
```

will produce

```
Albert Einstein
```

and

```
print("1\tone\n2\ttwo\n^51^\tthree\n")
```

will produce

```
1    one
2    two
3    three
```

In string constants some characters have a special representation as shown in table 4.2. The second part of the table uses the same conventions as for the characters.

Character	**Representation**
backslash \	\\
double quote "	\"
hat ^	\^
backspace	\b
form feed	\f
newline	\n
carriage return	\r
tab	\t

Table 4.2 - Meta-characters used in string constants.

Moreover,

1. Characters are examined from left to right.
2. If the ^ character is followed by a sequence of digits denoting a number between 0 and 255 followed by ^, then the whole sequence is replaced by exactly one character with ASCII code equal to that number. Otherwise, the first ^ character is interpreted literally, and the interpreter scans the right part of the string.

3. If the \ character is followed by a letter (λ) and if "\ λ" does not denote a meta-character as depicted in table 4.2, then the sequence is replaced by λ. The \ character is thus removed from the string.

Example:

The instruction

```
print("^1234^55^\^\\\h");
```

will print

```
^12347^\h
```

4.5 Lists

Lists are ordered collections of values. These values can be of any type (including list). Because lists belong to a basic type (list), simple operations can be applied on lists. Another type, cursor, is strongly associated to lists and will be discussed in next section. All the operations and operators available with this type are presented here below.

A programmer can directly enter a constant list in a program simply by specifying the components of the list between brackets. For example:

```
list: lint_ext, lint_exp;
begin
  lint_exp:=[1..20];
  lint_ext:=[1,2,3,5,8,13,21];
  print(lint_exp**lint_ext);
end
```

This program will print all the common values of the two lists: "1 2 3 5 8 13". The first list was defined as a range of values, and the second one was defined explicitly. The syntax of list constants is:

⟨list constant⟩ ← "[" ⟨list expressions⟩ "]" | "[" ⟨expression⟩ "..." ⟨expression⟩ "]"

⟨list expressions⟩ ← Ø | ⟨expression⟩ ("," ⟨expression⟩)*

More complicated constant list expressions follow:

Examples:

```
[1,[1..fact(1)],2,[1..fact(2)],3,[1..fact(3)],4,[1..fact(4)]]
[[],[1,[]],[2,[1,[]]],[3,[2,[1,[]]]]]
[1,2,3..10,11]                                  error: dots are not allowed here!
```

4.6 Cursor

Cursors are references to elements of lists. A cursor can either be null or be positioned. In latter case, it can be either active or passive. Let us examine cursors in these different cases:

- **null cursor**: the cursor is not attached to any list and is not indicating any value.
- **active cursor**: the cursor is positioned on a value in a list, and this value can be consulted, removed, ...
- **passive cursor**: let us suppose that a cursor *c* is positioned on the value *2* in the list *l=[1,2,3]*. Then the value 2 is removed from the list *l*. Therefore the cursor has no more meaning and is said being *passive*. If the program consults the value indicated by this cursor c, it gets an execution error. Although this situation looks like the *null reference*, the situation is quite different since the cursor is still attached to the list. This case will be discussed in the section 8.3.

4.7 Files

Objects of type file are references to files stored on disks. These objects become real references after a call to the function OpenFile whose first argument is the file name and second argument is an integer constant. This constant indicates the mode: *_W* if the file is created for writing, *_R* if the file is opened for reading or *_A* is the file is opened for appending. Depending on the mode, the program may read or write information. Writing always occurs at the end of the file and characters are read from the *current position*. Programs must close all the opened files before leaving. More details are found in 8.4.

4.8 References

As mentioned in the section 1, Voyager 2 is integrated in the DB-MAIN tool; therefore it has an access to the content of the repository. The full definition of the repository is presented in part II.

The repository of DB-MAIN is an object with relationships between object-types. The relationships are *one-to-many*; they are named *links*. The attributes of an object type are named *fields*. An object is sometimes also called a reference. The following table summarizes the equivalence between these concepts:

ER-schema	↔	**Voyager 2**
entity-type	↔	object-type
entity	↔	object, reference
attribute	↔	field
relation	↔	link

To each object type present in the definition of the repository of DB-MAIN corresponds a type in Voya ger 2. For instance, the type group corresponds to the "group" object type. Variables or expressions of this type can either be references to an object of this object-type, be void (a special value denoting *nothing*), or be invalid.

If a variable is a reference to an object, the value of its fields can be obtained with the "." operator. For instance, the following program prints the name of the object referenced by the variable *ent*:

```
entity_type: ent;
begin
  ...
  print(ent.name);
  ...
end
```

Expressions composed with the "." operator may also occur in the left side part of an assignment, like in the following example:

```
entity_type: ent;
begin
  ...
  ent.name:="CUSTOMERS";
  ...
end
```

The right part of the "." operator is in fact an integer value identifying one field among all the others. In this example, `name` is a predefined integer constant.

Chapter 5

Expressions

Expressions are classified into several categories depending on the type returned by the evaluation process. Some expressions are untyped mainly due to access to the repository and to lists, for these particular cases, the type verification is delayed until the evaluation time. The first subsection presents the operators used in expressions. Following subsections present operators and functions provided by the language for each type.

5.1 Precedence and associativity of operators

Each expression operator in Voyager 2 has a precedence level and a rule of associativity. When parentheses are not used explicitly to indicate the grouping of operands with operators, precedence rules are applied. If two operators have the same precedence, they are grouped following the associativity rule (left/right associativity). Table 5.1 defines the precedence and associativity rules of each operator. Lines separate operators according to their precedence. One line separates two groups of operators and each operator inside one group have the same precedence. If a group is above another one, then its operators will be evaluated before the operators of the other group. For instance, the precedence of * is higher than the precedence of +.

Example:

The following expressions may be evaluated as follows with the precedence/associativity rules:

Original expression	Equivalent expression
a+bc	a+(bc)
a=not b and c or d	a=(((not b) and c) or d)
a.length > 10 = 1	((a.length)>10)=1

Token	Operator	Class	Associates	Operands[a]
not	logical not	prefix	no	i, f
-	unary minus	unary	no	i, f
*	multiplicative	binary	left	i, f
/	division	binary	left	i, f
mod	modulo	binary	left	i
**	list intersection	binary	left	l

Table 5.1 - Operators: Precedence and Associativity rules

+	addition	binary	left	i, f, s
++	difference	binary	left	i
*	list concatenation	binary	left	l
<	less than	binary	left	i, f, c, s
>	greater than	binary	left	i, f, c, s
<=	less than or equal	binary	left	i, f, c, s
>=	greater than or equal	binary	left	i, f, c, s
<>	different	binary	left	i, f, c, s
=	equal	binary	left	i, f, c, s
and	logical and	binary	left	i, f
or	logical or	binary	left	i, f
xor	logical xor	binary	left	i, f
:==	functional assign-ment	binary	right	any
,	separator	binary	left	any

Table 5.1 - Operators: Precedence and Associativity rules

a. Operands must always be of the same type (except for integer and float). The following letters denote expected types by previous operators: l: list, i: integer, f: float, c: char, s: string, any: any type.

5.2 Arithmetic Expressions

Operators +[1], -, *, /, mod, and, or, not, <, >, <=, >=, <>, = require integer or float expressions as operand (except for mod that require only integer expressions). Their operands are fully evaluated before their own evaluation but the order is left unspecified. The +, -, *, / and mod operators are respectively the addition, subtraction, multiplication, division and remainder of division.

If a division by zero occurs (x/y and y=0), the result is 0 and the error register is set to DIV_BY_ZERO. The following table gives a formal definition to othyer operators:

a>b	returns : if a>b then 1 else 0
a<b	returns : if a<b then 1 else 0
a<=b	returns : if a≤b then 1 else 0
a>=b	returns : if a≥b then 1 else 0
a=b	returns : if a=b then 1 else 0
a<>b	returns : if a= b then 1 else 0
not (a)	returns : if a≠0 then 1 else 0

5.3 Reference Expression

The word "reference" groups several types together and do not allow distinction between them. Let us examine the following line:

```
attribute: att;
```

att is a variable that can reference an object in the repository. We will see in chapter 14 that the following statement:

```
att:=GetFirst(attribute[a]{TRUE})
```

puts into the variable att the reference to the first attribute found in the repository of DB-MAIN. This variable can be used to consult or modify properties of the object:

1. The + operator is overloaded in order to behave like the StrConcat function with strings.

Example:

```
print(att.name);
att.name:="FIRST-NAME";
```

The left part of the "." operator must be an identifier (global/local variable, parameter) denoting an object. The left part must be a valid field name for the object specified in the right part. The right expression must be either an integer or float expression or a string. String fields will be explained later in chapter 16.

5.4 Functional Assignment

The ":==" operator behaves like the assignment operator (:=) except that the left-hand-value is returned as the value of the expression.

Example:

```
integer: a,b;
string: s;
begin
  a:=(b:==0);
  if ((s:==read(_string))="Hello") then {
    print("World!");
  }
end
```

The first statement initializes the two variables a and b with the value 0. The second statement reads one string from the console, puts it into the variable s and then compares this string with the string "World!".

More explanations can be found in the section 7.2.

Chapter 6

List Expressions

6.1 Overview

Lists in Voyager 2 have no similar counterparts in Pascal and C. As explained in the subsection 4.5, lists are ordered collections of values. A list has an existence which is not directly linked to the scope of variables representing it. Values in lists may be of any type, even list, cursor, …A list exists in memory until the program can no more use the values contained in this list, and the programmer does not have to care about the memory management. Let us remember that values can be get through cursors or variables of type list.

Because lists in Voyager 2 are quite different from lists in other languages like Pascal, C and Lisp, some definitions are necessary.

Definition 6.1 (List) *Let l be a variable denoting a list of values $v_1, ...,v_n$, we write $[v_1,...,v_n]$ the content of this list.*

Definition 6.2 (Ghost) *We define • (a ghost) a special value having no meaning in Voyager 2. This value can belong to lists.*

Ghosts are invisible and therefore useless, but they will be used in graphical representation of lists and in explanations.

Definition 6.3 (∂) *We define a unary operator ∂. Let l be a list, then ∂l returns the list l from which all the ghosts have been removed.*

In other words, all the obsolete elements are removed from the list. Then ∂[•,1,2,[3,•],•]=[1,2,[3]].

Definition 6.4 (list equality, =) *$l_1 = l_2$ iff $\partial l_1 \equiv \partial l_2$ where the $\equiv$ operator has the usual meaning. This means that ghosts do not perturb lists comparison in Voyager 2.*

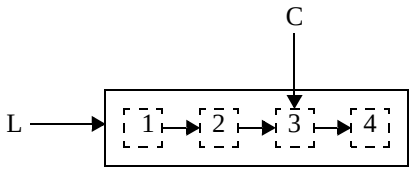
The list equality is the usual way to consider the equality between lists for the programmer.

We associate a graphical representation to the lists, to the variables of type list and to the cursors in order to ease the explanations. A list is represented by a rectangle containing values linked by arrows. Values are represented by dashed boxes. A variable v of type list is represented by an arrow towards the graphical representation of the list. A cursor c pointing to a value inside the list is denoted by an arrow towards this value. Let us consider the following program:

```
1:  List: L;
2:  cursor: C;
3:  begin
4:    L := [1..4];
```

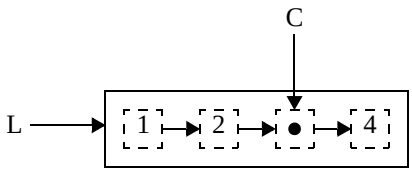
```
5:    attach C to L;
6:    C >> 2;
7:    kill(C);
8:    C <<;
9:    C +> [5..8];
10:   C >>;
11:   attach C to get(C);
12:   C >>;
13:   kill(C);
14:   C >>;
15:   kill(C);
16:   C >>;
17:   C <<;
17:  end
```

At line 4, the list L is assigned to the list [1,2,3,4], at line 5, the cursor C is attached to the list L and therefore is indicating the first value of L. The instruction at line 6 moves the cursor two elements forward, the cursor C is now indicating the value 3. The graphical representation of the state after line 6 is:



This is a convention. Newly attached cursors always point to the first element. If the list is empty (L=[]) then the cursor is said to be null and has the special value void.

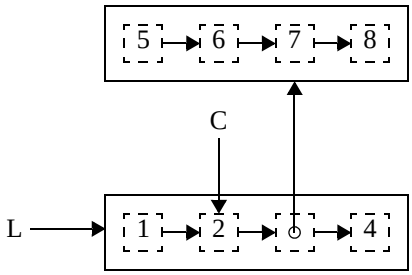
The instruction kill(C), at line 7, destroys the value under the cursor. So, the instruction kill(C) will destroy the value 3 in L. We represent this action by replacing the value 3 in L by the special value: •. The cursor is still attached to the list L but it is now impossible to consult the value under C or to replace this value by another one. The drawing becomes:



With respect to the definition 6.4, the following property holds:

$$L = [1,2,4]$$

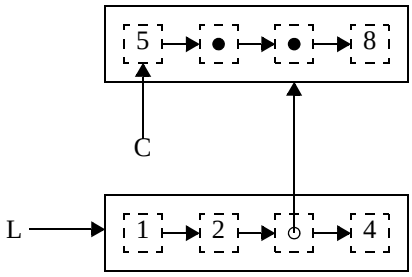
At lines 8 and 9, the cursor C is moved one element backward and the list [5,6,7,8] is inserted just after C. The drawing becomes:



Let us remark that the ghost value has disappeared. The reason is simple: the cell containing the ghost value was no more referenced. So it was safe to suppress it. From the user's point of view, it was impossible to detect the presence of the ghost value after the execution of the instruction "C <<;".

The value under a cursor can be consulted with the function get. Lines 10 and 11 use this function to attach C to the newly created list. The cursor is now indicating the first element of the new list: 5. The lines 12 to 16 destroy the two elements 6 and 7 of this list, so, after their execution, C is indicating the

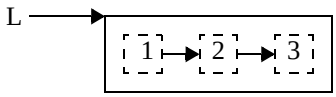
last value: 8. This list is now equivalent to the list [5,8] and for this reason if the cursor C is moved one element backward, at line 17, one finds the value 5 under C as illustrated by the drawing:



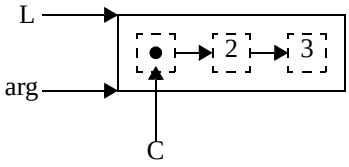
Although the general rule in Voyager 2 for passing values to functions is by value, list objects are always passed by address[1]. Let us examine the meaning of the following program:

```
1:  list: L,R;
2:  procedure RemFirst(list: arg)
3:     cursor: C;
4:  {  attach C to arg;
5:     kill(C);
6:  }
7:  begin
8:    L:=[1,2,3];
9:    R:=L;
10:   RemFirst(L);
10:   print([L,R]);
11:  end
```

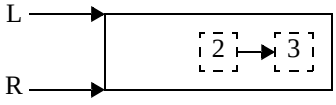
After line 8, the state is described by this schema:



at line 6, when the first element has been deleted and the RemFirst function has been called:



and finally at line 10:



and the result printed on the console will be:

[[2,3],[2,3]]

With respect to this principle, lists can be built inside a function and returned to the global environment. The following program is a good example of what happens when a list is returned from a function:

```
list: L;
function list foo()
   list: local;
{  local:=[1..3];
   return local;
}
begin
```

1. Except some cases like the ++ operator and a few other functions.

```
  L:=foo();
  print(L);
end
```

Because the list [1,2,3] is first referenced by the variable local, and finally referenced by the global variable L, the list was not destroyed when the function finished.

6.2 Operations

Several operators exist specifically for the lists. The section describes them.

6.2.1 Concatenation

The infixed operator ++ takes two distinct lists and returns the concatenation of both. Let us note that the arguments are detached of their body after execution. For instance, let us suppose that the cursor C is attached to the list L1 and that the instruction R:=L1++L2 is performed, then the cursor C is now attached to the list R and no more to L1 the value of which is the empty list [].

---

function list: r ++ (list: l1, list: l2)

Precondition. lists l1 and l2 are two list expressions denoting distinct lists.

Postcondition. lists l1 and l2 are now empty. If the list l1 is $[v_1,...,v_n]$ and l2 is $[w_1,...,w_m]$, then the result of the operation is a new list: $[v_1,...,v_n,w_1,...,w_m]$. After the call, l1=[] and l2=[].

The following examples show the effect of this operator:

Examples:

[1,2,3]++[4,5,6] → [1,2,3,4,5,6]

L1:=[[’a’,1]]; L1:=L1++[[’b’,2]] → [[’a’,1],[’b’,2]]

L2:=L1++L1; → **error!**

L2:=L1; L3:=L1++L2; → **error!**

6.2.2 Intersection

The infixed operator ** is used between two lists to compute all the common elements. There is no restrictions on the arguments of this operator.

---

function list: r ** (list: l1, list: l2)

Precondition. l1 and l2 are two lists. The type of the items stored in both lists may not be identical.

Postcondition. r is the list of all the values common to lists l1 and l2. If one object is present in both lists but with different types (one super-type[2] and one sub-type[3] for instance), they are considered distinct. The order of the returned list is left unspecified.

The following instances show the power of this operator:

[1,1,2,4]**[1,5,2] → [1,2]

[[1,2],[3,4],’b’,7]**[’a’,7,[3,4],["ab",GetCurrentSchema()]] → [[3,4],7]

l1:=[1,2,1,3,’a’,’a’,’b’];

l1**l1 → [1,2,3,’a’,’b’]

6.2.3 Insertion

To insert values in lists, several methods have already been presented. But none is as general as the operators +> and <+. These operators are infixed. For each one, the left operand must be an expression of type cursor and the right operand may be any expression that can be inserted in a list. The effect of the first (resp. second) operator is to insert the result of the right hand expression just after (resp. before) the value designated by the cursor specified in the left operand.

2. For instance: data_object
3. For instance: ent_rel_type

In order to remove any ambiguity, these two operators can be defined more formaly as follows:

Let us analyse the effect of: C +> E

where *C* is any expression of type cursor and *E* is an expression.

1. If *C* **is attached to a list** *L*,
   a) **if the cursor** *C* **is null**, then the value of *E* is inserted as the first element of the list *L*
   b) **if the cursor** *C* **is not null**, then the value of *E* is inserted just after the value designated by the cursor *C*.
2. If *C* **is not attached to a list**, the instruction fails, as well as the program.
   This is an error of the programmer.

The effect of: C <+ E

is:

1. If *C* **is attached to a list** *L*,
   a) **if the cursor** *C* **is null**. Then the value of *E* is inserted as the last element of the list *L*
   b) **if the cursor** *C* **is not null**. Then the value of *E* is inserted just before the value designated by the cursor *C*.
2. If *C* **is not attached to a list**. The instruction fails, as well as the program.
   This is an error of the programmer.

In all the cases, the cursor *C* is unchanged and is still designating the same value as before the call of the instruction.

6.2.4 Miscellaneous

---

function any: r get (cursor: c)

Precondition. The cursor c is attached to a list.

Postcondition. r is the value pointed by the cursor c.

on error: The program is interrupted and an error message is displayed.

Chapter 7

Statements

7.1 General rules

Each statement must be terminated by one semi-colon, except compound instructions after which this character is optional.

The empty statement is not allowed in Voyager 2, however a compound statement may be empty.

7.2 Assignment

Assignment statements must respect the following syntax:

⟨assignment-inst⟩ ← ⟨lhs⟩ := ⟨rhs⟩

⟨rhs⟩ ← ⟨expression⟩

⟨lhs⟩ ← ⟨variable⟩ | ⟨variable⟩.⟨field⟩

⟨field⟩ ← ⟨expression⟩

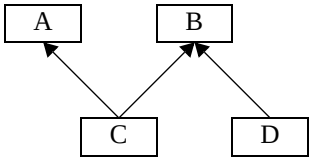
The *rhs*-expression must have a type compatible with the type of the *lhs*-expression. If the *lhs*-expression has a field, then the evaluation of the field must return an integer value. Fields are specific to variables denoting a reference to a repository's object and usually, the user will use a predefined constant (cfr. 2.6) in place of complex expression. When this instruction is executed, the *rhs*-expression is first computed and the result is then assigned to the *lhs*-expression.

Example:

```
int: a,b;
string: s;
begin
a:=1;
b:=a*2;
s:="Rob Roy";
end
```

When inheritance is involved during the assignment, this instruction is able to solve automatically the ambiguity - dynamic type casting. Let us suppose that we have the following schema representing the inheritance between four object types[1] :

1. The example is not a part of the real schema.



and the following program:

```
A: a;
B: b;
C: c;
begin
  ...
  b:=c;
  a:=b;
  ...
end
```

The last assignment is not trivial since the object referenced by b could be either of type C or D. But the assignment is able to find the correct path between the type of a and the type of the object referenced by b[2].

If the assignment fails - if types are not compatible - then the program is aborted and the conflicting types are displayed in the console.

The dynamic type casting performed by the assignment is not a general rule in Voyager 2. Therefore, unless it is explicitly mentioned[3], types must always be exactly identical. For instance, each time you define a new function, arguments and parameters must always have the same type. For this reason the following program is wrong:

```
C: c;
procedure foo(A:a){
  ...
}
begin
  ...
  foo(c);
end
```

The only way to pass the value c to the function is by using an explicit assignment; for instance:

```
C: c;
A: a;
procedure foo(A:a){
  ...
}
begin
  ...
  a:=c;
  foo(a);
end
```

Some suite of assignments can sometimes be optimized. Although performances are not critical for the Voyager 2 programmer, strings may slow down some programs like parsers. For instance, the following scheme is often observed in parsers:

2. Let us remark here, that the type of an object referenced by a variable may be different from the type of the variable! For instance, just after the first assignment, the type of the object referenced by b is C although the type of the variable b is B.
3. The dynamic type casting is applied for the arguments of the function create.

```
while neof(f) do {
  s:=read(_string) ;
  if s="begin" then
    ...
end
```

The scanned string will be put twice on the internal stack of the Voyager 2 virtual machine. The function `read` will read the string from the file and place it on the stack in order to put it into the variable s. The evaluation of the condition of the `if` instruction will place the value of s on the stack. One obvious optimization is not removing the value from the stack. This optimization can be achieved by the programmer by using the function assignment operator (:==) defined in the section 5.4.

7.3 Selection Statements

Selection statements direct the flow of control depending on the value of an expression.

7.3.1 The if-then Statement

The `if-then` statement executes a list of instructions if the evaluation of the condition is different from 0. The syntax is:

⟨if-then-statement⟩ ← if ⟨condition⟩ then { ⟨list-instruction⟩ }

The evaluation of the expression *condition* must return an integer value d. If the value d is nonzero then the list of instructions *list-instruction* is executed.

Example:

```
if n=0 then { n:=1; }
```

7.3.2 The if-then-else Statement

The `if-then-else` statement executes a list of instructions, either *success* or *failure*, depending on the evaluation of the expression *condition*. The evaluation of the expression *condition* must return an integer value (*d*). The flow of control is directed to the list of instructions *success* if the evaluation of the expression *condition* is nonzero, and to the list *failure* if *condition* is evaluated to zero.

⟨if-then-else-statement⟩ ← if ⟨condition⟩ then ⟨success⟩ else ⟨failure⟩

⟨success⟩ ← { ⟨list-instruction⟩ }

⟨failure⟩ ← { ⟨list-instruction⟩ }

Example:

```
if m<n then {
  v:=m;
} else {
  v:=n;
}
```

7.3.3 The switch Statement

The switch statement chooses one of several flows of control depending upon a criterion. The criterion must be either a variable or a variable with a field. Its type must be compatible with the values found in the case statements, as if they were used with the "=" operator. Its syntax is:

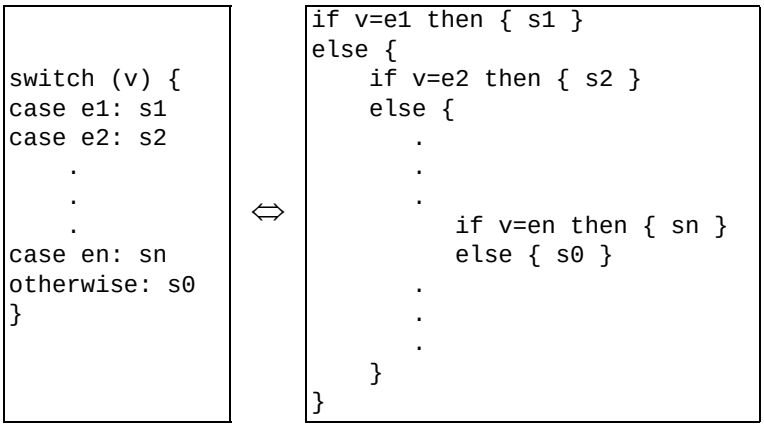
⟨switch-statement⟩ ← switch (⟨variable⟩)
{ ⟨case-list⟩ ⟨default⟩ }

⟨case-list⟩ ← Ø | ⟨case-list⟩ ⟨case-sttmt⟩

⟨case-sttmt⟩ ← case ⟨expression⟩ : ⟨list-instruction⟩

⟨default⟩ ← Ø | otherwise : ⟨list-instruction⟩

The meaning of such a statement can be described by another equivalent if-then-else statement as showed below:



If the *default* clause is not present, then the list s_0 is considered empty. This translation of the switch statement is in no way an explanation of the compilation process, so the order of evaluation of the e_i expressions is not garanteed by the Voyager 2 language. Therefore, expressions for which the evaluation has a side effect are discouraged since the semantics is unspecified.

When the switch statement is executed the value of the variable is compared to each expression e_i until values are equal. Once this condition is satisfied, the respective list of instructions is executed. If all tests fail, then no instruction is executed unless the *default* case is present in which its associated list of instructions is executed.

Example:

```
switch (letter) {
case 'B':
  print("Belgium");
  print(" (Belgique)");
case 'F':
  print("France");
case 'S':
  print("Spain");
  print(" (Espagne)");
otherwise:
  print("I don't know!");
}
```

7.4 Iteration Statements

Iteration statements are the `while`, `repeat` and `for` instructions.

7.4.1 The while Statement

The `while` statement has the following syntax:

⟨while-statement⟩ ← while ⟨condition⟩ do ⟨body⟩

⟨condition⟩ ← ⟨expression⟩

⟨body⟩ ← ⟨list-instruction⟩

The evaluation of the *condition* must be an integer value. While the evaluation of this expression will return non-zero, the *body* will be executed. The iteration stops when the evaluation of the *condition* returns the value 0.

Example:

```
f:=1;
while n>0 do {
  f:=f*n;
  n:=n-1;
}
```

7.4.2 The repeat Statement

The `repeat` statement has the following syntax:

⟨repeat-statement⟩ ← repeat ⟨body⟩ until ⟨condition⟩

⟨body⟩ ← ⟨list-instruction⟩

⟨condition⟩ ← ⟨expression⟩

The evaluation of the expression *condition* must return an integer value. The *body* is executed until the evaluation of the *condition* returns a nonzero value.

Example:

```
n:=read(_integer);
repeat {
  n:=n-1; } until n=0;
```

7.4.3 The for Statement

The `for` statement is probably the most usual instruction for doing iterations among a set of values. The original feature of this statement is the iteration through the elements of lists. This characteristic allows it to be used to visit references coming from the evaluation of a request. Its syntax is:

⟨for-statement⟩ ← for ⟨iterator⟩ in ⟨list⟩ do ⟨body⟩

⟨iterator⟩ ← ⟨variable⟩

⟨list⟩ ← ⟨expression⟩

⟨body⟩ ← ⟨list-instruction⟩

The evaluation of the expression *list* must return a value of type list. Moreover, each element of this list must be of the same type as the variable *iterator*. If the list is empty, this statement has no effect except the evaluation of the *list*. Otherwise, the variable *iterator* is instantiated with the first value found in the list, and the *body* is executed. After the first execution of the body, *iterator* is initialized with the second value of the list and the body is executed again; and so on with all the values of the list. When the for statement is completed, the variable *iterator* is instantiated with the last value found in the list (the value is unspecified if the list is empty).

Let us note that the time at which the list is evaluated is left unspecified, so it is recommended to avoid any instruction that could have a side-effect on the evaluation of the list. The following examples show very dangerous programs:

Examples:

```
for i in [1..n] do {
  print(i);
  n:=n+1; }

b:=1;
for i in [a,b] do {
  b:=2;
}

l:=[1..10];
for i in l do{
  l:=l++[11];
}
```

Here follows some correct use of the for instruction.

Examples:

```
for i in [1..5+5] do {
  print(i);
}

for data in DATA_OBJECT[data]{@SCH_DATA:[sch]
                              with data.name="Foo"} do {
  print(data.short_name);
  for gr in GROUP[gr1]{@DATA_GR:[data]} do {
    print(gr.name);
  }
}

for c in ['a','b']++['e','f'] do {
  print(CharToUpper(c));
}

for my_list in [[1,2],[3,4],[5]] do {
  print(my_list);
}
```

7.4.4 The goto Statement

The `goto` statement directs the flow of control to a statement labeled by an identifier. If this instruction is used inside a function, the flow of control cannot go out of the body of the function. In the same way, the control flow cannot be directed from the main body to a function. See 7.4.5 for a detailed example.

⟨goto-statement⟩ ← goto ⟨identifier⟩

7.4.5 The label Statement

The `label` statement is used to put a label in front of a statement. The syntax is:

⟨label-statement⟩ ← label ⟨identifier⟩

Example:

```
i:=0;
label loop;
if i<10 then {
  i:=i+1;
  goto loop;
}
```

7.4.6 The break Statement

The `break` instruction can only be used in `for-in-do`, `while` and `repeat` instructions. For the `while` and `repeat` instructions, the effect of this instruction is equivalent to a `goto` instruction to a label put just after the `while/repeat` instruction. This "breaks" the loop.

For the `for-in-do` instruction, the explanation depends on the list used for the iteration. If the list expression is a predicative query, then the effect of the break instruction is to skip to the brother of the father of the current item. Let us consider the following example:

Example:

```
1:  owner_of_att: o,o1,o2,o3;
2:  attribute: a;
        .
        .
3:      .
4:  for a in ATTRIBUTE[a]{@OWNER_ATT:[o1,o2,o3]} do {
5:    if GetFirst(OWNER_OF_ATT[o]{OWNER_ATT:[a]})=o1
6:    then { break; }
7:    print(a.name);
8:  }
```

where o1 owns three attributes: a1, a2 and a3 ; and o2 owns two attributes: b1 and b2. Then the first attribute at line 5 will be a1. Since the owner of a1 is o1, the test succeeds and the break instruction is called. Its effect will be to skip all the sons of the o1 owner and to go directly to the next owner: o2. The program will thus print: b1, b2, …

If the list expression is not a request, then the break instruction will break the loop and will continue with the instruction following the `for-in-do` instruction.

7.4.7 The continue Statement

The `continue` instruction can only be called inside `for-do-in`, `while` and `repeat` instructions. In the `while/repeat` instructions, `continue` will skip the rest of the body of the instruction, and cause the reevaluation of the *condition* expression.

If the instruction is used inside the `for-do-in` instruction with a request as list-expression, then its effect will be different. Let us get a look at the followiing example:

Example:

```
owner_of_att: o,o1,o2,o3;
attribute: a;
    .
    .
    .
for a in ATTRIBUTE[a]{@OWNER_ATT:[o1,o2,o3]} do {
  if a=a2
  then { continue; }
  print(a.name);
}
```

where the context is the same as in 7.4.6. When the current item becomes a2, then the `continue` instruction is called. The instruction will skip the rest of the body and will look for the brother of the current element, a2, which is a3. If a2 were the last son of the owner, o1, then the `for-do-in` instruction will terminate the processing the o1 ’s sons and go to the next owner: o2.

If the list expression is not a request, then the continue expression will just skip the rest of the body for the current element and will process the next value in the list.

7.4.8 The halt Statement

The `halt` instruction can be called anywhere in the program where an instruction is expected. This instruction stops the program. Similarly as a normal termination of the program, this instruction will not close opened files.

7.4.9 The interrupt statement

The `interrupt` instruction can be called anywhere in the program where an instruction is expected. This instruction pauses the execution of the program. The user can decide to restart it using the *Continue plug-in* function of the DB-MAIN case tool.

Chapter 8

Operations

Operations are statements having the form of a call to a predefined procedure/function. Because their syntax has already been defined, we have isolated them from other statements.

8.1 Operations on Characters

---

function integer: d CharIsDigit (char: c)

Checks whether a character is a digit or not.

Precondition. Ø

Postcondition. d ≠ 0 if c ∈ {0,...,9} and d=0 otherwise.

---

function integer: d CharIsAlpha (char: c)

Checks whether a character is a letter, either lower case or upper case, but always without accent.

Precondition. Ø

Postcondition. d ≠ 0 if c ∈ {a,...,z,A,...,Z} and d=0 otherwise.

---

function integer: d CharIsAlphaNum (char: c)

Checks if a character is either a digit or a letter, lower or upper case, but without accent.

Precondition. Ø

Postcondition. d ≠ 0 if c ∈ {0,...,9,a,...,z,A,...,Z} and d=0 otherwise.

---

function string: s CharToStr (char: c)

Builds a new string containig a single specified character.

Precondition. Ø

Postcondition. s is a string composed of the c character.

---

function char: c' CharToUpper (char: c)

Converts a letter to upper case.

Precondition. Ø

Postcondition. if c ∈ {a,...,z} then c' is the respective upper case letter. All other characters are left unchanged.

function char: c' CharToLower (char: c)

Converts a letter to lower case.

Precondition. Ø

Postcondition. if c ∈ {A,...,Z} then c' is the respective lower case letter. All other characters are left unchanged.

function char: c AscToChar (integer: d)

Initializes a character variable with a character whose ASCII code is specified.

Precondition. Ø

Postcondition. Character c has the ASCII code: d.

on error: c = '^0^'.

function integer: d CharToAsc (char: c)

Retrieves the ASCII code of a character.

Precondition. Ø

Postcondition. d is the ASCII code of the character c.

8.2 Operations on Strings

In order to simplify the notations, the definitions below use the following conventions:

- Let s denote a string and d a positive number. We note:
  - s_d the d^{th} character of the string s
  - $s_{d\rightarrow}$ the suffix of the string s starting at the position d (included)
  - $s_{d\rightarrow d+n}$ the substring comprised between positions d and d+n where n is a positive number such that d+n does not exceed the length of the string.
- Let us remember that the first character of a string is placed at the position 0, and thus if n is the length of s, the last character is placed at the position n-1.

The following operations are safe with respect to two criteria:

- The program can never write a character outside strings.
- The program can never place the null character inside a string[1].

This is a valuable guaranty against frequent bugs that C and Pascal programmers certainly know.

function string: s StrBuild (integer: d)

Builds a new string made of d spaces.

Precondition. d ≥ 0 and d ≤ MAX_STRING

Postcondition. s is a string composed of d space characters (' ').

on error: s is the empty string.

function string: s StrConcat (string: s_1, string: s_2)

Concatenate two strings.

Precondition. Ø

Postcondition. This function appends the string s2 at the end of s1 and the result is stored in s. The length of the resulting string is StrLength(s_1)+StrLength(s_2). The infix operator "+" can also be used in place of the StrConcat function.

1. By convention, the null character ends strings. Therefore such a possibility is troubling the memory manager.

function integer: r StrFindChar (string: s, integer: d, char: c)

Looks for the first appearance of a specific character in a string suffix.

Precondition. $0 \leq d < \text{StrLength}(s)$.

Postcondition. If $r \geq 0$ then $s_r = c$ and $\forall\ i, d \leq i < r: s_i \neq c$. Otherwise if $r = -1$ then $\forall\ i, d \leq i < \text{StrLength}(s): s_i \neq c$.

on error: r = -1.

function integer: r StrFindSubStr (string: s, integer: d, string: t)

Looks for the first appearnace of a string in the suffix of another string.

Precondition. $0 \leq d < \text{StrLength}(s)$.

Postcondition. If $r \geq 0$ then t is a prefix of $s_{d+r\rightarrow}$. Otherwise if $r = -1$ then $\forall\ i \geq 0$, t never is a prefix of $s_{d+i\rightarrow}$.

on error: r = -1

function char: c StrGetChar (string: s, integer: d)

Returns the d-th character of the string.

Precondition. $0 \leq d < \text{StrLength}(s)$.

Postcondition. $c = s_d$.

on error: c = '^0^'

function string: r StrGetSubStr (string: s, integer: d, integer: n)

Returns the substring of a starting at position d, of length n.

Precondition. $0 \leq d < \text{StrLength}(s) \wedge 0 \leq n \leq \text{StrLength}(s)\text{-}d$

Postcondition. $r = s_{d\rightarrow d+n}$

on error: if $d < 0$ then $d \leftarrow 0$; if $d \geq \text{StrLength}(s)$ then $d \leftarrow \text{StrLength}(s)$; if $n < 0$ then $n \leftarrow 0$; if $n > \text{StrLength}(s)\text{-}d$ then the function will consider that n-StrLength(s)+d space characters are added at the end of the string s.

function string: s StrItos (integer: d)

Converts the integer d into the string s.

Precondition. Ø

Postcondition. s is a string containing the ASCII representation of d.

function integer: d StrLength (string: s)

Returns the length of a string.

Precondition. Ø

Postcondition. d is the length of the string s.

function string: s' StrSetChar (string: s, integer: d, char: c)

Replaces the d-th character of the string s by character c.

Precondition. $0 \leq d < \text{StrLength}(s)$ and $c \neq$ '^0^'.

Postcondition. $\forall\ i \in \{0 \ldots \text{StrLength}(s)\text{-}1\} \setminus \{d\}: s'_i = s_i$ and $s'_d = c$, where s' stands for the state of the string after the execution of the function.

on error: s' = s.

function integer: d StrStoi (string: s)

Converts a string containing the representation of an integer number into that integer number.

Precondition. 1) The number represented by s is a number between INT_MIN and INT_MAX. 2) The string must match this regular expression: [\t]*[+-]?[0..9]+ (see appendix F for more details about regular expressions). The string may start with spaces or tabular characters but must end with a number. A number may have a sign (+ or -) and must have at least one digit.

Postcondition. Converts the longest prefix of s satisfying the above regular expression to an integer d. Space and tabular characters at the beginning of s are omitted.

on error: If the value d is outside the integer range, then the result d is undefined. If the string s does not match the regular expression, then d = 0.

---

function string: s' StrToLower (string: s)

Converts all the letters of a string to lower case.

Precondition. Ø

Postcondition. All the characters $c \in \{A,...,Z\}$ in the string s are replaced by their corresponding lower case letters, the result is stored in s'. No other characters are changed.

---

function string: s' StrToUpper (string: s)

Converts all the letters of a string to upper case.

Precondition. Ø

Postcondition. All the characters $c \in \{a,...,z\}$ in the string s are replaced by their corresponding upper case letters, the result is stored in s'. No other characters are changed.

---

function integer StrCmp (string: s_1, string: s_2)

Compares two strings and tells which one comes first in lexico-graphical order.

Precondition. Ø

Postcondition. Returns 0 if $s_1 = s_2$, 1 if $s_1 > s_2$ and -1 otherwise.

---

function integer StrCmpLU (string: s_1, string: s_2)

Compares two strings and tells which one comes first in alphabetical order. Note that letters with accents are not considered equivalent to the basic letter.

Precondition. Ø

Postcondition. Returns 0 if $s_1' = s_2'$, 1 if $s_1' > s_2'$ and -1 otherwise, where s_i' = StrToUpper(s_i) and $i \in \{1,2\}$.

---

function integer StrIsInteger (string: s)

Checks whether a string contains figures only or not.

Precondition. Ø

Postcondition. Returns 1 if CharIsDigit(s_i) = 1 $\forall$ $0 \le i \le$ StrLength(s)-1 and 0 otherwise.

See also MakeChoice and MakeChoiceLU in chapter 10 (pages 57).

8.3 Operations on Lists and Cursors

---

procedure AddFirst (list: l1, any: e)

Adds a new element at the begining f a list.

Precondition. Ø

Postcondition. After evaluation of the expression e, the result is added to the list l1 at the first position. If the expression is a list, this list is shared by l1.

procedure AddLast (list: l1, any: e)

Adds a new element at the end of a list.

Precondition. Ø

Postcondition. After evaluation of the expression e, the result is added to the list l1 at the last position. If the expression is a list, this list is shared by l1.

function any: r GetFirst (list: l)

Retrieves the first element of a list.

Precondition. l is a non-empty list

Postcondition. r is the first element of the list l. Of course, if the first element of a the list is a list, then the result is not a copy of it but shares it.

on error: the program is halted.

function any: r GetLast (list: l)

Retrieves the last element of a list.

Precondition. l is a non-empty list

Postcondition. r is the last element of the list l. Of course, if the last element of a the list is a list, then the result is not a copy of it but shares it.

on error: the program is halted.

function integer : n Length (list: l)

Return the length of a list.

Precondition. Ø

Postcondition. n is the number of elements found in the list l.

function cursor: c member (list: l, any : m)

Checks whether a value m is stored in the list l.

Precondition. Ø

Postcondition. if the element m occurs in the list l, then the cursor c points to this element. If the elements occurs more than once, then c points to the first occurrence. Elements that have a type different of the element m are omitted. If m does not belong to the list then the cursor c is void.

8.4 Operations on Files

function file OpenFile (string: FileName, integer: Mode)

Opens a file for reading or writing.

Precondition. *FileName* is the name of a file. *Mode* is an integer constant among: _W for the write mode and _R for the read mode and _A for the append mode.

Postcondition. Depending on the value of *Mode*:

- **_W :** If the file *FileName* exists then it is destroyed and the result is a handle to a new file opened for writing only. If *FileName* is not a valid name, the result is void and the error register is set to ERR_FILE_OPEN.
- **_R :** If the file *FileName* does not exist, the result is the value void and the error register is set to ERR_FILE_OPEN. Otherwise the result is a handle to the file opened for rea-

ding. The *current position* is either the first character of the file or the end of file if the file is empty.

– **_A :** If the file *FileName* exists then the function returns a handle to this file opened for writing, the cursor being placed at the end-of-file. Otherwise, the file is created and the function behaves like the mode was _W.

---

procedure CloseFile (file: f)

Closes an opened file. This function must be called for each opened file before a program terminates.

Precondition. *f* denotes an handle to a file opened with the instruction OpenFile.

Postcondition. The file is closed, and the value of *f* is undefined.

on error: The error register is set to ERR_FILE_CLOSE.

---

procedure printf (file: f, any: value)

Writes a string, a chart, an integher or a list of these types in a file.

Precondition. *f* denotes a file opened for writing and *value* is any expression among types string, char, integer, list.

Postcondition. *value* is written on the file denoted by *f*. If the type of *value* is list then all the values found in this list are written on the file surrounded by the string constants LEFT,RIGHT and separated by the string constant COMMA[2] (cfr. function SetPrintList page 41 for more details about these constants). Values not belonging to types string, char, integer, list are skipped.

on error: The behavior is undefined.

Let us remark that very deep and recursive lists perturb this procedure.

---

procedure print (any: value)

The procedure `print` bahaves like `printf`, but only requires the value argument, and writes the value on the console.

---

function any readf (file: f, integer: t)

Reads a value of a specific type in a file.

Precondition. *f* denotes a file opened for reading and *t* is an integer constant denoting the type of value to be read in the file. Following constants can be used: _integer, _char, _string.

Postcondition. Upon the value of *t*, the instruction will behave like this:

– **_integer :** The longest sequence of decimal digits optionally preceded by - or + is read from the *current position*. At the end, the *current position* is either the first character after the sequence or the end of file. If *current position* is either the end of file or is not indicating a number, then 0 is returned. If the sequence denotes a number outside the range [INT_MIN…INT_MAX] then the instruction returns a random integer.

– **_string :** The longest sequence of characters before either the end of file or the first character ’ \ n’ or the MAX_BUFFER*th* character after the *current position*. If the *current position* is the end of file or is indicating the end of line character, then the empty string is returned. The *current position* becomes either the end of file or the first character after the sequence.

– **_char :** The character under the *current position* is returned. If the end of file is reached, the ASCII code 0 is returned.

2. These constants are internal and are not visible.

function any readf (finteger: t)

The function `read` behaves like `readf` except that characters are read from the console.

function integer eof (file: f)

Checks whether the end of file is reached.

Precondition. The file *f* is opened.

Postcondition. eof returns 1 if the end of file is reached and 0 otherwise.

function integer neof (file: f)

Checks that the end of file has not been reached.

Precondition. The file *f* is opened.

Postcondition. neof returns 0 if the end of file is reached and 1 otherwise.

For files opened for writing, the function always returns 0. C programmers will note that the function neof is quite different of the function feof in this language.

Some other instructions are discussed here although they have no concern with the type file.

procedure rename (string: OldName, string: NewName)

Renames or moves a file.

Precondition. *OldName* is the name of an existing file. *NewName* is a file name that does not yet exist. Both expressions must denote files on a same physical device.

Postcondition. The file *OldName* is renamed *NewName*. If paths are different, then this instruction will move the file.

on error: the error register is set to ERR_ERROR.

procedure delete (string: filename)

Deletes a file.

Precondition. *filename* is the name of an existing file.

Postcondition. The file *filename* is deleted.

on error: the error register is set to one of the following values: ERR_PERMISSION_DENIED, ERR_PATH_NOT_FOUND.

function integer ExistFile (string: filename)

Checks whether a file exists with the specified name.

Precondition. *filename* is a valid file name for DOS. The file may not exist.

Postcondition. The function returns 1 if the file exists. Otherwise, error codes ERR_PERMISSION_DENIED and ERR_PATH_NOT_FOUND can be returned. The error register is not modified.

procedure SetPrintList (string: left, string: right, string: comma)

Configures the formating of printings in a file or in the console.

Precondition. *left*, *right* and *comma* are strings with no more than MAX_DELIM characters. Strings can be empty.

Postcondition. Strings *left*, *right*, *comma* are put into constants LEFT, RIGHT and COMMA.

The next example illustrates the use of the previous instructions.

Example:

```
file: f;
begin
  f:=OpenFile("c: \\ tmp \\ foo.txt",_W);
  SetPrintList("(",")",",");
  printf(f,[1,[1,2,3],4,' \ n']);
  SetPrintList(" \ /*","*/ \ n"," \ n");
  printf(f,["line comment 1","line comment 2","line comment 3"]);
  CloseFile(f);
end
```

The program will print the following text in the file *foo.txt*:

```
(1,(1,2,3),4)
/*line comment 1
line comment 2
line comment 3*/
```

8.5 Interface Operations

The following operations are illustrated with screen snapshots. Although the manual is written in English, my operating system has a French configuration, and therefore dialog boxes are a mix of French and English texts. French texts are system dependent messages and English are user's parameters defined below.

---

function string DialogBox (string : t, string : m, integer: s, string: d)

Shows a dialog box to allow the user to enter some text.

Precondition. Ø

Postcondition. A dialog box like the one shown in figure 8.1 is created with:

- **t**: the title (<TITLE> in the figure)
- **m**: the message (<MESSAGE> in the figure)
- **s**: the maximum length of the input area in characters
- **d**: the default string displayed in the input area (<DEFAULT VALUE> in the figure)

If the user clicks on the CANCEL button, the result is the empty string and the error register is set to ERR_CANCEL.

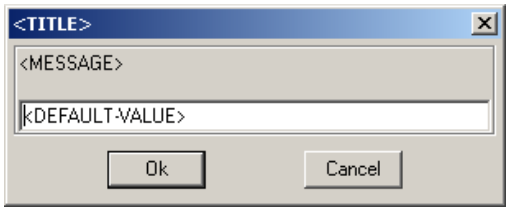


Figure 8.1 - A DialogBox Window.

---

function string BrowsePrint (string : t, string: e, string: d)

Shows a dialog box to allow the user to choose a filename and a place for saving a file.

Precondition. Ø

Postcondition. A dialog box like the one shown in figure 8.2 is created with:

- **t**: the title (<TITLE> in the figure 8.2).
- **e**: suggested extensions. This string is formatted as a list of pairs "`name_1|ext_1|name_2|ext_2|...|name_n|ext_n`" where `ext_i` is an extension ("*.v2" for instance), and `name_i` is its associated name ("Voyager 2 program" for instance).

– **d**: the initial directory to display ("C:\VOYAGER" in the figure 8.2).

If the user clicks on the CANCEL button, the result is the empty string and the error register is set to ERR_CANCEL. Otherwise, the result is the name of the selected file (with its path). The user may either choose an existing file or type a new name

.

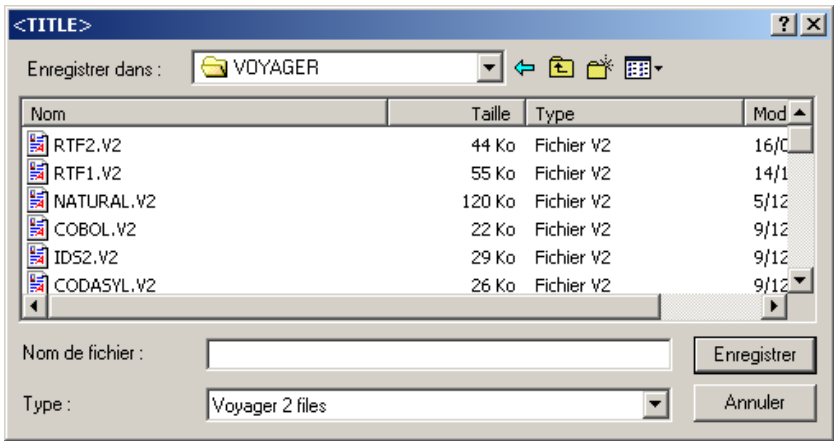


Figure 8.2 - A File Browsing Window.

---

function string BrowseRead (string : t, string : m, string: e)

Shows a dialog box to allow the user to choose a file to open.

Precondition. Ø

Postcondition. A dialog box as the one shown in figure 8.2 is created with:

– **t**: the title (<TITLE> in the figure 8.2).

– **e**: suggested extensions. This string is formatted as a list of pairs "`name_1|ext_1|name_2|ext_2|...|name_n|ext_n`" where `ext_i` is an extension ("*.v2" for instance), and `name_i` is its associated name ("Voyager 2 program" for instance).

– **d**: the initial directory to display ("C:\VOYAGER" in the figure 8.2).

If the user clicks on the CANCEL button, the result is the empty string and the error register is set to ERR_CANCEL. Otherwise, the result is the name of the selected file (with its path). Although file names are greyed, the user can type a new file name.

---

procedure MessageBox (string : t, string : m)

Displays a textual message for the user.

Precondition. Ø

Postcondition. A messge box like the one shown in the figure 8.3 is displayed with:

– **t**: the title (<TITLE> in the figure)

– **m**: the message (<MESSAGE> in the figure)

–

Figure 8.3 - A Message Box.

function integer: r Choice (string : t, list : L, integer: s, integer: m)

Precondition. Ø

Postcondition. A dialog box is shown to allow the user to choose one item among several ones.

- **t**: the title
- **L**: the list of possible items. The elements of L which do not match the string type will be omitted.
- **s**: the listbox will be sorted if $s \neq 0$.
- **m**: if not 0, an item must be selected before the user can click on the OK button.

If the user clicks on the CANCEL button, the result is -2. If the choice is not mandatory (m = FALSE) and if no item is selected, then the result is -1. Otherwise, the result is the index of the string in the list (the first index is 0). (See the example in figure 8.4).

Example:

```
Choice("Choose your favorite author",
       [ "Malet, Léo",
         "Steeman, Stanislas-André",
         "Ray, Jean", "Simenon, Georges",
         "Mayence, Bruce",
         "Tabachnik, Maud" ],
       TRUE,
       TRUE
      );
```

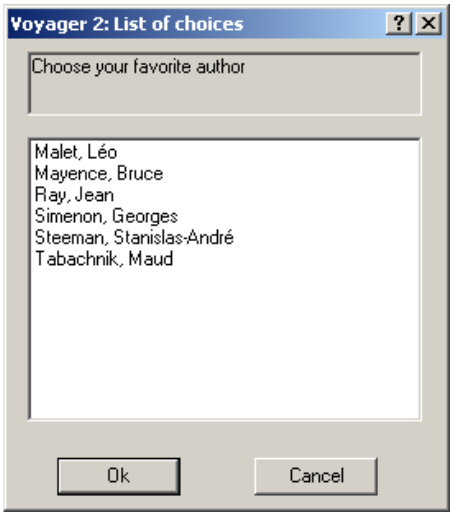


Figure 8.4 - A Choice Dialog.

8.6 Time Operations

function integer GetDay ( )

Gets the current day.

Precondition. Ø

Postcondition. returns the day (1-31) of the current date.

function integer GetHour ( )

Gets the hour of the current time.

Precondition. Ø

Postcondition. returns the hours (0-23) of the current time.

---

function integer GetMin ( )

Gets the minutes of the current time.

Precondition. Ø

Postcondition. returns the minutes (0-59) of the current time.

---

function integer GetMonth ( )

Gets the current month.

Precondition. Ø

Postcondition. returns the month (1-12) of the current date.

---

function integer GetSec ( )

Gets the seconds of the current time.

Precondition. Ø

Postcondition. returns the seconds (0-59) of the current time.

---

function integer GetWeekDay ( )

---

gets the day of the week of the current day.

Precondition. Ø

Postcondition. returns the day in the week (1-7) of the current date.

---

function integer GetYear ( )

Gets the current year.

Precondition. Ø

Postcondition. returns the year of the current date.

---

function integer GetYearDay ( )

Gets the number of the current day in the current year.

Precondition. Ø

Postcondition. returns the day in the year (1-365) of the current date.

8.7 Flag Operations

Each object of the repository of DB-MAIN contains an array of 32 boolean flags which are used for many purposes. The following functions can be used to get the value and set the value of these flags.

---

function integer: r GetFlag (integer: d, integer: p)

Gets the boolean value of a particular flag.

Precondition. Ø

Postcondition. Returns the bit stored at position p in the integer d. The value p is a binary mask to access the bit: to access the bit i of d, the mask should be the ith power of 2. For instance, to access flag 0, p should be 1. To access bit 5, p should be 32. Note that several bits can be checked at the same time: by example, bits 2 and 3 can be checked with $p = 2^2 + 2^3 = 12$

---

function integer: r SetFlag (integer: d, integer: p, integer: v)

Changes the value of a particular flag.

Precondition. Ø

Postcondition. a new array of flags is returned where the bits accessible through mask p are set to 1 if v is non-zero, or set to 0 if v is 0. All the other bits of the array have the same value as the bit at the same position in array d. The mask p should be build in the same way as explained in the operation GetFlag(d,p). For instance, if d is the array of flag 001...0011010 (32 bits, shortened for readibility), SetFlag(d,6,1) will return the result 001...001**11**10.

8.8 General Operations

function integer GetPosX (user_view: v, generic_object: g)

Precondition. $g \neq Ø$

Postcondition. Returns the X coordinate, in 1/1000 of millimeter, of the generic object g in the user view v.

function integer GetPosY (user_view: v, generic_object: g)

Precondition. $g \neq Ø$

Postcondition. Returns the Y coordinate, in 1/1000 of millimeter, of the generic object g in the user view v.

function integer GetColor (user_view: v, generic_object: g)

Precondition. $g \neq Ø$

Postcondition. Returns the colour of the generic object g in the user view v.

procedure UpdatePosX (user_view: v, generic_object: g, integer: x)

Precondition. $g \neq Ø$

Postcondition. Updates the X coordinate of the generic object g in the user view v. The x value is the new position in 1/1000 of millimeter from the left border of the drawing sheet.

procedure UpdatePosY (user_view: v, generic_object: g, integer: y)

Precondition. $g \neq Ø$

Postcondition. Updates the Y coordinate of the generic object g in the user view v. The y value is the new position in 1/1000 of millimeter from the left border of the drawing sheet.

procedure UpdateColor (user_view: v, generic_object: g, integer: c)

Precondition. $g \neq Ø$

Postcondition. Updates the color of the generic object g in the user view v. The c value is the new color, its interpretation depends on the systemm configuration, so it cannot be detailed here; the user should preferably use values previously obtained with the function GetColor().

function integer GetOID (generic_object: g)

Precondition. $g \neq Ø$

Postcondition. Returns the technical identifier of the generic object. This value is unique and stable.

procedure call (string: s)

Precondition. s denotes a Windows or DOS application with optional arguments.

Postcondition. The program s is started and the Voyager 2 program continues its execution.

on error: The error register is set to the constant ERR_CALL.

---

procedure CallSync (string: s)

Precondition. s denotes a Windows or DOS application with optional arguments.

Postcondition. The program s is executed, the Voyager 2 program waits for s to be finished before continuing its execution.

on error: The error register is set to the constant ERR_CALL.

---

procedure ClearScreen ( )

Precondition. Ø

Postcondition. The screen (Voyager console) is cleared.

---

function any GetCurrentObject ( )

Precondition. Ø

Postcondition. Returns the currently selected object. If no object, or many objects, are selected, then the function returns Void(GENERIC_OBJECT).

---

function schema GetCurrentSchema ( )

Precondition. Ø

Postcondition. Returns the reference of the current schema. If there no current schema (the active window does not show a schema), the function returns the value void.

---

function integer: e GetError ( )

Precondition. Ø

Postcondition. Returns the value of the error register. The call puts the value 0 in the error register.

---

function string: s GetOxoPath ( )

Precondition. Ø

Postcondition. Returns a string with the path of the "oxo" file that contains the current program.

---

function integer: r GetType (any: v)

Precondition. Ø

Postcondition. This function returns the value denoting the accurate type of the value passed as argument. For instance, if the argument is a variable defined as ent_rel_type, then the possible results are ENTITY_TYPE and REL_TYPE. This function is useful for processing values of various types coming from heterogeneous lists.

---

function integer: r IsNoVoid (any: v)

Precondition. Ø

Postcondition. Returns not IsVoid(v). See function IsVoid for more details.

---

function integer: r IsVoid (any: v)

Precondition. Ø

Postcondition. Returns TRUE if the argument is null. If the type of the argument is list or string the result is always FALSE. For integers and characters, this predicate is true if the value is the integer 0 or the character '^0^'.

---

function any: o Void (integer: t)

Precondition. The integer constant t denotes a valid type of the language, except the types: list and string.

Postcondition. o is the special value *void* of the type denoted by t.

on error: The program halts.

---

procedure Transfo (schema: s, string: t)

Precondition. Ø

Postcondition. If the string t contains a valid global transformation script, it is applied to the schema s. Otherwise, the schema s is unchanged and the error register is set to the constant ERR_BAD_TRANSFO.

on error: The error register is set to the constant ERR_BAD_TRANSFO

The script is used by the advanced global transformation assistant of the DB-MAIN CASE tool. For ease of writing such a script, it is recommended to use that assistant and to copy the script from the assistant to the clipboard, then to paste it in the V2 source program (some characters may have to be escaped with "\" by hand).

A small example of the use of an advanced global transformation script follows:

```
schema : sch;
integer : err;

begin
  sch := GetCurrentSchema();
  if IsNoVoid(sch) then {
    Transfo(sch,"RT_into_ET(ROLE_per_RT(3 N) or ATT_per_RT(1 N))");
    err := GetError();
    if err=ERR_BAD_TRANSFO then {
      MessageBox("TestTrsf error","Invalid transformation script");
    }
  }
end
```

8.9 Blackox

8.9.1 BlackBoxP

---

procedure BlackBoxP (integer: c, ...)

The general format of the BlackBoxP procedure. The value of the first parameter determines the format of the next ones. Various correct values of c and the corresponding parameters are listed below.

Precondition. c is the identifying code of a function of DB-MAIN. The "..." denotes the arguments of the procedure. This procedure is described in the help file of DB-MAIN.

Postcondition. Postconditions depend on c

---

procedure BlackBoxP (BBP_NEW_LOG, string: file_name, schema: sch)

Precondition. Ø

Postcondition. The file file_name contains the log file of the schema sch.

---

procedure BlackBoxP (BBP_ADD_POINT_LOG, string: check_point, schema: sch)

Precondition. Ø

Postcondition. Adds the check point check_point in the log file of schema sch.

---

procedure BlackBoxP (BBP_TRACE, integer: on_off, schema: sch)

Precondition. Ø

Postcondition. Sets the trace of schema sch to on_off.

procedure BlackBoxP (BBP_REPLAY_AUTO, schema: sch, string: file_name, integer: r)

Precondition. Ø

Postcondition. Replays the file_name log file on sch (the errors are displayed if r = 1).

procedure BlackBoxP (BBP_DISPLAY_REF_VAR, text: fl, System: sys, integer: var_id)

Precondition. Ø

Postcondition. ???.

procedure BlackBoxP (BBP_SAVE_PS_CONSOLE, string: file_name)

Precondition. Ø

Postcondition. Saves the Voyager console in file_name.

procedure BlackBoxP (BBP_INTEGRATE_SCHEMA, schema: slave_sch, schema: master_sch, string: file_name)

Precondition. Ø

Postcondition. Integrates slave_sch into master_sch and stores the integration report into file_name.

procedure BlackBoxP (BBP_COPY, list: l)

Precondition. Ø

Postcondition. Copies the list of objects into the clipboard.

procedure BlackBoxP (BBP_PASTE, schema: sch)

Precondition. Ø

Postcondition. Pastes the content of the clipboard into the schema.

procedure BlackBoxP (BBP_DBL_CLICK, schema: sch, generic_object: go)

Precondition. Ø

Postcondition. Simulates a double click on go in schema sch.

procedure BlackBoxP (BBP_OPEN_WIN, schema : sch)

Precondition. Ø

Postcondition. Opens a window for showing schema sch.

procedure BlackBoxP (BBP_CLOSE_WIN, schema : sch)

Precondition. Ø

Postcondition. Closes the window of schema sch.

procedure BlackBoxP (BBP_MARK_SELECTED, schema : sch)

Precondition. Ø

Postcondition. Marks the selected objects of schema sch.

procedure BlackBoxP (BBP_DELETE_SELECTED, schema : sch)

Precondition. Ø

Postcondition. Deletes the selected objects of schema sch.

procedure BlackBoxP (BBP_CENTER_SELECTED, schema : sch)

Precondition. Ø

Postcondition. Places the selected object of schema sch in the center of the graphical or textual view.

procedure BlackBoxP (BBP_SELECT_MARKED, schema : sch)

Precondition. Ø

Postcondition. Selects the marked object of schema sch.

procedure BlackBoxP (BBP_CREATE_VIEW, schema : sch, string: name)

Precondition. Ø

Postcondition. Defines the view name on the marked objects of schema sch.

procedure BlackBoxP (BBP_GENERATE_VIEW, schema : sch, string: name, integer: ssatt, integer: proc, integer: rel)

Precondition. Ø

Postcondition. Generates the view name of schema sch with the sub-attributes (if ssatt = 1), the processing units (if proc = 1) and the relations (if rel = 1).

procedure BlackBoxP (BBP_DELETE_VIEW, schema : sch, string: name)

Precondition. Ø

Postcondition. Deletes the view name of schema sch.

procedure BlackBoxP (BBP_COPY_VIEW, schema : sch, string: name, string: cname)

Precondition. Ø

Postcondition. Copies the view name in the new view cname of schema sch.

procedure BlackBoxP (BBP_RENAME_VIEW, schema : sch, string: name, string: rname)

Precondition. Ø

Postcondition. Renames the view name of schema sch into rname.

procedure BlackBoxP (BBP_REFRESH_WIN, schema : sch)

Precondition. Ø

Postcondition. Refreshs the window containing schema sch.

8.9.2 BlackBoxF

The general format of the BlackBoxF function. The value of the first parameter determines the format of the next ones. Various correct values of c and the corresponding parameters are listed below.

function any BlackBoxF (integer: c, ...)

Precondition. c denotes a unique code that must correspond to some defined operation. The "..." denotes the arguments of the procedure. This procedure is described in the help file of DB-MAIN.

Postcondition. Postconditions depend on c

function integer BlackBoxF (BBF_SCHEMACOPY, schema: sch_org, schema: sch_targ)

Precondition. sch_targ is an empty schema.

Postcondition. sch_org is copied into sch_targ.

Returns 1 if the copy succeed, 0 otherwise

---

function integer BlackBoxF (BBF_IS_VALID_GR_COMPONENT, group: gr)

Precondition. Ø

Postcondition. Returns 0 if the group is valid.

< 0, if there is an error.

> 0 = number (+1) of the element that couldn't be part of the group.

---

function integer BlackBoxF (BBF_IS_VALID_IDENTIFIER_GR, group: gr)

Precondition. Ø

Postcondition. Returns 0 if the elements of gr are valid elements of an identifier.

Otherwise returns <0.

---

function integer BlackBoxF (BBF_IS_VALID_EXISTENCE_GR, group: gr)

Precondition. Ø

Postcondition. Returns 0, if gr is made up of valid components for an existence group.

Otherwise, returns <0

---

function list BlackBoxF (BBF_GET_DECLARED_VAR, document: doc)

Precondition. Ø

Postcondition. Returns the list '[var_id, var_name, level],...' of the variables declared in the COBOL source code 'doc'.

---

function list BlackBoxF (BBF_DIR, string: path)

Precondition. Ø

Postcondition. Returns the list of files of path. path can contain wildcards (à la DOS).

---

function schema BlackBoxF (BBF_IMPORT_ISL, string: sch_name, string: sch_version, string: file_name)

Precondition. file_name is an ".isl" file that contains the schema "<sch_name>/<sch_version>".

Postcondition. Returns the imported schema if the import succeed. Otherwise, returns Void .

---

function integer BlackBoxF (BBF_ET_TO_ATT, integer: elem, entity_type: et)

Precondition. Ø

Postcondition. Transforms the entity type et into an attribute.

Returns 1 if the transformation succeed, 0 otherwise.

If elem = 0, the newly created objects have default names, otherwise a dialog box asks for the name of the newly created objects.

---

function integer BlackBoxF (BBF_ATMUL_TO_LIST, integer: elem, attribute: att)

Precondition. Ø

Postcondition. Transforms att, if multivalued, into a list of single-valued attributes (= instantiation).

Returns 1 if the transformation succeed, 0 otherwise.

If elem = 0, the newly created objects have default names, otherwise a dialog box asks for the name of the newly created objects.

---

function integer BlackBoxF (BBF_RT_TO_ATT, integer: elem, rel_type: rt)

Precondition. Ø

Postcondition. Transforms rt into a reference attribute (foreign key).

Returns 1 if the transformation succeed, 0 otherwise.

If elem = 0, the newly created objects have default names, otherwise a dialog box asks for the name of the newly created objects.

---

function integer BlackBoxF (BBF_ISA_TO_RT, integer: elem, entity_type: et)

Precondition. Ø

Postcondition. Transforms the is-a relation (for which et is a super-type) into relationship type.

Returns 1 if the transformation succeeds, 0 otherwise.

If elem = 0, the newly created objects have default names, otherwise a dialog box asks for the name of the newly created objects.

---

function integer BlackBoxF (BBF_ATT_TO_ET_INST, integer: elem, attribute: att)

Precondition. Ø

Precondition. Transforms the attribute att into an entity type using the instance representation of duplicate values of attributes.

Returns 1 if the transformation succeeds, 0 otherwise.

If elem = 0, the newly created objects have default names, otherwise a dialog box asks for the name of the newly created objects.

---

function integer BlackBoxF (BBF_ATT_TO_ET_VAL, integer: elem, attribute: att)

Precondition. Ø

Postcondition. Transforms the attribute att into an entity type using distinct attribute values (*value representation*).

Returns 1 if the transformation succeeds, 0 otherwise.

If elem = 0, the newly created objects have default names, otherwise a dialog box asks for the name of the newly created objects.

---

function proc_unit BlackBoxF (BBF_FIND_PU_BY_NAME, owner_of_proc_unit: opu, string: s)

Precondition. Ø

Postcondition. Searchs the proc_unit that has the name s in the owner_of_proc_unit opu.

Returns the proc_unit if the search succeeds, Void otherwise.

Chapter 9

Functions and Procedures

9.1 Definition

Functions and procedures are program chunks that are identified by a name and which can be invoked from other procedures or functions, or from the main program. The scope of a function or a procedure is the whole program. The function or procedure definitions must be placed between the last global variable definition and the main program. Functions and procedures can have local variables (their scope is restricted to the body of the function) and return a value (the result) of any type (except function or procedure).

The syntax of a function definition is the following:

```
function <type> <identifier> (<arg>,< arg>,...)
[explain (* ... *)]
<definition-line>
...
{
  <instruction>
  ...
}
```

The syntax of a procedure definition is the following:

```
procedure <identifier> (<arg>,< arg>,...)
[explain (* ... *)]
<definition-line>
...
{
  <instruction>
  ...
}
```

In these definitions, <identifier> is the name of the function, <type> is the type of the result of the function, <arg> is an argument of the function, <definition-line> is a variable declaration as presented in Chapter 3 except that the scope of the variable is the body of the function and not the whole program, and <instruction> is the series of instructions to be executed when the function is invoked.

The flow of the executed instructions will stop either after the execution of the last instruction, or when a return instruction is encountered. In a function, the return instruction should be followed by an expression, of the same type as the function, whose result is the result of the function. If the function ends without encountering a return instruction, its result will be undefined and the execution of the program will probably be aborted a bit further.

When a function or procedure ends, all the local variables disappear from the environment and the memory is cleared. The eecution flow is then directed to the instruction following the function or procedure call.

All the arguments, except lists, are passed by value. That is to say that, at each call, the parameters are in fact a copy of the arguments, and modifiying the value of these parameters has no influence on the value of the argument in the calling function or procedure. But:

1. lists are passed by *reference*. This means that all the operations performed on a list parameter are also performed on the argument, every modification of the list inside the procedure or function will affect the list in the calling procedure or function.
2. when arguments are references to objects, although this argument is passed by value, the behavior of the program is the same as if the value was passed by reference. Indeed, only the reference is passed by value, not the whole object which is stored in the repository.

Note that a local list returned by a function will not be destroyed when the function ends, as explained before, because lists are managed by a *garbage collector* which sees that the list is used by both a local variable and the program calling the function. For example, let us suppose that `l` is a local list variable, and its value before the call to the return instruction is [1,2,3]. This list will still valid after the return instruction although the local variables should be destroyed when the function exits.

Finally, there is no implicit type casting of the values passed to a procedure or function to the type of the arguments specified in the signature of the procedure or function. This job must be done by the programmer. For instance, if a function expects a *data_object* as first argument, it is forbidden to pass expressions of another type (even *entity_type*) that is a subtype.

The following program illustrates the use of functions and procedures to print lists of factorials:

```
function integer fact(integer: n)
  integer: i, f;
{  f:=1;
  for i in [1..n] do {
    f:=f*i;
  };
  return f;
}

procedure PrintFact(integer: i, integer: j)
  integer: z;
  list: l;
{   for z in [i..j] do {
    l:=l++[fact(z)];
  };
  print(l);
}

begin
  SetPrintList("","",",");
  PrintFact(2,5);
end
```

The export and explain clauses that appear in the syntax are outside the scope of this chapter and will respectively be described in 17.2 and 17.5.

9.2 Recursiveness

The scope of a function is the whole program, including its own body. So a function can call itself. This principle is called *recursiveness*. For instance the factorial function shown above could be shortened in the following way:

```
function integer fact(integer: n){
  if n=0
  then { return 1; }
  else { return n*fact(n-1); };
}
```

In the same way, the PrintFact procedure could be rewritten as:

```
procedure PrintFact(integer: i, integer: j){
  if i<j
  then {
    print(fact(i));
    print(',');
    PrintFact(i+1,j);
  } else {
    print(fact(i));
  };
}
```

Chapter 10

Lexical Analyzer

Because strings are passed by value to functions and procedures in Voyager 2 and because characters are read one by one from files, writing an efficient lexical analyzers in Voyager 2 with the general purpose functions presented in the previous chapter is a real challenge. Since text file analysis is a common task, Voyager includes some specificic functions.

All these functions use the same *input stream* that is initialized by the function SetParser. The input stream can be either a file or a string. Because a stream is a little bit more sophisticated than a normal file (OpenFile), usual functions for files cannot be used with this *input stream*.

---

procedure SetParser (τ : sf)

Initialization of the input strem.

Precondition. τ is either the string type or the file type. This instruction specifies which stream will be used by the subsequent functions of the lexical library. If the argument is a string, then all the lexical functions will read characters from a "virtual" file initialized with the argument. Otherwise, if it is a file, characters are read from the file itself.

Postcondition. The input stream is initialized.

---

function string : r GetTokenWhile (string : s)

This function returns the longest string from the input stream made up of characters in s.

Precondition. The input stream is initialized. The argument must be a literal string: the value of s must be known at compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension (with a range of characters) or in expansion (with a list of characters). The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded in this way: for each occurence of a range "α - β" where α, β denote any character, this substring is replaced in the pattern by the set of characters γ: $\alpha \le \gamma \le \beta$. Thus the following pattern : "-a-d0-4" is equivalent to the string: "-abcd01234".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. The result of this function is the string $(\alpha_i)_1^m$ where $0 \le m \le n$ and $\forall i \in 1..m: \alpha_i \in s$ and if $m < n$ then $\alpha_{m+1} \notin s$. After the call, the input stream is replaced by $(\alpha_i)_{m+1}^n$.

function string : r GetTokenUntil (string : s)

This function returns the longest string from the input stream made up of characters not belonging to s.

Precondition. The input stream is initialized. The argument must be a literal string: the value of s must be known at compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension (with a range of characters) or in expansion (with a list of characters). The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded in this way: for each occurence of a range "α - β" where α, β denote any character, this substring is replaced in the pattern by the set of characters γ: $\alpha \le \gamma \le \beta$. Thus the following pattern : "-a-d0-4" is equivalent to the string: "-abcd01234".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. The result of this function is the string $(\alpha_i)_1^m$ where $0 \le m \le n$ and $\forall i \in 1..m: \alpha_i \notin s$ and if $m < n$ then $\alpha_{m+1} \in s$. After the call, the input stream is replaced by $(\alpha_i)_{m+1}^n$.

procedure SkipWhile (string : s)

This procedure skips, in the input stream, the longest series of characters in s.

Precondition. The input stream is initialized. The argument must be a literal string: the value of s must be known at compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension (with a range of characters) or in expansion (with a list of characters). The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded in this way: for each occurence of a range "α - β" where α, β denote any character, this substring is replaced in the pattern by the set of characters γ: $\alpha \le \gamma \le \beta$. Thus the following pattern : "-a-d0-4" is equivalent to the string: "-abcd01234".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. After the call, the input stream is replaced by $(\alpha_i)_{m+1}^n$ where m is defined as $0 \le m \le n$ and $\forall i \in 1..m: \alpha_i \in P$ and if $m < n$ then $\alpha_{m+1} \notin P$ where P is the pattern.

procedure SkipUntil (string : s)

This procedure skips, in the input stream, the longest series of characters not belonging to s.

Precondition. The input stream is initialized. The argument must be a literal string: the value of s must be known at compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension (with a range of characters) or in expansion (with a list of characters). The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded in this way: for each occurence of a range "α - β" where α, β denote any character, this substring is replaced in the pattern by the set of characters γ: $\alpha \le \gamma \le \beta$. Thus the following pattern : "-a-d0-4" is equivalent to the string: "-abcd01234".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. After the call, the input stream is replaced by $(\alpha_i)_{m+1}^n$ where m is defined as $0 \le m \le n$ and $\forall i \in 1..m: \alpha_i \notin P$ and if $m < n$ then $\alpha_{m+1} \in P$ where P is the pattern.

procedure UngetToken (string : s)

This procedure puts some characters back to the input stream.

Precondition. The input stream is initialized. Let $(\alpha_i)_1^n$ be the input stream. α_1 is the first character. Let $(\sigma_j)_1^m$ be the sequence of letters composing s.

Postcondition. The input stream is replaced by $(\sigma_j)_1^m$ o $(\alpha_i)_1^n$ where o is the "append" operator for lists.

---

function char : c GetChar ( )

Reads one character from the input stream.

Precondition. The input stream is initialized. There is at least one character in the input stream.

Postcondition. The first character of the input stream is removed and returned.

---

function integer seof ( )

This function checks whether the end of the input stream is reached or not.

Precondition. The input stream is initialized.

Postcondition. Let n be the value returned by this function. n = 0 if there is at least one character left in the input stream and $n \neq 0$ otherwise.

---

function integer nseof ( )

Precondition. This function checks whether the end of the input stream is not yet reached or not.

Precondition. The input stream is initialized.

Postcondition. Let n be the value returned by this function. $n \neq 0$ if there is at least one character in the input stream and 0 otherwise.

The following functions are not really in the lexical library. They should be defined in the section 8.2. But they are often used with lexical functions.

---

function integer: d MakeChoice (string: s, list: l)

This function looks for a string in a list of strings. For example, it can be used to check if a word that was just read from the input stream is in a list of reserved words with case sensitivity ("word", "Word" and "WORD" are different strings).

Precondition. l is a literal list of strings: this value must be known at compilation time. Let us note $(\sigma_i)_1^n$ the list l where σ_1 is the first element. All the values should be distinct.

Postcondition. if $\exists$ i $\in$ 1..n: $\sigma_i = s$ then d = i otherwise d = 0. The complexity of the this function is $\Theta(\log_2 n)$.

on error: If one value occurs several times in the list l, then the result is random. The compiler prints a warning message.

---

function integer: d MakeChoiceLU (string: s, list: l)

This function looks, case insensitively, for a string in a list of strings. For example, it can be used to check if a word that was just read from the input stream is in a list of reserved words, without taking care of the lower/upper case letters ("word", "Word" and "WORD" are the same strings).

Precondition. l is a literal list of strings: this value must be known at compilation time. Let us note $(\sigma_i)_1^n$ the list l where σ_1 is the first element. All the values should be distinct.

Postcondition. if $\exists$ i $\in$ 1..n: StrCmpLU (σ_i,s) = 0 then d = i otherwise d = 0. This comparaison is case insensitive. The complexity of the this function is $\Theta(\log_2 n)$.

on error: If one value occurs several times (*case insensitive*) in the list l, then the result is random. The compiler prints a warning message.

PART II

THE REPOSITORY

Chapter 11

Repository Definition

This chapter is devoted to a presentation of the repository of DB-MAIN. The repository is composed of objects. These objects are formally defined in chapter 12. The present chapter gives an overview as well as a semantic description of the repository.

The repository is an object base. It will be represented as a DB-MAIN ERA schema. Entity types represent classes. All the rel-types are many-to-one. All the attributes are single-valued.

The repository is too large to be presented on a single page. For this reason, the repository definition has been "exploded" in six views according to the various ontologies the repository can model. The first view (see Fig. 11.1) is a "macro-view". This corresponds mainly to the objects the software engineer can observe in the project window. The "data-schema" view (see Fig. 11.2) corresponds to the former definition, that is, the representation of the entity-relationship schemas. The third view (see Fig. 11.3) shows how notes are attached to all the objects of the repository. The fourth view (see Fig. 11.4) represents the process schema (statements, functions, expressions, …). The fifth view (see Fig. 11.5) denotes the persistent data that underlie the graphical representation of the schema (data and process). Finally, the last view (see Fig. 11.6) is just an overview of all the objects which inherit from the generic_object. Of course, all those views are intimately linked together and some objects appear in several views.

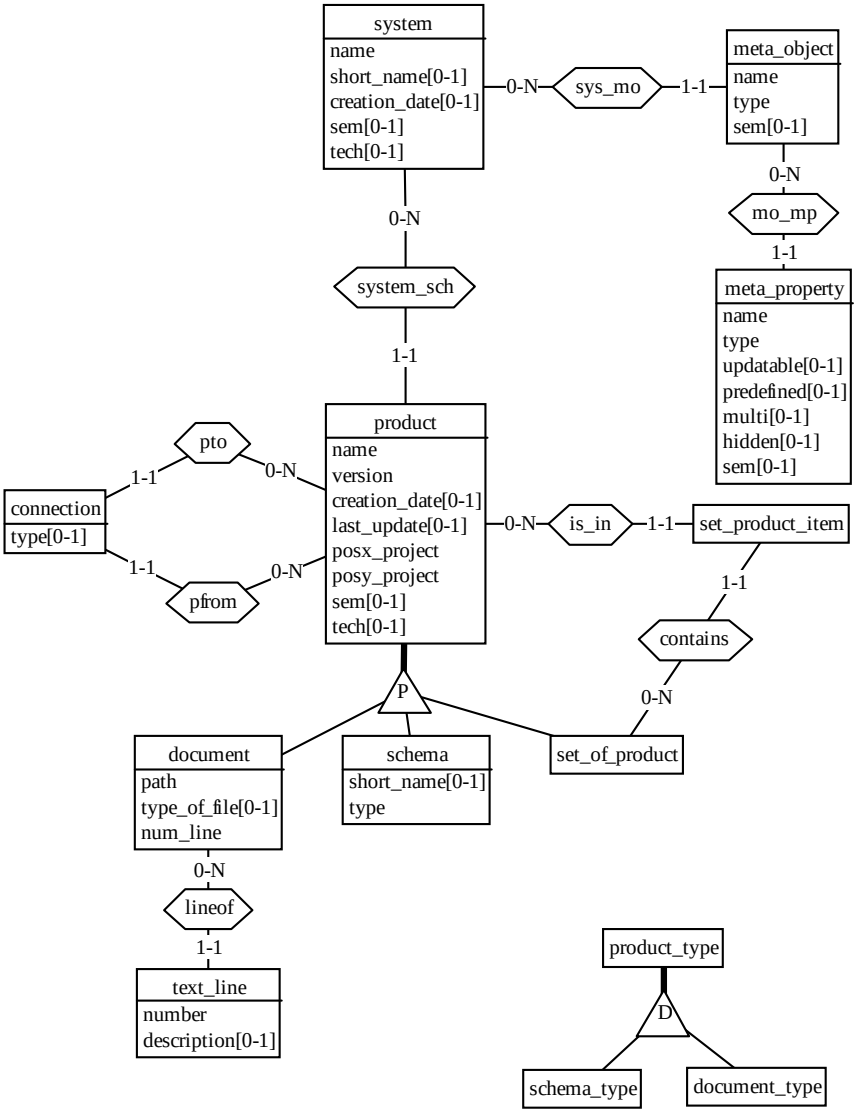


Figure 11.1 - The "macro" view.

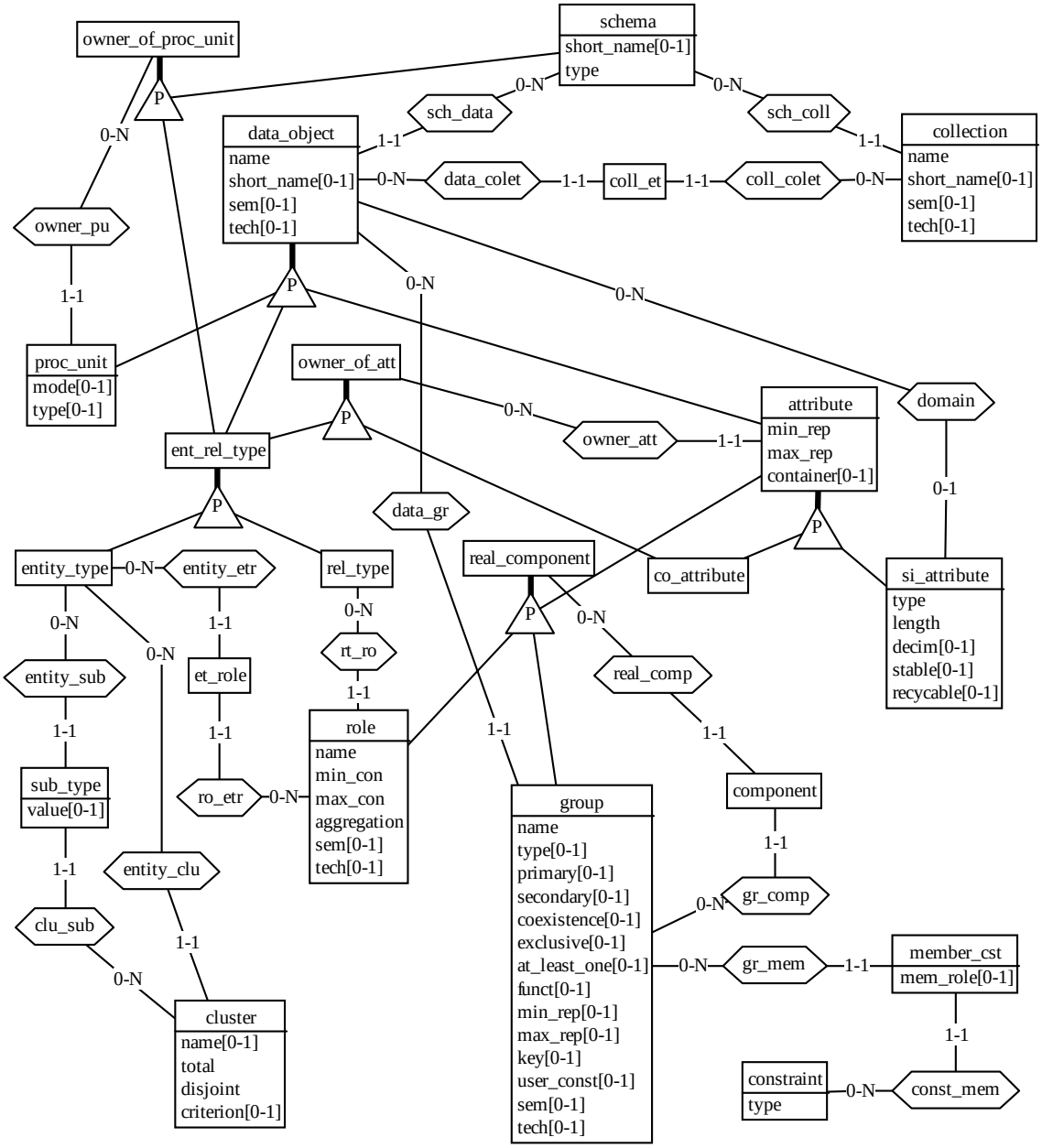


Figure 11.2 - The "data" view.

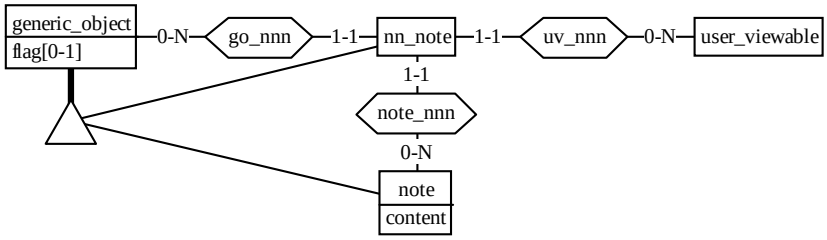


Figure 11.3 - The "notes" view

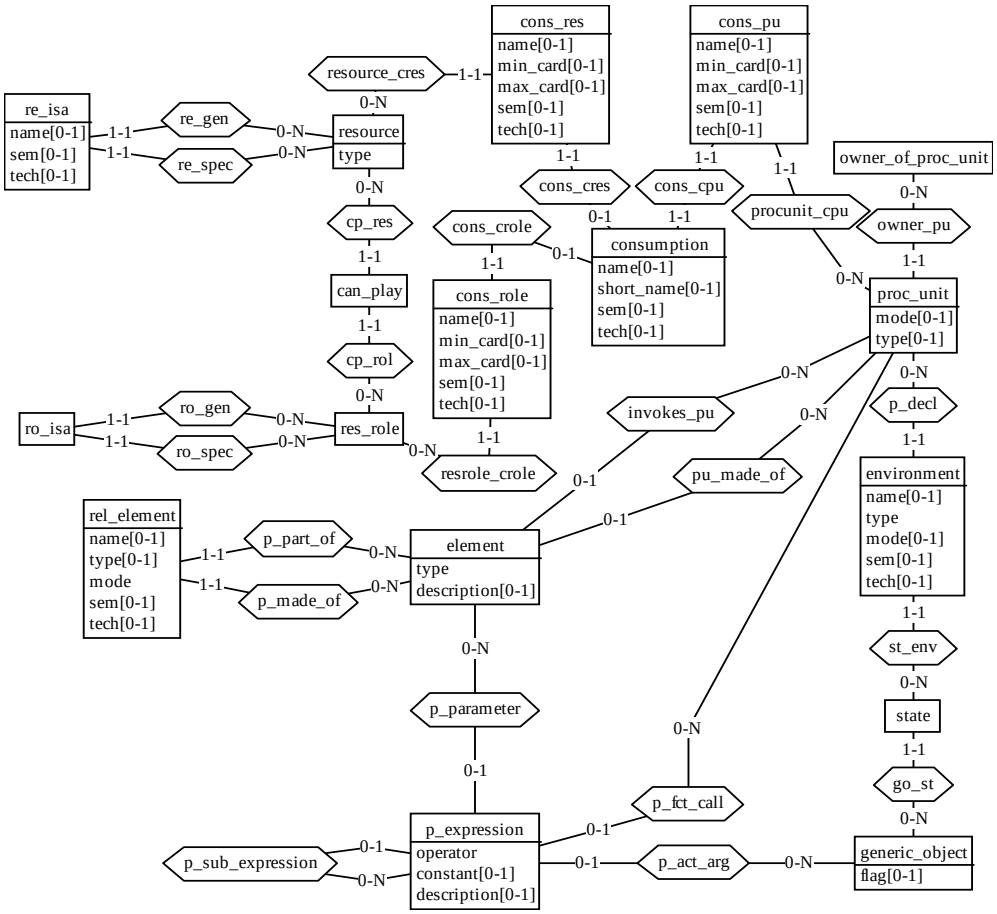


Figure 11.4 - The "process" view.

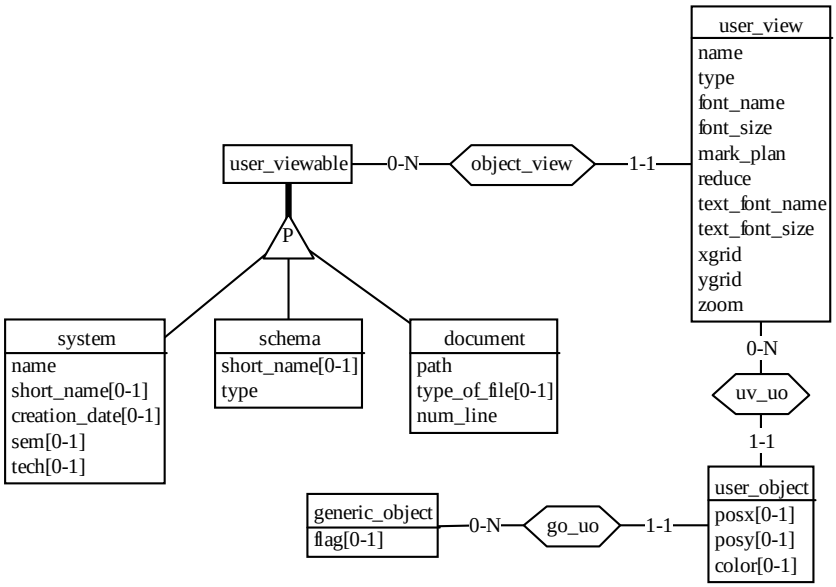


Figure 11.5 - The "graph" view.

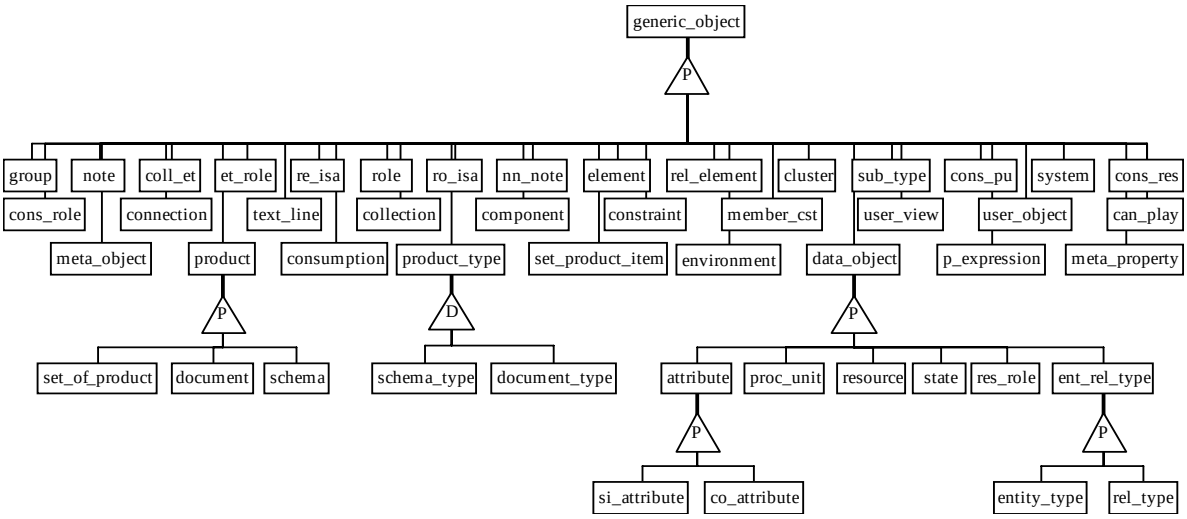


Figure 11.6 - The "inheritance" view.

The schema of the DB-MAIN repository presented above deserves some comments, some explanations about how DB-MAIN projects, schemas, and all the constructs in the schemas are stored in the repository. Each object type will be individually fully described and defined in chapter 12.

The repository is aimed at storing whole projects made of several products, possibly of different kinds, such as database schemas, processing schemas, textual documents, and product sets. For instance, to store an Entity/Relationship (ER) schema, concepts like entity types, attributes, relationships,... must be represented. The folowing pages show how each product and each concept that can be represented in these products are stored in the DB-MAIN repository.

To each concept corresponds an object type in the repository, such as the *entity_type* object type that corresponds to the entity type concept and the *co_attribute* object type to the compound attribute concept. However, some object types do not correspond to pertinent concepts. It is the case of the *component* object type, for example. The presence of these object types is due to technical reasons[1]. Finally, some object types are **virtual**, which means that they are a partition super-class of several other object classes. Instances of a virtual object type cannot exist in themselves, but instances of the object types which inherit of the virtual types can be seen and used as objects of the virtual object type. For instance, a object of type *attribute* cannot be created, but objects of types *si_attribute* or *co_attribute* can all be used as if they were of the type *attribute*.

11.1 Project

The very first object that must be created in the repository, when a project starts, is of the *system* type. There can be only one instance/object of this type in a project. All the products that appear in the project window of DB-MAIN are schemas, textual documents and sets of products, which are respectively represented by the object types *schema*, *document* and *set_of_product*. These three object types share common properties which are gathered in the *product* virtual object type. *product* instances are linked to the object of type *system* by the *system_sch* link.

Important note: Voyager 2 does not allow DB-MAIN users to manage the structured histories of DB-MAIN. Only the compact view of the project is accessible and manageable through Voyager. Only one exception to this rule, for supporting process modelling with MDL: when a new product is created with Voyager 2, a product type can be specified.

1. Our repository technology can not represent *many-to-many relationship types*.

11.2 ERA schema

11.2.1 Schema

An ERA schema is an instance of *schema* object type whose attribute *type* is set to the constant *ERA_SCHEMA*. A schema is made of data objects which can be gathered into collections. A data object is either an entity type, a rel-type, an attribute or a processing unit. *data_object* is a virtual object type which allows us to link all the data objects composing the schema to the schema through the *sch_data* link. All the collections of data objects of a schema are linked to the schema by the *sch_coll* link. The *schema* object type, inheriting the *owner_of_proc_unit* object type, can receive processing units linked by *owner_pu*.

11.2.2 Entity type

An entity type is stored as an object of the type *entity_type*. Using the inheritance mechanism, an *entity_type* being an *ent_rel_type* which is itself a *owner_of_att*, an entity type can receive several attributes, all of the first level, linked to it by the *owner_att* link. In the same way, an *ent_rel_type* being an *owner_of_proc_unit*, an entity type can receive some processing units using the *owner_pu* link. Finally, an *ent_rel_type* being also a *data_object*, an entity type can receive groups by the *data_gr* link.

11.2.3 Rel-type

A rel-type is stored as an object of the type *rel_type*. A *rel_type* being an *ent_rel_type*, a rel-type can possess attributes, processing units and groups in the same way as entity types.

11.2.4 Attribute

The representation of an attribute depends on its type:

- Compound attributes are represented by the *co_attribute* object_type. *co_attribute* inherits from *owner_of_att*, so each compound attribute can receive sub-attributes by the *owner_att* link. By this mechanism, any compound attribute of level *n* receives attributes of level *n+1*.
- Object attributes are represented by objects of the type *si_attribute* whose property *type* is set to the constant *OBJECT_ATT* and which are connected to a *data_object* by a *domain* link. This *data_object* used as the type of the object attribute must be an entity type.
- User-defined attributes are represented by objects of the type *si_attribute* whose property *type* is set to the constant *USER_ATT* and which are connected to a *data_object* by a *domain* link. This *data_object* used as the user-defined type must be an attribute. This attribute must belong to the special entity type *]DOMAINS-ATTRIBUTES[* which belongs to the special schema named *]DOMAINS[* (with version *]USER-DEFINED[*).
- All other attributes are represented by objects of the type *si_attribute* that are not linked by the *domain* link. The property *type* identifies the type of the attribute.

11.2.5 Processing unit

In an ERA schema, a processing unit can be attached either to a schema, to an entity type or to a rel-type. It is only shown by its name. In the repository it is represented by an object of the type *proc_unit*. For representing a more precise definition of a processing unit, the DB-MAIN user will use a UML activity or use case diagram.

11.2.6 Role

A role is represented in the repository by an instance of the *role* object type. Each role is always linked to one rel-type by a *ro_etr* link. But, the role can be played by several entity-types. And an entity type can play a role in several rel-types. *et_role* is a technical object type that represents a many-to-many rel-type between a role and an entity type. Each *et_role* is identified by its link *entity_etr* to *entity_type* and by its link *role_etr* to *role*.

11.2.7 Generalization/specialization

Entity types can be generalized or specialized. The repository allows an entity-type to be specialised in several ways. For instance, customers can be divided in male/female and have different characteristics

according to the gender, or they can be divided into children and adults with different characteristices too. To do so, an *entity_type* object can be linked to several objects of type *cluster*, one for the gender and one for the age in the example, and each cluster can have several sub-types. In the example, one cluster should have two sub-types which are entity types males and females, and the other cluster should have two sub-types too, which are young-people and adult-people.

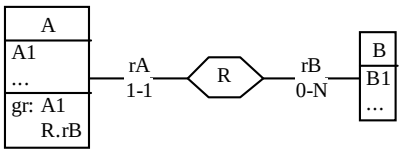
On the other end, an entity type can be generalised several times. For instance, adult-people can be a specialization of both customers and V.I.P. The many-to-many rel-type between *entity_type* and *cluster* is materialized by the *sub_type* object type, each *sub_type* being identified by its link *entity_sub* to *entity_type* and its link *clu_sub* to *cluster.*

In practice, DB-MAIN does not support multiple clusters, so each *entity_type* should not be linked to more than one *cluster.*

11.2.8 Group and constraint

A group can be made up of several components which can be attributes, roles or other groups. *real_component* is a virtual object type which is a generalization of these attributes, roles and groups. On the other hand, each of these *real_components* can be part of several groups. This many-to-many rel-type between *group* and *real_component* is materialized by the *component* technical object type, which is identified by its *gr_comp* link to *group* and its *real_comp* link to *real_component*.

Note that, in an entity type, the roles that can be part of a group are the "far" roles of the rel-types into which the entity type participates. For instance, if A and B are two entity types, R is a rel-type, A plays a role rA in R and B plays a role rB in R, then the groups of A can only include the role rB, not rA.



A group can be the support of one or several constraints. A constraint is represented by an object of the type *constraint*. The various constraints can be classified in three categories:

- A constraint on the members of a group, which should be validated on each instance of the object type owning the group. For instance, the members of the group should coexist or are mutually exclusive,... These constraints are represented by the value given to the *type* property of the *constraint*.
- A constraint on the group as a whole, which should be validated on the set of all the instances of the object type owning the group. For instance, the group is an identifier of an entity type. These constraints are represented in the same way as the previous ones.
- A constraints between several groups. For instance, a referential contraint between a reference group and an identifying group. In this case, the constraint must be linked to all the groups and the link must show the role of the group in the constraint. *member_cst* is the object type which act as the many-to-many relationship between *constraint* and *group*. It is identified by both links *const_mem* to constant and *gr_mem* to *group*. *Member_cst* is characterised by the role of the group in the constraint stored in the *mem_role* property. DB-MAIN presently supports constraints between two groups only. The group origin of the constraint should be linked to a *member_cst* object with the property *mem_role* set to the constant *OR_MEM_CST*, and the target group should be linked to a *member_cst* object with the property *mem_role* set to the constant *TAR_MEM_CST*, these two *member_cst* objects being linked to the same *constraint* object.

11.2.9 Collection

A collection can contain several data objects. In fact, in a ER schema, DB-MAIN only supports entity types. And each entity type can be part of several collections. This many-to-many rel-type between *collection* and *data_object* is marterialized by the object type *col_et*. It is identified by its link *coll_colet* to *collection* and its link *data_colet* to *data_object.*

11.3 UML class diagram

A UML class diagram is an instance of *schema* object type whose attribute *type* is set to the constant *UMLCLASS_SCHEMA*.

Basically, in the repository, A UML class diagram is the same as an ER schema. Only the graphical view of these schema is different. In particular, the roles of a rel-types are stored in the repository in the same way for UML class diagrams as for ER schemas, even if they are inverted on screen.

Note that non-binary rel-types are difficult to manage in a UML class diagram, so they should be avoided, even if they are permitted.

11.4 UML activity diagram

11.4.1 Schema

A UML activity diagram is an instance of *schema* object type whose attribute *type* is set to the constant *UMLACTIVITY_DIAGRAM*. A schema is made up of processing units, with various roles and various graphical representaions, object states and relations between all these components. An object state represents a particular state of a data object at a given time. A data object is either an *internal* attribute, or an *external* attribute, entity type, rel-type or collection. *Internal* means that the attribute is part of the schema, and *external* means that the data object is in fact a reference to a data object defined in a data schema (ER or UML class diagram). *data_object* is a virtual object type which allows us to link all processing units and internal data objects to the schema through the *sch_data* link.

11.4.2 Action state

An action state is stored in the repository as a *proc_unit*. Its *mode* property should be left to ' '. Its *type* field can be used for documentation, it can indicate if the action state should be seen as a method, a procedure, a function, a trigger,... The *name*, *short_name*, *semantics_desc*, and *technical_desc* fields are inherited from *data_object*. *proc_unit* also inherits the roles it plays in the *owner_group* and *domain* links, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash.

11.4.3 Initial state, final state, synchronisation, decision, signal sending and receipt

Initial states, final states, horizontal and vertical synchronisation bars, decision states, signal sendings and signal receipts are special kinds of action states (see 11.4.2). Their *mode* field and their *name* should be set to the following special values:

	mode	name
initial state	I	INITIAL_STATE
final state	F	FINAL_STATE
horizontal synchronisation	H	SYNCHRONISATION
vertical synchronisation	V	SYNCHRONISATION
decision	D	DECISION
signal sending	S	SIGNAL_SENDING
signal receipt	R	SIGNAL_RECEIPT

Table 11.1 - Special action states

11.4.4 Object state

An object state is an instance of the *state* type which is linked through *go_st* to a *generic_object*. It is identified by its *name*, inherited from *data_object*, among all the states of the schema it is part of. The *short_name*, *semantics_desc*, and *technical_desc* fields are inherited from *data_object* too. *proc_unit* also inherits the roles it plays in the *owner_group* and *domain* links, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash.

11.4.5 Control flow

A control flow between two action states (*proc_unit*), according to its most general meaning (including initial and final state, synchronisation,...), is represented by an instance of *rel_element*, and two instances of *element*. An *element* with its *type* field set to 'C' is linked with *pu_made_of* to the *proc_unit* from which the control flow is originating. Another *element* with its *type* set to 'A' is linked with *invokes_pu* to the *proc_unit* to which the control flow is targeted. These two *elements* are linked by a *rel_element* whose *type* and *mode* fields can be left undefined. A *rel_element* has a *name*, a *semantics_desc* and a *technical_desc*. Note that each *proc_unit* is linked to at most one *element* with the *type* set to 'C' which is shared by all the control flows originating from this *proc_unit*.

11.4.6 Object flow

An object flow is represented by an instance of *environment* which is linked to an action state (*proc_unit*) with the *p_decl* link and to an object state (*state*) with the *st_env* link. An *environment* has a *name*, a *semantics_desc* and a *technical_desc*. The orientation of the link is stored in the *mode* field: 'I' for input, 'O' for output and 'U' for update. The *type* field reminds whether the object whose, state is linked by *st_env*, is internal ('I') or external ('E').

11.5 UML use case diagram

11.5.1 Schema

A UML use case diagram is an instance of *schema* object type whose attribute *type* is set to the constant *UMLUSECASE_DIAGRAM*. A schema is made up of use cases, actors and relations between these components.

11.5.2 Use case

A use case is stored in the repository as an instance of *proc_unit*. Its *type* and *mode* fields are not used. It inherits its *name*, *short_name*, *semantics_desc*, and *technical_desc* from *data_object*. *proc_unit* also inherits the roles it plays in the *owner_group* and *domain* links, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash.

11.5.3 Actor

An actor is a special kind of resource and is stored as an instance of *resource* in the repository. Since *resource* inherits from *data_object*, an actor has got a *name*, a *short_name*, a *semantics_desc* and a *technical_desc*. It also inherits the roles played by *data_object* in the *owner_group* and *domain* links, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash.

11.5.4 Extend, Include and use case generalization relations

An extend, include or generalization relation between two use cases (*proc_unit*) is represented by an instance of *rel_element*, and two instances of *element*. An *element* is linked with *pu_made_of* to the *proc_unit* from which the relation is originating. Another *element* linked with *invokdes_pu* to the *proc_unit* to which the relation is targeted. These two *elements* are linked by a *rel_element* whose *type* and *mode* fields can be left undefined. A *rel_element* has a *name*, a *semantics_desc* and a *technical_desc*. The *type* field of the two elements should have the following values, according to the kind of relation:

	element linked to the originating use case	*element* linked to the target use case
extend	E	X
include	I	N
use case generalization	G	R

Table 11.2 - Type of elements for relations between use cases

Note that each originating *proc_unit* is linked to at most one *element* with the same *type* value which is shared by all the relations of the same kind originating from this *proc_unit*.

11.5.5 Actor generalization

An actor generalization is represented in the repository by an instance of *re_isa* linked to the most general *resource* with *re_gen*, and to the specialized *resource* with *re_spec*. A *re_isa* has a *name*, a *semantics_desc* and a *technical_desc*.

11.5.6 Association

An association between a use case (*proc_unit*) and an actor (*resource*) is stored as a *consumption*, *cons_pu* and *cons_res* in the repository. Since *consumption* inherits from *data_object*, a consumption has got a *name*, a *short_name*, a *semantics_desc* and a *technical_desc*. The *consumption* is linked to the *cons_pu* with an instance of the *cons_cpu* link and to the *cons_res* with an instance of the *cons_cres* link. The *cons_pu* is linked to the *proc_unit* with an instance of the *procunit_cpu* link. A *cons_pu* has a *name*, a *semantics_desc* and a *technical_desc*. The *min_card* and *max_card* fields show how many actors of the same type should be associated with the use case. The *cons_res* is linked to the *resource* with an instance of the *resource_cpu* link. A *cons_res* has a *name*, a *semantics_desc* and a *technical_desc*. The *min_card* and *max_card* fields of *cons_pu* show in how many instances of the use case each actor should participate. At the present time no instance of *res_role* can be linked to the *consumption*.

11.6 Textual document

A textual document is represented by an object of type *document*. Such an object mainly contains a reference to a text file on a disk. Such a document is made up of lines of text. A DB-MAIN user may need to mark or to annotate some of the lines. To do so, an object of type *text_line* can be created for each line that deserve such an attention.

11.7 Set of products and connection

A set of product can contain any number of items, technical objects of type *set_product_item* linked to the *set_of_product* by *contains*, which can be any other product, linked to *set_product_item* by *is_in*. An integrity constraint is that a set cannot contain itself. A product can be an item of several sets.

Products can also be connected with an oriented connection. An object of the *connection* type is linked to the origin product by *pfrom* and to the target product by *pto*.

11.8 Note

To each element of a schema, diagram or of the project, can be attached a note. In the repository, notes, represented by objects of type *note*, are attached to the *generic_object* object type, altough, in practice, DB-MAIN only supports notes on the following object types: *schema*, *entity_type*, *rel-type*, *si_attribute*, *co_attribute*, *role*, *group*, *collection*, *cluster*, *proc_unit*, *rel_element*, *environment*, *state*, *resource*, *consumption*, *cons_pu*, *cons_res*, *document*, *set_of_product*.

A note attached to one element. This element can appear in one or many views (like an entity type that can appear in a UML class diagram and in a UML activity diagram). Sometimes the note should be visible in all the views, sometimes not. Moreover, an object may receive several notes, possibly in a single view or in many views. This complex situation is managed by technical objects of the type *nn_note* which materializes a ternary rel-type between the *note*, the *generic_object* to which the note is attached, and the *user_viewable* generalizing the *schema* or the *system* in which view the note should appear. A *nn_note* object is identified by its three links *note_nnn* to the *note*, *go_nnn* to the *generic_object*, and *uv_nnn* to the *user_viewable*.

11.9 Meta object and meta property

In every project, DB-MAIN creates some *meta_objet* objects which are linked to the *system* object with the *sys_mo* link. A Voyager 2 program can add its own *meta_object* for its own needs, but these new *meta_objects* will not be managed by DB-MAIN.

To each *meta_object* created by DB-MAIN, some *meta_properties* are created automatically too, and linked by *mo_mp*. The user can manage all of them and create new ones, either to DB-MAIN *meta_objects* or to the user-defined *meta_objects*. The new *meta-properties* will be managed by DB-MAIN.

Chapter 12

Objects Definition

This chpater presents all the object types of the repository presented in Chapter 11.

Each object type is presented in a formal way with its generalizations, its attributes and its links to other objects. It is then described in English with some comments. The instruction to perform in order to create a new object of the type completes the description. These presentations use the following conventions.

The formal presentation of each object type is presented as a table, like in the following exemple:
:

employee		
isa *person*		
name	string[L_NAME]	1-1
own	0-∞	cars

Figure 12.1 represents graphically the employee table.

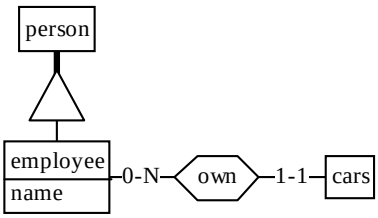


Figure 12.1 - Graphical representation of the employee table.

The first line contains the object name. The second line is the list of generalizations: *employee* can be generalized by *person*.Then come all the properties of the object: fields and roles.

Each field is described by its name, its type and its minimum-maximum cardinality. When an attribute is optional, a default value is given to it anyway. This default value is mentioned between parentheses. When the type is a string, the maximum length of the string is indicated between square brackets[1], using predefined integer constants. It is recommended to use the name of the constants rather than their value for obvious reasons of maintenance of the Voyager 2 programs with future versions of DB-MAIN.

1. Although strings size is unlimited in Voyager 2, the need to store this information in a persistent way on disk requires to limit the size of each string to a reasonable length.

Each role is described by its name[2], the minimum-maximum cardinality and finally the object type at the other end of the link.

Each time a new object is created, its environment must be defined. The environment of an object is the collection of data required to create it: field values, roles, positions,... Objects are created by calling the function *create* with the components of the environment in argument. Such a call looks like:

```
et := create(ENTITY_TYPE,
     name:  "employee",
     short_name:  "emp",
     sem:  "male/female",
     @SCH_DATA:  GetCurrentSchema());
```

where the first argument is a predefined integer constant denoting the type of the object to create. The following arguments are items composed of two expressions: a constant denoting the component of the environment being specified, and its value. All the mandatory components (minimum cardinality>0) must be specified. Their order of the components is not important. If a component is specified twice or more, *create* chooses one of them randomly. When the left expression of an item is underlined, it is required. Otherwise, it is optional but subject to the restrictions explained in the comments.

If a mandatory component (field, role, ...) is lacking, or if an integrity constraint is violated then the program is aborted, and a possible explanation is printed on the console. Note that the message can sometimes be confusing. It is the case for instance, when unicity/integrity constraints are violated[3]: the program is aborted and a message about the memory is displayed[4]. But, in any case, the compiler shows which instruction is causing the error.

Note that the binary relationships in the repository are called links, in this book, in order to make easier the distinction between relationships used by the repository (link) and relationships defined by the repository (see 12.16). The links always define a relationship between two object types: the father and the son. The father may have zero, one or more sons and sons may have zero or one father, but no more. To each link corresponds one integer constant from the table 2.5.

12.1 generic_object

generic_object		
flag	integer	0-1 (=0)
p_act_arg	0-∞	p_expression
go_st	0-∞	state
go_uo	0-∞	user_object
go_nnn	0-∞	nn_note

This object type has no other purpose than being a supertype of all the other object types. This property if often used when programmers have to deal with objects for which they do not know the exact type. For instance: "What is the current selected object in the schema window?" (GetCurrentObject()) will return one object whose type is unknown. The programmer can store this value in a variable of type *generic_object*.

The *flag* field is an array of bits that stores various information. "mark" is such an information. The programmer can use the functions GetFlag and SetFlag[5] to access this array.

Let us note that the *flag* array can store information users are not aware of. Therefore, when accessing some bits, programs should be careful to preserve all the the others. The five following indexes are defined:on a *generic_object*: MARK1, MARK2, MARK3, MARK4, MARK5. The flag of index $MARK_i$ of

2. Let us note that roles name are sometimes preceded by the symbol @ to indicate which direction of the link is used.
3. For instance: you are defining for the second time an entity-type with the same name.
4. This is due to improve the efficiency of the system.
5. cfr. page 45

an object is true if, and only if, the object is marked in the marking plan number i (see figure 12.2). The following example shows how to use this flag:

```
schema: sch;
data_object: dta;
integer: the_flag;

begin
  sch := GetCurrentSchema();
  for dta in DATA_OBJECT[dta]{@SCH_DATA:[sch] with GetType(dta)=ENTITY_TYPE} do {
    the_flag := dta.flag;
    if GetFlag(the_flag,MARK2) then {
      dta.flag := SetFlag(the_flag,MARK2,FALSE);
    } else {
      dta.flag := SetFlag(the_flag,MARK2,TRUE);
    }
  }
end
```

This program marks unmarked entity types and unmarks marked entity types (wrt. the 2th marking plan). .

Figure 12.2 - Window sample showing the "mark" interface.

The flag index SELECT is used to select/unselect an object in the current view.

12.2 user_object

user_object		
isa *generic_object*		
posx	integer	0-1 (unknown)
posy	integer	0-1 (unknown)
color	integer	0-1 (unknown)
@go_uo	1-1	generic_object (unknown)
@uv_uo	1-1	user_view

This object type stores a few information about the graphical representation of an object in a user view. The first time DB-MAIN draws a new object in a window, if this object has no associated *user_object*, DB-MAIN creates it and sets default values for the attributes. Voyager 2 programs are not allowed to create *user_objects*.

12.3 note

note		
isa *generic_object*		
content	string	1-1
note_nnn	0-∞	nn_note

This object type denotes a note (free text) attached to one or more generic_objects through a *nn_note*.

To create a new *note*:

what := create(NOTE,
content: string,
GENERIC_OBJECT: generic_object,
USER_VIEWABLE: user_viewable);

The content, the *generic_object* and the *user_viewable* must be all specified, or none of them. In the latter case, a *nn_note* must be created before the content can be updated. Indeed, the content is saved in the same file as all the descriptions of the *schema* or of a *project*, so a path between the new *note* and a *schema*, or the *system*, must exist for the content of the *note* to be saved.

Rem: the owner is either a schema, an entity type, a rel-type, an attribute, a role, a group, a cluster, a processing unit, or a collection.

12.4 nn_note

nn_note		
isa *generic_object*		
@go_nnn	1-1	generic_object
@note_nnn	1-1	note
@uv_nnn	1-1	user_viewable

A *nn-note* is a relationship entity type between *user-viewables*, *generic_objects* and *notes*. It is automatically deleted when either the *note* or the *generic_object* or the *user_viewable* is deleted.

To create a new *nn_note*:

what := create(NN_NOTE,
@GO_NNN: generic_object,
@NOTE_NNN: note,
@UV_NNN: user_viewable);

12.5 system

system		
isa *user_viewable*		
name	string[L_NAME]	1-1
short_name	string[L_SNAME]	0-1 (="")
creation_date	string[L_DATE]	0-1 (see ♦)
sem	string	0-1 (="")
tech	string	0-1 (="")
sys_sch	0-∞	product

An object of type *system* represents the project under development, made up of schemas, processes, files (document),... The repository can contain only one object of type *system* since DB-MAIN cannot manage several projects at the same time.

To create a new system:

what := create(SYSTEM,
name: string,
short_name: string,
creation_date: string,
flag: integer);

♦ When the *creation_date* field is not mentioned, this field if automatically updated with the date of the computer.

12.6 product

product		
name	string[L_NAME]	1-1
version	string[L_VERSION]	1-1
creation_date	string[L_DATE]	0-1 (see ♦)
last_update	string[L_DATE]	0-1 (see ♦)
sem	string	0-1 ("")
tech	string	0-1 ("")
posx_project	integer	0-1 (=0)
posy_project	integer	0-1 (=0)
@system_sch	1-1	system
pfrom	0-∞	connection
pto	0-∞	connection

Objects of type *product* do not exist per se. They only have a sense as a generalization of *schemas* (see-section 12.7), *documents* (see section 12.10) and set of *products* (see section 12.8). Therefore it is impossible to create directly such an object. Each product is identified by its mandatory components.

♦ It is recommended not to use the *creation_date* and *last_update* fields since they are updated automatically. In case these fields are used, they must follow the following format : "YYYYMMDD", exactly four digits to represent the year, exactly two digits for the month and exactly two digits for the day.

12.7 schema

schema		
isa *product*		
isa *user_viewable*		
isa *owner_of_proc_unit*		
short_name	string[L_SNAME]	0-1 ("")
type	char	0-1 (=ERA_SCHEMA)
sch_coll	0-∞	collection
sch_data	0-∞	data_object

Objects of type *schema* are generalized entity-relationship schemas. Each schema belongs to only one system[6] and is identified by its *name*, its *version* and the *system*.

The *type* field denotes the kind of schema. The possible values of this field are:

- *ERA_SCHEMA*: It denotes an Entity-Relationship-Attribute schema.
- *UMLCLASS_DIAGRAM*: It denotes an UML class diagram schema.
- *UMLACTIVITY_DIAGRAM*: It denotes a UML activity diagram schema.
- *UMLUSECASE_DIAGRAM*: It denotes a UML use case diagram schema.

The following indexes in the *flag* field denote the following properties:

- *RTSQUARE*: the shape of the rel-types has square corners.
- *RTROUND*: the shape of the rel-types has round corners.
- *RTSHADOW*: the rel-types are displayed with a shadow.
- *ETSQUARE*: the shape of the entity types has square corners.

6. There is only one system instance in the repository of DB-MAIN.

- *ETROUND*: the shape of the entity types has round corners.
- *ETSHADOW*: the entity types are displayed with a shadow.

To create a new *schema*:

```
what := create(SCHEMA,
    name: string,
    short_name: string,
    version: string,
    creation_date: string,
    last_update: string,
    type: char
    flag: integer,
    sem: string,
    tech: string,
    SCHEMA_TYPE: schema_type,
    @SYSTEM_SCH: system);
```

Note: the SCHEMA_TYPE field is to be used only in conjunction with the MDL language for defining methods [7].

12.8 set_of_product

set_of_product		
isa *product*		
isa *owner_of_proc_unit*		
contains	0-∞	set_product_item

Objects of type *set_of_product* denote sets of products. Every object belongs to only one system[7] and is identified by its *name*, its *version* and the *system*.

The following indexes in the *flag* field denote the following properties:

- *HIDEPROD*: indicates whether DB-MAIN has to display the elements of the set (false) or to hide them (true).

To create a new *set_of_product*:

```
what := create(SET_OF_PRODUCT,
    name: string,
    version: string,
    creation_date: string,
    last_update: string,
    flag: integer,
    sem: string,
    tech: string,
    @SYSTEM_SCH: system);
```

12.9 set_product_item

set_product_item		
isa *generic_object*		
@contains	1-1	set_of_product
@is_in	1-1	product

7. There is only one system instance in the repository of DB-MAIN.

Each *set_product_item* object denotes a tuple (*p*, *s*) where *p* is a product and *s* is a set of products meaning that the product *p* belongs to the set of products *s*.

To create a new *set_product_item*:

what := create(SET_PRODUCT_ITEM,
flag: integer,
@CONTAINS: set_of_product,
@IS_IN: product);

12.10 document

document		
isa *product*		
isa *user_viewable*		
path	string	1-1
type_of_file	string	0-1 (="")
lineto		

Objects of type *document* denote any textual document like files, documentation,… The *type_of_file* field is a string describing the type of the file (eg. "text", "COBOL", "annual report 95",...). The *creation_date* and *last_update* fields are optional fields (see section 12.6 for more details about the default values). *Documents* are attached to one and only one *system*.

To create a new *document*:

what := create(DOCUMENT,
name: string,
version: string,
path: string,
creation_date: string,
last_update: string,
flag: integer,
type_of_file: string,
DOCUMENT_TYPE: document_type,
@SYSTEM_SCH: system);

Note: *DOCUMENT_TYPE* is only used in conjunction with the MDL language [7].

12.11 text_line

text_line		
isa *generic_object*		
number	integer	1-1
description	string	0-1 (="")
@lineto	1-1	document

A *text_line* stores some information concerning a particular line of a document. The line is identified by its *number* in the document. The information can be stored either in the *description*, in the *flag* inherited from the generic object, or in the associated *user_object* (special colour for example) created automatically the first time that DB-MAIN shows the line on screen.

To create a new line:

what := create(TEXT_LINE,
number: integer,
description: string,
@LINETO: document);

12.12 connection

connection		
isa *generic_object*		
type	string[L_ROLE]	0-1 ("")
@pfrom	1-1	product
@pto	1-1	product

Objects of type *connection* represent oriented links between objects of the project like schemas and documents (file, ...). If the field *type* is not mentioned, its default value is "". This field can have any value but a few constants with precise semantics are defined:

- *CON_COPY*: The target product is a copy of the origin product.
- *CON_DIC*: The target product is a "printed" copy (File/Print dictionary).
- *CON_GEN*: The target product is some code generated on the basis of the origin product.
- *CON_INTEG*: Undocumented feature (so far).
- *CON_XTR*: The target product is the result of the extraction process.

To create a new object of type *connection*:

```
what := create(CONNECTION,
    type: string,
    flag: integer,
    @PFROM: product,
    @PTO: product);
```

12.13 data_object

data_object		
isa *generic_object*		
isa *owner_of_proc_unit*		
name	string[L_NAME]	1-1
short_name	string[L_SNAME]	0-1 ("")
sem	string	0-1 ("")
tech	string	0-1 ("")
@sch_data	1-1	schema
data_gr	0-∞	group
domain	0-∞	si_attribute

Objects of type *data_object* are generalization of objects representing entity types, relationship types, attributes, processing uniots or resources. Objects of this type cannot be created directly.

12.14 ent_rel_type

ent_rel_type	
isa	*data_object*
isa	*owner_of_att*

Objects of type *ent_rel_type* are generalizations of entity types (section 12.15) and relationship types (section 12.16). Such objects cannot be created directly.

12.15 entity_type

entity_type		
isa *ent_rel_type*		
entity_etr	0-∞	et_role
entity_sub	0-∞	sub_type
entity_clu	0-∞	cluster

Each object of type *entity_type* denotes an entity type. It can belong to collections, have attributes, play roles in relationship types, and be generalization/specialization of other entity types.

To create a new object of type *entity_type*:

what := create(ENTITY_TYPE,
name: string,
short_name: string,
sem: string,
tech: string,
flag: integer,
@SCH_DATA: schema);

12.16 rel_type

rel_type		
isa *ent_rel_type*		
rt_ro	0-∞	role

Objects of type *rel_type* denote relationship types. Each relation is attached to exactly one *schema* and its *name* identifies it among all other relationship types of the same schema. Rel-types can own attributes, groups and roles.

To create a new relation:

what := create(REL_TYPE,
name: string,
short_name: string,
sem: string,
tech: string,
flag: integer,
@SCH_DATA: schema);

12.17 attribute

attribute		
isa *data_object*		
isa *real_component*		
min_rep	integer	1-1
max_rep	integer	1-1
container	char	0-1 (=SET_CONTAINER)
@owner_att	1-1	owner_of_att

The *attribute* object type is a super-type of types *si_attribute* and *co_attribute*. Therefore, such objects cannot be created directly. See sections 12.18 and 12.19 for more details.

All the fields of an attribute are shared with sub-types by the inheritance principle. The *min_rep* (resp. *max_rep*) is the minimum (resp. maximum) cardinality of the attribute. This value must be comprised

between 0 and N_CARD which is equivalent to the infinite value[8]. The minimum cardinality must be smaller than the maximum cardinality. If one of the rules cited above is transgressed, then the program is aborted.

The *container* field shows how multivalued attributed must be interpreted. The possible values for this field are:

- *SET_CONTAINER*: It is the default value. It denotes a set, in the mathematical sense.
- *BAG_CONTAINER*: It is a set with possible duplicates. There is no order inside a set/bag.
- *ARRAY_CONTAINER*: It is an array, each item can be referenced with an index. Arrays can contain several identical items.
- *UNIQUE_ARRAY_CONTAINER*: It is an array into which each element appears only once.
- *LIST_CONTAINER*: A list is an ordered collection of items (duplicates are allowed).
- *UNIQUE_LIST_CONTAINER*: A list with no duplicates.

Note: the role *@sch_data* is redundant with the mandatory role *@owner_att*, since attributes and their owners must belong to identical schemas, but it is necessary for DB-MAIN, for optimization reasons. Voyager programs should manage this redundancy correctly.

12.18 si_attribute

si_attribute		
isa *attribute*		
type	char	1-1
length	integer	1-1
decim	integer	0-1
stable	integer	0-1 (=FALSE)
recyclable	integer	0-1 (=TRUE)
@domain	0-1	data_object

Objects of type *si_attribute* represent simple, atomic, attributes. The *type* field is the type of the attribute. Special constants are available in Voyager 2 to denote the different possible types:

- *CHAR_ATT*: string with a constant length.
- *VARCHAR_ATT*: string with a variable length, possibly with a maximum length.
- *NUM_ATT*: numerical value with fixed integer and decimal part lengths.
- *DATE_ATT*: date.
- *BOOL_ATT*: boolean value.
- *FLOAT_ATT*: real value with specified precision.
- *USER_ATT*: user-defined domain whose definition is available with the *domain* relationship.
- *OBJECT_ATT*: the entity-type[9] linked with the *domain* relationship.
- *INDEX_ATT*: The value of the corresponding attribute, unique, is automatically computed by the DBMS.
- *SEQ_ATT*: The value of the corresponding attribute is automatically computed, in sequence, by the DBMS.

A user-defined domain *attribute* must be attached to a *data_object* stored in a special *schema* that can ba accessed with the following query: GetFirst(SCHEMA[sch]{sch.name=SCHEMA_DOMAINS}).

8. Only the maximum cardinality may take the infinite value.
9. Although the data_object is the object type found on the other side of the domain relationship, only the entity-type instances are pertinent

New user-defined domains should be added to this *schema* too. They will automatically appear in the suitable dialog boxes of DB-MAIN.

The *length* field denotes the size of the value and the *decimal* field denotes the precision of the decimal type. The following table indicates when the length and decimal values are required. Cells marked with a star show that a value is expected if the corresponding type is mentioned. If a value is shown in a cell, that value is forced for the corresponding field.

	length	**decimal**
CHAR_ATT	*	
VARCHAR_ATT	*	
NUM_ATT	*	*
DATE_ATT	10	
BOOL_ATT	1	
FLOAT_ATT	*	
USER_ATT		
OBJECT_ATT		

The *stable* field indicates that the value of the attribute cannot be changed. The *recyclable* field shows that each value of the attribute can be reused by the same attribute of other owners at any time; the value of a non-recyclable attribute can never be reused. These fields are mainly useful with identifiers.

To create a new si_attribute:

what := create(SI_ATTRIBUTE,
name: string,
min_rep: integer,
max_rep: integer,
type: char,
length: integer,
short_name: string,
stable: integer,
recyclable: integer,
container: char,
decim: integer,
sem: string,
tech: string,
where: attribute,
flag: integer,
@OWNER_ATT: owner_of_att,
@DOMAIN: data_object);

where shows where the new attribute must be placed among its siblings. If this information is not supplied, the new attribute is put in first position in the list of attributes of the father, specified in *@OWNER_ATT*. Otherwise, *where* must be another attribute of the same father, and the new attribute will be created just after that specified one in the list of attributes of the father. For example, to create an object attribute of type *ent*:

```
si_attr:=create(SI_ATTRIBUTE,type: OBJECT_ATT,min_rep: 1,max_rep: MAX_CON,
                container: BAG_ATT,@OWNER_ATT: the_owner,@DOMAIN: ent);
```

12.19 co_attribute

co_attribute	
isa	*attribute*
isa	*owner_of_att*

Objects of type *co_attribute* represent compound attributes. Only the *min_rep* and *max_rep* fields must be mentioned in the environment as well as the *name* and the owner (*owner_of_att*).

To create new composed attributes:

what := create(CO_ATTRIBUTE,
name: string,
short_name: string,
min_rep: integer,
max_rep: integer,
container: char,
sem: string,
tech: string,
where: attribute,
flag: integer,
@OWNER_ATT: owner_of_att);

See 12.18 for more details about the where field.

12.20 owner_of_att

owner_of_att		
isa *generic_object*		
owner_att	0-∞	attribute

The type *owner_of_att* has no creator. Objects of this type are just a generalization of objects of *ent_rel_type* and *co_attribute* types to allow them to own attributes.

12.21 component

component		
isa *generic_object*		
@real_comp	1-1	real_component
@gr_comp	1-1	group

Technical objects of *component* type are artifacts to represent many-to-many relationships. Since the order of the components of a group matters, the order of the objects inside that many-to-many relationship is important. In the creator, a special field, *where*, is used to denote the component preceding the new one in the set of components owned by a group. If this information is not provided in the environment, then the new component becomes the first one of the group.

To create a new object of type component:

what := create(COMPONENT,
flag: integer,
where: component,
@REAL_COMP: real_component,
@GR_COMP: group);

For instance, let us suppose that the attribute *at* must be inserted at the second place in the group *gr*, the first item of the group being the component *co*. The instruction will be:

```
new_one:=create(COMPONENT,@REAL_COMP:at,@GR_COMP:gr,where:co);
```

12.22 group

group		
isa *generic_object*		
isa *owner_of_proc_unit*		
name	string[L_NAME]	1-1
type	char	0-1 (=ASS_GROUP)
primary	integer	0-1 (=0)
secondary	integer	0-1 (=0)
coexistence	integer	0-1 (=0)
exclusive	integer	0-1 (=0)
atleastone	integer	0-1 (=0)
key	integer	0-1 (=0)
user_const	integer	0-1 (=0)
funct	integer	0-1 (=0)
min_rep	integer	0-1 (=0)
max_rep	integer	0-1 (=0)
sem	string	0-1 ("")
tech	string	0-1 ("")
@data_gr	1-1	data_object
gr_mem	0-∞	member_cst

A *group* in the repository of DB-MAIN is a list of properties like attributes, roles and other groups. Elements of groups are *real_component* that is generalization of *attribute*, *role* and *group*. A group can belong to an entity type, a rel-type or an attribute, therefore there is a link between data_object and group.

The fields of a group are:

- *name*: The group name is mandatory. It is usually a technical name. All the groups attached to the same data_object must have distinct names.
- *type*: The type is either ASS_GROUP (normally) or COMP_GROUP (shouldn't be used anymore).
- *primary/secondary*: The primary and secondary fields denote respectively primary and secondary identifiers and must be used as boolean values.
- *coexistence*: This field is used as a boolean value to indicate if all the items of the group, optional, must be instantiated together or not.
- *exclusive*: This field is used as a boolean value to indicate whether at most one item of the group can be instantiated, or not.
- *atleastone*: This field is used as a boolean value to indicate whether at least one item in the group must be instantiated, or not.
- *key*: The group is an access key for its owner.
- *user_const*: This field is used as a boolean value to indicate that the *meta_property* "User-constraint" contains the user contraint type.
- *funct*: This field is an array of bits that stores the different functions of the group (primary, secondary, coexistence, exclusive, atleastone, key, constraint). The corresponding constants are ID_GR, SEC_GR, COEX_GR, EXCL_GR, AL1_GR, KEY_GR and CST_GR.
- *min_rep* and *max_rep*: cardinalities of the group. The values of these fields must be comprised between 0 and N_CARD which is equivalent to the infinite value[10].

The following instruction creates a new group:

10. Only the maximum cardinality may take the infinite value and min_rep <= max_rep.

what := create(GROUP,
name: string,
type: char,
primary: integer,
secondary: integer,
coexistence: integer,
exclusive: integer,
atleastone: integer,
key: integer,
constraint: integer,
funct: integer,
sem: string,
tech: string,
flag: integer,
@DATA_GR: data_object);

12.23 constraint

constraint		
isa *generic_object*		
type	char	1-1
const_mem	0-∞	member_cst

A *constraint* object defines a constraint between two groups. Only the *type* field is required in the environment of this object. The possible values of *type* are the five constants:

- *INC_CONSTRAINT*: Let g_1 and g_2 be two groups playing respectively the origin and target roles for the **referential** constraint, g_2 being a primary/secondary identifier group. Then all the components of g_1 (attributes and/or roles) must take their values in the domain built on the values of all the components of g_2.
- *EQ_CONSTRAINT*: Let g_1 and g_2 be two groups participating in an **equality** constraint, then the inclusion constraint holds for both (g_1,g_2) and (g_2,g_1). Therefore, this constraint should not be oriented: g_1 and g_2 play the same role, but, some physical models require that a group must be the origin of the other one[11].
- *INCLUSION_CONSTRAINT*: the constraint is an **inclusion** if each instance of the first group must be an instance of the second group (the second group must not be an identifier, the inclusion constraint is the generalization of the referential constraint).
- *INV_CONSTRAINT*: An **inverse** constraint must be declared between two identifier (primary or secondary) groups made up of one object attribute each. It shows that the value the each object attribute of each group is the owner of the other group.
- *GEN_CONSTRAINT*: the constraint is a **generic** constraint between any groups (without preconditions).

More generally, a constraint is a property attached to a tuple of groups (g_1,g_2, ..., g_n). Each component of the tuple plays a special role in the constraint. The name of the role is specified in objects of type *member_cst*. The constraints defined in DB-MAIN are binary and their roles have the same names: OR_MEM_CST (for *origin*) and TAR_MEM_CST (for *target*).

For instance, if the group g_1 (composed of the *name* and *first_name* attributes of an entity e_1) is a foreign key to a group g_2 (composed of the *N_F_names* attribute), then g_1 is the origin of an inclusion constraint and the target is the group g_2.

11. This constraint is usually implemented in SQL by a foreign key followed by a check. The orientation takes a sense here.

To create a new constraint:

what := create(CONSTRAINT,
type: char,
flag: integer);0

12.24 member_cst

member_cst		
isa *generic_object*		
mem_role	char	0-1 ("")
@const_mem	1-1	constraint
@gr_mem	1-1	group

The domain of the *mem_role* field is a character, its possible values are restricted to two constants: OR_MEM_CST and TAR_MEM_CST. See section 12.23 for more details.

To create a new *member_cst* object:

what := create(MEMBER_CST,
mem_role: char,
flag: integer,
@CONST_MEM: constraint,
@GR_MEM: group);

12.25 collection

collection		
isa *generic_object*		
name	string[L_NAME]	1-1
short_name	string[L_SNAME]	0-1 ("")
sem	string	0-1 ("")
tech	string	0-1 ("")
@sch_coll	1-1	schema
coll_colet	0-∞	coll_et

Objects of type *collection* denote files, clusters, areas, ... Each collection is identified by its name and a schema (*sch_coll* link).

To create a new object of type collection:

what := create(COLLECTION,
name: string,
short_name: string,
sem: string,
tech: string,
flag: integer,
@SCH_COLL: schema);

12.26 coll_et

coll_et		
isa *generic_object*		
@coll_colet	1-1	collection
@data_colet	1-1	data_object

Objects of this technical type represent many-to-many relationships between collections and data_object instances, in particular between collections and entity types in an ERA schema. Therefore the creation of these objects is derived from the attachment of an entity-type to a collection.

To create a new object of type *coll_et*:

what := create(COLL_ET,
flag: integer,
@COLL_COLET: collection,
@DATA_COLET: data_object);

12.27 cluster

cluster		
isa *generic_object*		
name	string[L_NAME]	0-1 ("")
total	integer	1-1
distinct	integer	1-1
criterion	string[L_CRITERION]	0-1 ("")
@entity_clu	1-1	entity_type
clu_sub	0-∞	sub_type

Objects of type *cluster* denote groups of distinct entity types being a specialization of a super-type entity-type. The *total* and *disjoint* fields are boolean values characterizing the semantics of the group:

- *total*: is TRUE if all the instances of the super-type must be specialized by an instance of at least one sub-type and FALSE otherwise.
- *disjoint*: is TRUE if each instance of the super-type can be specialized by an instance of at most one sub-type and FALSE otherwise.

The *criterion* field allows users to specify a criterion in order to know by which sub-type the super-type is specialized.

To create a new object of type *cluster*:

what := create(CLUSTER,
name: string,
total: integer,
distinct: integer,
criterion: string,
flag: integer,
@ENTITY_CLU: entity_type,
@CLU_SUB: sub_type);

12.28 sub_type

sub_type		
isa *generic_object*		
value	string[L_VALUE]	0-1 ("")
@clu_sub	1-1	cluster
@entity_sub	1-1	entity_type

The technical object-type *sub_type* is the materialization of a many-to-many relationship-type between a *cluster* and its sub-*entity_types*. The *value* field is the criterion (field of *cluster*) value by which sub-types may be distinguished.

To create a new object of type *sub_type*:

what := create(SUB_TYPE,
value: string,
flag: integer,
@CLU_SUB: cluster,
@ENTITY_SUB: entity_type);

12.29 role

role		
isa *generic_object*		
name	string[L_NAME]	1-1 (cfr ♦)
min_con	integer	1-1
max_con	integer	1-1
aggregation	char	0-1 (=" ")
sem	string	0-1 (="")
tech	string	0-1 (="")
@rt_ro	1-1	rel_type
ro_etr	0-∞	et_role

♦ Objects of type *role* denote the roles played by the entity types in the relationship types. Each role is identified by its *name*, its cardinalities (*min_con* and *max_con*) and the *rel_type* it depends on. The values of the cardinality must be comprised between 0 and N_CARD which is equivalent to the infinite value[12]. Although the name is mandatory, this field can be omitted if the following constraints are satisfied:

- the role is not a multi-entity role
  ∀r∈ROLE[...]{TRUE}: Length(ET_ROLE[...]{@RO_ETR:[r]})=1
- the name of the entity type that should play the new role, must not be a name of another role in the relationship type.

If these constraints are satisfied, the name of the role is the name of the entity-type playing the role.

The field *aggregation* gives a particular meaning to the relationship type and to the role itself. The possible values of this field are:

– *AGGREGATION_ROLE*: It denotes an aggregation relationship type (should be binary). The role having this *aggregation* value is played by the entity type is a part of another.

– *COMPOSITION_ROLE*: It denotes a composition relationship type (should be binary). The role having this *aggregation* value is played by the entity type which is a component of another.

To create a new role:

what := create(ROLE,
name: string,
min_con: integer,
max_con: integer,
sem: string,
tech: string,
flag: integer,
@RT_RO: rel_type);

12. Only the maximum cardinality may take the infinite value and min_con <= max_con.

12.30 et_role

et_role		
isa *generic_object*		
@entity_etr	1-1	entity_type
@ro_etr	1-1	role

Each *et_role* object denotes a tuple (*e*, *r*) where *e* is an entity type and *r* is a role. Each tuple (*e*, *r*) means that the entity type *e* participates in the role *r*. If several entity types participate in a role *r*, *r* is said to be a *multi-ET* role.

To create a new et_role:

what := create(ET_ROLE,
flag: integer,
@ENTITY_ETR: entity_type,
@RO_ETR: role);

12.31 real_component

real_component		
isa *generic_object*		
real_comp	0-∞	component

The *real_component* type is a supertype of the *attribute*, *role* and *group* types. Its only function in the repository is to be a component of a group.

12.32 proc_unit

proc_unit		
isa *data_object*		
mode	char	0-1 (unknown)
type	char	0-1 (unknown)
@owner_pu	1-1	owner_of_proc_unit
procunit_cpu	0-∞	cons_pu
pu_made_of	0-∞	elemnt
invoke_pu	0-∞	element
p_fct_call	0-∞	p_expression
p_decl	0-∞	environment

This object represent a processing unit, in the large. Valid values for the fields *mode* and *type* depend on the context into which processing is used. More information about this can be found in Chapter 11.

what := create(PROC_UNIT,
name: string,
short_name: string,
mode: char,
type: char,
sem: string,
tech: string,
flag: integer,
where: proc_unit,
@SCH_DATA: schema,
@OWNER_PU: owner_of_proc_unit);

where shows where the new processing unit must be placed among its siblings. If this information is not supplied, the new processing unit is put in first position in the list of processing units of the father, specified in *@OWNER_PU*. Otherwise, *where* must be another processing unit of the same father, and the new processing unit will be created just after that specified one in the list of processing units of the father.

12.33 element

element		
isa *generic_object*		
type	char	1-1
description	string	0-1 (="")
@pu_made_of	0-1	proc_unit
@invokes_pu	0-1	proc_unit
p_made_of	0-∞	rel_element
p_part_of	0-∞	rel_element

This object represents an element of a processing unit. Its genuine semantics depends on the context, as described in Chapter 11. Exactly one of the two roles *@pu_made_of* and *@p_made_of* is mandatory. But the *create* command below can only specify the *@pu_made_of* link, so, if a *@p_made_of* has to be created, the user should create a *rel_element* object just after creating the *element* object.

what := create(ELEMENT,
type: char,
description: string,
flag: integer,
where: element,
@PU_BODY: proc_unit,
@INVOKES_PU: proc_unit);

where shows where the new *element* must be placed among its siblings. If this information is not supplied, the new element is put in first position in the list of elements of the parent *proc_unit*, specified in *@PU_MADE_OF*. Otherwise, *where* must be another element of the same processing unit, and the new element will be created just after that specified one in the list of elements of the father.

12.34 rel_element

rel_element		
isa *generic_object*		
name	string[L_NAME]	0-1 (="")
mode	char	0-1 (=0)
type	char	0-1 (=0)
sem	string	0-1 (="")
tech	string	0-1 (="")
@p_made_of	1-1	element
@p_part_of	1-1	element

This object denotes the decomposition of a processing unit element into its sub-elements. The *type* and *mode* fields denote the kind of decomposition.

what := create(REL_ELEMENT,
name: string,
type: char,
mode: char,

sem: string,
tech: string,
flag: integer,
where: rel_element,
@P_MADE_OF: element,
@P_PART_OF: element);

where shows where the new *rel_element* must be placed among its siblings. If this information is not supplied, the new *rel_element* is put in first position in the list of *rel_elements* of the father, specified in @P_MADE_OF. Otherwise, *where* must be another *rel_element* of the same *element*, and the new sub-element will be created just after that specified one in the list of sub-elements of the father.

12.35 p_expression

p_expression		
isa *generic_object*		
operator	char	1-1
constant	string	0-1 (="")
description	string	0-1 (="")
@p_sub_expression_of	0-1	p_expression
p_sub_expression_of	0-∞	p_expression
@p_parameter	0-1	element
@p_fct_call	0-1	proc_unit
@p_act_arg	0-1	generic_object

This object denotes an expression. At least one of the two roles *@p_parameter* (the expression is part of a processing unit element) or *@p_sub_expression_of* (a sub-expression part of a more complex expression) must be defined in the creator. The expression can be either a *constant* (*operator* = 'C'), an object of the repository (using *@p_act_arg*), an operation (operator = '+', '-',...) on sub-expression(s) (using *p_sub_expression_of*), or a call to a function (using the link *@p_fct_call*).

what := create(P_EXPRESSION,
operator: char,
constant: string,
description: string,
flag: integer,
where: p_expression,
@P_SUB_EXPRESSION: p_expression,
@P_ACT_ARG: generic_object,
@P_PARAMETER: p_statement,
P_FCT_CALL: proc_unit);

where shows where the new *p_expression* must be placed among its siblings. If this information is not supplied, the new *p_expression* is put in first position in the list of *p_expressions* of the father, specified in @P_PARAMETER or in @P_SUB_EXPRESSION. Otherwise, *where* must be another *p_expression* of the same *element* or *p_expression*, and the new *p_expression* will be created just after that specified one in the list of *p_expressions* of the father.

12.36 environment

environment		
isa *generic_object*		
name	string[L_NAME]	0-1 ("")
type	char	1-1
mode	char	0-1 (=0)
sem	string	0-1 ("")
tech	string	0-1 ("")
@p_decl	1-1	proc_unit
@st_env	1-1	state

An *environment* object shows which states of any other object (generally *data_object*) is part of the environment (local variables, parameters,...) of a processing unit.

what := create(ENVIRONMENT,
name: string,
type: char,
mode: char,
sem: string,
tech: string,
flag: integer,
where: p_environment,
@P_DECL: proc_unit,
@P_GO_ENV: generic_object);

where shows where the new *environment* must be placed among its siblings. If this information is not supplied, the new *environment* is put in first position in the list of *environments* of the *proc_unit*. Otherwise, *where* must be another *environment* of the same *proc_unit*, and the new *environment* will be created just after that specified one in the list of *environments* of the parent proc_unit.

12.37 state

state		
isa *data_object*		
@go_st	1-1	proc_unit
st_env	1-1	environment

A *state* object denotes a particular state of any *generic_object*. For instance, an *entity_type* may represent a cloth, and two states of the cloth can be "new" when it has just been made, and "delivered" when it has been delivered to a shop.

The following commande should be used to create a new state:

what := create(STATE
name: string,
short_name: string,
sem: string,
tech: string,
flag: integer,
@GO_ST:generic_object,
@SCH_DATA: schema);

12.38 consumption

consumption		
isa *data_object*		
cons_cpu	1-1	cons_pu
cons_cres	0-1	cons_res
cons_crole	0-1	cons_role

A *consumption* object shows which *resource* plays which role (*res_role*) in a processing unit (*proc_unit*). Three technical objects have been implemented to show respectively how many resources can be consumed (*cons_res*), how many instances of the role of the resource can be consumed (*cons_role*), and how many processing units can consume these resources playing this role (*cons_pu*). Note that both the *cons_res* and the *cons_role* are optional.

To create a *consumption*:

what := create(CONSUMPTION
name: string,
short_name: string,
sem: string,
tech: string,
flag: integer,
CONS_CPU: cons_pu,
CONS_CRES: cons_res,
CONS_CROLE: cons_role);

12.39 cons_pu

cons_pu		
isa *generic_object*		
name	string[L_NAME]	0-1 ("")
min_card	integer	0-1 (=1)
max_card	integer	0-1 (=1)
sem	string	0-1 ("")
tech	string	0-1 ("")
@procunit_cpu	1-1	proc_unit
@cons_cpu	1-1	consumption

A *cons_pu* object shows how many processing units can play a role with the consumption.

To create a *cons_pu*:

what := create(CONS_PU
name: string,
min_card: integer,
max_card: integer,
sem: string,
tech: string,
flag: integer,
@PROCUNIT_CPU: proc_unit,
@CONS_CPU: consumption);

12.40 cons_res

cons_res		
isa *generic_object*		
name	string[L_NAME]	0-1 ("")
min_card	integer	0-1 (=1)
max_card	integer	0-1 (=1)
sem	string	0-1 ("")
tech	string	0-1 ("")
@resource_cres	1-1	resource
@cons_cres	1-1	consumption

A *cons_res* object shows how many resources can play a role with the consumption.

To create a *cons_res*:

what := create(CONS_RES
name: string,
min_card: integer,
max_card: integer,
sem: string,
tech: string,
flag: integer,
@RESOURCE_CRES: resource,
@CONS_CRES: consumption);

12.41 cons_role

cons_role		
isa *generic_object*		
name	string[L_NAME]	0-1 ("")
min_card	integer	0-1 (=1)
max_card	integer	0-1 (=1)
sem	string	0-1 ("")
tech	string	0-1 ("")
@resrole_crole	1-1	res_role
@cons_crole	1-1	consumption

A *cons_role* object shows how many resource roles can play a role with the consumption.

To create a *cons_role*:

what := create(CONS_ROLE
name: string,
min_card: integer,
max_card: integer,
sem: string,
tech: string,
flag: integer,
@RESROLE_CROLE: res_role,
@CONS_CROLE: consumption);

12.42 resource

resource		
isa *data_object*		
type	char	1-1
resource_cres	0-N	cons_res
re_gen	0-N	re_isa
re_spec	0-N	re_isa
cp_res	0-N	can_play

A *resource* object can represent any human or material resource need by a processing unit, like programmers, consultants, memory, printers, coffee pot,...

To create a *resource*:

what := create(RESOURCE
name: string,
short_name: string,
sem: string,
tech: string,
type: char,
flag: integer,);

12.43 re_isa

re_isa		
isa *generic_object*		
name	string[L_NAME]	0-1 ("")
sem	string	0-1 ("")
tech	string	0-1 ("")
@re_gen	1-1	resource
@re_spec	1-1	resource

The object type *re_isa* is used to define resources in a hierarchical way. For instance, a computer technician is an employee.

To create a *re_isa* object:

what := create(RE_ISA
name: string,
sem: string,
tech: string,
flag: integer,
@RE_GEN: resource,
@RE_SPEC: resource);

12.44 res_role

res_role		
isa *data_object*		
resrole_crole	0-N	cons_role
ro_gen	0-N	ro_isa
ro_spec	0-N	ro_isa
cp_rol	0-N	can_play

A *res_role* object represents a role that a resource can play. For instance, An employee can either play the role of analyst or programmer.

what := create(RES_ROLE
name: string,
short_name: string,
sem: string,
tech: string,
flag: integer,);

12.45 ro_isa

ro_isa		
isa *data_object*		
@ro_gen	1-1	res_role
@ro_spec	1-1	res_role

The object type *ro_isa* is used to define roles of resources (*res_role*) in a hierarchical way. For instance, a project leader is an analyst.

To create a *re_isa* object:

what := create(RO_ISA
flag: integer,
@RO_GEN: res_role,
@RO_SPEC: res_role);

12.46 can_play

can_play		
isa *data_object*		
@cp_res	1-1	resource
@cp_rol	1-1	res_role

The object type *can_play* is aimed at storing a list of all the roles (*res_role*) that a resource (*resource*) can play. When a schema uses objects of both types *resource* and *res_role*, each *consumption* of a *resource*, linked by *cons_res*, playing a *res_role*, linked by *cons_role*, should correspond to a *can_play* linking the same *resource*, by *cp_res*, to the same *res_role*, by *cp_rol*.

To create a *can_play* object:

what := create(CAN_PLAY,
flag: integer,
@CP_RES: resource,
@CP_ROL: res_role);

12.47 owner_of_proc_unit

owner_of_proc_unit		
owner_pu	0-∞	proc_unit

owner_of_proc_unit is a virtual object type which is inherited by all the object types that can own processing units (*proc_unit*). Objects of that type only cannot exist, and so cannot be created.

12.48 meta_object

meta_object		
isa *generic_object*		
name	string[L_NAME]	1-1
type	integer	1-1
sem	string	0-1 (="")
@sys_mo	1-1	system
mo_mp	0-∞	meta_property

A *meta_object* object can be created for two purposes:

- To mirror an object type of the repository, in order to extend this object type by adding meta-properties to it (see 12.49). When a project is created in DB-MAIN, a series of *meta_object* objects are automatically created. Voyager 2 programs can access them, but should not modify them (which can result in crashing DB-MAIN).
- To extend the repository of DB-MAIN. A Voyager 2 program can add as many objects types as needed to the repository, for its own needs, but objects of these types will not be taken in charge by DB-MAIN and will not be shown on screen. For instance, a program could add new object types like *agent*, *module*,…

A description of the fields follows:

- *name*: the name of the object type ("entity_type", "system", "si_attribute", …).
- *sem*: a semantics description of the meta-object, an informal text.
- *type*: an integer constant identifying the type of the object type. Predefined constants are SCHEMA, ENTITY_TYPE, GROUP,…(cfr. 2.4).

In order to create a new *meta_object*, the following command should be used:

```
what := create(META_OBJECT,
    name: string,
    type: integer,
    sem: string,
    flag: integer,
    @SYS_MO: system);
```

12.49 meta_property

meta_property		
isa *generic_object*		
name	string[L_NAME]	1-1
type	integer	1-1
updatable	integer	0-1 (=TRUE)
predefined	integer	0-1 (=FALSE)
multi	integer	0-1 (=FALSE)
hidden	integer	0-1 (=FALSE)
sem	string	0-1 ("")
@mo_mp	1-1	meta_object

If an object type is described by an instance of *meta_object*, it is possible to dynamically add new fields to it (cfr. 16.2 for more details). These fields are named **meta-properties**. Each meta-property is described by an instance of the *meta_property* object type.

A description of the fields follows:

- *name*: The name of the property.
- *type*: The type of the property. It cabn be one of the constants listed in 12.18: integer: NUM_ATT, string: VARCHAR_ATT, char: CHAR_ATT, float[13] : FLOAT_ATT, boolean: BOOL_ ATT.
- *updatable*: This field is an integer value used as a boolean value. If the value is equivalent to the constant TRUE, then the meta-property may be updated in the DB-MAIN environment. Otherwise, the value of the meta-property can only be read and not modified. However, Voyager 2 programs can always modify this field, no martter if it is *updatable* or not.
- *multi*: This field is an integer value used as a boolean value. If this field has the value TRUE, the meta-property may be multivalued. The value of such a meta-property is then a list of values of the same type, defined in the *type* field.
- *predefined*: This field is an integer value used as a boolean value. If this field has the value TRUE, then the possible values of such a meta-property must be taken in a predefined list of values. For instance, a meta-property called *gender* could have as predefined values *female* or *male*. The predefined values must be compliant with the type of the meta-property and are stored in the semantics description of the meta-property as a textual property. Note that although the choice of a predefined value is enforced in the DB-MAIN tool, there is no automatic validation in Voyager 2, so the programmer should always take care of taking values in the list only.
- *hidden*: This field is an integer value used as a boolean type. When it is TRUE, the meta-property is predefined by DB-MAIN for its own needs. It can normally not be accessed by DB-MAIN users. Voyager 2 programs should be very careful when modifying them, which can crash DB-MAIN.
- *sem*: A semantics description of the dynamic property.

A new *meta_property* can be created by the following command:

```
what := create(META_PROPERTY,
    name: string,
    type: integer,
    sem: string,
    updatable: integer,
    predefined: integer,
    multi: integer,
    hidden: integer,
    flag: integer,
    @MO_MP: meta_object);
```

13. This type is not yet supported by Voyager 2.

The following example shows how to create meta-properties.

```
meta_property: mp;
meta_object: mo;
entity_type: ent;

begin
  // add the meta-property 'local' to each collection
  // the meta-property is predefined and the allowed values are:
  //   Bruxelles, Paris, Madrid, London
  mo:=GetFirst(META_OBJECT[mo]{mo.type=COLLECTION});
  mp:=create(META_PROPERTY,name:"local",type: VARCHAR_ATT,
      predefined:1,@MO_MP:mo);
  mp.sem:=SetProperty(mp.sem,"VALUES",
      "Bruxelles\nParis\nMadrid\nLondon");
  // add the meta-property 'owners' to each entity-type
  // the meta-property is multivalued
  mo:=GetFirst(META_OBJECT[mo]{mo.type=ENTITY_TYPE});
  mp:=create(META_PROPERTY,name:"local",type: VARCHAR_ATT,
      multi:1,@MO_MP:mo);
  ...
  // let 'ent' be an entity-type
  // add 'tintin' to the list of owners.
  ent."owners":=ent."owners"++["tintin"];
  print(ent."owners");
end
```

12.50 user_viewable

user_viewable		
object_view	0-∞	user_view
uv_nnn	0-∞	nn_note

The *user_viewable* object type is a virtual type inherited by object types which are shown in their own window in DB-MAIN, such as *schema*.

Some objects of some types, such as *note* or *si_attribute* for example, can be shown in several views. For instance, a *si_attribute* object can be shown in a UML class diagram, and also in a UML activity diagram. In each diagram, this *si_attribute* should have different positions, stored in different *user_object* objects (see 12.2). One of the *user_object* object is linked to the *user_viewable* object inheritating the schema of type UMLCLASS_DIAGRAM, and a second *user_object* is linked to the *user_viewable* inheritating the schema of type UMLACTIVITY_DIAGRAM. Furthermore, different notes (*note*) can be attached to the same *si_attribute* in the various views. Several objects of type *nn_note* will link the various *note* objects to the *si_attribute* and to the *user_veiwable* objects corresponding to the view into which the notes must appear.

12.51 user_view

user_view		
isa *generic_object*		
name	string	1-1
type	char	1-1
font_name	string	1-1
font_size	integer	1-1
mark_plan	integer	1-1
reduce	integer	1-1
text_font_name	string	1-1
text_font_size	integer	1-1
xgrid	integer	1-1
ygrid	integer	1-1
zoom	integer	1-1
@object_view	1-1	user_viewable
uv_uo	0-∞	user_object

This object denotes the graphical representation of a product (schema, document, or a system). The programmer is not allowed to create new user views.

A description of the fields follows:

- The *font_name* and *font_size* attributes denote the font used in graphical views.
- The *text_font_name* and *text_font_size* attributes denote the font used in textual views.
- The *mark_plan* attribute is the current mark plan used in a schema (see page 76 for more details).
- The *reduce* attribute is the current reduce factor (in percent) in the graphical views.
- The *zoom* attribute is the current zoom factor (in percent) in the graphical views.
- The *xgrid* and *ygrid* attributes denote the size of the page drawn in the graphical views.

12.52 product_type

product_type		
isa *generic_object*		
title	string	1-1
min_mul	integer	1-1
max_mul	integer	1-1
description	string	0-1 (="")

This object type is used to give a type to products created by Voyager 2 procedures used with an MDL method [7]. *product_type* are read-only objects: their fields can be read but their values cannot be changed and new *product_type* cannot be created.

A description of the fields follows:

- *title*: the name of the product type.
- *min_mul*: the minimum number of products that should be created with this type.
- *max_mul*: the maximum number of products that should be created with this type.
- *description*: a small text describing the semantics of the product type.

Note that *min_mul* and *max_mul* are guidelines which are not enforced.

12.53 schema_type

schema_type	
isa	*generic_object*

A type for schema created within an MDL method [7].

12.54 document_type

document_type	
isa	*generic_object*

A type for document created within an MDL method [7].

Chapter 13

Predicative Queries

13.1 Introduction

Voyager 2 provides predicative queries to access the repository of DB-MAIN. The aim of these queries is to hide and to factorize boring and technical details when querying the repository. With predicative queries, the programmer has not to tell how to obtain a result but simply what he wants. These queries guarantee the same performance as a *hand-written* algorithm. Although navigational queries are provided too in Voyager 2 (see Chapter 14), the use of predicative queries is recommended.

Examples:

To get all the optional attributes of a project, the following query will return the expected result in a list:

```
ATTRIBUTE[att]{att.min_rep=0}
```

This query is an expression that can be used everywhere a list-expression is expected. Another form of query is:

```
for dat in DATA_OBJECT[dat]{@SCH_DATA:[GetCurrentSchema()]}
do {
  .
  .
  .
};
```

this example shows how to iterate through all the data-objects from the current schema.

Finally, a more complex query:

```
list_result:=DATA_OBJECT[dat]{@SCH_DATA:[GetCurrentSchema()]
                              with soundex(dat.name,"bank")};
```

The result is the list of data-objects from the current schema having a name similar to "bank"[1].

There are two kinds of predicative queries: *global scope* and *restricted scope* queries. A formal specification is given in the following sections.

1. soundex is not a primitive of Voyager 2.

13.2 Specifications

All the queries have to respect the following syntax:

⟨query⟩ ← ⟨ent-expr⟩ "["⟨variable⟩ "]" "{" ⟨constraint⟩ "}"

where

- ⟨ent-expr⟩ is an integer expression denoting an object of the repository. Although any integer expression is valid, programmers will usually use constants from the table 2.4. The meaning of each constant has been explained in the chapter 12.
- ⟨variable⟩ denotes a variable whose type must be exactly the same as the object type represented by the ⟨ent-expr⟩ expression[2].
- ⟨constraint⟩ is used to sort out the pertinent objects.

Once the query has been evaluated, the value of the ⟨variable⟩ is undefined. During the evaluation, the program must not modify its content. The scope of the ⟨variable⟩ is just a portion of the ⟨constraint⟩. This last characteristics will be explained below. The ⟨variable⟩ is called the *iterator* for convenience.

13.2.1 Global Scope Queries

Global-scope queries look the whole repository for objects satisfying the constraint. The constraint, any integer expression, is used as a boolean expression. It is in the scope of the iterator.

For instance, the query

```
ENTITY_TYPE[e]{TRUE}
```

will look for all the entity-types of the project. All these entity-types are then stored in a list.

The following example shows how to use the variable specified in the query to express more accurate constraints:

```
ATTRIBUTE[a]{a.min_rep=1 and a.max_rep=1}
```

The query looks for all the single-valued mandatory attributes of the project.

Objects having one or more sub-types cannot be used in *global scope* queries. If such a cuse occurs, the query will return an empty list.

13.2.2 Restricted Scope Queries

In restricted-scope queries, constraints have two components: the *link-constraint* and the *boolean-constraint*. Although the first part is mandatory, the second one is not. The syntax of this constraint is:

⟨constraint⟩ ← ⟨link-constraint⟩ [" with" ⟨boolean-constraint⟩]

⟨link-constraint⟩ ← ⟨link-expr⟩ : ⟨list-fathers⟩

⟨boolean-constraint⟩ ← ⟨integer-expression⟩

⟨link-expr⟩ ← [" @"] ⟨integer-expression⟩

⟨list-fathers⟩ ← any expression of type list

where:

- The ⟨link-expr⟩ is an integer expression denoting a link between two objects. Usually, programmers will use the constants of table 2.5 rather than complex expressions. Each constant is explained in chapter 12. Because links are oriented, it suffices to reverse the sign of a ⟨link-expr⟩ to reverse the corresponding link. In queries, the special symbol "@" is equivalent to the unary operator "-", and programmers are encouraged to use it in order to make the query more readable. Therefore, if L is a constant denoting a link, then L ≡ @@L.

2. Although <ent_expr> is an expression whose value can be changed at execution time, the <variable> must be typed at compilation time and its type cannot be changed during execution.

- The ⟨ent-expr⟩ expression denotes an object type (see above). This object type must be exactly the same as the one playing the role @L, if L is the value of "⟨link-expr⟩".
- ⟨list-fathers⟩ denotes any expression the evaluation of which returns a list. Let *l* be such a list and let L be the value returned by the evaluation of "⟨link-expr⟩". Then each item of the list *l* must be compatible with the object type playing the role @L. ⟨list-fathers⟩ is not in the scope of the iterator.
  Example:
  If the value of L is @PFROM, then the object type playing the role @@PFROM (≡ PFROM) is *product*. All the object types compatible with *product* are: *document*, *schema* and *generic_object*[3].
- The ⟨boolean-constraint⟩ must be preceded by the *with* keyword. This part is optional in the constraint. This expression is in the scope of the iterator and can use it.

For example, let A, A_0, A_1, B, B_0 and B_1 be object types, and a, a_0, a_1, b, b_0 and b_1 be their respective fields. Let the following relations hold: A_1 isa A, A isa A_0, B_1 isa B and B isa B_0. Let L be a link between A and B: $A \xrightarrow{L} B$. This schema is depicted in figure 13.1.

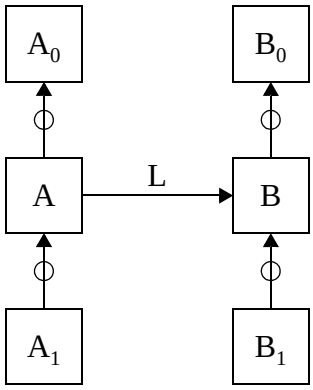


Figure 13.1 - Acadamic Schema.

Exercises: Are these requests correct?

a) A[α]{L:β1, β2}
   where A:α; B:β_1, β_2

b) $A_0[\alpha_0]${L:β1, β2}
   where $A_0:\alpha_0$; B:β_1, β_2

c) A[α]{L:β_0,β_1}
   where A:α; $B_0:\beta_0$; $B_1:\beta_1$

d) B:[β]{@L:A[α]{α.a>=3} with β.b<=5 and β.b>0}
   where A:α; B:β

e) A[α]{L:B[β]{@L:A[α_1]{α_1.a=β.b} with β.b=3}}
   where A:α,α_1; B:β

f) A[α]{L:B[β]{@L:A[α]{α.a=6} with β.b=3}}
   where A:α; B:β

g) A[α]{L:B[β]{@L:A[α_1]{α_1.a=6} with β.b=3}}
   where A:α,α_1; B:β

h) A[α]{L:B[β]{β.b=1}++B[β]{@L:A[α_1]{α_1.a=2}}}
   where A:α,α_1; B:β

3. Note that each document or schema is a product, but all the generic objects are not product. So, if the list contains generic objects, The program must have the insurance that they are all compatible with the product object type.

Solutions:

a) **YES**: without any comment.

b) **NO**: L is a role played by the object type A. A_0 cannot be used in place of A in this query.

c) **YES**: The list of fathers $[\beta_0,\beta_1]$ may be composed of any value for which the type is compatible with the object type playing the role @L.

d) **YES**: without any comment.

e) **NO**: The constraint α_1.a=β.b is not in the scope of the β iterator. The value of β is undefined.

f) **NO**: The same variable α is used two times as iterator in the query.

g) **YES**: without any comment.

h) **YES**: The query asks for all the objects from A that play the role L for one element of the list B[β]{β.b=1}++B[β]{@L:A[α_1]{α_1.a=2}}. This list is a new expression, independent of the query. This expression is made of two queries. Their evaluations return two lists. The concatenation of these two lists is used as a "list of fathers" in the main query. Note: this query is not very efficient. How can it be rewriten more efficiently?

Chapter 14

Iterative Queries

Although predicative queries (see Chapter 13) are very useful, they cannot be used in every situation. For this reason, Voyager 2 offers basic primitives to access the content of the repository more directly: *_GetFirst*(), *_GetNext*(), *TheFirst* and *TheNext*.

---

function generic_object: o TheFirst (integer: t)

Precondition. t must be an integer expression. The evaluation of t must return a value which denotes an object type, represented by a constant of table 2.4.

Postcondition. o is the first object of type t found in the project. If there is no object of type t, then o is void.

---

function generic_object: o TheNext (integer: t, any: p)

Precondition. t must be an integer expression. The evaluation of t must return a value which denotes an object type, represented by a constant of table 2.4. p must be a reference to an object type t. p must be different of void.

Postcondition. o is the object that follows p in the list of objects of type t. If p is the last object, then o is void.

on error: The behaviour is uncertain.

---

function any: s _GetFirst (integer: l, any: f)

Precondition. The value of l must denote a link (see table 2.5). f must be different of void and its type must be compatibe with the object type that plays the role l.

Postcondition. If $[s_1,..., s_n]$ is the list of objects linked to f by l, then o is the first element of this list. If the list is empty, then s is void. The type of o is the type of the object type that plays the role @l.

on error: The behaviour is uncertain.

---

function any: b _GetNext (integer: l, any: f, any: s)

Precondition. the value of l must denote a link (see table 2.5). f is an object whose type must be compatibe with the object type that plays the role l. s must be one of the objects linked to f by l.

Postcondition. Let $[s_1,...,s_i,s_{i+1},...,s_n]$ be the list of all the objects linked to f by l, and let i be the index of s (s_i = s). If i < n then o = s_{i+1}, and if i = n then o = void.

on error: The behaviour is uncertain.

It is recommended to avoid the use of these functions as much as possible since predicative queries have the same performance and are less error prone.

Example:

This example shows how to use the _GetNext function:

```
entity_type: ent;
attribute: att;
begin
  ent:= one entity type expression;
  for att in ATTRIBUTE[att]{@OWNER_ATT:[ent]}
  do  {
      print(att.name);
      if IsNoVoid(_GetNext(OWNER_ATT,ent,att)) then {
        print(',');
      };
  };
end
```

Chapter 15

Object Removal

The removal of an object from the repository is done by a call to the *remove* procedure. This procedure requires one argument: the object to remove. This procedure will remove the specified object as well as other objects, in cascade, in order to preserve the integrity of the repository. For instance, if a rel-type is removed, all its attributes and all the roles played by entity types in this rel-type will be removed too.

---

procedure remove (object: o)

Precondition. o may be any object of the repository. This value cannot be *void*.

Postcondition. The object and all the other objects that depend on it are removed.

Example:

```
remove(GetFirst(ENTITY_TYPE[ent]{name="CLIENT"}));
```

Chapter 16

Properties

Although the repository of DB-MAIN is statically defined with C++ classes, it can be dynamically extended with **Textual Properties** and **Dynamic Properties**.

These two kinds of properties have the same aim: "to add new fields to one object-type in the repository". These two technics are completely different. They will be explained in this chapter.

16.1 Textual Properties

Textual properties are new properties attached to an object of the repository. These information are stored in either the semantic description field or in the technical description field of the object. These properties are not completely supported in Voyager 2, but two functions can help the programmer to manage them: GetProperty and SetProperty. Each function is fully described hereafter. For an example of a *property*, let us suppose that *o* is one object (ex: an entity-type variable) whose technical description is :

```
"This object denotes a car bought by a firm.ø
Each car is pink.ø
eof"
```

To add a new property to some entity-types, like "the average number of instances in the database[1]", The technical description can be updated in the following way:

```
"This object denotes a car bought by a firm.ø
Each car is pink.ø
#average=26ø
eof"
```

This information can be retrieved by a lexical analysis of the text.

Conventions are defined in DB-MAIN to represent this kind of information inside texts. The definition is:

> A *textual properties* is made up of two parts: the *field* and the *value*. The representation of these information can occur anywhere in a text and must respect the following rules: the field is the list of characters found between the ’#’ character placed at the beginning of a line, and the first occurrence of the ’=’ character placed on the same line. All the characters are useful and their interpretation is

1. WIth a relational database, the terms "tuple" or "line" should be used instead of "instance".

case sensitive. The value associated to the field is the list of characters found just after that '=' character, until the first '#' character starting a new line, or until the end of the text. In the first case, the '#' character and the preceeding *new-line* character do not belong to the value. If several properties with the same field exist in the text, only the first occurrence is taken into account.

The two functions defined hereafter help the programmer to manage the textual properties stored in the semantic and technical descriptions of any object (and in any text in general).

---

function string: value GetProperty (string: s, string: field)

Precondition. Ø

Postcondition. If the field *field* is found in the text *s*, then the associated value is returned to the user. Otherwise, the constant *PROP_NOT_FOUND* is returned. This message is distinct of any possible value The text *s* is left unmodified. If the text is corrupted, then the message PROP_CORRUPTED is returned.

---

function string: r SetProperty (string: s, string: field, string: value)

Precondition. Ø

Postcondition. This function returns the string *s* where the value associated with the field *field* has been replaced by the text *value*. If the *field* is not present in *s*, it is added at the end of *s*, with its associated *value*.

Example:

```
schema: sch;
integer: i;
string: s;
begin
  sch:=GetCurrentSchema();
  s:=GetProperty(sch.sem,"color");                    (1)
  if s=PROP_NOT_FOUND then { i:=0; }
  else { i:=StrStoi(s); };
  sch.sem:=SetProperty(sch.sem,"color",StrItos(-i));
end                                                   (2)
```

At the point **(1)**, the semantic description of the schema *sch* is:

```
This schema will be printed on our printer with the color:↵
#color=4↵
#end↵
But if the color is negative, this color is used for the↵
background! ↵
eof"
```

and after the execution of the program the semantic description of the schema has been updated with the opposite of the color field:

```
"This schema will be printed on our printer with the color:↵
#color=-4↵
#end↵
But if the color is negative, this color is used for the↵
background! ↵
eof"
```

Let us note that the special line "#end" is used to mark the end of the property. This line could have been replaced by any other property like this:

```
"This schema will be printed on our printer with the color:↵
#color=-4↵
#font=Arial↵
```

```
#end↵
But if the color is negative, this color is used for the↵
background! ↵
eof"
```

Last but not least, this last function retrieves all the properties from a string with their associated values and put them in a list returned to the user:

function list: l GetAllProperties (string: s)

Precondition. Ø

Postcondition. *l* is a list of pairs [p_1,v_1,p_2,v_2,...,p_n,v_n] where v_i is the value of a property called p_i. The p_i are all the properties present in the text *s*.

16.2 Dynamic Properties

16.2.1 Introduction

Dynamic properties is the second mechanism provided in DB-MAIN to extend the repository. This is the mechanism of choice, the one that should be prefered by any DB-MAIN user.

The repository is described in one of its parts called "*meta-repository*". This part contains object types: *meta_object* and *meta_property*. To some object types of the repository corresponds one instance of the *meta_object* object type. Similarly, each attribute/property of an object type is described by an instance of the *meta_property* object type. Chapter 12 describes these two object types in more details.

This meta-description is very interesting since it allows DB-MAIN users to dynamically extend the repository. It suffices to add a new instance of *meta_property* and to link it to an instance of *meta_object* in order to add a new property to the object type described by the meta-object. Once this is done, the new property is available both in the CASE tool and in Voyager 2.

16.2.2 Explanation

Dynamic properties can be illustrated by an example. In a company, ER-schemas are defined by several people at the same time. To manage this complexity, it is decided to add to each entity type and to each relationship type a new field indicating who added this object to the schema.

First, a new property should be added to the *entity_type* object. This property will represent a person ("Albert", "Bill", "Jessica", …), its type will be *string*. The project leader can do it by using the CASE tool, with the menu File → Meta... → Properties. He can also do it with a Voyager 2 program:

```
meta_object: mo;
meta_property: mp;
begin
  mo := GetFirst(META_OBJECT[mo]{mo.type=ENTITY_TYPE});
  mp:=create(META_PROPERTY,name:"owner",type: VARCHAR_ATT,@MO_MP:mo);
  ...
end
```

A new property named "owner" is now added to the entity_type object. Every entity-type of the project has a new field initialized with an empty string. A similar work can be done with rel_types.

If the entity type "CLIENT" is created by Mr Sherlock Holmes, the new field can be set in this way:

```
entity_type: ent;
...
  ent:=GetFirst(ENTITY_TYPE[ent]{ent.name="CLIENT"});
  ent."owner":="Sherlock Holmes";
  ...
  print([ent.name," has been created by ",ent."Owner",' \ n']);
end
```

Remark: a dynamic property can be referenced like any other field with two exceptions:

1. the name of the field is between double quotes. It is a string.
2. the mechanism is not case sensitive.

The sentence found after the "." may be any expression. If the evaluation of the expression returns an integer, then the field is static, otherwise the value should be a string, and the field is *dynamic*. When the field is static (*name*, *short_name*, *sem*, *tech*, *min_rep*,...), the name of the field is predefined in Voyager 2 as an integer constant.

PART III

MODULAR PROGRAMMING

Chapter 17

Library and process

One of the main innovations of the version 3.0 of the Voyager 2 language is the ability to use Voyager 2 programs as libraries or as processes, due to a new architecture of the abstarct machine used to run Voyager 2 programs. This chapter explains how to use these characteristics.

17.1 The New Architecture

The abstract machine is composed of two memory blocks. The code of a program (ie. the .oxo program) is stored in the first one, and the second one is the memory used during the execution of the program. They are respectively called the **image** and the **stack**. For example, the program "foo.oxo" can be executed. It can be loaded and stored in one or many images. Once the program is loaded, it can be executed. A stack is created as a working space for the new process. But, from the same image, another process can be run which needs another stack to be created. So, two stacks are created for a single image. But the architecture allows another program to be loaded, and so a new process and a new stack. The situation is depicted in figure 17.1.

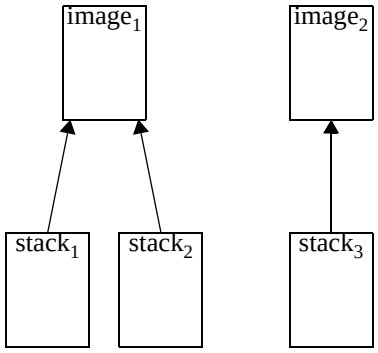


Figure 17.1 - This schema depicts the memory state with two programs and three running processes.

An image stores all the instructions of a program, and therefore each function or procedure has its representation inside the image. It is possible to execute either the whole program (ie. the body) or simply one function or one procedure (from DB-MAIN assistants, for instance).

As long as a stack is preserved, the "memory" of the process is preserved. So, if one process is executed, its global variables will be left in some state (S) at the end of its execution, and the same state will be retrieved at next execution.

With this architecture, two new types have been added to the language. The first one is *program*. A value of type *program* denotes a process. Once a variable of this type is correctly initialized, the associated program can be executed. The second type is *lambda*[1]. It denotes an entry in the image of a specific *program* corresponding to a procedure or a function. Such a value, once initialized, can be used to start the execution of the associated function or procedure using the stack of the process. These two types are presented in more details in section 17.2.

17.2 Voyager 2 Process

The first step in the use of a process is the declaration of a variable of type *program*.

```
program: p;
```

Once this variable is declared, it can be initialized with the *use* instruction.

```
p := use("c:\\foo.oxo");
```

The *use* instruction has only one argument, a string, which denotes the program to be loaded into a newly created image. Once the *p* variable is initialized, the program can be called in such a way:

```
p!(a1,...,an);
```

The ! character is a suffix unary operator. a_1,...,a_n are values passed as arguments to the process.

A new characteristic of Voyager 2 is that a program can return a value, and its type does not have to be specified. Hence, a program can either be executed as a procedure (an instruction) or as a function (an expression) depending on the usefulness of the returned value in the calling program[2]. So, another call to the program *foo.oxo* is:

```
v := p!(a1,...,an);
```

Functions and procedures can be called separately. The first step to call such a procedure is to get a "handle" to the function from the program/process. This is done with the binary operator :: . The left operand denotes the process, and the right operand is a string that denotes the name of the function or the procedure. The result of this operator is a value of type *lambda*. This result can be stored in a variable for later use. This can be done as follows:

```
lambda: fct;
...
fct := p::"my_function";
```

where the *fct* variable is declared of type *lambda*. Once this variable is initialized, it can be used to call the function in the following way:

```
x := fct::(y1,...,ym);
```

The suffix unary operator :: calls and executes the function "my_function" inside the process p. The values y_1,..., y_m denote the arguments of the function. The way to execute a procedure is similar.

So a complete program could be:

```
program: p;
lambda: fct;
```

1. The "*lambda*" term derives from an area of mathematical logic called the "*lambda calculus*" in which many of the theorical foundations of functional languages are based [4], [2]. lambda expressions in Voyager 2 have very few common characteristics with this concept but the author did not have enough imagination to invent a new term.
2. This approach is close to the philosophy of the C language, where functions can be used as procedures.

```
string: result;

begin
  p:=use("c:\\foo.oxo");
  fct:=p::"FormatDate";
  result:=fct::(13, "Feb", 1968);
  print(result);
end
```

The following program is more concise and is strictly equivalent to the previous one:

```
begin
  print((use("c:\\foo.oxo")::"FormatDate")::(13, "Feb", 1968));
end
```

The second form shows that *program* and *lambda* values can be used anywhere where such types are expected.

After examining the calling programs, it is interesting to have a closer look at the called functions and procedures. In fact, they look like any other V2 program. The only difference is that functions or procedures that can be called from outside must be preceded with the *export* keyword. So the FormatDate function declaration should have the following syntax:

```
export function string FormatDate(integer: d, string: m, integer: y) ...
```

The syntax of a procedure is the same. The *export* keyword allows programmers to define functions for private use and other ones with unlimited access.

Another important principle is the preservation of the stack as long as it could be needed to execute the process or a function or a procedure attached to this process. This can be shown by the two following programs:

```
/******************/
/* calling program */
/******************/

program: p;
lambda: fct;

begin
  p:=use("c:\\foo.oxo");
  fct:=p::"nestor";
  p!();
  print(fct::(1));
  print(fct::(3));
end
```

and

```
/*****************************************/
/* called program. Stored in file c:\foo.oxo */
/*****************************************/
integer: n;

export function integer nestor(integer: a){
  n:=n+a;
  return n;
}

begin
  n:=1;
end
```

The execution of the first program calls the body of the *foo* program and stores 1 into the global variable *n*. Then, the function *nestor* is called. This function uses the variable *n* from the stack in the state left by the previous execution. In the example, *n* is 1 and the function returns 1+1=2. The second call returns 2+3=5.

Several processes can be initialized from a program. For instance, if the previous calling program is rewritten as follows, its execution prints 2 and 4 as a result, because the execution of the first function does not influence the stack of the second function.

```
program: p1,p2;
lambda: fct1,fct2;

begin
  p1:=use("c:\\foo.oxo");
  p2:=use("c:\\foo.oxo");
  fct1:=p1::"nestor";
  fct2:=p2::"nestor";
  p1!();
  p2!();
  print([fct1::(1),fct2::(3)]);
end
```

A program called from another program has to retrieve arguments and return a result. In the following extract, the "foo.oxo" program is called two times. The first call tests the result of the program, which represents an error code, and displays a message if necessary. For the second call, the programmer is more confident into the result and does not test the error code.

```
program: prog;
begin
  prog:=use("c:\\foo.oxo");
  if prog!(1,2,3)=0 then {
    print("error message");
  }
  prog!(4,5);
end
```

The number of arguments may change from one call to another. Let the called program, "foo.oxo", be defined as:

```
integer: sum,i;
begin
  if Length(Environment) then {
    for i in Environment do {
      sum:=sum+i;
    }
    print(sum);
    return 1;
  } else {
    return 0;
  }
end
```

This version contains two important changes from previous versions:

1. The global variable *Environment* is not defined in the program although it is used. This variable is predefined in Voyager 2 and denotes a list. This variable is initialized with the list of the arguments present in the call. This variable can be used as any other variable.
2. The *return* instruction is now allowed in the body of the program. It returns the result of the calling program. The type of this value is not specified and can be of any type (integer, string, list, ...).

The user does not have not to care about process unloading. Like for strings and lists, the Voyager 2 language automatically unloads processes from the memory (image and stack) once it is no more needed.

Because, process and lambda values are first order classes, these values can be manipulated as any other value. Such expressions can be passed as arguments to functions (extern or not), stored into lists,... Only inputs and outputs are not allowed: a process or a value cannot be printed to or read from a device (file/keyboard).

Sometimes, the process loading can fail (not enough memory, .oxo file corrupted, security failure, mistyped filename,...). In such events, the loading mechanism returns either a program or a lambda value *void*. The user has to explicitely test these values with the usual functions (IsVoid or IsNoVoid) in order to insure the program correcteness.

Some details were not fully explained in this description. The complete formal definition of each concept is given later in this chapter (section 17.4).

17.3 Libraries

Libraries are just a convenient cosmetic cream on top of the previous concepts. Processes are more often used as "libraries" than as "real" processes (as known in operating system). But the dynamic management of such processes is too heavy and unnecessary. For this reason, Voyager 2 provides a special syntax to declare libraries.

For instance, a program can contain several functions to manage advanced functionalities (trees, associative lists,...). To use all these functions, this library/program should be loadeds each time, lambda variables should be declared for each used function/procedure and so on. But Voyager 2 allows the declaration of a library at the beginning of the program. Here is an example:

```
/************************/
/* libraries declaration */
/************************/
use "c:\\lib\\tree.oxo" as mylib;
use mylib.DisplayTree as DisplayTree;
use mylib.ErrorProcess as TreeError;
use mylib.ComputeDepth as Depth;
use "c:\\lib\\assoc.oxo" as associative_list;
use associative_list.SetAssoc as SetAssoc;
/* Global Variables */
integer: a,b,c;
...
begin
  ...
  DisplayTree::(MyTree,File);
  print(Depth::(MyTree));
  ...
end
```

The *use* keyword is used in two different ways. The first one gives a logical name to a process/library. The second one gives a logical name to function/procedure inside a library.

Libraries, contrary to *program*/*lambda* declarations, always allow processes to be initialized, no matter if the body is executed or not. This characteristic is important a library needs another library. For instance, if the program needs the library "tree.oxo" and if this last one needs the library "record.oxo", "record.oxo" must be loaded before any function/procedure of "tree.oxo" is called. This is ensured by the *use* statements at the beginning of the program. Otherwise, the process should be executed before any other function can be called from this process.

To ease the programming job, the compiler produces a file with the extension ".ixi" that contains all the needed *use* declarations for each function/procedure defined with the *export* keyword. This file can be included with the directive explained in chapter 18.

17.4 Formal Definitions

17.4.1 The use Function

function program: p use (string: s)

Precondition. *s* is a string that denotes the name of a Voyager 2 program. The name must follow the syntax of paths in MS-DOS. This program must have the same security privileges as the calling program (see chapter 19).

Postcondition. *p* denotes a new process that has been loaded in memory with a newly created stack. The process is just loaded and no execution has been launched.

on error: The value *void* is returned.

17.4.2 The GetLambda Function

function lambda: f GetLambda (program: p,string: s)

Precondition. *p* denotes a process and *s* is a string that denotes the name of a function or a procedure in *p*.

Postcondition. *f* is a lambda expression that can be used to call the function or procedure *s* in *p*.

on error: The value *void* is returned.

17.4.3 The ! suffix unary operator

Syntax: P ! (a_1,...,a_n)

P denotes a *program* expression. If *P* is not *void*, then the program or process denoted by *P* is called using the a_1,...,a_n values as arguments. Otherwise, nothing happens. If the ! operator is used inside an expression, the called program must necessarily return a well-typed value as expected according to the context. If the operator is used as an instruction, the returned value is ignored.

17.4.4 The :: suffix unary Operator

Syntax: F::(a_1,...,a_n)

F is any expression that is evaluated as a lambda value *F'*. Depending on the nature of *F'*, the function or procedure denoted by this value (*F'*) is called using the expressions a_1,...,a_n as parameters. The number of arguments must be strictly the same as the number specified in the definition of the function or the procedure. The compiler cannot check if the number of arguments is correct because it misses information to do so. Arguments must have exactly the same types as declared in the definition. If *F'* denotes a function, the *F*::(a_1,...,a_n) expression is evaluated as the result of this function. Otherwise, the procedure is just called. Once again, the compiler does not have enough information to check that functions are not used as procedures or reciprocally. If *F'* denotes the value *void*, nothing happens and the execution is aborted with an error message. If the number of arguments is wrong, the stack will be corrupted and the program will stop without any explicit message. The same event occurs if a procedure is used as a function and reciprocally. Such errors are programming errors that cannot be catched by the compiler or the abstract machine.

17.4.5 The :: binary Operator

Syntax: P :: N

P is a program expression and *N* is a string expression. The :: operator is evaluated as a lambda expression that corresponds to a function or procedure named *N* and defined in the program *P*. The stack of *P* will be used during the execution of the function or the procedure. Let *R* be the result, then if *IsVoid*(*P*)=*TRUE* then *R*=*Void*(*_lambda*). Otherwise if the *N* string does not match a function or a procedure in the specified program, then the value *void* is returned.

17.5 Literate Programming

Programmers often document their programs with comments. But few environments allow to recover the program documentation from comments[3]. Voyager 2 tries to use comments found in programs to document the ".ixi" files produced by the compiler. In the Voyager 2 syntax, *explain* clauses can appear in the head of programs and inside each function or procedure. *Explain* clauses are used by the compiler to produce documented ".ixi" files. Figures 17.2 and 17.3 show how the compiler works on an example:

```
explain (*Factorial Library. Author: Nestor Burma *)

export function integer fact1(integer: a)
explain (*fact1 computes the factorial of its argument in using
a recursive algorithm *)
{ if a=0 then { return 1; }
  else { return a* fact1(a-1); }
}

export function integer fact2(integer: a)
explain (*fact2 computes the factorial of its argument in using
an iterative algorithm *)
  integer: i, result;
{ result:=1;
  for i in [1..a]
  do { result:=result*i; }
  return result;
}

begin
end
```

Figure 17.2 - Litterate Programming: an example of Voyager 2 program including an *explain* clause.

3. "*Literate Programming*" first appeared in the WEB programming language, which is a mix of TeX sentences and Pascal statements. WEB was defined by D. Knuth.

⇓

```
/* FILE GENERATED ON: 23/IX/1997 at 10:44,24 secs
** Please, does not modify this file.
** Voyager 2 Declarations
** Compiled with Version 3 Release 0 Level 2 */

use "facto.OXO" as facto;

/******************
 * Documentation: *
 ******************

Factorial Library. Author: Nestor Burma */

use facto.fact1 as fact1;

/* FUNCTION returns integer
   Arguments:
    1) integer: a
   EXPLAIN:
fact1 computes the factorial of its argument in using
a recursive algorithm */

use facto.fact2 as fact2;

/* FUNCTION returns integer
   Arguments:
    1) integer: a
   EXPLAIN:
fact1 computes the factorial of its argument in using
an iterative algorithm */

/* IXI file completed */
```

Figure 17.3 - Literate Programming: The ".ixi" file is produced from the Voyager 2 program shown in Figure 17.2. Each "explain clause" is used to document exported functions as well as the library itself.

Chapter 18

The Include Directive

Recurrent needs were observed during the programming phase. For instance, the same constants, functions and procedures are always necessary. For this reason, programmers often have to duplicate sections of code from one program into another. Such programs become rapidly difficult to maintain.

It is now possible to include files into a program by using a directive statement. Its syntax is:

```
#include "..." (same as in C language)
```

Spaces are not allowed between the # character and the *include*. However, the # may appear anywhere in a line.

This directive may appear anywhere in the program: in the library section, in the global variables declarations, between statements, or even inside an expression. The semantics of this directive is quite simple: the compiler replaces the directive with the content of the file specified in argument. The backslash characters do not need (and cannot) to be escaped. Here follows an example:

Example:

```
#include "c:\lib\tree.ixi"
...
integer: n;
#include "c:\lib\rtf_cst.h2"
...
function char foo(){
  ...
}

#include "c:\misc\error.h2"

begin
  #include "c:\misc\copyrigh.h2"
  ...
end
```

The language does not enforce the file extensions. However, it is stringly recommend to use ".ixi" for libraries declarations and ".h2" for other files. Although any other extension name can be chosen, life would be easier if everyone respects this convention (as in C).

It is also recommended to avoid as much as possible[1] the use of this directive. Libraries can often be used in place of this directive and should be prefered for methodological reasons.

If an error occurs in an included file, then the compilation process fails as for any other reason. However, if the compiler fails in opening an included file, this one is simply skipped and the compiler produces a warning. Very often, this will cause other errors to appear later.

1. The include directive really includes the content of a file and thus, enlarges the .oxo file size. Since the size of .oxo files is limited, you could exceed this limit.

Chapter 19

Security

Only licensed users of a specific version of DB-MAIN can compile Voyager 2 programs. This license is materialized by an electronic key which stores the following information:

- **A user ID**: A unique number associated with each user group. A company having several keys will have a single user ID shared by all its keys.
- **A key ID**: A unique number associated with each electronic keys. Two keys have distinct key ID.
- **Compile capability**: If this flag is set, the user can use the Voyager compiler. Otherwise, the compiler will stop its execution with an explicit error message.
- **Distribution capability**: If this flag is set, Voyager programs can be developed for users with a different user ID.

The compiler cannot be used without an electronic key having the *compile capability* flag set.

When the *compile capability* flag is set, the compiler produces ".oxo" files that can only be used with an electornic key having the same *userID* as the key used for compiling.

When the *distribution capability* flag is set, the user can specify an explicit *user ID* or an explicit *key ID* that restricts the use of the ".oxo" to computers with an electronic key with that *user ID* or *key ID*. These IDs should be specified with the following switches in the commande line of the compiler:

-Kclient: to specify a user ID (cfr. -infokey).

-Kkey: to specify a key ID (cfr. -infokey).

-Kall: the generated ".oxo" file can be run by everybody, with or without electronic key.

-infokey nnn: to specify the ID used by the compiler (cfr. -Kclient,-Kkey). nnn is the number/ID information.

Here follow some examples:

```
comp_V2 foo.v2 -Kclient -infokey 31414
comp_V2 foo.v2 -Kkey -infokey 27182
comp_V2 foo.v2 -Kall
```

The command line "comp_V2 foo.v2" is the same as the command line "comp_V2 foo.v2 -Kclient -infokey x " where x is the user ID of the electronic key used to compile.

PART IV

APPENDIX

Appendix A

The Voyager 2 Abstract Syntax

This chapter gives the abstract syntax of the Voyager 2 language. The syntax is defined as a set of rules written in extended BNF. The following conventions are respected.

- **head** ← body: a rule.
- "example": literal characters.
- example: a keyword.
- *example*: a terminal word that represents a class of tokens. (cfr. 2 for more details)
- **example**: a non terminal word which must be defined by another rule.
- the | operator denotes a disjunction in a body.
- $\langle \text{example} \rangle_{0,\infty}$: example is repeated 0 or more times.
- $\langle \text{example} \rangle^{\alpha}_{0,\infty}$: example is repeated 0 or more times and items are separated by "α".
- $\langle \text{example} \rangle_{1,\infty}$: example is repeated 1 or more times.
- $\langle \text{example} \rangle^{\alpha}_{1,\infty}$: example is repeated 1 or more times and items are separated by "α".
- ∅: the empty word.

A.1 The Syntax

program ← **explain-clause** ⟨**use-clause**⟩$_{0,\infty}$ ⟨**def-var**⟩$_{0,\infty}$ ⟨**def-fct**⟩$_{0,\infty}$ **body**

explain-clause ← ⟨explain "(*" *text* "*)"⟩$_{0,1}$

use-clause ← use *string* as *identifier* ";" | use *identifier* "." *identifier* as *identifier* ";"

def-var ← *type* ":" ⟨**one-var**⟩$^{,}_{1,\infty}$ ";"

one-var ← *identifier* ⟨"=" **expr**⟩$_{0,1}$

def-fct ← **def-function** | **def-procedure**

def-function ← ⟨export⟩$_{0,1}$ function *type identifier* "(" ⟨**arg**⟩$^{,}_{0,\infty}$ ")" **explain-clause** ⟨**def-var**⟩$_{0,\infty}$
"{" ⟨**instr**⟩$_{0,\infty}$ "}"

def-procedure ← ⟨export⟩$_{0,1}$ procedure *identifier* "(" ⟨**arg**⟩$^{,}_{0,\infty}$ ")" **explain-clause** ⟨**def-var**⟩$_{0,\infty}$
"{" ⟨**instr**⟩$_{0,\infty}$ "}"

arg ← *type* ":" *identifier*

body ← begin ⟨**instr**⟩$_{0,\infty}$ end

instr ← ∅

| **designer** ":=" **expr** ";"

| goto *identifier* ";"

| continue ";"

| break ";"

| halt ";"

| label *identifier* ";"

| return ⟨**expr**⟩$_{0,1}$ ";"

| **attach-stmt** ";"

| **move-stmt** ";"

| **addcursor** ";"

| **setval** ";"

| **dynamic-call** ";"

| **loop-stmt** ⟨";"⟩$_{0,1}$

| **ifthenelse** ⟨";"⟩$_{0,1}$

| **while-stmt** ⟨";"⟩$_{0,1}$

| **switch-stmt** ⟨";"⟩$_{0,1}$

| **repeat-stmt** ";"

| **call-procedure** ";"

designer ← *identifier* | *identifier* "." **expr**

expr ← **expr omega2 expr**

| **designer** ":==" **expr**

| "-" **expr**

| not "(" **expr** ")"

| "(" **expr** ")"

| "[" ⟨**expr**⟩'$_{0,\infty}$ "]"

| "[" **expr** ".." **expr** "]"

| **expr** "[" **designer** "]" "{" **constraint** "}"

| use "(" **expr** ")"

| **expr** "::" "(" ⟨**expr**⟩'$_{0,\infty}$ ")"

| **expr** "::" **expr**

| **expr** "!" "(" ⟨**expr**⟩'$_{0,\infty}$ ")"

| **designer**

| *integer*

| *float*

| *char*

| *string*

| *file*

| **call-procedure**

| **create-inst**

omega2 ← "-" | "+" | "*" | "<" | ">" | "<=" | ">=" | "=" | "<>" | "++" | "**" | and | or | xor | mod

constraint ← **expr** | **expr** ":" **expr** ⟨with **expr**⟩$_{0,1}$

call-procedure ← *identifier* "(" ⟨**expr**⟩'$_{0,\infty}$ ")"

create-inst ← create "(" **expr** "," ⟨**simple-field**⟩'$_{0,\infty}$ ")"

simple-field ← **expr** ":" **expr**

attach-stmt ← attach *identifier* to **expr**

move-stmt ← *identifier* { "⟨" | "⟩" } ⟨**expr**⟩$_{0,1}$

addcursor ← **expr** { "<+" | "+>" } **expr**

setval ← **expr** "<--" **expr**

dynamic-call ← **expr** "::" "(" ⟨**expr**⟩'$_{0,\infty}$ ")" | **expr** "!" "(" ⟨**expr**⟩'$_{0,\infty}$ ")"

loop-stmt ← for **designer** in **expr** do "{" ⟨**instr**⟩$_{0,\infty}$ "}"

ifthenelse ← if **expr** then "{" ⟨**instr**⟩$_{0,\infty}$ "}" ⟨else "{" ⟨**instr**⟩$_{0,\infty}$ "}"⟩$_{0,1}$

while-stmt ← while **expr** do "{" ⟨**instr**⟩$_{0,\infty}$ "}"

switch-stmt ← switch "(" **designer** ")" "{" ⟨**case-stmt**⟩$_{0,\infty}$ **default-case** "}"

case-stmt ← case **expr** ":" ⟨**instr**⟩$_{0,\infty}$

default-case ← ⟨otherwise ":" ⟨**instr**⟩$_{0,\infty}$⟩$_{0,1}$

repeat-stmt ← repeat "{" ⟨**instr**⟩$_{0,\infty}$ "}" until **expr**

A.2 Remarks

The #include directive does not appear in the Voyager 2 syntax since it is replaced everywhere by the content of the specified file.

Appendix B

The VAM Architecture

Although the only visible tool is the compiler, it is worthwhile to know that Voyager 2 is interpreted. In fact, there are two languages: Voyager 2(V2) and Voyager 1 (V1). The first one is described in this manual. The second one, V1, is an intermediate language between V2 and the abstract machine: the VAM[1].

Figure B.1 shows how a V2 program is translated into a binary file (prog.oxo) that can be loaded directly into DB-MAIN in order to be executed.

As a simple example, the following table shows extracts from three files:

prog.v2	prog.v1	prog.oxo

begin
print(1+2);
end
	push-int 1
	push-int 2
	add
	print
		13/1
		13/2
		65
		43

The prog.oxo is just a binary file composed of 6 words (16 bits): 13,1,13,2,65 and 43.

V1 looks like an assembler language and the ".oxo" file is just a binary translation of each instruction of the ".v1" file with its operands. The ".oxo" file can be loaded fast into the DB-MAIN tool because the parsing has already been done. In Figure B.1 the compiler is represented by a box that contains two hidden processes: the real V2 compiler (named *comp_v2*) and the V1 compiler (named *comp_v1*). What users can see is just the translation of the V2 program into the binary file. Once this compilation is completed (without error), the program can be loaded into the tool and can be executed.

1. Voyager Abstract Machine.

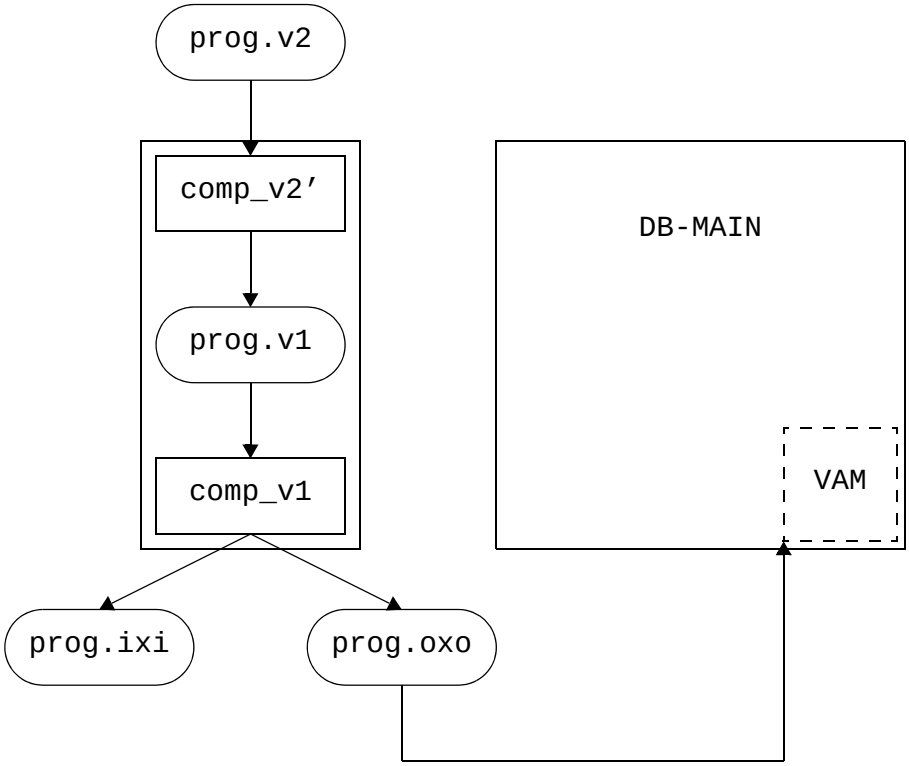


Figure B.1 - The Voyager Architecture.

Appendix C

Error Messages when Compiling

The compiler produces three types of error messages during the compilation:

- **warning**: The error is not important and the compiler is smart enough to continue producing the right code. Example: a *return* instruction is followed by an expression in a procedure.
- **error**: The error prevents the compiler from generating the right code. The compiler skips the error and continues its job looking for other errors. No code is produced.
- **fatal**: The error is too important to continue the compilation. No code is produced.

All the errors produced by the compiler are documented hereafter. The compiler always tries to display the line number of the error. Sometimes this number does not correspond to the right line. It is the case for compound instructions. This is a known bug[1].

1. Buffer too small. (maybe a line with too many characters)

   A line of the program is too long for the buffer of the compiler. It should be split into several lines.

2. Parsing error.

   There is a syntax error. The compiler should show the content of the line as well as the line number where the error occurred. The character '¨' indicates where the syntax is not verified. Very often, a ';' separator is lacking in the previous line.

3. Error while opening a file.

   It is impossible to open the specified file (.v2) or to create the output file (.v1). Is it shared by other applications? Does it exist? Are the *path* valid?

4. A function with the same name already exists.

   A function/procedure is being defined for the second time.

5. The left hand expression of an assignment must be a variable or a field.

   The program contains an instruction *E := T ;* where E is not a variable. Only variables (possibly with a field) are allowed in the left part of assignments.

6. A procedure is used but not defined.

   One instruction is calling a procedure that is not defined. Note that lower and upper case characters are considered as distinct in Voyager 2.

7. Unknown type.

1. "It is not a bug but a feature." Indeed, the compiler only knows the line number when one instruction is fully parsed. For this reason, the line number corresponds to the last line of the instruction and not the first one.

A bad type is used. Probably a misspelled type like "integger" or "siattribute".

8. The identifier is neither a variable nor a constant.

   A variable is expected here and the identifier is not defined as a variable.

9. The procedure is used as a function.

   The procedure is used in an expression and not as an instruction.

10. A variable is expected.

    A field with a variable used as loop-variable is specified in a request. Only variables are allowed here. Example: ENTITY_TYPE[e]{...}.

11. The identifier is already defined.

    An identifier is defined for the second time. Functions/procedures names, global variables, parameters names and local variables must be all different.

12. Incorrect number of arguments.

    A call to a function or a procedure does not use the same number of arguments as required by the definition of the function or theprocedure.

13. A field with a constant name is not allowed.

    You are using a constant name where a variable is expected.

14. The identifier is neither a function nor a built-in.

    An undefined function is used. The spelling should be checked.

15. Only a variable can be passed by address.

    A value is passed to a built-in procedure or function which expects a variable passed by reference.

16. The local variable is already defined.

    There is a conflict between a local variable and either a global variable, a constant name or a function or procedure name.

17. Local constants not supported.

    This feature is not yet implemented although the parser understands it. Note that the feature is not explained in this reference manual.

18. Return statements must have an expression in functions.

    A *return* instruction is used inside a function with no value. Note that the returned value must have the same type as the one specified in the head of the function.

19. Return statements cannot have an expression in procedures.

    A value is specified after the *return* instruction inside a procedure. This value is not allowed in Voyager 2.

20. Return statements in the body of the program cannot have an expression !

    A value is specified after the return instruction inside the main body (the part between the begin and end keywords). This value is not allowed.

21. "continue" statement not allowed here.

    The *continue* instruction is only allowed inside *while*, *repeat* and *for* instructions.

22. "break" statement not allowed here.

    The *break* instruction is only allowed inside *while*, *repeat* and *for* instructions.

23. The first character of identifiers cannot be '_'.

    The '_' character in first position of names is used for reserved keywords.

24. Internal Error.

    The compiler has reached a dangerous state. Please warn the DB-MAIN team.

25. Not enough memory.

Not enough memory to allocate dynamic objects. Some applications should be closed or Windows restarted.

26. Unterminated string.

    The string is not closed. A double quote should probably be added at the end.

27. The word is a reserved keyword.

    The spelling of the identifier should be changed in order to avoid conflicts.

28. Procedure expected in a call.

    A function is used where a procedure is expected.

29. Sub-process cannot be executed.

    The compiler cannot run a sub-process. The ".oxo" file has not been produced. Too many applications are running, not enough memory is available or the process was not found. Its name is comp_v1.dll.

30. A */ is probably missing

    A comment is not terminated or is too long.

31. A *) is probably missing

    An explain clause is not terminated or is too long.

32. A literal string is expected here.

    The expression must be a string (ie. "...").

33. Fields are not allowed for iterator variables in loops.

    Expressions of the form *variable.field* are not allowed in the loop statements (for-in-do).

    Let us note that the semantics of Voyager 2 prevents the compiler to trap some errors. The main reason is that Voyager 2 is weakly typed and the type verification is sometimes impossible at compile time.The VAM will stop them at execution time.

    For instance: the compiler will compile the following program without any error or warning message:

```
file: F;
begin
  F:="Hello"+(5++'i');
  print(F);
end
```

34. The identifier does not denote a "program". It should be defined in a use clause.

    The program uses an identifier in a "use" clause that has not been declared as a library in a previous "use" clause.

35. Include directive skipped. File cannot be opened.

    The filename specified in an *include* directive does not denote a file or this file cannot be opened.

Appendix D

Error Messages during Runtime

The VAM (Voyager Abstract Machine) is able to trap most of the problems that can occur during the execution. Each time it is possible, an error message is displayed indicating the cause of the problem. Unfortunately, every possible error cannot be trapped. This means that wrong voyager programs may mislead the VAM, possibly crash DB-MAIN.

Here are the various error messages that can pop-up:

1. Another type is expected.

   The type of an operand does not match the expected type of an operation.

2. Internal Error: please stop here.

   This error should not appear. Please report it to the DB-MAIN development team with all available information.

3. You are using an invalid field.

   The field used is not valid for this type of object. Example: "print(dto.identifier);" where *dto* is a *data_object*.

4. No enough memory for allocation.

   No more memory is available for dynamic data. Some applications can be closed or Windows restarted.

5. A list is expected here.

   A non-list value is passed to an operation that expects a list.

6. No program loaded.

   This error is obsolete.

7. Impossible to open file.

   The ".oxo file" cannot be openend. Is it compiled?

8. Illegal Instruction.

   The ".oxo" file is probably corrupted. If it is the case, then it should be recompiled. If this does not work, then the memory should be corrupted, Windows should be restarted. If this fails again, then the DB-MAIN development team should be warned.

9. The instruction is not defined so far.

   This message should never appear. The DB-MAIN development team should be warned.

10. Cell must be active.

An invalid cell is referenced. This message will occur during execution of the following program:

```
cursor: c; ...; kill(c); print(get(c));  ←  error 10.
```

11. The list cannot be empty.

    The expected argument should be a list with at least one active cell.

12. The file is corrupted.

    The ".oxo" file is corrupted. The program should be compiled again.

13. Error while reading the file.

    The ".oxo" file is corrupted. The program should be compiled again. This may occur when old versions of ".oxo" files are executed with a more recent version of DB-MAIN.

14. Argument of SetPrintList too large for the buffer.

    Arguments of the SetPrintList are stored in static strings and, therefore, their size is limited. Some characters should be removed.

15. This type cannot be read/written.

    Data of some types cannot be printed or read. For instance, values of type list or reference to objects like data_object cannot be printed.

16. Invalid field in Create operation.

    An invalid field is specified in the create instruction.

17. A mandatory field is lacking in Create operation.

    All the mandatory fields must be present in a *create* operation. Chapter 12 lists all the required fields for every type of object type.

18. Integrity rules are not checked.

    A *create* instruction violates an integrity rule during execution. Chapter 12 gives for more details about the integrity rules.

19. The buffer of the lexical analyzer is to small.

    The lexical analyser tries to parse a text which is too large. MAX_LEX_BUFFER is the size of the buffer of that lexical analyser. This size should not be exceeded.

20. Several choices are identical in MakeChoice.

    Several arguments of the MakeChoice statement are identical. It is probably a mistyping error.

21. A semi-formal field in the text is corrupted.

    The syntax of a textual property is invalid.

22. The dynamic property does not exist.

    The dynamic property is not referenced as an instance of the specified *meta_property* object type.

23. Remove is not allowed for this type.

    The *remove* instruction must be used to remove objects of the repository. The type of the value in parameter is invalid.

24. Impossible to start the AsbtractMachine with this program.

    A wrong name or path of a Voyager program has been specified. Other causes can be a damaged file, or a problem with dynamic ressources (file handle/memory/...).

25. Bad lambda expression.

    The *lambda* expression is invalid. The program problaby tries to use a function that does not exist or that is mispelled.

26. Bad code in blackbox.

    The code passed in parameter to the blackbox is invalid.

Appendix E

Frequently Asked Questions

E.1 Environment Relation Questions

E.1.1 How do I compile a program?

Once you have saved your program in a file (with your favorite text editor), say "prog.v2", you can compile it by using the Voyager 2 compiler.

The compiler is a 32-bits application that can be executed from the DOS prompt. If the program is correct, the compiler produces a file called "prog.oxo". This file can then be loaded in the DB-MAIN tool. Otherwise, the compiler stops, prints an error or warning message, and waits for a key tp be pressed on the keyboard. The effect of the entered character is:

c: continue compiling and do not stop any more.

s: stop now.

other characters: continue and stop again at next error or warning.

More options are allowed on the command line of the compiler. They are described hereafter:

syntax:

```
comp_v2 <option>*
```

where *option* can be:

filename: any string with no space character and with the first character different from '-', or any string enclosed in double quotes, can be a filename, according to the syntax of the operating system. The first filename found in the list is the name of the input file. The second filename is the output file. If these filenames do not have an extension, the compiler automatically adds the ".v2" extension to the first one and the ".oxo" extension to the second one. If the output file is not specified then the name is the same as the input file, but with the ".oxo" extension.

-date: prints the version of the compiler and stops immediately.

If the filename is missing in the list of options, the compiler asks for it during execution.

E.1.2 Question How do I write efficient programs ?

Voyager 2 was not built to be particularly efficient but to write programs rapidly and easily. For this reason, it is bad idea to try to write a number crunching program in Voyager 2. But there are some recipes to improve program performances:

- Lists should be avoided in the print(x) instructions:
  ```
  print(x); print(y);
  ```
  is better than
  ```
  print([x,y]);.
  ```
- Requests should be expanded rather than browsed using an intermediate list.

  For instance, the program at right runs faster than the program at left:

```
l:={request};
for x in l do {
.
.
.
}
```

```
for x in {request} do {
.
.
.
}
```

- Built-in instructions should be prefered when possible: prefer a *for-in-do* to a *while* instruction.
- In the *for-in-do* instruction, when the list looks like [a..z], the z expression is evaluated at each loop. If this expression is quite complex, it is better to evaluate it before the loop and use a variable to hold the result.

E.1.3 I cannot close the console ! Why?

The console is locked until the Voyager 2 program is finished.

E.1.4 When I load program, DB-MAIN tells me that the version of the program is too old.

Although DB-MAIN is backward compatible with all the versions of the language, the binary format may change from one version to another. When this message appears, the ".v2" file should be compiled again with the new compiler.

E.1.5 Why does the compiler find errors in my program although it was working fine with older versions?

It probably means that some identifiers used as variable/constant/function name are now reserved keywords in the new version. It is especially true for "dot-expression" (*variable.expression*). In older versions of Vayager 2, the compiler was able to distinct a variable declared as *integer: sem* from a field used in a *dot-expression* like *sch.sem*. Because right-hande-side expression of the "." operator may now be any expression, the compiler is no more able to distinguish them. Renaming variables should solve the problem.

E.2 Language Specific Questions

E.2.1 In a predicative query, DB-MAIN tells me that there is an invalid assignment. Why?

Probably that a sub-type is used where a super-type is expected. For instance, the following query is invalid:

```
ENTITY_TYPE[ent]{@SCH_DATA:[sch] with GetType(ent)=ENTITY_TYPE}
```

Although the list returned by the query is be composed of entity types only, the *ent* variable used as an iterator in the generated code to find all the instances of data objects linked to the sch schema should also be temporarily used to reference rel-types and attributes. So, it is possible that the VAM tries to assign an attribute to the *ent* variable that should be defined as *entity_type*. A correct query is:

```
DATA_OBJECT[dto]{@SCH_DATA:[sch] with GetType(dto)=ENTITY_TYPE}
```

E.2.2 Is there a nil value like in Pascal?

No, because there are no pointers in Voyager 2. However, special values denoted by *void* can be used for references. To obtain the *void* value of the entity_type, the function *Void(ENTITY_TYPE)* can be used. This function cannot be called for other types like *integer* or *char.*

E.2.3 Why is my request looping?

It is prohibited to modify the value of a variable used as the iterator in a query. For instance, the following program is wrong because the variable *dto* cannot be modified in the body of the *for-in-do* instruction:

```
data_object: dto;
schema: sch;
begin
  sch:=GetCurrentSchema();
  for dto in DATA_OBJECT[dto]{@SCH_DATA:[sch]} do {
    dto:=Void(DATA_OBJECT);
  }
end
```

E.2.4 How can I empty a list L?

```
L:=[1,2]; L:=[];
```

The last instruction empties the list L.

E.2.5 How can I test if a list is empty ?

There are at least two ways to proceed. The first one is to check the equality of the candidate list with the empty list:

```
if mylist=[] then ...
```

The second solution is to test the length of the list:

```
if Length(mylist)=0 then ...
```

Appendix F

Regular Expressions

A *regular expression* is a pattern description using a "meta" language. The characters that form regular expressions are:

.	Matches any single character.
*****	Matches 0 or more occurrences of the preceding expression.
+	Matches 1 or more occurrences of the preceding expression.
[...]	Matches any character within the brackets.
?	Matches 0 or 1 occurrence of the previous expression.
"..."	Matches exactly the text enclosed between quotes.
x..y	Is a notation for a range of characters, e.g., [0..4] means [0,1,2,3,4].
\ t,\ n,\x	Denotes respectively the tabulation, the newline and the x character when this last one is already used by the regular expression language ([].*+...).

For instance [a..zA.. Z][a..zA.. Z0..9]* denotes the syntax of identifiers in Pascal and [0..9]+[.[0..9]+]? denotes the syntax of real numbers (12,012,19.021,...).

BIBLIOGRAPHY

[1] A. Aho, R. Sethi, and J. Ullman. *Compilateurs. Principes, techniques et outils*. InterEditions, 1989.

[2] A. Church. *The calculi of lambda conversion*. Princeton University Press, 1941.

[3] V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. *Description du méta-schéma de l'atelier logiciel DB-MAIN version 1.0*. Technical report, FUNDP, 1995.

[4] A. J. Field and P. G. Harrison. *Functional Programming*. International computer science series. Addison-Wesley, 1989.

[5] K. Jensen and N. Wirth. *Pascal. Manuel de l'utilisateur.* Eyrolles, 1978.

[6] D. E. Knuth. *The TeXbook*. Addison-Wesley, 1990.

[7] D. Roland. MDL: Programmer's guide. Technical report, FUNDP, 2000.

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

[9] R. Wilhelm and D. Maurer. *Les compilateurs: théorie, construction, génération*. Masson, 1994.

INDEX