The I-GET User Interface Programming Language: User's Guide

Anthony Savidis, January 2004

Abstract

The I-GET User Interface Programming language is a fourth (4^{th}) generation programming language which provides declarative constructs for dialogue programming like: agent classes with preconditions, variable monitors, and unidirectional constraints. It supports type safe call-backs for interaction objects, type safe handlers implementing Event Response Systems (ERS), and a C-based programming kernel with dynamic memory allocation. It supports engagement of arbitrary pointer expressions within declarative constructs, implementing automatic run-time update detection, while it offers a powerful method for bridging C++ code with code written in the I-GET language. Finally, it offers a model for creating abstract interaction objects, together with a method for defining schemes for the polymorphic mapping to alternative physical interaction objects.

Keywords

User-Interface Development. User Interface Management Systems. Programming Languages.

I-GET USER INTERFACE MANAGEMENT SYSTEM

Programming Language Manual

Anthony Savidis,
HCI Laboratory,
ICS-FORTH,
as@ics.forth.gr

Table of Contents

List of figures

Chapter 1

1. Lexical conventions of the language

A program written in the I-GET language is read by the I-GET compiler, which then generates C++ code. Compilers are typically split into four layers: (i) lexical analysis, (ii) syntactic analysis, (iii) semantic analysis, and (iv) code generation. Lexical analysis is analogous to the regulations governing the construction of legal linguistic tokens in verbal languages. In this Section, we will present all the various categories of tokens which are allowed when writing I-GET programs.

1.1 Keywords

The I-GET language has 62 keywords from which: 19 are borrowed from C / C++, 9 concern data types, 8 concern constructs commonly supported in other programming languages, and 26 are new keywords due to the new types of constructs introduced in the I-GET language.

agent
bool
break
case
channel
char
constructor
continue
create
daid
default
destroy
destructor
else
enum
eventhandler
export
in
inputevent
instantiation
int
lexical
local
longint
longreal
longword
me
message
method
myagent
new
nil
noexport
objectid
private
public
real
ref
release
return
scheme
sharedid
shcreate
shdestroy
shread
shupdate
string
struct
terminate
type
virtual
void

extern	**of**	**while**
for	**out**	**word**
header	**outputevent**	
if	**parent**	

1.2 Comments

C and C++ styles supported for comments.

Comments in the I-GET language can be defined following the C or C++ styles. In case the old C style is followed, nested comments are also supported.

```
/* this is a comment */ /* and another */
// this is a C++ style comment
/* this is a /* nested comment */ */
/* a comment may span across
multiple lines of text  */
```

1.3 Identifiers

Identifiers are applied in programming languages for construct classes, where instance resolution is addressed via user-supplied names. Identifiers in the I-GET language must start with a letter, followed by any number of letters combined with decimal digits or the underscore character. The I-GET compiler treats identifiers in a *case sensitive* manner.

```
pid // ok
my_var_12 // ok
OJECT_0_1 // ok
12_a_object // error, starts with a digit
a__2 // ok
_a_name // error, starts with an underscore
```

1.4 Literals

Three categories of literals are supported in the I-GET language: string literals, number literals and character literals.

String literals are provided as a sequence of any number of characters surrounded by double quotes; to include double quotes within a string write the sequence `\"`. All escape characters supported in C / C++ may be also included within a string literal, and they will be replaced accordingly. If you supply the special null character `\0` within a string literal, the result is that all characters following `\0` will be discarded.

```
"Hello"
"Hello\n\tworld"
"Hello,\"world\""
""
"this is included\0but this not"
```

Character literals are single characters enclosed in single quotes. Escape sequences, as those used within string literals, are also allowed for character literals.

```
'a' '\0'
'\n' '$'
```

Number literals are numeric constants. Real and integer numeric constants are supported. The examples below show the specific data type of each numeric literal defined.:

```
3.141628 // real, single precision
```

```
3.141628L // real, double precision

1759 // signed integer

1759W // unsigned integer

1345638L // long signed integer

1345638LW // long unsigned integer - WL not legal
```

1.5 White space between tokens

White space in the I-GET language is any sequence of: tab stops, carriage return (ASCII CR), line feed (ASCII LF), and spaces. Language tokens are: identifiers, literals and all kinds of operators. An I-GET program is composed of tokens; white space can be freely used between tokens.

1.6 Important separators

Group and item separators.

There are two key separators (also called punctuators) in the I-GET language: group separator `[ ]` (opening and closing square brackets), and item separator `;` (semicolon). Typical use of the group separator is for defining either a block of statements, or the body of a construct class in the I-GET language. Typical use of the item separator is for defining statements, declarations or function prototypes. People familiar with the C / C++ group separator `{ }` (braces) will have to use square brackets in the I-GET language, in those similar cases where braces are needed in C / C++.

1.7 Operators and precedence

There are 11 precedence levels, some of which contain only one operator. Operators in the same level have equal precedence with each other. In case some operators are mentioned twice in the table below, the first appearence denotes a unary role, while the second a binary role. Each category has an associativity rule: left to right, or right to left. The precedence of each operator level in the table below is indicated by its order in the table, having the first category being the one with the highest precendence.

Operators	Associativity
() [] -> . ::	Left to right
! ++ -- + - * &	Right to left
* / %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
^	Left to right
&&	Left to right
||	Left to right
= *= /= -= += %=	Right to left
,	Left to right

1.8 Preprocessor support

The I-GET compiler does not include any built-in preprocessor, however, it provides compiler flags which allow a command-line preprocessor to be used, prior to I-GET compilation. The first compile flag is **`-nocpp`**, which indicates that a preprocessor will not be used (note that "cpp" stands for code preprocessor, it is not necessary to be only the well known `cpp` C-preprocessor). The default action of the I-GET compiler is to use the standard C-preprocessor.

The second compile flag is **`-namecpp <name>`**, through which the executable name of the preprocessor to be used is supplied; if no name is specified, the default is `cpp` (the original C-peprocessor name).

The third flag is **`-pathcpp <path>`**, through which the path in which the preprocessor executable can be found is provided; the I-GET compiler will concatenate the path name with the executable name, and will pass the input file (i.e. dialogue file) for compilation; intermediate files are handled by I-GET itself. If no path is defined, the default is `/lib` (the path for `cpp` in UNIX installations).

Only the C-preprocessor special marks for line numbers and file names are recognised directly.

There is a small problem when trying to use an external preprocessor with language compilers. Usually preprocessors "inline" all included files together into a large stream. In order to enable compilers "know" the original input file, while compiling the whole stream, special line and file marks are put on the resulting stream. It is the responsibility of the particular compiler to recognise such marks and use accordingly. The I-GET compiler has been developed to directly recognise such marks for the `cpp` C-preprocessor. It will not recognise other types of marks from other precprocessors, and those will be normally considered as compile errors. What you can do in this case is to firstly disable generation of such marks, if the preprocessor provides flags to do that (most commonly used preprocessors support that). Secondly,

The I-GET compiler offers special constructs to preserve line and file information, when preprocessors other than CPP are to be utilised.

since line and file information will be normally lost, you can use two very simple, special purpose facilities offered in the I-GET language, to set line number or file name directly within a dialogue specification segment. When the I-GET compiler parses these items, it will continue as if the line number or the file name are those provided by the developer. An example which follows will make clear how these two constructs can be used to avoid loosing line and file information, when C-preprocessor is not used.

```
// file1.txt
<include somehow file2.txt>
@"file1.txt"
@4

// file2.txt
@"file2.txt"
@3
```

The first line of `file1.txt` contains a comment, while the second a preprocessor directive to include a file (in a form the preprocessor used understands). The third line uses the special I-GET construct, `@"file1.txt"`, to set the file name directly. Even when the included file is expanded in-line, when the I-GET compiler parses this line, it will set the file name to `file1.txt`. At line 4, there is a similar construct setting line number to 4 - `@4`. Hence, the I-GET compiler will consider that at this point, the line number is 4. The result is that, even though a file has been expanded in-line, the compiler will still "see" the correct file and line number, due to the use of these constructs. The approach is similar for `file2.txt`.

I-GET directly understands line and file directives from standard C-preprocessor.

The I-GET compiler is capable of understanding line information generated by the standard C-preprocessor, which is in the form **# <line> "<file>"**, so in case you employ a standard C-preprocessor you don't need to worry about preserving line and file information due to expanded includes.

Compile flags to inform the I-GET compile how to construct a correct command-line call of the preprocessor to be used.

The I-GET compiler will call prior to compilation the preprocessor you specify via the flags **-pathcpp** and **-namecpp**. However, calling a preprocessor that is not known usualy requires passing more flags, while the command line calling syntax may not be standard across preprocessors. The I-GET compiler supplies the following compile flags which enable to provide enough information on the calling convention as well as all the necessary flags for command line preprocessors:

- **-cppoutsw <switch>**, which defines the output redirection switch through which the output of the preprocessing stage can be redirected to a file. Examples of such switches commonly used are `-o` and >. Hence, in that case, it suffices to specify `-cppoutsw -o` or `-cppoutsw >`.

- **-cppflag <flag>**, which defines an argument to be directly supplied in the command line call of the preprocessor. Multiple such switches can be provided to the I-GET compiler, in order to construct multiple switches and arguments for the preprocessor call. No spaces will be placed in-between such successive arguments. For instance, if you supply the following arguments to the I-GET compiler: `-cppflag -I -cppflag /usr/include`, the resulting string constructed for the preprocessor call is `-I/usr/include`. To add spaces between the various parts of the preprocessor arguments use the following I-GET compile flag:

- **-flagspace**, which simple adds a single space at the end of the current argument string constructed through the `-cppflag` I-GET compile op-

tion. Hence, to have multiple include directives for the preprocessor call, you could add in the previous example the following: `-flagspace -cppflag -I -cppflag /home/include`. Finally, the I-GET compiler needs to now the "call pattern" of the preprocessor to be used. The call patterns is the syntax which defines how the various preprocessing flags, the input file, the output redirection switch and the output file are placed in the command line, in order to form a correct preprocessor call. For this purpose, the I-GET compiler offers the following flag:

- **`-cppcall <call pattern>`**. The call pattern is a string composed of the following characters, representing argument classes: (i) **F**, representing the argument string constructed via the use of `-cppflag` and `-flagspace`; (ii) I, representing the input file name (file is automatically set by I-GET); (iii) O, representing the output file name (file is automatically set by I-GET); (iv) S, representing the output redirection switch, supplied via `-cppoutsw`; and (v) **-**, representing a space. No other characters are alowed in a call pattern. All argument class characters (i.e. F, I, O, S) should appear once in the call pattern, and only once. The call pattern should not begin or end with a **-**, while as many "spaces" (i.e. -, zero also allowed) maybe put in-between argument class characters.

An example of how to use the preprocessor-specific compile options.

Examples of how to use the above compile options, in order to inform the I-GET compiler on the way to construct a legal preprocessor, will now be discussed. Assume you want to use the standard C-preprocessor under UNIX. Also, assume you want to specify some include directories `/dir1` and `/dir2`; the I-GET compiler supports already include directives, which, however, in order to be used, must be utilised in accordance to a standard C-preprocessor (i.e. the I-GET simply passes these flags to the preprocessor). We will use directly the prerocessor, and ignore all I-GET supported flags; the sequence of compile options for using the C-preprocessor as above is:

```
-pathcpp /lib

-namecpp cpp

-cppoutsw -o

-cppflag -I/dir1

-flagspace

-cppflag -I/dir2

-cppcall F-I-S-O
```

The argument string constructed is '`-I/dir1 -I/dir2`' (i.e. **F** argument). The output redirection switch is `-o` (i.e. **S** argument). Hence, the I-GET compiler will construct the preprocessor call as: '`/lib/cpp -I/dir1 -I/dir2 <input file> -o <output file>`', which is the correct calling convention, with all the required preprocessor options.

An important notice: you will need standard CPP to compile I-GET libraries written in the I-GET language.

The I-GET software distribution includes various libraries which have been implemented in I-GET code itself. In these libraries, there are some macros within the code complying to the C-preprocessor, while the compile options for those files, directly use the I-GET compiler include directives, which require an external C-preprocessor in order to work. Hence, for the utilisation of the supplied libraries you will necessarily employ a standard C-preprocessor (please be careful to use only a preprocessor, since compiler vendors usually supply preprocessors which perform, apart from preprocessing, some language-specific lexical checking, which may conflict with I-GET language lexical conventions - standard UNIX CPP and GNU CPP do not present such problems).

Chapter 2

2. Data types and variables

Variables have and identity, storage area, type and value.

All items which can be manipulated in the I-GET language, have an *identity*, a *storage area*, a *type* and a *value*. In most programming languages such items are commonly mentioned as "objects", or "variables". In the I-GET language, we will mostly use the term "variable", since the term "object" is primarily associated to "interaction objects" (there are built-in categories of interaction objects in the I-GET language). Variables are used in any place where typed items are required (dialogue variables, attributes of interaction objects, arguments for parameterized constructs, etc).

The I-GET language is a strongly typed language.

The I-GET language is a *strongly typed language*; in other words, all items with a storage (i.e. place-holders) are declared prior to use, in association with a specific type, while all expressions supported in the I-GET language are based on typed items, the type of which is clearly known and well defined in compile time. This is similar to typical programming languages like C / C++ / Pascal / Java, while it is opposite to commonly used scripting languages such as PERL or Python (where dynamic typing is supported). Many similarities with C will be apparent in the I-GET language, with respect to the data-types supported.

2.1 Built-in data types

There are 12 built-in data types. 9 of them correspond to simple data types common in programming languages, while 3 of them refer to special object categories of the I-GET language.

Data types	Explanation
`int`	C++ `int`
`longint`	C++ `long int`
`word`	C++ `unsigned int`
`longword`	C++ `unsigned long`
`real`	C++ `float`
`longreal`	C++ `double`
`bool`	I-GET boolean class
`string`	IGET string class
`char`	C++ `char`
`sharedid`	Reference identifier for shared objects
`daid`	Reference identifier applicable to instances of any agent class
`objectid`	Reference identifier applicable to instances of any lexical object class

2.2 Variable declarations

All variables have identifiers, supplied during variable declaration. An identifier cannot be legally used before (in the source code) its definition as part of a variable declaration. Variables are manipulated within a program through their associated identifier.

2.2.1 Introduction to the declaration syntax and categories of variables

The declaration of variables in the I-GET language is very simple, and is based on the C declaration syntax, though highly simplified. A simple declaration has the following form:

TypeName PtrPrefix VarIdentifier ArrayPostfix;

```
int a_number;

int *a_number_ptr;

string* a_string_ptr_array[10];

objectid an_objid_array[5][20];

int i,*j, k[20], q[10][20]; // multiple vars allowed.
```

Four categories of variables in the I-GET language.

In the above examples only built-in type names are used, however, user-defined type names are normally allowed in the same manner. The I-GET language supports four categories of variables: (i) conventional variables, gaining as much storage as their respective type requires; (ii) pointer variables; (iii) array variables; and (iv) arrays of pointers. **PtrPrefix** can be any number of *stars* * (zero allowed), to denote the indirection depth, while **ArrayPostfix** can be any sequence of *[integer number]* (zero allowed) to denote the array dimensions and respective sizes. It should be noted that in the case of arrays, arithmetic expressions are not allowed between square brackets. Hence, all of `[n+10]`, `[f(i,j)]`, `[10+20]`, `[10-1]` are illegal in the I-GET language. Integer literals (i.e. constants) must be directly provided (even constant expressions such as `100-10+5` are illegal).

Expressions are not allowed to be given as array dimensions; integer numbers (i.e. literals) must be directly supplied.

2.2.2 Scope categories for variables

The scope of an identifier, which corresponds to a variable, is that part of the program in which the identifier can be used to access its respective variable. There are four categories of scope for data variables in the I-GET language, which correspond to four, out of six, scope categories supported in C++. The scope categories in the I-GET language depend on where and how variable identifiers are declared.

- *Block*. The scope of an identifier with block scope (also called local scope) starts at its declaration point and ends at the end of the block containing the declaration (i.e. the enclosing block). Formal arguments of functions have a block scope, limited to the scope of the block that defines the function.

- *Prototypes for parameterized constructs*. Examples of such constructs are functions (i.e. function prototypes), agents (agent prototypes, look like function prototypes), input events (look like structure declarations), and output events (slightly different from structure declarations). The scope of parameters defined in such constructs ends at the end of the prototype definition.

- *File*. Identifiers of file scope are known as global variables, and are declared outside of blocks or any other type of construct; their scope starts at the point of declaration and ends at the end of the source file.

- *Class*. In the I-GET language there are four built-in class categories: (i) agent classes; (ii) virtual interaction object classes; (iii) lexical interaction object classes; and (iv) event handler classes (those are embedded within agent classes always). There are no facilities for creating general purpose OOP classes, but only classes belonging to one of the above four categories. Class scope for variables in the I-GET language has more complex

regulations and is presented in detail when discussing those class categories in their respective Sections of documentation, in this Volume.

The I-GET language introduces the notion of scope spaces.

Apart from variables, there are various other categories of constructs in the I-GET language, for which different scope categories apply. Those categories of constructs in which name collision among identifiers will generate compile errors, are said to belong to the same scope space. A detailed discussion on the issue of scope spaces, as well as the identifier / name collision rules in the I-GET language is provided under Section 14; it should be noted that in C++ there is a single scope space, in which all types of identifiers declared which happen to belong to the same scope category will collide (e.g. it is a compile error to declare a global variable and define a structure type with the same name, both at file scope).

External and local storage qualification.

Global variables may be qualified with a storage qualification which defines whether declared variables are to be stored in local storage, and be hidden to the rest of the modules (`local` qualification, corresponds to `static` in C / C++), or if declared variables are explicitly referenced from global data storage (`extern` qualification, as in C / C++). Relevant examples follow below.

```
extern string user_id, host_machine, passwd;
local int login_users;

// semantic error, previous was extern.
//
extern int login_users;
int windows_opened;

// ok, set as local now.
//
local int windows_opened;
objectid* object_insts;

// ok, extern is default.
//
extern objectid* object_insts;
```

2.3 User-defined data types

Three categories of user-defined types: enumerated, structures and type synonyms.

There are three categories of user-defined data types allowed in the I-GET language: (i) enumerated types; (ii) structure types; and (iii) type synonyms. Data types can be defined either in file scope (globally) or within class scope (i.e. within the definition body of I-GET supported class categories). In the former case a data type may be used at any place following its definition point in the source file, while in the latter case there are more complex scope rules which are discussed in the context of the various I-GET class categories.

2.3.1 Enumerated types

Enumerated type definition in the I-GET language is based on the C style; however, from the type checking point of view, the I-GET approach is more close to Pascal, since enumerated variables can be assigned to- or compared with- only enumerated variables of the same type (as opposed to C which treats enumerated variables as integers). Examples of enumerated type definition follow, and variable declarations follow below.

```
enum Days=[Sun, Mon, Tue, Wed, Thu, Fri, Sat];
enum ToggleState=[ToggleOn,ToggleOff];
enum ShapeStyle=[Oval, Rectangle, RoundedRectangle];
enum State=[On, Off];

// semantic error, On & Off redefined.
//
enum Bool=[On, Off];

// Syntax error, assigning integers  as in C not
// supported.
//
enum Graphic=[Circle=1, Rect=2];

// semantic error, redefinition.
//
enum State=[True, False];
```

```
Graphic element, *element_ptr, element_arr[10];

Bool one_flag, many_flags[5];
```

2.3.2 Structure types

The definition of structure types has a syntax similar to C++, except that it does not support variable declarations together with the structure type definition. Forward structure definitions are also supported, in case that a pointer to a structure type is needed before the structure is defined. Structure types cannot be defined within other structures (however, fields of already defined structure types are allowed). Examples of defining structure types as well as declaring variables of that types follows below.

```
struct Point [ int x; int y ];

struct Employee [

       string name, surname;

] my_emp; // syntax error, var declaration not sup-
          ported with structure definition.

struct BinaryTree; // that's a fwd definition.

BinaryTree* root; // ok, just a pointer.

BinaryTree can_I_declare_it; // semantic error, incom-
                             plete type.

Struct BinaryTree [

       string data;

       BinaryTree* left;

       BinaryTree* right;

];

BinaryTree  can_I_declare_that; // ok now, type com-
                                pletely known.
```

The fields of a structure type can be only of variable category; functions, or object instances are not allowed, however, reference variables to I-GET object classes are allowed.

2.3.3 Type synonyms

Type synonyms are a nice way to produce data types on the basis of patterns of variables. The declaration syntax is similar to that of C. Type synonyms help break-down complex variable types into a sequence of more readable data-types. In fact, in he I-GET language this is the only way to do it, since the syntax is highly simplified with respect to C / C++ declaration syntax. The definition syntax, as well as examples of defining type synonyms and declaring corresponding variables follows below.

type SingleVariableDecl ;

```
type string string_arr_10[10];

type int Int;

type Int* arr10_ptrint[10];

type arr10_ptrint* ptr_arr10_ptrint;

type ptr_arr10_ptrint arr20_ptr_arr10_ptrint[20];

arr20_ptr_arr10_ptrint my_var;

type string String1, String2; // syntax error, single
                              var declaration ex-
                              pected.

type extern int ext_int; // error, storage qualifiers
                         not allowed in type synonym
                         definitions.
```

2.4 Initializers for variables

Initializers set the initial value that is stored in a variable, are are provided together with the declaration. If a variable is not explicitly initialized via an initializer, then it will be initialized by default in the following manner:

Data-type	Default value
All numeric data types	0
`string`	""
All pointer types	`nil`
`char`	' ' (space)
`bool`	`false`
`objectid, sharedid, daid`	`nil`

Three types of initializers in the I-GET language.

There are three types of initializers: (i) initializers for simple types (all built-in types, and enumerated types), requiring a single expression; (ii) initializers for arrays, requiring multiple expressions of the same type; and (iii) initializers for structures, requiring multiple expressions, possibly of different types.

2.4.1 Initializers for simple variables

All types of expressions, including functions calls, can be used in initializers, as far as their type matches the variable being initialized (expresisons and type matching rules are discussed within Section 4 of this Volume).

```
int a=16,*b=&a, c=-17;
string file="a1.txt", path=GetPath(file);
bool flag=true, *flag_ptr=&flag, new_flag=*flag_ptr;
char get_from_string=file[2], my_c='\0';
real r1=r2, r2; // error, r2 not declared yet.
bool b1=false, b2=true, b3=b1 && b2;
```

2.4.2 Initializers for array variables

Initializers for arrays provide the values for all elements of the arrays. Type matching is also performed to assert that value match correctly with the array element type. Undefined array dimensions are not allowed in the I-GET language, hence, you have to explicitly supply all dimensions. Arrays are initialized by providing initializers for all elements of an array, in increasing array subscript order, separated with commas and enclosed in square brackets. In case ofmulti-dimensional arrays of type *D1*...*Dn,* the array is considered to be a *[D1]* array of *[D2][D3]...[Dn]* elements. Then, you have to supply an initializer list of *n* initializers, all of the reduced array type, which can be done by applying the reduction rule recursively, until a single-dimensional array is met. Normally, many opening / closing lists will be written down, depending on the total number of dimensions. Examples for initializing arrays are given below.

```
int a[5]=[1,2,3,-4,23*56];
int b[4]=[a[0], a[1]+34, a[2]*a[3], readint()];
string A[2][2]=[ ["one","two], [ readstr(), "four"] ];
bool B[3]=[false, false, B[0]]; // error, B not known
                                   before declaration is
                                   completed.
real AA[1][1][1]=[ [ [ 3.1416 ] ] ];
```

```
real BB[2][1][1][1]=[ AA, AA ]; // ok, can assign ar-
                                   rays if they match.

real CC[1][1]=AA[0]; // ok, AA[0] is real[1][1]
```

2.4.3 Initializers for structure variables

Structure variables are initialized by providing a list of initializers for each field, in increasing field order, all separated with commas and enclosed in square brackets. Example for initializing structure variables follow below.

```
Point alpha = [13, 27];

struct Rect [

      word width;

      word height

];

struct MachineInfo [

        string Processor;

        word Mhz;

        word Mbytes;

        Rect resolution;

        word size_inch;

];




MachineInfo machine = [

        "Pentium",

        130,
```

```
                1300,
                [ 1020, 860],
                26
        ];
```

Chapter 3

3. Functions

Supported semantic features for functions in the I-GET language are similar to the Pascal language, though being more closely to the C syntax. The syntax in the I-GET language does not allow to define returned types which include an array postix (i.e. dimensions defined by sequences of `[]`), however, the use of type synonyms can be employed to define the return type of a function to be an array, thus avoiding this problem.

3.1 Function implementation

Functions are defined by means of a function header, which includes the returned type, the function name and the formal arguments, and an implementation block. Some functions doe not need to return a type, so their type is declared to be `void`. Also, some functions support an empty list `()` of formal arguments. We provide examples of function implementation below; we will provide also some sample implementation blocks, all including simple I-GET statements, since those are discussed in detail under Section 13 of this Volume.

```
int Inc (int* i) [ return ++*i; ]

longing Factorial (int n) [
        if (n==1) return n;
        return n*Factorial(n-1);
]
```

```
void Sort (string* items, int N) [
      /* implementation here */
]
Point* ReplicatePoly (Point* points, int N) [
       Point* replica;
       new(replica,N);
       int i;
       for (i=0; i<N; replic[i]=points[i], i++)
                ;
       return replica;
]
int CopyArr[10] (int arr[10]) [ // syntax error.
       return arr;
]
type int arr10[10]; // use a synonym for array type.
arr10 CopyArr (int arr[10]) [ // ok now.
       return arr;
]
```

The syntax does not allow to specify directly an array returned type, but this can be easily managed through the use of an auxiliary type synonym.

As it is evident from the above example, typed functions may return values through the use of `return <expr>;` statement; `void` functions may include a simple `return;` statement with no value specified. All sorts of user-defined types (i.e. structures, enumerated and type synonyms) can be used as the returned type of a function. The I-GET tool supports a run-time error detector, which automatically identifies the source point and the reason for a run-time error to occur. One typical error is to reach an exit point for a typed function which does not include a `return <expr>;` statement; since this may cause an erroneous behaviour to the caller, the I-GET tool will report this as a run-time warning.

3.2 Function prototypes

Argument identifiers are always required in function prototypes.

Variable lists of arguments as in C, and function name overloading as in C++ is not supported.

A function prototype allows calling a function prior to its use. This is a very good hiding implementation details, while exporting all that is needed to use a function: returned type (if any), name and formal arguments (if any). The scheme for defining function prototypes in the I-GET language is similar to C, with a small restriction: when specifying formal arguments, argument names must be explicitly supplied (these need not be the same names, as those used within the argument list of the function implementation). This has been decided to promote readability of function prototypes, assuming that descriptive identifiers are given to formal arguments. . Variable number of arguments is not supported in the I-GET language, neither function overloading. Examples of function prototypes follow below.

```
void CloseWindows (objectid* win_list, int N);

string UserLogingPsswd (string user_id);

daid DiallogBox (bool* ok, bool* no, bool* cancel);

void AskUserNamesFromApplication ();

struct UserInfo [ string id, passwd, access; ];

UserInfo* GetUsers ();

void CloseDown(void); // error, use () instead.

void FindWindow (string); // error, need arg name.
```

3.3 Storage qualification

Storage qualification is similar to variables. Functions by default gain the `extern` storage qualification, which allows access from any other source file in an interactive application. You can access a function implemented

within another file by declaring an `extern` qualified function prototype at any point in file scope, before the place in the source code where the function is to be called. To hide a function from other files use the `local` storage qualifier. There is a small difference with these two qualifiers. The extern qualifier may be only used with function prototypes, while the local qualifier may be used either to qualify an implementation or a prototype; however, extern linkage is the default for functions if no qualifier is supplied (either within an implementation or a prototype). Examples of storage qualification on functions follow below.

```
int F(); // extern is default.

extern int F(); // ok, same as before.

int F() [ /* implementation */ ] // ok, no conflict.

extern void G() [] // syntax error, extern not allowed
                   for implementations.

local void H() []; // ok

local void H(); // ok, storage class still matches.

extern void H(); // semantic error, conflict in storage
                 class from previous definitions.

void H(); // ok, local still considered.
```

3.4 Calling functions

The syntax for calling functions is very easy. A function is called with actual arguments placed in the same sequence as their matching formal arguements. The actual arguments are converted as if by initialization to the declared types of the formal arguments. A function (i.e. within its implementation block) may modify the values of its formal arguments, but this has no effect on the actual arguments in the calling context. To implement functions

which change the content of actual arguments, pointer types can be used as formal arguments (as in C); changing the content of data though the use of a pointer formal argument within a function, is not "re-done" after exiting this function at run-time. Examples of calling functions are provided below.

```
void Swap (int* a, int* b) [

     int temp=*a; *a=*b; *b=*a;

]

int i=10,j=20; Swap(&i,&j);

F( G( H(i), H(j) ) );

int foo() [return -1; ]

foo(); // ok, return value discarded.

foo(10); // semantic error, no arguments expected.

&foo; // semantic error, can't get address of a func-
       tion.

foo; // semantic error, 'foo' variable expected, since
      only variables are allowed with this syntax.
```

3.5 Built-in functions

Small set of built-in functions, since those can be easily expanded.

The I-GET language provides a small set of built-in functions. These function serve two purposes only: (i) terminal character-based standard I / O; and (ii) some type conversions of built-in numeric data types from / to `string` data type.

Basic terminal I/O functions for debugging / testing purposes.

The set of functions is very small due to the fact that in the I-GET tool it is very easy to implement I-GET versions of common C++ functions (such as mathematical functions). This is possible through the capability of I-GET for mixing with C++ code, which allows a function to be implemented with a function header (i..e. returned type, name and parameters) in I-GET, and an implementation body in C++. The terminal I / O functions are provided for

primarily debugging and testing purposes, since it is common to put messages and progress guards (by requesting terminal input), in order to control execution sequencing in programs. Type conversion functions allow all types of numeric types to be easily converted from / to `string` type.

	Function	Description
Terminal input functions for built-in data types.	`int readint();`	`int` read from std input.
	`longint readlongint();`	Same for `longint`
	`word readword();`	Same for `word`.
	`longword readlongword();`	Same for `longword`.
	`real readreal();`	Same for `real`.
	`longreal readlongreal();`	Same for `longreal`.
	`char readchar();`	Same for `char`.
	`string readstr();`	Same for `string`.
Conversion of `string` data type to built-in numeric data types.	`int strtoint(string);`	Reads `int` from `string`.
	`longint strtolongint(string);`	Same for `longint`.
	`word strtoword (string);`	Same for `word`.
	`longword strtolongword (string);`	Same for `longword`.
	`real strtoreal (string);`	Same for `real`.
	`longreal strtolongreal (string);`	Same for `longreal`.
Terminal output for string, and conversion of real numeric types to `string` with presentation format control.	`void printstr (string);`	`string` to std output.

	`string realtostr (`
	`  real number,`
	`  int intPart,`
	`  int fractPart`
	`);`
		`real` to `string`, enabling control of precision on integer and fractional part.

`string longrealtostr (`
`  longreal number,`
`  int intPart,`
`  int fractPart`
`);`
	Same as before, for `longreal` to `string`.

Chapter 4

4. Expressions

The I-GET language offers a large repertoire of expressions, which considerable expressiveness for dialogue implementation. Expressions are recursively composed of combinations of operators and expressions, while there is a specific set of primitive expressions serving as a starting point to the expression composition process. We will start with more primitive expression classes, and then move to expressions built by combining any other types of expressions.

4.1 Primary expressions

Primary expressions are the most primitive type of expressions. These are the main starting points for building expressions, and they are usually based directly on user-variables or built-in language constructs. They do not involve expression manipulation operators at this point, but "produce" expressions which may then play the role of arguments to particular expression-making operators. The syntax for primary expressions, as well as corresponding examples, follows below.

Primary expression:

Literal or

lexical (ID) :: ID . ID or

virtual :: ID . ID or

agent :: ID . ID or

Lvalue or

FunctionCall or

AgentInstantiation or

ObjectRef . parent or

ObjectRef . myagent or

ObjectRef

ID is a token for identifiers in the I-GET languages (i.e. a user supplied name). Here ID is used for toolkit qualification.

ObjRef:

{ Lvalue } or

{ Lvalue } ID or

{ me } or

{me } ID or

{ myagent }

The **Lvalue** grammar symbol indicates a storage locator; in other words an expression that designates a variable with a reserved storage in memory. They have been traditionally called *Lvalues*, *L* standing for left, meaning that an *Lvalue* could legally stand on the left (the receiving end) of an assignment statement. *Lvalues* are discussed in detail within the next Section (4.2).

The **ObjRef** grammar symbol denotes legal expressions representing instances of the various I-GET object classes (i.e. agent classes, virtual object classes and lexical object classes). The **me** and **myagent** constructs are similar to the this C++ construct; the **{Lvalue}ID** provides the way for accessing the physical instance of virtual object {Lvalue} for toolkit **ID**; please refer to virtual and lexical object classes documentation for more details. Object references are used to access members of object instances (e.g. functions and variables local to object classes). The second, third and four grammar rules, which have apparent similarities, concern access to enumerated constants for enumerated types which are defined as public locally within classes (e.g. lexical, virtual and agent classes). The rule requires an explicit specification

of the referring class, so that the enumerated constant can be correctly checked within the proper owner class.

FunctionCall:

ID (ExpressionList) or

AgentScopeRes ID (ExpressionList) or

ObjRef -> ID (ExpressionList)

AgentScopeRes:

:: or

ID ::

Functions can be either called directly by their name, if in the calling context the name is directly visible, or though an explicit context resolution. Context resolution means simply either an object reference (i.e. the **ObjRef->** prefix in the standard function call syntax), or agent scope resolution for locating a function within a higher-hierarchically agent or globally (to resolve name collision with a function that is "closer" in scope), when calling within hierarchies of agents (i.e. the **AgentScopeRes** prefix); please refer to the documentation on agent classes and agent hierarchies for more details.

Examples for syntactically correct primary expressions in the I-GET language.

```
my_var                  ::my_var                DBoxAgent::my_var
F();                    ::F();                  MsgAgent::G();
{button}.x              {button}.fg             {Vbutton}Xaw.fg
{button}.parent         {my_menu}.agent         {my_menu}
{my_menu}->H();         {my_agent}->F();        readstr()
my_arr[Pos("A")];       *my_ptr                 "hello,world"
'a'                     nil                     3.141628L
{vi_obj}Xaw->G();       {myagent}->Dpy();       {me}->F();
```

Examples for referring to enumerated constants of enu-

```
lexical(Rooms)::Menu::Xor
```

merated types defined within object classes

```
virtual::Command::Direct

agent::Dbox::ApplyIncluded
```

4.2 Lvalues

Grammar for Lvalues in the I-GET language.

Lvalue:

ID or

AgentScopeRes ID or

Lvalue [Expression] or

*** Lvalue** or

Lvalue . ID or

Lvalue -> ID or

(Lvalue) or

ObjRef . ID

Examples for Lvalues in the I-GET language, for all grammar rules.

```
my_var                  (*ptr_2_ptr)[n]

::my_var                arr2[10][2+arr[j]]

AnAgent::my_var         *arr_ptr[10]

arr[F(j)]               str_var.field

*ptr                    str_ptr->field

ptr[i+j]                {obj_ref}->local_member
```

4.3 Expressions from unary operators

*The indirection operator * can be applied only to Lvalues, while in C++ it may be applied to virtually any kind of expression.*

Unary operators are applied to a single *Lvalue* or expression. At this point you may have noticed that the grammar rules for operators and expressions are different from the corresponding in C++. The same holds also for unary expressions, where the most obvious difference is the absence (in the I-GET language) of the indirection operator (i.e. *) from unary expressions; the reason is that in the I-GET language the indirection operator can be applied only to *Lvalues* and not to any type of expression (as in C++). The grammar for unary expression and examples follow below.

Grammar rules for unary expressions.

UnaryExpr:

& Lvalue

+ Expression or

- Expression or

++ Lvalue or

Lvalue ++ or

-- Lvalue or

Lvalue – or

(Expression) or

! Expression

Examples of syntactically correct unary expressions.

```
&my_var // address of        +45.67
-32                          ++i
i++                          --j
j—                           (arr[j]-arr2[i])
! ptr // logical not         ! ( box_on || !flag )
! F(j)                       ( F(q) * H(p) )
```

4.4 Expressions from arithmetic operators

There are five binary arithmetic operations supported in the I-GET language: addition, multiplication, subtraction, division and modulo. The syntax as well as examples follow below.

Grammar for arithmetic expressions.

ArithmeticExpr:

Expression + Expression or

Expression - Expression or

Expression * Expression or

Expression / Expression or

Expression % Expression

Examples for arithmetic expressions.

```
(a+b)/c-d                       arr[j]*arr[j+1]/arr[j-1]
-(N % 8) + (N % 9)              j++ - ++j
a / (a-1) * b / (a-b)           N % 1 - N
{button}.x+{button}.width       {me}.width / letter_size
```

4.5 Expressions from boolean operators

There are three binary boolean operators supported in the I-GET language: *logical or*, *logical and*, and *logical xor*. The syntax for expresions resulting from these operators and examples follow below.

Grammar for boolean expresions.

BooleanExpr:

Expression || Expression or

Expression && Expression or

Expression ^ Expression

Examples of boolean expresions

```
a || b // logical or          b && c // logical and
flag_a ^ flag_b // xor        (a || b) && (!c ^ d)
!{toggle}.state || ok         !(F(a) || H(b))
```

4.6 Expressions from assignment operators

There are six (6) assignment operators in the I-GET language, which form a sub-set of the assignment operators supported in C. The basic = operator, known as the simple assignment operator, and the rest five (5) assignment operators, known as compound assignment operators. The syntax and and examples for assignment expressions follow below.

Grammar for assignment expressions.

AssignExpr:

Lvalue = Expression or

Lvalue += Expression or

Lvalue -= Expression or

Lvalue *= Expression or

Lvalue /= Expression or

Lvalue %= Expression

Examples of assignment expressions.

```
a=b=c=d=f                     a+=16
n/=2 // gets half of n.       i-=step
n*=n // gets n square.        n-=n // gets 0.
j%=(k-j%k)                    X+=Y+=Z+=W=10
```

4.7 Expressions from relational operators

Six relational operators in the I-GET language: two equality checking and four ordering checking operators.

The relational operators in the I-GET language are distinguished in equality operators, for checking whether two expressions are of the same value or not, and ordering operators, which provide the relative position of two expressions (e.g. larger, less) in an ordering of the value domain. There are two (2) equality operators and four (4) ordering operators. The result of relational expressions if of `bool` type. The grammar for relational expressions as well as examples follow below.

Grammar for relational expressions.

RelationalExpr:

EqualityExpr ***or***

OrderingExpr

EqualityExpr:

Expression == Expression or

Expression != Expression

OrderingExpr:

Expression < Expression or

Expression <= Expression or

Expression > Expression or

Expression >= Expression

Examples for relational expressions.

```
a==b // equality check.        x+1 > x // true.
a!=b // inequality check.      x >= x // true.
(a+1)*b <=a*b                  a[j] < a[i]
a+1 != a // true.              a+1==a // false.
```

4.8 Type checking rules and type conversions

We will provide the type checking and type conversion rules for all operators. Each operator allows only a particular set of expression types to be engaged, while they are specific regulations as to the type of the resulting expression after the operator is applied. We present in detail the types of expressions allowed for each operator (*type checking*), as well as the rules governing the data type of each resulting expression (*type conversion*).

Operator	Type checking	Type conversion
()	• All expressions allowed.	• Original type not affected.
[]	• Arrays. • Pointers.	• If array, the result is of the array item type for the current array dimension. • If pointer, the result is of the item type for the current indirection depth.
->	• Pointer to structure, accessing a field variable. • Object reference, accessing a member function.	• If pointer to structure, the result is of the field type. • If object reference, then member function is called, hence, the result if of that of the member function type (typeless, if `void`).
.	• Structure variable, accessing a field. • Object reference, accessing a local member variable.	• If structure, the result is of the field type. • If object reference, the result if of the local member type.
!	• `bool`, `int`, `longint`, `word` and `longword`.	• The result is a type `bool` expression always.

	• Pointer variables.	• Non-zero numbers are converted to `true`, while zero valued numbers are converted to `false`. The `!` operator negates this value. • All pointers with a `nil` value are converted to `false`, else to `true`. Again the `!` operator negates this value.
++ --	• Variables from any numeric data type. • Pointer variables.	• In numeric variables, the result is of the same numeric type. • If pointer variables, the result is of the same pointer type.
+ (unary)	• All numeric expressions.	• Result is of the same numeric expression type.
- (unary)	• All numeric expressions.	• Result is of the same numeric expression type.
*** (unary)**	• All pointer types.	• Result is of the item type for the current indirection depth.
&	• All types of variables.	• Result is a pointer to the type of the variable.
*** (binary)**	• All numeric expressions.	• Result is of the major numeric type. Numeric types are listed below with increased majority: `int`, `word`, `longint`, `longword`, `real`, `longreal`.
/	• All numeric expressions.	• Result is `longreal`, if a `longreal` is engaged, else `real`.

%	• All integer numeric data types (all number types except `real` and `longreal`).	• Result is of the major numeric data type involved in the module expression.
+ (binary)	• All numeric expression types. • `string` expression added with with a numeric, `string`, or `char` expression. • Pointers added with integer numeric expressions.	• If numeric, result is of the major numeric type. • If `string`, result is string. • If pointer, result is of the same pointer type.
- (binary)	• All numeric data types. • Integer numeric expressions subtracted from pointers.	• If numeric, result is of the major numeric type. • If pointer, result is of the same pointer type.
< <= > >=	• All numeric expression types. • `string` expression types.	• Resulting expression is always of `bool` type. • If numeric, arithmetic comparison is performed. • If `string`, lexicographic comparison is performed.
== !=	• All numeric expression types can be engaged. • All pointer expressions, compared with pointers of the same indirection depth and basic type, or `nil`. • All rest built-in types with expressions of the same built-in type; id built-in types can be compared with `nil`.	• Result is in all cases of `bool` type. Value matching is performed.

	• Object references of any class, when compared to object references of the same type or `nil`. • Agent class references compared with `daid`. • Lexical class references compared with `objectid`. • No arrays or structures can be engaged.	
^ && ||	• All `bool` type expressions. • All expressions which can be converted to bool (i.e. po and nters, integer numeric expressions).	• The result is always a `bool` type expression.
=	• All numeric expressions. • `string` assigned with: string, char and any numeric expression. • Pointers assigned with pointers of the same indirection depth and basic type, or `nil`. • Pointers *P* assigned with arrays *A*, if **P* and *A[]* expressions are of the same type. • Arrays assigned with arrays matching in all dimensions and array element type.	• The result in all cases gains the type of the Lvalue (i.e. variable on the left, being assigned the value).

	• Structure variables, assigned with expressions of the same structure; matching in structure type name is required. • `daid` assigned with `daid`, agent class reference expressions or `nil`. • `objectid` assigned with `objectid`, lexical class reference expressions or `nil`. • `sharedid` assigned with `sharedid` expressions or `nil`. • Rest of built-in type variables assigned with expressions of the same type.	
***= /= += -= %=**	• Such expressions are of the form *op*=. The matching rule is that both the *op* as well as the = should be allowed, following the above type matching rules.	• Resulting expression is defined as in the case of assignments (=).

Chapter 5

5. Monitors

What are monitors to objects / variables.

A monitor is a code segment attached to an Lvalue (i.e. an expression that designates a variable with a reserved storage in memory), which is automatically executed, exactly after an Lvalue is changed. Monitors are declarative language constructs which are not supported in typical programming languages like C / C++ / Java / Ada. Programmers are not bothered by the point in code-, as well as they way in which-, an Lvalue changes; monitors have similarities with the notion of event handlers for interaction objects (objects corresponding to Lvalues, specific event being value modification).

5.1 Basic scheme and examples

MonitorDef:

LvalueList : Compound

LvalueList:

Lvalue (**, Lvalue**) ∗

A single code segment (i.e. block - **Compound** statement), can be attached to a list of Lvalues, and not only to a single Lvalue. If any of the Lvalues listed changes, then the block is executed. Since any type of Lvalue an be engaged within a monitor, pointer expressions and array indexing expressions can be involved as well. In all cases, the I-GET run-time system will detect changes, independently of the way an Lvalue has been updated (i.e. it may happen via pointers and indirection). Also, another important property

The notion of dynamic object redefinition, and an innovative feature of the I-GET tool.

of the I-GET language is that supports in a fully documented manner a phenomenon called *dynamic object redefinition*. Some Lvalues do not always designate the same storage, and they depend on other expressions. For instance, the expression `a[i]`, indicates an array element and it is an Lvalue. If however, i changes, the then a different storage location is represented by `a[i]`, in other words, the object `a[i]` has been dynamically redefined, a phenomenon which is different from changing the content of `a[i]`. The situation is similar for an Lvalue `*p`, if the value of `p` pointer (i.e. address) changes; then `*p` denotes a different storage area (i.e. different Lvalue). In case a monitor is attached to such types of Lvalues, which are subject to dynamic object redefinition, in all cases that dynamic redefinition occurs at run-time, the I-GET run-time system will cancel the associated monitor(-s) and will re-install it / them to the re-defined Lvalue(-s). Examples of monitors, including cases in which dynamic object redefinition is observed, follow below.

```
int a, *b, c[10], i=0;
a: [ printstr("'a' has changed to " + a +'\n'); ]
a, *b : [
      printstr("Second monitor for 'a' " + a + '\n');
      printstr("'*b' has changed " + *b + "\n");
]
c[i]: [ printstr("c["+i+"] has changed "+c[i]+'\n'); ]
i: [ printstr("'i' changed, c[i] redefined.\n"); ]
bool ok=false, no=false, cancel=false;
ok: [
      if (ok) [ /* here actions for ok. */ ]
]
```

```
cancel: [
      if (cancel) [ /* here actions for cancel.*/ ]
]
no: [
      if (no) [ /* here actions for no. */ ]
]
```

Multiple monitors on the same Lvalue allowed.

In the examples above, the employment of monitors to implement asynchronous dialogue control is shown (shaded code segment, with monitors for `cancel`, `ok` and `no` boolean variables); in this case, the boolean variables are to be only set within the call-backs of their corresponding dialogue-box button objects, while the "reaction logic" is separately implemented by means of monitors. The capability of having multiple monitors on the same Lvalue is also illustrated (for variable `a`). In case that multiple monitors are defined for a single Lvalue, the execution rule is very simple:

Order of execution in case of multiple monitors.

Order of installation for monitors at run-time.

Monitors are executed at run-time following the exact order of registration. Hence, most recently installed monitors are the last to be called.

A monitor is installed at run-time when the scope in which it belongs is instantiated. Monitors can be defined either globally, or locally within I-GET specific classes. File scope is instantiated at run-time, if the particular file is included in the dialogue control component. When file scope is instatiated, global variables are initialized in their order of declaration. A monitors is installed exactly after all monitors / variables preceeding that particular monitor in file scope are installed / initialized.

Hence, in the following code example, the order of execution steps, upon start up, is: `my_var` is initialized to `3`, monitor for `my_var` is installed, `my_str` is initialized to `"3"` (from `my_var`), monitor for `my_var` is called (since `my_var` changes in `InitValue` function), and `my_int` is initialized to `5` (returned from `InitValue` function).

```
int my_var=3;

my_var: [

      printstr("'my_var' changed to " + my_var + '\n');

]

string my_str=myvar;

int InitValue () [ return my_var=5; ]

int my_int=InitValue();
```

If multiple monitors are installed in file scope for the same variables, however, from different files, the order of installation is undefined.

Using storage qualification, it is possible to access variables from other files. In such cases, it is also possible to attach monitors to external variables, thus being able to "monitor" information residing in other modules. In this case, if other monitors have been also installed in file scope from other files, on the same variable, the order of installation is undefined, since it depends on the order of initialization of generated (by the I-GET compiler) C++ files. Hence, you should not assume any particular registration order in such cases. This scenario is illustrated below.

```
// file1.txt

int shared_var;

shared_var: [ printstr("Monitor from file1.txt.\n"); ]
```

Which monitor is installed first? You should not write code by making any assumption on this.

```
// file2.txt

extern int shared_var;

shared_var: [ printstr("Monitor from file2.txt.\n"); ]
```

5.2 Cycle elimination

What if you try to change an Lvalue from within a monitor which is attached to it ? Potentially, this could cause a new Lvalue update notification to be generated, resulting in calling all associated monitors to this Lvalue, including the one changing the Lvalue, thus changing again the Lvalue, thus calling again the monitor, and so on, causing an infinite number of such cycles to be executed. The I-GET policy to resolve this situation is very simple:

Cycle elimination policy for monitors in the I-GET tool.

All changes to a monitored Lvalue, while the execution of an associated to it monitor is still in progress (i.e. the execution control is still within the monitor block), are disabled by the run-time system, while the Lvalue retains its previous content.

Through such *cycle elimination* behaviour, the update effect will be simply discarded while the Lvalue will preserve its previous content. This is a safe behaviour and will cause neither a run-time error nor any exception. Examples are given below for monitors which could cause cycles.

```
int my_var=23, *my_var_ptr=&my_var;
```

An example which demonstrates the issue of cycle elimination.

```
my_var: [
     my_var++; // the effect of this is disabled.
     (*my_var_ptr)--; // also this effect is disabled.
     printstr("my_var is " + my_var + "\n");
]
```

```
int Init() [ // a file initialization  function.
     return my_var+=my_var; // causes monitor call.
]
local int dummy=Init(); // trick to call Init() func.
```

5.3 Monitors on aggregate variables

Monitors can be attached to any type of Lvalue, even those of aggreegate types (i.e. *arrays* and *structures*). A monitor attached to an aggreegate Lvalue will be called if its content is affected in any manner. A structure variable is affected if any of its field is affected (the rule recursively applies for structure-type fields as well). An array type variables is affected if any of its elements is affected. When a monitor for an aggregate variable is called, there is no a-posteriori way to distinguish which particular element was the one changed, that caused the monitor to be called. Examples of monitors on aggregate variables follow below.

```
struct Point [ word x,y; ];
Point my_point=[0,0];
Point points[2]=[my_point, my_point];
my_point: [ printstr("'my_point' changed.\n"); ]
points: [ printstr("'points' changed.\n"); ]
my_point, points: [
      printstr("'my_point' or 'points' changed.\n");
]
```

The problem with multiple monitor calls in case of monitors to aggregate variables.

When attaching monitors to aggregate variables, you should take care of the fact that any change of a basic element will cause the monitor to be called. In case that multiple elements are changed by successive statements executed at run-time, the monitor will be called as many times as the number of elements changed. This may either introduce performance problems for large aggregate variables or may cause execution of monitors at a point where this is not desired (e.g. you may want the monitor to be executed only after all target elements have been changed to the new values). To avoid such problems, you may do the following trick:

A trick to avoid multiple monitor calls when monitoring

Choose an appropriate element of an aggregate variable which you will always change last in your code, when updating the various elements of the aggregate variable. Attach a monitor only to this single element; this moni-

aggregate variables.

tor will be executed, only after all elements (i.e. the whole aggregate variable) have changed.

An example showing the application of the technique to avoid multiple calls of monitors on aggregate variables.

```
struct Margings [ real left, right, up, down; ];
struct ParagraphSettings [
      bool hyphenate;
      real line_spacing;
      Margings margins; // 'down' field is marked.
];
ParagraphSettings pgph_set;
pgph_set.margins.down : [
      // Code here as if update of whole pgph_set.
]
string str_arr[10][10][20]; // [9][9][19] is marked.
str_arr[9][9][19] : [
      // Code here as if update of whole str_arr.
]
```

5.4 Constraining variables together through monitors

Monitors grasp changes on variables in a manner independent of the source point, as well as the update implementation approach (i.e. direct / indirect). Due to this capability, it is possible to construct *monitor schemes* which exhibit distinctive properties. For instance, consider the code below.

```
#define FORCE_SAME(a,b) a: [b=a;] b: [a=b;]
```

```
int i,j; FORCE_SAME(i,j)

string s,w FORCE_SAME(s,w)
```

The FORCE_SAME macro defined, declares two monitors, having the effect of keeping the a pair of variables passed as macro parameters always the same. This is achieved by implementing two symmetric monitors: one which "catches" the update on the same variable, and applies the change to the second, and the opposite. It is still possible to update manually one of the two variables (i.e. via a conventional statement), but always the update will be propagated to the other variable. Some languages, like I-GET itself, support explicitly constraints; however, in the case of such closed cyclic constraints (as in the above example), external updates may not be allowed; in such situations, the employment of monitors may be an easier solution.

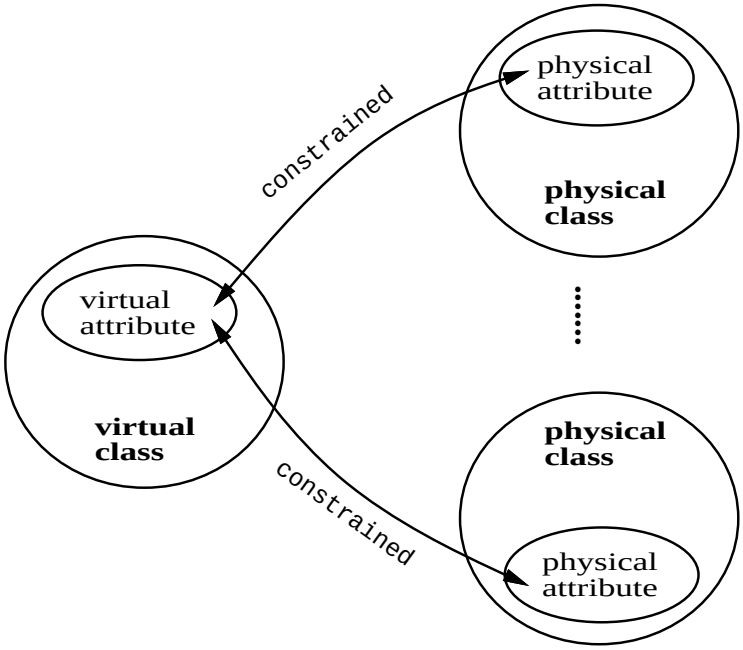


Figure 5.4–1: Attributes of virtual objects are at run-time constrained with the corresponding attributes of all active physical instances.

The use of monitors for simulating con-

Another powerful use of monitors, in place of constraints, is when there is a cluster of constraints, which is, however, incrermentally constructed. For instance, later we will discuss the definition of virtual classes, and how these can be mapped to multiple physical object classes (i.e. polymorphism), sup-

straints is particularly appropriate in the case of constraining virtual attributes with physical attributes.

porting multiple active instances (i.e. at run-time, a virtual object may be linked to more than one physical instances). In this case, the attributes of physical instances must be constrained some-how to the attributes of the virtual instance and vice versa, so that the virtual and all concurrent physical instances may share consistently attribute values (see Figure 5-1). But when defining virtual classes, the number of physical classes is unpredictable, hence, a single constraint cannot reside at the virtual object class side. In the same manner, it cannot reside either at the physical class side.

The only place is within distinct instantiation schemes, which must be defined separately for each alternative mapping to a physical class. If a constraint of the form *virtual attribute*←*physical attribute* is added in each such scheme, only the last established constraint would be active (since, multiple constraints would be added on the same variable). The solution is to constrain the virtual attribute through the *FORCE_SAME(virtual attribute, physical attribute)* scheme of monitors. This issue will become more clear in the discussion of instantiation schemes, and you should not worry of the above scenario seems obscure for the time being.

Chapter 6

6. Constraints

Constraints are relationships which are forced on engaged variables. Various categories of such relationships may be supported by different constraint systems, such as: equality, inequality, ordering (based on comparison / ordering operators), etc. In the I-GET tool, e*quality constraints* are supported (i..e. constraint equations), forced only on one direction (i.e. *unidirectional constraints*). Unidirectional equality constraints look like conventional assignments for typical Lvalues, which, however, must be preserved automatically (i.e. re-evaluated somehow) by the *constraint satisfaction* system, when engaged Rvalues change.

6.1 Basic scheme and examples

Syntax for unidirectional constraints.

Constraint:

Lvalue := Expression ;

Examples of simple constraints in the I-GET language.

```
real a, b, x, y;

y := a * x + b;

{obj1}.x := {obj2}.x + {obj2}.width + SPACE;

string name, sur_name, all; real salary;

all := "Name:" + name + "\n" +

      "Surname:" + sur_name + "\n" +

      "Salary:" + salary + "\n";
```

All types of Lvalues engaged in assignments are also allowed in constraint equations, while all types of expressions engaged as Rvalues in assignments are also allowed on the right part of constraints; the only syntactic difference between assignments and constraints is that instead of the assignment operator =, the constrain operator := is used. Constraints are only allowed within *file* and *class* scope.

In the above examples, each time a variable is engaged in the right part of the constraint is changed, the expression is re-evaluated and assigned to the constrained Lvalue. Constraints, in addition to monitors, provide another facility for declarative programming control, and are appropriate for a large repertoire of programming tasks such as: forcing particular topological relationships among displayed graphical objects, automatic display update for internal data structures when those constrain attributes of displayed interaction objects, variable associations needed in programming modules, etc.

The constrained Lvalue is considered as dependent only on those variables which are directly visible within the constraining expression.

The above rule helps avoid any misinterpretations as to what happens if a function call is engaged within the expression, which in its implementation body accesses global variables affecting the returned value. If such a global variable is changed, will the run-time system re-evaluate the constraint ? The answer to this question is this:

If a function call is engaged in a constraining expression, re-evaluation due to this function call will be carried out only if a variable engaged within its actual arguments is modified.

Examples which

```
int a, b, c, d, e;

int F() [ return a+b; ]

int H (int a, int b) [ return a + b; ]

e := F() + H(d,e);
```

show how function calls are treated within constraining expressions.

```
a=10;  // no re-evaluation - invisible in F().
b=20;  // no re-evaluation - invisible in F().
d=30; // yes, 'd' is actual argument in H(d,e).
e=40; // yes, 'e' is actual argument in H(d,e).
```

6.2 Constraints and *dynamic object redefinition*

As it has been briefly introduced in the previous Section, when discussing monitors, dynamic object redefinition may cause the dynamic change, at run-time, of the actual Lvalues on which certain language constructs, such as monitors, are attached. This issue also applies to constraints, since Lvalue are those which are subject to constraint equations. The effect of dynamic object redefinition on constraints is dual:

Effects of dynamic object redefinition on constraints.

- *Constrained Lvalue is dynamically redefined*, which causes the constraint object to vary at run-time, even though the constraint equation is still the same.

- *Constraining expression is dynamically redefined*, which means that, apart from the content of the expression as such (i.e. value update), some particular engaged variables are redefined; as a result, the same variable update that caused a particular constraint to be re-evaluated at a certain execution point in time, may have no effect at another time, if due to dynamic object redefinition the updated variable no longer participates in the constraining expression.

In the next sub-section we will discuss all the details of dynamic object redefinition, and its effect on Lvalues (Lvalue is an expression denoting a *variable*) engaged in constraints.

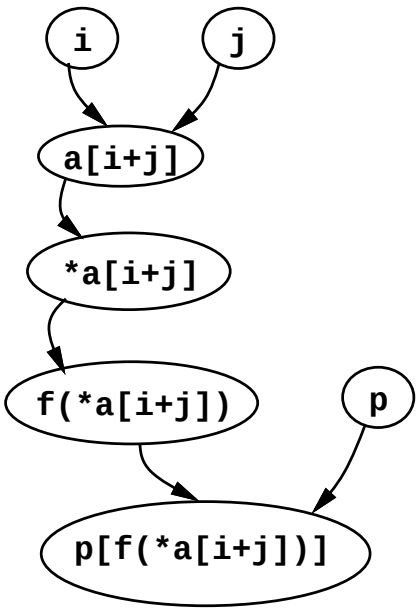
6.2.1 Dynamic object redefinition variable graphs

Consider the Lvalue `a[i]`, with say `i` being `3`. Then, clearly if `i` changes,

e.g. `i` becomes `6`, then `a[i]` denotes a different variables (from `a[3]` before, to `a[6]` now). Hence, there is an intuitive relationship between `i` and `a[i]`, since we now that if `i` changes, then `a[i]` is dynamically redefined. We can define such a relationship more formally:

*There is a dynamic redefinition relationship **x**→**y** between objects **x** and **y**, if changing the object **x** caused the object **y** to be dynamically redefined.*

Graphs resulting from such relationships are called dynamic object redefinition variable graphs, and show how the effect of changing a single variable is affecting dynamically the definition of other variables. An example follows below.




```
int i, j;

int* a[100];

int* p;

int f (int k);
```

In the above example, the Lvalue `p[f(*a[i+j])]` is dynamically redefined if any variable in the graph on the left, from which there is a path to this Lvalue, is modified.

Such dynamically redefined Lvalues, may be engaged either as the constrained Lvalues, or just participate in the constraining expression. In both cases, I-GET will dynamically update correctly variable engagement.

6.2.2 *Lmembers* and *Rmembers* in unidirectional constraints

Lmembers *are all those Lvalues appearing in the dynamic object redefinition graph of the constrained Lvalue; the latte is also included in this graph and is called* ***Major Lmember****.*

Rmembers *are all those Lvalues appearing in the dynamic object redefinition graphs of the various Lvalues directly engaged in the constraining; the latter are also called* ***Major Rmembers****.*

Next we will explain the effect caused by changing either Rmembers or Lmembers, on the constraint(-s) in which they are engaged. We will consider the following constraint to explain the various effects:

```
int j, *p, *q, a[10], b[20], c[30];

a[*p] := b[j] + c[*q];
```

Changing the Main Lmember.

It is allowed to change it via a statement, only if there is a constraining expression of an active constraint, in which the Main Lmember is engaged as an Rmember (i.e. it constrains at least another variable). If this is not the case, then passing explicitly via a statement a value to the Main Lmember will have no effect; for the above example, the Main Lmember `a[*p]` does not constrain other variables, hence, any assignment directly or indirectly is disabled:

```
a[*p]=27; // no effect.

int *pp=&a[*p]; // lets try indirectly.

*pp=29; // also no effect.
```

Changing any other Lmember.

Changing any other Lmember will cause the Main Lmember to be dynamically redefined. The effect is that the constraint on the old Main Lmember variable is cancelled, while the same constraint is installed on the re-defined Main Lmember. When a constraint is installed, it is also evaluated for the first time. For our previous constraint example, the following code shows what happens when a particular Lmember is changed.

```
int new_index=11;

p=&new_index; // 'p' is an Lmember that is changed.

// now, constrained a[*p] is actually a[11].

new_index= 13; // '*p' is an Lmember that is changed.

// now, constrained a[*p] is actually a[13].
```

Changing a Major Rmember.

When a Major Rmember is changed, a variable that is directly engaged within the constraining expression is modified, thus requiring re-evaluation of the constraint so that the constrained Lvalue is "refreshed" accordingly. Updates on Major Rmembers only cause re-evaluation and do not lead to cancellations and re-installations of constraints. Examples of updating Major Rmembers follow.

```
b[j]=b[j+1]; // 'b[j]' change causes constraint satis-
             faction.

int index=*q;

c[index]=b[j]; // 'c[*q]' is 'c[index]', hence again
                  constraint satisfaction is called.
```

Changing any other Rmember.

Changing a non-Major Rmember will cause one or more Major Rmembers to be dynamically re-defined. Hence, after such an update, the Main Lmember becomes constrained by different Rmember variables; this will require cancellation and re- installation of the constraint with the re-defined Rmembers. Examples of this case follow below.

```
j++; // 'j' is an Rmember that is changed.

int an_index=26;

q=&an_index; // 'q' is an Rmember that is changed.

an_index++; // 'an_index' is '*q' that is an Rmember.

*q+=10; // '*q' is an Rmember.
```

6.2.3 More examples on constraints

```
string s; int a, b;

s:="(" + a + b + ")";

void F() [

      a := b; // syntax error, constraints allowed only
              in file- or class- scope.

]

a := s; // semantic error, assignment type conflict.

int c, d, e=10;

a := b = c = d = e; // ok, legal, though unorthodox.

b+1 := a+1; // syntax error, not an Lvalue on the left.

a := 5; // ok, 'a' is now a constant (changes dis-
        abled).

int a_const=10;

a_const := a_const+10; // this constraint is legal,

a_const=35; // but take care, 'a_const' is 45 now, and
            not 35 (10 added due to constraint).

int arr[10], index=0, value=23;

arr[index] := value; // initially, its arr[0].


// Calling SetValue(<arr val>); will cause

// all 'arr' elements to become <arr val> !.

//

void SetValue (int val) [

      value=val;

      for (index=0; index<10; index++)

            ;

]
```

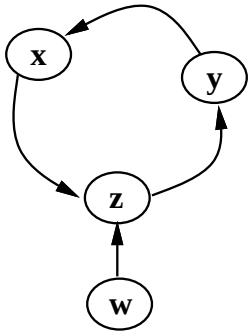
6.3 Cycle elimination

*When an Lvalue **x** appears as an Rmember (of any type) in the constraining expression of a Major Lmember **y**, then **y** is dependent on **x**, defininf a formal relationship **x**→**y**.*

When multiple constraints are defined, the graph resulting from such dependency relationships is called the *constraint graph*. It is possible that cycles are included within a dependency graph. For instance, consider the example constraints below (left) and their corresponding constraint graph (right).

An example where a cycle within the constraint graph is created.

```
int x, y, z, w;

x := y;

y := z;

z := x+w;
```



In the above example, there is apparently a cycle in the constraint graph. For the time being lets forget what this cluster of constraints aims to express, and try to focus on its behaviour at run-time. Since there is a cycle, it is allowed to "manually" (i.e. via a statement) change any of the variables which are engaged in the cycle, since all these variables which are constrained (i.e. *bounded* variables - have an incoming edge), constrain also other variables (i.e. have outgoing edges); variable `w` is a *free* variable (has only outgoing edges). Assume that `w` is changed; this will affect `z`, which will affect `y`, which will affect `x`, which will affect again `z`. At this point, a constraint system should be able to stop, without causing re-evaluation due to a cycle.

The I-GET constraint system performs optimal constraint evaluation (i.e. ***less required constraint evaluations****), ensures that all dependent variables will be re-evaluated (i.e.* ***completeness****) once and only once (i.e.* ***cycle elimination****).*

6.3.1 Two-phase algorithm for constraint satisfaction

The constraint satisfaction algorithm of the I-GET tool is being split in two phases (see Figure 6.3-1): (i) the *planning phase*; and (ii) the *evaluation phase*. During the planning phase, an optimal cycle-free sequence for evaluating constraints is constructed, while during the valuation phase, all constraints involved in this sequence are evaluated.

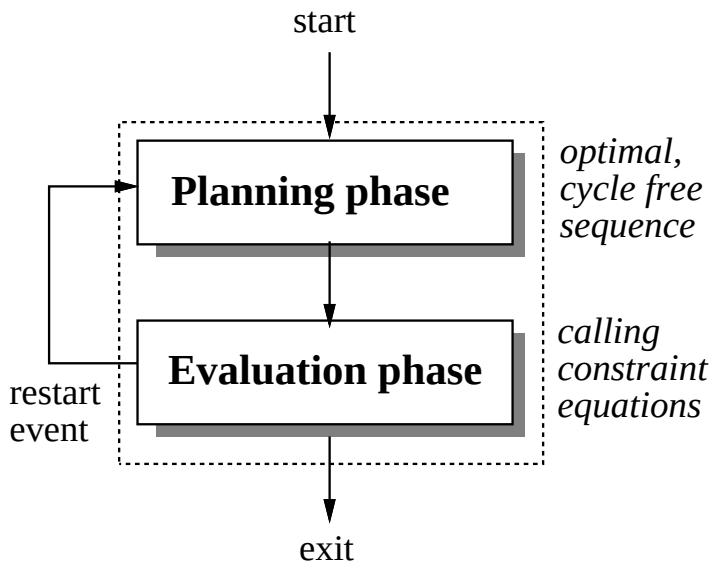


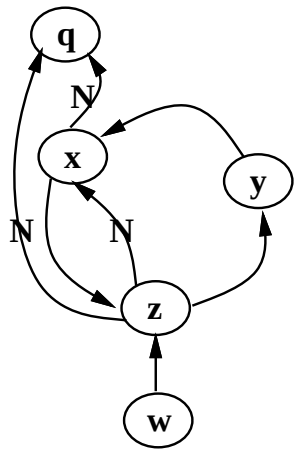
Figure 6.3–1: Two phase constraint satisfaction in I-GET.

The planning phase involves internal processing in which no program code is executed and no variables are affected. However, within the evaluation sequence, the execution of the various constraint assignments will cause the involved constrained variables to change. Such changes, while the evalua-

tion phase is in progress, may trigger other types of processing / events, such as calling associated monitors, or evaluating preconditions. As it will be discussed later on when discussing agent classes, it is possible that the evaluation of certain preconditions may lead to the instantiation of associated agent classes. If such newly created agent instances encompass locally constraints, then during their instantiation those constraints will be also installed. Similarly, it is possible that some preconditions cause destruction of agent instances which may result in cancelling some constraints. Hence, if this scenario is realized in practice, then during the evaluation phase the constraint space will be affected. Such an event will not affect the evaluation phase if the variables engaged in the planning sequence are not related to the installed / cancelled constraints. But if they are, the evaluation process will be interrupted and the constraint satisfaction algorithm will restart to consider the new changes in the space of active constraints.

This issue is illustrated below; we assume that after the evaluation of variable `y` from our previous example, some new constraints are installed, which also affect the constraint graph (new egdes marked with **N**). The evaluation sequence will be stopped, while the constraint satisfaction will restart considering the update of variable `y` as the starting point, and calculating for all active constraints.

```
/*

The following constraints
are activated when 'y'
variable is evaluated.

*/

x := y+z;

int q;

q := x+z;
```



6.3.2 Cycle elimination during *planning phase*

The cycle elimination capability of the I-GET constraint satisfaction algorithm is included in the planning phase. The fact that cycle elimination is supported, enables bi-directional constraints to be easily modelled via decomposition into a number of unidirectional constraints. For instance, consider that we want to force three variables to have the same value. Then, the following set of unidirectional constraints achieves this goal.

```
real a, b, c;
a := b;
b :=c;
c := a;
b=27; // will propagate 27 to 'a' and 'c' as well.
```

6.3.3 Handling constraints generating infinite *restart* events

Even though the planning phase eliminates cycles, in the sense that constraints present in the planning sequence are evaluated only once, there are certain types of constraints which could still cause the constraint satisfaction algorithm of I-GET to fall in an infinite loop; we do not know yet any other constraint system that would not "hang" with these scenarios.

If an Rmember changes due to the evaluation of a constraining expression in which it is engaged, then re-evaluation may be caused.

The action that the I-GET constraint system takes is to ignore all notifications for updates on Rmembers which are engaged in the constraint being currently evaluated. We will show some examples which would normally cause the system to "hang" in other constraint systems, due to re-evaluation triggered, just after a particular constraint evaluation has been performed.

An example causing re-evaluation, which would lead to an infinite loop.

```
int i, j;

// This would cause always re-evaluation.
i := j++;

// An infinite loop could start.
j=1;

// 'i' and 'j' are 2 here - notification for j ignored.
//
int a, b; int F(int c) [ return b=10; ]

// Change is not so visible as before.
a := F(b);

// Here 'a' and 'b' are 10 - no infinite loop.
int arr[10], index=0; int G() [ return index ++; ]

// Main Lmember continously redfined.
arr[index] := G();

// 'index' is 1, arr[0] is 1, arr[1] now constrained.
//
int N;
extern int counter;

N := Process(&counter);

int Process (int* q) [
    if (*q==JUMP_POINT)
         *q+=JUMP; // Rmember 'counter' changes here.
    return *q % GROUP_SIZE;
]
```

Re-evaluation caused due to (less visible) Rmember change, and to continuous Main Lmember redefinition.

In this case, re-evaluation would be caused only once, since the Rmember is conditionally changed.

In the previous example, re-evaluation would not cause any problem, even if the I-GET tool would not ignore the problematic Rmember update notifications. The reason being that re-evaluation is caused only once, due to putting conditionality in the change of variale `*q`. But even after ignoring notifications, the variable `N` still gains the new value, since the result is returned after the change. The examples aims to show that in some cases, re-evaluation may be something you could take advantage of; but there is always an alternative way to go, through the more safe approach of I-GET to disable the previously described “dangerous” type of Rmember modification.

6.4 Constraints on aggregate variables

Constraints may engage aggregate variables (i.e. arrays and structures) either as Rmembers or as Lmembers. Of particular interest is the case of having aggregate variables as Rmembers, and the effect it has on constraint satisfaction as the value of such variables is modified. Firstly we provide some examples of constraints on aggregate variables, and then we discuss the special case of Rmember aggregates.

Examples of constraints on aggregate variables.

```
struct Point [ word x, y; ];
Point mouse, draw_pixel_at;
draw_pixel_at := mouse;
draw_pixel_at : [ /* draw a pixel here */ ]
int arr[10], copy_arr[10];
copy_arr := arr;
```

When aggregate variables are Rmembers, the variable is considered as modified if any of the basic / primitive elements change. For instance, in the following code we provide examples of statements and indicate at which point update notification will be generated (for the constraint satisfaction, monitor management and precondition management mechanisms).

```
mouse.x=34; // one notification.
mouse.y=mouse.x=0; // two successive notifications.
Point my_point=[100,100];
mouse=my_point; // two successive notifications.
int new_arr[10]=[1,1,1,1,1,1,1,1,1,1];
arr=new_arr; // ten successive notifications.
```

In the above examples, when more than one notification is generated, these are not handled by means of a massive notifications, indicating the number of updates and the variables being updated, but a series of distinct notifications, all generated at the time of that the variable changes (this is the only policy that can be followed without consistency problems). As a result, each time one of the above notifications is processed, the constraint satisfaction algorithm is called.

Lets study the case of changing the `arr` variable via an assignment statement. The result of the assignment `arr=new_arr;` is that all each elements of `new_arr` is copied / assigned to its corresponding element of `arr`. However, each time such an action takes place, the arr variable is considered to be changed. Hence, due to the fact that `arr` is an Rmember in a constraint, it will trigger constraint satisfaction and cause `arr` (with its particular current value of elements) to be assigned on `copy_arr`.

Only after the last element from `new_arr` is copied to the corresponding in `arr`, then `arr` would have been completely assigned the `new_arr` contents and be in a stable value state. This final element assignment will still trigger constraint satisfaction, giving to `copy_arr` a stable value content. Practically, all the previous constraint satisfaction rounds have been unnecessary, however, they cannot be avoided. Why ? Because there are cases in which we want such element updates to trigger notifications, as it is shown in the example below.

```
struct Employee [
       string name, surname, address;
       real years;
];
Employee my_emp, copy_empl;
```

```
copy_empl := my_empl;
my_empl.surname := {surname_obj}.text;
my_empl.name := {name_obj}.text;
my_empl.address := {address_obj}.text;
my_empl.years := strtoreal({year_obj}.text);
```

In the above example, there is a variable `my_empl`, which holds an Employee data value. The fields of this structure variable are directly constrained with the text-content attribute of text-field objects (i.e. `{<field>_obj}.text` expression). Hence, each time the user edits such a field, the constraint satisfaction algorithm is triggered, causing the corresponding field of the `my_empl` variable to be changed, subsequently causing an assignment of the whole `my_empl` structure to `copy_empl`. Apparently, this is a correct behaviour, though applying unnecessary assignments. You can solve this problem by breaking down a constraint to the various basic elements; this could be easily handled for structures, but it may be difficult in practice for arrays. This is the main reason that in case of arrays, the employment of monitor-based techniques for constraining array variables results in better run-time performance.

6.5 Defining multiple constraints on the same variable

Multiple constraints on the same variable (i.e. Major Lmember) may be installed at run-time. In such case, the following rule is applied:

Constraints on the same variable are placed in the same stack, when they are installed. When the scope of a constraint is destroyed, this constraint is also removed from the stack. The constraint at the top of stack, is always the

active constraint.

When constraints belong to the same scope, if their scope is instantiated, the order of activation reflects the order of declaration in their particular scope context. In the example below we indicate the order of activation for each constraint; it should be noted that when a constraint is activated (i.e. installed) it is evaluated directly for the first time (so, the constrained varibale is assigned to the constraining expression).

Order of activation of constraints in the same scope.

```
int a, b=10, c=20;

a := b;   // 1st activated, 'a' becomes 10.

a := c;   // 2nd activated, 'a' becomes 20.

a := b+c; // 3rd activated, 'a' becomes 30.
```

As it has been discussed also for monitors, using storage qualification, it is possible to access variables from other files. In such cases, it is also possible to attach constraints to external variables, thus being able to "constrain" information residing in other modules. In this case, if other constraints have been also installed in file scope from other files, on the same variable, the order of installation is undefined, since it depends on the order of initialization of generated (by the I-GET compiler) C++ files. Hence, you should not assume any particular registration order in such cases. This scenario is illustrated below.

Which constraint is installed first ? You should not write code by making any assumption on this.

```
// file1.txt

int shared_var;

local constraining_var;

shared_var := constraining_var;
```

```
// file2.txt

extern int shared_var;

local constraining_var;
```

```
shared_var := constraining_var;
```

Chapter

7

7. Hooks and bridges: mixing I-GET code with C / C++ code

There various development scenarios in which it is required that within particular programming languages, mixing with other languages is required. The type of such mixing allowed varies, depending on the importance that is has for the particular programming language:

- Functions / procedures written in other languages and can be called, while their object files can be linked together. Typically, this will work only across compilers of a particular vendor.

- Code can be written in-line, within independent source segments, while a single compilation phase illusion is preserved, transparently calling a native compiler of the second language. Typically, this will work only when calling in-line assembly code.

- Mixing code inline (i.e. *hooks*), calling foreign functions in both directions, well defined exchange of data types, cross language assignments (i.e. a foreign expression to a native variable) and mixing among statements (i.e. *bridges*). This is the type of mixing allowed in the I-GET language with C / C++; we do not know other 4GLs (like I-GET language) allowing such powerful mixing with 3GLs (like C / C++).

7.1 Hooking globally C / C++ code

Hooking globally C / C++ code means enabling in-line source segments at file scope to be written in C / C++. It is important to emphasize that this is allowed only within file scope, C / C++ hooked program slices must be globally defined. A hooked source segment starts with the tag **_c++[** (no space in-between characters allowed) and ends with **]_c++** (again no spaces between the five characters). Simple examples follow below.

```
// This is an I-GET source file.
int a, b; string c, d;
void IGET_func () [ a=b; c=d; ]
_c++[
#include <stdio.h>
void CPP_func (void) {}
char* CPP_var_here;
]_c++
string IGET_var_here;
_c++[ ]_c++ // empty allowed.
```

7.1.1 Possible sources of compile errors

When hooking C++ code, the rule for possible compile errors is simple. If something is C / C++ compiler error then it will "ring" when compiling the code generated modules (which are in C++) with your C++ compiler. The I-GET compiler will place line- an file- information in the generated files, so that compile errors due to hooked code are reported with respect to original source code position. For instance, the following is an I-GET source code which will generate a C++ compile error.

```
1  // I-GET source file.
2  int a, b, c;
3  _c++[
4  #include "myheader.h"
5  int static void a; // an error.
6  ]_c++
7  // back to I-GET code.
I-GET compilation: <all ok>
C++ compilation: Error: Line 5 <description>.
```

Of particular importance are errors appearing due to conflicts from the combination of I-GET and C / C++ code (i.e. no errors if stand-alone I-GET or C / C++ code). The most common source of problems is *name collision*. In such cases, identifiers of C / C++ program slices will conflict with identifiers of I-GET generated code. To avoid name conflicts we provide proper guidelines in the next Section.

7.1.2 Avoiding name collisions

All code generated I-GET identifiers start with a pre-defined prefix. We will list below all these prefixes. Selecting identifiers in C / C++ code avoiding such prefixes will ensure that name collisions will not occur.

_ECO	**_IEV**	**_PIC**
_STR	**_OEV**	**_PII**
_SYN	**_VAR**	**_AGN**
_ENU	**_VIC**	**_EVH**
_FUN	**_VII**	**_MET**

All the previous prefixes start with an underscore, and continue with three capital letters; if you do not want to remember those prefixes in detail, just keep in mind this mnemonic rule. The prefixes are quite cryptic anyway, and it should be unlike that you will need to name identifiers in your program with prefixes from those in the list. Please keep in mind that only names with the above prefixes have the possibility to conflict; all the examples below will compile (passing both I-GET and C++ compilation phases) with no problem.

```
// I-GET source file.
int a, b, c; string d; void F();
_c++[
// The following will not conflict.
int a, b, c; char* d; void F (void);
int H (int, char*);
]_c++
// This will not conflict as well.
int H (int a, real b);
```

7.1.3 Transparency among pre-processor directives

In the I-GET tool, there are two compilation phases: (i) the I-GET compilation phase, which generates code in C++; and (ii) the native C++ compilation phase, which compiles generated code into object code. These two phases have distinct pre-processing phases: (a) the I-GET pre-processing phase, in which a user-defined pre-processor is employed for pre-processing only native I-GET code; and (b) the C++ pre-processing phase, for standard C++ pre-processing of the generated C++ code. In the first pre-processing phase, none of the hooked C / C++ block are processed, hence, all pre-processing directives and developer-supplied macros within C++ hooked

code are left untouched and unprocessed. This policy is taken for two reasons. Firstly, the I-GET compilation phase should be made open to other pre-processors which could simply fail to process C-preprocessor directives in hooked code. Secondly, native C / C++ compilers define some standard macros which can be used by developers for conditional compilation; these macros are not defined by the I-GET tool.

The effect of the above policy is the complete transparency on preprocessor directives between I-GET code and C++ code, even if a standard C-preprocessor is employed for I-GET pre-processing stage. The example below passes the I-GET & C++ pre-processing & compilation phases reporting errors as indicated in the comments below.

```
// I-GET source file.
#define MID(x,y) (((x)+(y)) / 2.0)
#define EMPTY_METHOD(obj,name) \
        method {obj}.name []
_c++[
#include <stdio.h>
#ifdef _cplusplus
// some C++ code here.
#endif
EMPTY_METHOD(<parms>) // error, macro not known here.
#define MID(x,y) (((x)+(y)) / 2.0)
#define MAX(a,b) ((a)<=(b) ? (b) : (a))
]_c++
int Max (int a, int b) [
    return MAX(a,b); // error, macro not known here.
]
```

In the previous example, since all pre-processor directives between hooked code and normal code are completely transparent, the macro `MID(x,y)` defined twice will not cause any redefinition problem since it is defined once in I-GET code and once in hooked code; parameters could be changed, still with no problem. Similarly, within hooked code the `_cplusplus` define can be checked for the presence of a C++ compiler. The macro `EMPTY_METHOD` is not visible in hooked code, hence, there will be a C++ compilation error at that point. Similarly, the macro `MAX(x,y)` is defined within hooked code and is not visible in I-GET code segments; hence, in the `Max` function, calling of `MAX` macro will cause an I-GET compilation error.

Finally, when defining multiple hooked code segments within one file, all these will be assembled within the *same* code generated file as global blocks; hence, you can refer directly to identifiers defined in previous (in the same source file) hooked code segments. Also, you may access externals defined in hooked code within other files. The following examples demonstrates these properties.

```
// file1.txt I-GET source file.
_c++[ int a=10, b=20; ]_c++
_c++[ int c=a, d=b; ]_c++
```

```
// file2.txt I-GET source file.
_c++[
extern int a, b, c, d;
int e=a+b+c+d;
]_c++
```

7.2 Hooking locally C / C++ statements

The I-GET language supports a plethora of statements, which are discussed in detail within Section 13 of this Volume. One such statement is the *hooked C / C++ statement*, which allows an arbitrary statement in C / C++ to be mixed with normal I-GET language statements. Any legal C / C++ statement can be defined as a hooked statement; the syntax for hooked statements, which is very simple, and examples, follow below; notice that the I-GET compiler does not append explicitly a semicolon, but extracts the statement which is defined between doubles quotes as it is (hence, you will have to supply a semicolon explicitly within the defined statement).

HookedStmt:

_c++ " <statement> " ;

```
_c++[
#include <stdio.h>
]_c++
void IGET_func () [
     int a;          // thats an I-GET variable.
     _c++ "int a;    // no conflict." ;
     _c++ "int b=a; // the C++ 'a'." ;
     _c++ "a++;" ;
     _c++ "fprintf(stderr,\"a is '%d'.\n\",a);" ;
]
```

Take care of hooked C / C++ statements including double quotes.

Since the hooked statements are written within double quotes, in case that you have to supply string constants within such statements, you will have to do the following:

firstly, write the statement as it would originally be in C / C++; then, substitute every appearance of " with \" in the statement, and finally surround it with doubles quotes. For example, the statement:

```
printf("\"Hello,world.\"");
```

will become:

```
_c++ "printf(\"\\"Hello,world.\\"\");" ;
```

7.3 Exporting I-GET statements within C / C++ code

Consider the following programming scenario: you write hooked code in C/ C++, and you want to call some I-GET statements. The first thing that you could try is to break the C / C++ code into multiple hook segments, according to the points in which the I-GET statements need to be placed, and then inject the I-GET statements as normal I-GET code, external, but in-between, C / C++ hooks. This idea, which we should mention will not work, is illustrated below.

```
// I-GET source file.
_c++[
void CPP_Function (void) {  // function starts here.
]_c++
// I-GET statements here, to be part of C++ function.
_c++[
// rest of CPP_function here.
}                           // function ends here.
]_c++
```

What is problematic with the above scenario ? The fact that I-GET language statements appear globally, since this does not comply with the I-GET language syntax. To solve this issue easily, the I-GET language provides a *statement qualifier*, through which it is allowed to have globally I-GET statements. This qualifier is **`export`**, while such statements are fully parsed and checked; all engaged variables must be visible at the point the statement is defined in I-GET file scope. Examples of such *exported statements* follow below.

```
_c++[
void F(int a, int b) {
]_c++
export [ int a; int b; a++; b=a; ]
export _c++ "if (a>b) a-=b; else a=b-a" ;
_c++[
printf("|a-b|=%d.\n", a);
]_c++
```

Since any I-GET statement can be exported, so do block statements, since they are also statements (see Section 13 - statements - in this Volume). Also, hooked C / C++ statements are I-GET statements, hence the second `export` simply adds a native C / C++ statement in the `F` function through I-GET code (alternatively, this statement could be directly placed within the hooked code). The first `export` statement does nothing useful, but only shows that no conflict of the I-GET variables declared will arise with those defined in C / C++ code, since the latter have been given names following the rules defined in Section 7.1.2. Next, we will discuss two special classes of I-GET statements which allow data exchange between the I-GET language code and C / C++ code in both directions.

7.4 Assigning C / C++ expressions to I-GET variables

This statement class addresses the issue of passing the result of arbitrary C / C++ expressions on I-GET variables. Such a statement is called a bridge. There is one type matching rule for this statement class:

Only some simple data types are allowed; take care of the regulations.

- The type of the I-GET variable receiving the result of the C / C++ expressions is allowed to be only of built-in data types or enumerated, except `sharedid` and `daid` which are not allowed. Pointers, structures, type synonyms and arrays are not allowed. The `objectid` type is allowed, however, the C / C++ expression must be of `int` type, the value of which must have been gained before through a bridge of the opposite direction (from I-GET to C/ C++, see next Section), by assigning an I-GET expression of `objectid` type to an `int` C / C++ Lvalue.

The I-GET compiler will only check the type of the I-GET Lvalue receiving the result; any faults on the type of the C / C++ expression will generate compile errors during the C++ compilation phase.

Attention on how exchange of enumerated type expressions is managed.

Regarding enumerated types, their definition must be part of the source code written in the I-GET language; the I-GET compiler will generate a C++ enumerated definition at the same source code point, so that you can use the defined enumerated types in C / C++ exactly as they are defined in I-GET code. The syntax of such bridge statements, and examples follow.

Bridge_CPP2IGET_Stmt:

Lvalue [<-] "<C / C++ expression>" ;

```
enum Days=[Sun, Mon, Tue, Wed, Thu, Fri, Sat];
void MixedFunction (Days* day) [
     _c++ "Days my_day=Mon;" ; // ok, 'Days' visible.
     *day [<-] "my_day";       // ok, safe conversion.
```

```
    int arr[10];
    _c++ "int arr[10];" ; // ok, no conflict.
    arr [<-] "arr" ; // error, arrays not allowed.
]
```

7.5 Assigning I-GET expressions to C / C++ variables

The opposite (or symmetric) of the previous statement class is to allow I-GET language expressions to be assigned on C / C++ variables. This is the second bridge statement class. The same regulations on data types hold, while `objectid` types will be automatically translated to `int` data type, on the basis of an internal I-GET registration mechanism; you should store this unique integer id, and use it when you have to assign back to an `objectid`. A special value `-1` indicates a `nil objectid`. The syntax, as well as examples follow.

Bridge_IGET2CPP_Stmt:

"<C / C++ Lvalue>" [<-] Expression ;

```
longreal sqrt (real num) [
        _c++ "float num;" ;
        "num=" [<-] num;
        longreal result;
        result [<-] "sqrt(num)" ;
        return result;
]
```

The previous example shows how to call C / C++ functions with I-GET actual arguments. Since the I-GET language does not provide any built-in mathematical functions, we should rely upon C++ for mathematical calculations. Firstly, we declare as many auxiliary C++ variables as the number of

arguments of the C++ function; for our example, the `sqrt` C++ function has only one parameter, hence, we declare a `float num` variable (notice that the name could be the same). Then, we define bridge statements to pass the actual arguments from the I-GET expressions to the auxiliary C++ variables (e.g. `"num=" [<-] num;` in our example). Finally, we call the C++ function, with the argument variables (e.g. `sqrt(num)` in our example) and if necessary, we pass the result (though an appropriate bridge) to an I-GET variable (e.g. `result` in our example). Notice that the C / C++ Lvalue receiving the result should include explicitly the = sign (also, type casting is allowed to be added for more safe conversions).

Calling I-GET expressions in C / C++ may not be so important for simple programming statements, however, it is valuable when it comes to special statement categories of the I-GET tool, such as controlling agents, changing attributes of interaction objects, manipulating objects in shared space, sending messages to the Application Component, etc. The reason that we have put so much emphasis in enabling a very strong mixing of C / C++ code with I-GET code, is for supporting dialogue developers in re-using existing C++ libraries for particular programming tasks, while building a dialogue control and coordination framework directly in the I-GET language. Next, we provide some examples which demonstrate the power of the hooks and bridges features of the I-GET language.

```
// I-GET source file.
int a, b;
_c++[ #include <stdio.h> ]_c++
a : [ _c++ "printf(\" a changed.\n\");" ; ]
b: [
   b [<-] "10" ; // no effect, since b's monitor.
]
```

```
struct Data [ string info; ];
_c++[
struct Data { char info[256]; } // replication needed.
extern void Store (Data&);
]_c++
void Store (Data item) [
     _c++ "Data item; char* info;" ;
     "info=" [<-] item.info;
     _c++ "strcpy(item.info,info);" ;
     _c++ "Store(item);" ;
]
```

Special attention when passing strings from I-GET to C / C++.

When assigning `string` type I-GET expressions to C / C++ `char*` variables, the null terminated character sequence is stored in a static character array, which is then passed to the C / C++ Lvalue. You should copy this string, if further processing is needed, since it will be overwritten by a subsequent execution of a similar bridge statement.

Chapter 8

8. Separate compilation in I-GET

The I-GET language is a 4GL for dialogue implementation. Usually, 4GLs for interface development provide various features for dialogue implementation as such, while little support is supplied for overall project development and re-usability, as it is common with mainstream programming languages. Separate compilation, though taken for granted for 3GLs, is a facility which is rarely met in 4GLs. The I-GET tool supports separate compilation, for the development of dialogue control components consisting of multiple modules, each separately compiled and tested. The approach in the I-GET tool is not as simple, intuitive and flexible as it is in the case of C / C++, however, it provides adequate features so that separate compilation can be realized in practice.

8.1 Qualifying code with *headers* blocks

Headers and implementations: the rue story from the C / C++ analogy.

In C / C++, the relationship between header files and source implementation files is not hard-coded in the language, but became a commonly applied convention to distinguish between sources of prototypes of constructs (i.e. light-weight constructs - headers), and sources of implementation of constructs (i.e. heavy weight constructs - implementations). The only built-in feature in the language has been the distinction between prototype / implementation and issue / storage for all entity classes. Hence, we have function prototypes and function implementations, class definitions and class implementations, extern variable declarations and real declarations, data type definitions

which posses no storage requirements, etc. Such a clear distinction exists for all kinds of language constructs in C / C++; this allows code to be written which provides the "protocol" for utilising implementations residing in other sources (i.e. header - implementation distinction). Such an approach resolves the necessity of explicitly qualifying code as being a "header" or "implementation" part, since the regulations for code generation clearly identify those types of constructs which are simply "issuing" the presence of some implementation, rather than causing an implementation to be code generated.

Explicit qualification of code segments in the I-GET language.

The I-GET language is not so powerful, with respect to the prototype / implementation distinction. More specifically, regarding the three basic class categories which can be constructed (i.e. agent classes, virtual classes and lexical classes), there is no explicit support for separating definition from implementation in these classes; a specification scheme is supported in which definition and implementation are defined together. In order to overcome all these barriers of the language for separate compilation, which have reflected some conscious design decisions having other (than separate compilation) objectives, the I-GET languages provides support for explicit qualification of code segments to force code generation to be disabled (i.e. no implementation in C++ is generated by the I-GET compiler for such code segments).

As a result, when qualifying this way particular I-GET language sources, which naturally include implementations of various constructs, their compilation will have two effects: (i) it will generate no code, hence, they will be treated as if they are lightweight constructs; and (ii) it will "issue" all such constructs, since they will be normally parsed by the compiler, so that they can be directly used from within other sources. The qualification of code as being light-weight code, i.e. appropriate for headers, is made possible by enclosing code within a special type of blocks, called *header blocks*.

Syntax for qualifying code with header blocks.

Header_QualfiedCode:

headers [***Code*** **]**

Code:

I-GET language legal code.

A simple example for qualifying code with header blocks.

```
// File a.gdd.
void AusefulFunc () [ /* implementation */ ]
struct ADataType [ /* fields */ ];


// File b.gdd (uses a.gdd).
headers [
#include "a.gdd"            // code included is header.
]
ADataType my_struct;        // use type.
void F() [ AusefulFunc(); ] // call function.
```

In the above example, the idea is that from within file *b.gdd* we want to utilise elements implemented as part of *a.gdd*. We directly include the whole implementation file, however, qualified with header blocks. The result is that during the compilation of *b.gdd*, all constructs from a.gdd will be just lightweight and will not cause any code generation. Notice that when *a.gdd* is included, the whole implementation of function `AUsefulFuc` is parsed, though presenting no problem since it will be simply ignored. Off course, since prototypes are supported for functions, we could make an approach similar to C in this case as follows:

```
// File a.hdr (header).
void AUsefulFunc();
struct ADataType [];
```

```
// File a.gdd (implementation).
#include "a.hdr"    // no headers, YES code generation.
void AUsefulFunc() []


// File b.gdd (client).
headers [
#include "a.hdr"    // headers, NO code generation.
]
ADataType my_struct;        // use type.
void F() [ AusefulFunc(); ] // call function
```

In the above example, we create a separate file, for more clean and understandable (to clients) header files, relieving from implementation details. Since the I-GET language is translated to C++, even data types will cause code generation; hence, when including a file which provides data types, unless it is surrounded with header blocks, some code will be generated. In our example, code generation is enabled for the original implementation file (i.e. *a.gdd* when including *a.hdr*), while for the client file (i.e. *b.gdd*) code generation is disabled.

Header blocks may be arbitrarily nested. This allows any file including header qualified blocks, to be also included also as being header qualified. In the example below, the file *b.gdd* is included from another file *c.gdd*, as being also header qualified.

```
// File c.gdd (client for b.gdd).
headers [
#include "b.gdd"
]
extern AdataType my_struct;
```

```
ADataType second_struct=my_struct;

void G() [ F(); ]
```

From the previous example, apart from directly including the *b.gdd* file, we notice that an extern definition of a variable declared within the b.*gdd* file is also provided. This is necessary if variables from other files need to be accessed; simply including the original variable declaration within header blocks will only issue the variable and enable it to be referenced. Even though this will compile successfully with the I-GET compiler, it will create problems to the C++ compilation of the generated for *c.gdd* file, since this will make reference to variable `my_struct`, though there is no `extern` declaration for that (the declaration within header blocks generates no code). Hence, an explicit `extern` declaration is needed, within the file referring to that variable. The same holds true also for functions; hence, in the first example, when directly including *a.gdd* within *b.gdd,* we should add an `extern` function prototype definition to ensure that the code generation of the *b.gdd* file would be now complete, with respect to making reference to externally (i.e. within *a.gdd*) implemented functions. This policy for data types, variables and functions should be always taken; the complete details for producing header files in the I-GET language are provided under Section 8.3 of this Volume.

8.2 Resolving dependencies among generated files

When including a file *X* with header qualification from a file *Y*, it means that external implementations from *X* are practically utilised by *Y*. For both *X* and *Y* files, the I-GET compiler produces a header and an implementation file (say *X.hh* & *X.cc*, and *Y.hh* & *Y.cc*). Normally, the *Y.hh* and / or *Y.cc* files will still make references to constructs implemented in *X.cc*; the prototypes of such constructs being defined within the file *X.hh*. For the *Y.cc* file to compile successfully in C++, it is necessary that the file *X.hh*, providing the

prototypes of constructs implemented in *X.cc*, is included by *Y.hh* and / or *Y.cc*. Hence, we should ensure that the header file *X.hh*, from which constructs are utilised, is included within *Y.hh* (this suffices, since it will then be included also by *Y.cc*). In the I-GET language, this dependency is resolved explicitly through header blocks:

```
// File Y.gdd.
headers [
#include "X.gdd"    // Y.gdd explicitly includes X.gdd
extern ["X.hh"]     // Y.hh explicitly includes X.hh.
]
```

As it is shown from the above example, the preprocessor `#include <file>` directive is employed to "issue" locally external constructs during the I-GET compilation phase, while the `extern [ <list of file names> ]` ensures that all files listed (here only *X.hh*) will be included from the generated C++ header file (here *Y.hh*) of the I-GET source file currently being compiled (here *Y.gdd*), during the C++ compilation phase.

The definition of a list of C++ header files through the extern [<file list>] list is mandatory within header blocks; empty list is not allowed, while all names supplied in the file list must be legal file names.

8.3 Making header files in the I-GET language

In this Section we will discuss the strategy for making I-GET header files, so that separate compilation can be supported, including a discussion on all relevant file relationships and compile dependencies.

8.3.1 Header files and separate compilation - the I-GET strategy

Header files in the I-GET language.

The I-GET header files play the role of enabling "client" modules to utilise constructs implemented externally, within other modules. The creation of an I-GET header file is based on the construction of a file which includes all the necessary information for using elements which are made open to other modules, however, by hiding implementation details. In contrast to C / C++ header files, the I-GET header files are made to be included only by "client" modules, and almost never (except from some cases) by the real implementation module(-s). In the previous Sections we have used the suffix *.gdd* for files to denote I-GET language compliant code (I-GET dialogue definition); this is not mandatory, though suggested for readability. Now we introduce another suffix, *.hdr*, which denotes I-GET special header files (again the suffix is not mandatory, but suggested).

Making and using I-GET header files: an overview.

Assume that there is a target module *b.gdd*, for which we want to enable other modules to use a particular sub-set of its elements. In order to do that, when the implementation of *b.gdd* is stabilized, we compile this file with the I-GET compiler. As a result, a C++ implementation- and header- files are generated; say that *b.cc* and *b.hh* are those generated files. Now, lets assume that there is a particular module *a.gdd*, which needs to utilise some of the elements that *b.gdd* makes available to external modules. Then, the *a.gdd* file should *include* the *b.hdr* header file at a point prior to any use of elements from *b.gdd* module. It is important to mention that this should be a standard preprocessor *include* command, and you should not surround it with header blocks (as we will explain later-on, header blocks will be embedded within the *b.hdr* file).

```
// a.gdd "client" file.

#include "b.hdr"

// Use elements from b.gdd beyond this point.
```

Each time the *b.gdd* implementation file changes, the *b.hdr* file will need to be changed, only if the *programming interface* of the various elements that

b.gdd makes available to external modules is modified. Also, as it has been previously mentioned, the *b.hdr* file is not normally "appropriate" to be included within the *b.gdd* file (we will discuss the exception to this rule in the next sub-section).

8.3.2 Constructing I-GET header files

Independently of the extensive discussion we provide to familiarize the reader with the separate compilation facility in the I-GET language, the process of creating I-GET header files is very simple. The centre of the process is the original implementation file, say *b.gdd* to comply with our previous examples, while the outcome is the header file, say again *b.hdr*. For each element of *b.gdd* which you want to make available to other modules, you will have to do the following: (i) identify the *element category*; (ii) identify the *form* of this element category, when engaged in header files (we provide a table below, showing all such forms for all element categories); (iii) identify whether the form of the target element needs to be *included with header blocks*; and (iv) write the element form in the header file, placing header blocks if required (in the latter case, you should also add an `extern ["b.hh"]` to resolve the dependency between the *a.cc* "client" C++ generated module and the C++ generated module *b.cc* for *b.gdd*).

In the table below, we provide the specific form with which an element should appear in a header file, for all element categories; also, the necessity or not of surrounding the element form with header blocks is indicated.

Element category	Form of inclusion in header	Qualify with header blocks ?
• Function.	Function prototype with `extern` qualification.	NO.
• Variable.	`extern` qualified variable declaration.	NO.
• User-defined data type.	Data-type definition as in original file.	YES. `extern[]` added.
• Parameterized agents classes. *When "client" needs to only create / destroy agent instances.*	Agent prototype with parameters.	YES. `extern[]` added.
• Lexical classes. • Virtual classes. • All agent classes. *The "client", apart from creating instances, needs to access all public members via instance references.*	• Copy the original class definition. • Remove all private members - *optional*. • Remove all those elements which cannot be externally referenced (monitors, constraints, event handlers, callback implementation) - *optional*. • Remove the implementation of member functions, and put empty blocks for constructors and destructors - *optional*.	YES. `extern[]` added.

In the above table, if with the definition of the target form of elements for inclusion in header files the word *optional* is attached to a particular action, it means that this action need not be necessarily taken. For instance, when "transforming" particular classes for header files, it is allowed to simply copy-paste the class definition to a header file, and only surround it with header blocks; this is possible, since all rest actions for further processing of class definitions are completely optional. However, you may want to follow the suggested actions, since they will result in simpler class forms by removing unnecessary implementation details. The rule defining whether the resulting header file can be included or not from within its original implementation file is very simple:

If the header file includes header blocks, it is not allowed to be included by the original implementation file, else it is allowed.

8.3.3 Example for creating and utilising I-GET header files

The examples which will follow do not concern I-GET class element categories (i.e. agents, lexical objects and virtual objects). Examples for making and using header files for the various I-GET class categories are discussed in detail within their respective Sections. The example below shows: (i) how we create a header file *b.hdr* from an implementation file *b.gdd*; and (ii) how we include the header file from a "client" file *a.gdd*, and utilise elements implemented in *b.gdd*.

Any b.gdd file may implement many more elements than those decided to be made available to external modules.

```
// File b.gdd.

int a, b;

local string d;

enum Days=[Sun, Mon, Tue, Wed, Thu, Fri, Sat];
```

```
void Command (int argc, string* argv) [
     // Implementation here.
]
void F() [ /* Implementation here. */ ]
local void G() [ /* Implementation here. */ ]
```

The header file will normally consist of extern qualified variables & functions, and header blocks for classes & data types.

```
// File b.hdr (a header file for elements from b.gdd).
extern int a, b;
headers [
extern ["b.hh"]
enum Days=[Sun, Mon, Tue, Wed, Thu, Fri, Sat];
]
extern void Command (int argc, string* argv);
```

Even though a conventional include directive is provided, the header blocks are still defined, since they have been now moved (for readability) within header files.

```
// File a.gdd (using b.gdd elements via b.hdr).
#include "b.hdr"          // a normal include.
local Days my_day;        // Days from b.gdd.
local void F() [
      int i=a, j=b;       // the a and b from b.gdd.
      string args[3]=["cpp", "-C", "-P"];
      Command(3, args);   // again from b.gdd.
]
```

Chapter 9

9. Lexical interaction objects

The lexical level of interaction is the physical interaction layer, where all interaction primitives have a particular physical substance. This not only concerns entities with a well defined physical space such as interaction objects, but also entities with a primarily temporal nature such as events, notifications and requests. The I-GET language supports three categories of lexical interaction elements: (a) interaction objects, which are discussed in this Section; (b) input events, which are discussed under Section 12; and (c) output events, which are also discussed under Section 12. Lexical interaction objects can be defined in two ways:

- As *lightweight objects*, in which case their specification primarily includes structure and properties, rather than functioning and behaviour. Such lexical objects gain a real implementation by connecting to a particular toolkit, while their specification is primarily an object class programming interface, which does not include dialogue implementation.

- As *heavyweight objects*, in which case their structure, properties, behaviour and user-dialogue are all implemented explicitly in the I-GET language. In other words, new lexical interaction objects will result in this case.

9.1 Explicit toolkit qualification

The I-GET language allows multiple sets of interaction elements to be manipulated at once. Such sets of interaction elements result by grouping elements together under the umbrella of a common *toolkit name*; in fact, any interaction element defined in the I-GET language must be explicitly associated to a particular toolkit name. Normally, such a toolkit name will refer to a real underlying toolkit (the name may clearly indicate the logical connection), which is integrated within the I-GET tool, in order to provide the implementation of some, or all, of those lexical interaction elements specified, which are associated with its name.

Before associating a particular interaction element with a toolkit name, the toolkit name must be first *issued*. Issuing a toolkit name is done at file scope with the following definition:

lexical (<toolkit name>) ;

Examples of issuing toolkit names locally (i.e. does not affect toolkits as such) within the I-GET language.

```
lexical (Xaw);      // local name for Athena widget set.
or
lexical (Athena);   // could choose another local name.
lexical (WINDOWS); // name for WINDOWS object library.
or
lexical (WIN32);    // that could be another name.
lexical (Xaw);      // no problem with multiple defs.
```

After defining toolkit names as above, any interaction element specified needs to be associated explicitly with such a pre-defined toolkit name. All lexical interaction elements associated with the same toolkit name *A*, will be said to belong in the same toolkit *A*. At the impementation level, there should be some kind of an underlying toolkit *A* connected to the I-GET tool (via a toolkit server - see *Volume 4, Integrating Toolkits*) which will provide

the physical implementation of all lightweight lexical interaction objects which, at the specification level, belong to toolkit *A*. You may even define heavyweight lexical objects (i.e. providing their own dialogue implementation) which are still associated with toolkit *A*; you may want to do that if the dialogue implementation of those heavyweight objects, which you explicitly construct through the I-GET language, utilises elements belonging to toolkit *A* (hence, it can be considered as an extension to the original toolkit *A* elements).

9.2 *export* and *noexport* categories of lexical objects

There two categories of lexical interaction objects. The **export** category, which is the default unless otherwise specified, and the **noexport** category. The latter indicates a lexical interaction object class which is completely invisible outside the Dialogue Control component and is hidden from the underlying toolkit with which it is associated; **noexport** objects will be usually heavyweight objects providing their own local implementation. The former, concerns lightweight lexical object classes, whose physical implementation is provided by the underlying toolkit (through toolkit integration) in which they belong. A lexical interaction object implementation has the following syntax (we omit the details of class body definition for clarity):

The header of a lexical class definition.

LexicalClass:

ExportQualifier lexical ID^{name} **(ID**^{toolkit} **) LexicalClassBody**

ExportQualifier:

export or

noexport or

empty

Some examples showing how the definition of lexical interaction objects looks like (only class headers defined).

```
lexical Command (Xaw) [        // default is export.
// class body here (lightweight).
]
noexport lexical Gauge (Xaw) [ // a new object for Xaw?
// class body here (heavyweight).
]
export PushButton (WINDOWS) [  // explicit export.
// class body here (lightweight).
]
```

In the above examples, we have also indicated which objects are expected to provide their own local implementation (i.e. heavyweight - `Gauge`), and those which are expected to have an implementation provided by the underlying associated toolkit (i.e. lightweight - `Command`, `PushButton`). In the I-GET language, there is no differentiated syntax among lightweight objects and heavyweight objects; their names simply indicate the different specification / implementation strategies. There is a single lexical class definition syntax, supporting a comprehensive range of constructs, some of which are more implementation oriented; those are expected to be mainly utilised for heavyweight **noexport** objects, rather than for lightweight **export** objects. In the next sub-section, we continue by discussing the lexical class definition structure and present various examples.

9.3 Specification structure and examples

The class definition body for lexical interaction objects is defined as a block starting with of an opening `[` and ending with closing `]` square bracket. In contrast to C++, the class definition body for lexical objects (and also for all other class categories of the I-GET language) should include the implementation of all members. Hence, for instance, the implementation of member

functions is also included within the lexical class body. From the syntactic point of view, it may look like in-line member functions in C++, however, the functions are not made in-line upon code generation. This scheme has been decided for many reasons. Firstly, we wanted to gather all implementation aspects for lexical objects (and for all rest class categories as well) into a single logical unit. In C++ this unit is a file, while in the I-GET language it is the class body itself. Secondly, we aimed to support various declarative constructs such as monitors, constraints and event handlers, within lexical object classes. Putting them directly into the class body is more natural and easier to control (i.e. context and related variables are directly visible). In order to avoid a hybrid syntax for classes, in which some of the constructs could be defined inside- while others outside- the class body, we have finally considered as more appropriate to support all definitions within the class body.

The implementation and definition of all member elements of lexical classes must be defined within its class body, as opposed to definition / implementation physical separation in C++.

The class body of a lexical class (and also of all other class categories in the I-GET language) should always end with a constructor `constructor […]` and a destructor `destructor […]` blocks, in this order (empty blocks `[]` are naturally allowed). Within the body of lexical classes, various instances of language constructs may optionally be defined. Hence, the simplest lexical class definition would be (assuming a toolkit name `MyToolkit` has been issued prior to the point of the class definition):

Mandatory constructor and destructor blocks at the end.

```
lexical SimplestLexicalClass (MyToolkit) [
        constructor []
        destructor []
]
```

An example of a minimal lexical class definition.

Next we will provide the full list of all various constructs which can be defined within lexical object classes, and we will discuss each such construct in detail. The syntax for the class body of lexical classes follows.

Grammar for body of lexical object classes.

LexicalClassBody:

[

(**LexicalClassConstruct**)∗

constructor Compound

destructor Compound

]

LexicalClassConstruct:

VariableDeclaration *or*

MethodIssue *or*

ExportQualication *or*

ScopeQualification *or*

MethodImplementation *or*

Monitor *or*

Constraint *or*

FunctionPrototype *or*

FunctionImpl *or*

UserDefinedDataType *or*

EventHandler

9.3.1 Variable declarations and method issue: the *toolkit interfacing contract*

Local variables: attributes of interaction objects.

Variables can be normally defined, supporting also initializers to be supplied. Storage qualification (i.e. use of `extern` or `local` qualifiers) is not allowed, since it can be only applied to file scope declarations. When variables are defined in the context of lexical classes they are called *lexical attributes*. The role of attributes is very important in the context of toolkit integration. When an **export**ed lexical class is defined, the toolkit server,

which is responsible for physically instantiating such lightweight objects (lightweight in the context of the Dialogue Control component), will have to maintain a consistent mapping among the attributes of the lexical object instances (residing in the Dialogue Control component) and the attributes of the corresponding physical object instances (residing in the underlying integrated toolkit). Hence, in case of exported lexical classes, the object attributes defined play also the role of a contract with the toolkit server, through which toolkit interfacing is performed, when integrating a particular toolkit.

Method issue: defining call-back categories.

A similar role has been also defined for methods. Methods in the I-GET language are synonymous to call-back categories in interface toolkits. For instance, one call-back category for `Button` objects could be called `Pressed`. Call-back categories are typically implemented in toolkits as call-back lists, allowing programmers to register various functions within such a list. When during interaction a call-back category is *notified*, it means that the logical action indicated by the call-back category name (e.g. `Pressed`) has just taken place (e.g. the user has pressed the button). The result is that all registered functions are called by the toolkit run-time library. The I-GET language supports such a similar scenario, enabling to define the various call-back categories during lexical class definition.

A method issue is simply the definition of such a call-back category name (like `Pressed`); as it will be explained later on, developers may implement methods for lexical object instances (i.e. call-backs) by attaching code to such method categories. The toolkit server will have to notify the lightweight object instances at the Dialogue Control component for such methods, each time their corresponding physical instances are also notified. The syntax for issuing methods (i.e. *lexical methods*), and an example of a lexical class definition including declaration of attributes and issuing of methods follows.

MethodIssue:

method Id[method name] **;**

Examples for attribute declaration and method issue within a lexical class definition.

```
lexical Command (Xaw) [
        word x=0, y=0, width, height, borderWidth=1;
        string fg="black", bg="white";
        string borderColor=bg, label="Command";
        bool roundCorners=false;
        method FocusIn;
        method FocusOut;
        method Activated;
        constructor []
        destructor []
]
```

9.3.2 *export* / *noexport* qualification for attributes

In some cases, even when defining **export** type categories of lexical interaction objects, you may desire to leave some of the object attributes completely hidden to toolkit servers. This is naturally a task that you will be concerned with only if you will be engaged somehow in toolkit server development (hence, you will also have to study *Section 3 - Toolkit Integration*). Why you want to hide certain attributes from the toolkit server, even if the toolkit server should not be responsible for such attributes ? couldn't the toolkit server simply ignore these attributes ? The answer is that you could just have the toolkit server ignore any messages for changing attributes for which it is not responsible; but this could still consume valuable network traffic, in the channel of toolkit interfacing. To avoid that, we support the explicit qualification of attributes of exported classes as `noexport`, if these attributes for some reason cannot be managed by the toolkit server (e.g. a premature im-

plementation of the toolkit server not yet supporting the full range of attributes in keeping consistency mapping; an attribute of the lexical class not originally met in the corresponding native object class of the integrated toolkit).

By default, all attributes gain the export qualification of their class. Hence, if a class is of `export` type, unless otherwise specified, all attributes will be `export`; similarly, the default qualification of attributes for `noexport` classes is `noexport`, while in this case trying to qualify some attributes as `export` generates a semantic compile error. Export qualification has the following syntax:

ExportQualification:

export: *or*

noexport:

After an `export:` or `noexport:` definitions, and until the next (if any) scope qualification within the same class, all variables gain the `export` or `noexport` qualification respectively.

Why not an explicit export qualification for methods ?

One question arising from the discussion on explicit export qualification is "why export qualification does not apply to methods as well ?". The answer is "because saving network traffic due to methods which are not supported by native object classes is straightforward". Notification for methods takes place in one direction at run-time, from the toolkit server to the Dialogue Control component. Hence, the toolkit server will have to either disable such notifications (if method defined does map to a physical call-back category, but we want to exclude notifications coming from the physical objects), or simply will cannot send such notifications (if the corresponding physical classes do not support a call-back category mapping to an issued method).

```
lexical Command(Xaw) [
        noexport:
        word timesSelected=0;
        string taskAssociated="UNKNOWN-TASK";
        export:
        // normally exported attributes, and
        // rest of class definition body goes here.
]
```

Examples of attributes which are not supported by the native physical classes and need to be defined as no-export.

9.3.3 *export* qualified attributes should be of *export* enabled data types

There is an additional rule judging if qualifying an attribute as **export** is legal According to this rule, and **export** qualified attribute sould be of an export enabled data type. From the built-in data types, the following are **export** enabled directly:

int	**longint**	**word**
longword	**real**	**longreal**
char	**string**	**bool**
objectid		

When data types are defined within lexical classes, they gain the export qualification and scope qualification (i.e. `public:` or `private:` - will be discussed within next sub-section) applicable at the point of definition. Regarding attributes of user-defined data types, either of the following should hold for the particular user-defined data type, so that the attribute can be defined as **export** qualified; in all the following cases, user-defined data types become **export** enabled.

- *The data type is globally defined (file scope).*
- *The data type is defined within agent- or virtual- classes with public access scope.*
- *The data type is defined within a lexical class at a point where a **public** access scope and **export** qualification apply.*

An example illustrating the symphony that is required for export qualification between attributes and their data types.

```
struct Font [
      int pts;
      string Family;
      bool underline, bold, strikeout;
];
lexical AnyClass (MyToolkit) [
      // default is export, since class is exported.
      Sharedid sharedObj;    // error-1.
      private:
      enum ShapeStyle=[Oval, Rect, RoundRect];
      public:
      ShapeStyle shapeStyle; // error-2.
      Font font;             // ok.
constructor []
destructor []
]
```

error-1: 'export' disabled type for 'export' qualified attribute.

error-2: 'private' data type for 'export' qualified attribute.

We provide another example, which also shows how data types defined

within lexical classes (called the type-owner classes), with public access scope, can be employed for declaring variables externally to their owner class. In the I-GET language, data types are subject to scope rules; hence, you may declare the same data type in different classes without any name collision problem, while you may also disable external access to a defined data type by adding a private scope qualification (more on next sub-section on scope qualification). In the example below, we skip the standard constructor and destructor blocks for clarity, though they are mandatory.

```
lexical OneClass (MyToolkit) [
      public:
      struct Point [ word x, y; ]; // ok, public type.
      private:
      enum MoveStyle=[ByMouse, ByShortcut, NotEnabled];
]
lexical OneClass (AnotherToolkit) [ // no collision.
]
lexical::OneClass(MyToolkit)::Point my_point;
lexical AndAnotherClass (MyToolkit) [
      export: public:
      struct Point [ real x, y; ]; // ok, public type.
      lexical::OneClass(MyToolkit)::Point pos1; // ok
      Point pos2; // ok
      lexical::OneClass(MyToolkit)::MoveStyle move;
]
// The underlined line generates an error: type
    'MoveStyle' in 'OneClass' for 'MyToolkit' is pri-
    vate.
```

In the above example, several observations can be made. Firstly, it is shown that two lexical classes may share a name, as far as they belong to different

toolkits (e.g. `OneClass` belonging to `MyToolkit`, and `OneClass` belonging to `AnotherToolkit`). Secondly, when a data type is defined within a lexical class, accessing the data type from outside that class requires an explicit definition of: the class category (e.g. `lexical` for lexical classes), the object class (e.g. `OneClass` for `(MyToolkit)` in our example) and the data type (`Point` and `MoveStyle` in our example). Hence, the expressions `lexical::<class name>(<toolkit name>)::<typename>` can be used to access data types within class `<class name>` belonging to toolkit `<toolkit name>`.

The fact that data types are subject to local class scopes is also shown. A `Point` structure type is defined both within `OneClass` and `AnotherClass`, without any type conflict problems. Also, within `AnotherClass`, one variable of the `lexical::OneClass(MyToolkit)::Point` is declared, and another variable of the local `Point` data type as well, showing how type resolution is easily achieved.

The previous example generates one semantic compile error, due to the declaration of a `lexical::OneClass(MyToolkit)::MoveStyle` variable called `move`, since the data type is privately defined. If we change the type access scope to `public:` and the export qualification to `noexport:`, again a semantic compile error will be caused, since now the data type being **export** disabled, cannot be used for an **export** variable.

9.3.4 Scope qualification for class members

There are two scope categories within lexical classes (and also all other class categories in the I-GET language): **public** scope, which allows access to member elements (those which can be referenced via an identifier) from outside the owner class, and **private** scope, which restricts access to those members only from within the owner class. Public scope is defined through

Two scope qualifiers, private: (the default) and public:.

the `public:` scope qualifier, while private scope is defined through the `private:` qualifier; the default class scope is private (i.e. all class members, starting from the beginning of class definition, and until a scope qualifier is met, gain private scope. Scope qualifiers are applied to the following member element categories:

Member element categories on which scope qualification is applied.

- Variables.
- User-defined data types (user means programmer here).
- Functions.

Variables with private: scope are not allowed to have export: qualification - though this rule can by bypassed in header files.

When a variable is of `private:` scope, it is not allowed to have `export:` qualification. The idea is that for lightweight objects, all lexical attributes which are exported (hence, they are maintained at the corresponding instances by an appropriate toolkit server) must be also accessible to programmers outside the class scope (which is not the case if those are made private). If, however, for some reason, particular exported attributes need to be hidden to programmers, then this can be done when creating the header file in which the class definition will be placed (see Section 8 - Separate compilation); simply those attributes are removed, or even declared as private (the header files of the I-GET tool are not compiled to construct any memory model for classes, but simply provide a member access contract based on naming and data typing). Examples for scope qualification in lexical classes follow below.

```
lexical OneClass (MyToolkit) [
        // export: and private: are the default.
        int a, b;  // error-1.
        public:
        string c;  // ok, no conflict now.
```

```
        noexport:

        string d[10];     // public, but not exported.
        private:
        longreal e;       // private and not exported.
        void F (int* i); // same as 'e'.
        struct PrivateType [ string a; ];
        public:
        void F (int* i) [ i++; ] // error-2.
        PrivateType publicAttr;  // error-3.
]

error-1: 'a' and 'b' are 'private' and also 'export',
which is not allowed.

erorr-2: 'F' defined with a 'private' prototype and a
'public' implementation.

Error-3: 'publicAttr' which is a 'public' attribute, if
of 'private' type 'PrivateType'.
```

From the previous example, some interesting features and rules are illustrated, related to scope access.

- Firstly, it is allowed to define prototypes for member functions, separate from their implementation definition. This is good if you want to logically split the class body in two parts: (i) one mostly related to definitions and prototypes; and (ii) another related to implementation aspects. In this case, the scope of the prototype and the implementation should match, or otherwise a compile error will be generated (`error-1`).

- Secondly, all public attributes should be of public data types; all built-in data types are public. In the previous example, the public attribute `publicAttr` is defined to be of type `PrivateType`, a `struct` type which has been defined in private scope; this causes a conflict which generates a

compile error (`error-3`).

- Finally, when an `export:` qualified class is defined, the default qualifiers are `export:` and `private:` at the beginning of the class body, if not explicit qualifiers are supplied. However, since these two qualifiers conflict, any attribute or data type that you declare with this status will cause a compile error. For instance, in the previous example, the declaration `int a, b;` at the beginning of the class, declares two both **export**ed and **private** variables, which is not allowed (`error-1`).

9.3.5 Method implementation and artificial method notification

Method implementation means making one or more registrations of implementation blocks for a particular call-back category. It is also known as call-back implementation. In the I-GET tool, it is possible to provide multiple concurrently active implementations for a particular call-back category, with registrations done at different run-time points; we will discuss later in this sub-section the issue of multiple registrations. Lets study the syntax of method implementation within lexical classes.

Firstly, we expect to implement methods for a particular object instance of a lexical class outside the class body. This is natural since call-back registrations are done at the object instance level. Then why is method implementation supported in the body of lexical classes ? The answer is “for two reasons”: (a) to enable default code to be executed, if required, upon method notification; and (b) methods implemented in the body of a lexical class may not necessarily belong to the owner class, but can register methods for any lexical object instance. Implementing methods for lexical object instance is very simple, and has the following syntax:

MethodImplementation:

method ObjRef . ID$^{\text{method name}}$ **Compound**

ObjRef is a grammar symbol introduced in Section 4 - Expressions, and concerns all types of expressions which represent object instances. Firstly, we will provide examples of implementing methods of the owner lexical class. Access to the object instance from within the lexical class is done via the `{me}` expression, which is analogous to `this` in C++.

```
lexical Command (Xaw) [
      int pressCounter=0;
      method Pressed;         // issuing a method.
      method {me}.Pressed [  // implementing a method.
            printstr("I am pressed.\n");
            pressCounter++;
            printstr("Press No " + pressCounter + '\n');
      ]
      method {me}.Pressed [
            printstr("I am impressed !\n");
      ]
      method {me}.Pressed [
            printstr("Now I am depressed.\n");
      ]
]
```

In the previous example, it is shown how a method implementation is provided in the I-GET language. Access to all members visible at the point of method implementation is allowed, from within the implementation block (e.g. accessing `pressCounter` in our example). The previous example illustrates also the capability of multiple registrations. The regulations for

multiple method registration in the I-GET tool follow below.

Multiple registrations on the same method: regulations.

*Methods are registered at run-time, for their respective object instances, in the exact order of definition. When multiple implementations are registered on an object **O** for the same method **M** at run-time, then, upon notification for method **M** for object **O**, all registered methods are sequentially executed, following the ascending order of registration.*

Hence, when defining a lexical class, by providing default implementations for some of, or all, the methods of this class, you can be sure that at run-time, upon method notification, the default implementations you have supplied within the class body are always the first to be executed. he I-GET language supports a statement for generating artificially method notifications for any object instance. This statement class is described in more detail under Section 13 - Statements. We provide the syntax here as well, with some simple examples for lexical object classes.

Artificial method notification.

ArtificialMethodNotification:

ObjRef -> ID$^{\text{method name}}$ **;**

```
noexport lexical Gauge (Xaw) [

      method FocusIn;

      method FocusOut;

      method Turned;

      void DisplayGauge();

      void GaugeDialogue () [

            if (<user action to turn gauge ok>)

                  {me}->Turned;
```

```
        ]
]
```

In the previous example, a non-exported (i.e. heavyweight) lexical class is defined, called `Gauge` (assuming for the `Xaw` toolkit). When you will implement such classes, normally you will supply your own object display logic and dialogue implementation. The scheme described in the previous example is very simplistic, and you will normally employ other types of constructs for dialogue implementation (like event handlers), however, it demonstrates the idea of when artificial method notification is to be applied. So, at some point in the physical dialogue implementation, you will detect that the user actions have reached a point, causing the logical action of "turning" the `Gauge`. Hence, you will need a way to generate a notification to the particular `Gauge` instance, so that all implementations registered for the `Turned` method are executed. This is made possible through the `{me}->Turned` statement.

Artificial method notification is particularly useful when you will implement non-exported heavyweight object through the I-GET tool.

9.4 Instance declaration and member access

Creating instances of lexical object classes in the I-GET language is allowed only within agent classes, and only in class scope in the form of declarations (i.e. instantiating lexical objects is not allowed directly via statements). We will present the style for *declaring lexical object instances*, and we will show how members can be accessed via such instances.

LexicalObjectInstance:

lexical (ID [toolkit name] **) ID** [lexical class] **ID** [instance name] **OptionalParent ;**

OptionalParent:

: parent = Expression [objectid type]

Grammar for declaring lexical object instances.

```
lexical(Xaw) ApplicationWindow a_win;
```

Examples of declar-

ing lexical object instances.

```
lexical(Xaw) Box box :parent={a_win};
lexical(Xaw) Command quit :parent={box};
lexical(WIN) FrameWindow f_win;
objectid my_id={f_win};

lexical(WIN) PushButton hello :parent=my_id;
void F() [
      lexical(Xaw) Listbox l_box; // syntax error.
]
```

Explanation of the object declaration syntax.

The declaration syntax is simple. Firstly it requires *toolkit resolution*; hence, `lexical(Xaw)` indicates that an instance of a lexical class belonging to the `Xaw` named toolkit will be declared. Secondly, the lexical class name follows with the instance identifier. Hence, `Command quit` indicates that the class is `Command`, while the instance variable will be called `quit`.

Passing parent objects for creating object hierarchies.

Finally, the declaration may either terminate at that point with a semicolon `;`, or continue with the provision of a parent object. Normally, top-level containers, such as windows providing window-management facilities, will be *parentless,* while all other objects classes will require an explicit parent to be supplied (this depends on what the toolkit, or your object implementation requires). The parent object is defined as an `objectid` expression type. We recall that `objectid` represents a generic type for lexical object instances; if `obj` is an instance variable (as `win`, `box` or `quit` above), then `{obj}` is of `objectid` type. In the previous example, supplying a parent object within the declaration of the `quit Command` object is defined through `:parent={box}`, which means that the parent of the `quit` instance is the `box` instance (the expression `{box}` as standalone is of `objectid` type).

9.4.1 Accessing member attributes

Attributes of a lexical class can be accessed via instance variables only if they have public scope. The syntax for accessing attributes, with corresponding examples, follows.

MemberAttribute ^{Lvalue}**:**

{ ID $^{instance\ variable}$ **} . ID** $^{attribute\ variable}$

Assignment.
```
{quit}.x=10;
```

Assignment.
```
{quit}.label="Quit";
```

Constraint.
```
{quit}.y := {ok}.y+{ok}.width+SPACE;
```

Monitor.
```
{quit}.y : [ printstr("'quit' button moved.\n"); ]
```

Assignment.
```
{quit}.label=readstr();  // label from terminal input.
```

Expression
```
( {quit}.y + {quit}.width ) / 2.0
```

In the examples above, we have considered the `Command` sample object class for `Xaw` toolkit, previously defined in this Section. Notice that the above code segment is not a single syntactically correct code segment; it simply gathers together some constructs which demonstrate attribute access. Object attributes are just variables in the I-GET language, and may be subject to any programming manipulation applicable to variables.

Lexical object instances are not Lvalues.

It should be noted that instance variables resulting from object declarations are not actually "variables", in the sense that they could be assigned with different instances; they are actually constants. For instance, the `quit` object instance identifier cannot appear alone on the left side of an assignment, while you cannot gain the address of `quit` object, attach a monitor to it or even constraint it; in other words instance identifiers are not Lvalues. As it will be made clear in the next sub-section, there is a special reference variable category for object classes, which can be used exactly as real object instances to access members, and may be assigned with object instances.

Built-in attributes of lexical object classes.

Apart from user-defined attributes, there are some built-in attributes for lexical object classes in the I-GET language; these attributes are read-only, hence you cannot assign values to them. Their names, how they can be referenced, and their data types are described in the following table.

Attribute name	Access syntax	Data type
parent	{*<obj>*}.parent	objectid
myagent	{*<obj>*}.myagent	daid

9.4.2 Calling member functions

Member functions of a lexical class can be accessed via instance variables only if the functions have public scope. The syntax for accessing attributes, with corresponding examples, follows.

MemberFunctionCall:

{ID *instance variable* **} -> ID (ExpressionList** *actual args* **)**

```
lexical OneClass (MyToolkit) [
      public:
      void F (int* a) [ (*a)++; ]
      string H (real a) [ return a; ]
      objectid MyId () [ return {me}; ]
      private:
      void G() []
      constructor [ G(); ]  // ok, local call.
]
lexical(MyToolkit) OneClass inst;
int a=10;
```

```
{inst}->F(&a);   // ok, 'a' becomes 11.
{inst}->H(3.14); // ok, result ignored.
objectid id={inst}->MyId();
{inst}->G();     // error, private function called.
```

In the previous example, examples of calling member functions are shown. It should be noted that the `export:` and `noexport:` qualifiers have no effect on functions. Hence, you may define and implement as many member functions you want within lexical classes, without affecting the toolkit interfacing status if those lexical classes are physically implemented by an underlying integrated toolkit.

9.5 Reference variables for lexical object classes

As it has been previously discussed, when declaring lexical object instances, the instance identifiers do not actually represent variables, but are constants. Hence, throughout your implementation, an instance identifier is constantly attached to a single object instance. To enable manipulate object instances with variables, the I-GET language provides the lexical object reference type; you can declare variables and even make type synonyms on the basis of this type. The syntax for defining an object reference type follows with some declaration examples.

LexicalReferenceType:

ref lexical (ID *toolkit name* **) ID** *class name*

```
ref lexical(Xaw) Command cmmd_ref;
ref lexical(MyToolkit) OneClass oneClass_ref;
type ref lexical(Xaw) Command CommandRef;
CommandRef cmmd_ref2, *cmmd_ref_ptr;
```

Lexical references can be assigned to / from: (i) lexical references of only the same lexical class; (ii) `objectid` type expressions; (iii) `nil`. The `objectid` type is a super-class type for all lexical references. Hence, if you store a lexical reference to an objectid variable, you can then assign back to the original lexical reference type and use it with no problems. You may also assign the `objectid` of an object instance (via the `{ }` operator) to an object reference, and use the reference to access member elements. Examples of using lexical references follow.

```
lexical OneClass (MyToolkit) [
      public:
      int x, y;
      string label;
      void Redraw();
      void Reshape();
]
lexical(MyToolkit) OneClass inst;
ref lexical(MyToolkit) OneClass my_ref={inst};
objectid id1={inst}, id2=my_ref;
id1==id2;   // true, both equal to {inst}.
{my_ref}.x=={my_ref}.y=10;
{my_ref};   // 'objectid' type for my_ref, i.e. inst.
{id1}->Redraw();        // error, not a ref type.
{my_ref}->Redraw)();    // ok now.
ref lexical(MyToolkit) OneClass secnd_ref=id1;
{secnd_ref}->Reshape(); // ok, 'id1' is 'inst'.
```

In the previous example, we demonstrate the key aspects of using lexical references. Firstly, you may think of lexical references as a built-in pointer

type. Using lexical references un-initialized will certainly generate undesirable run-time errors (try it and use the `-dbg` I-GET compiler flag, through which the source of a run-time error is also indicated). Secondly, the only way that instances can be directly engaged in an expression as such (i.e. without a member access) is via the `{ }` operator which returns the `objectid` value; such a value can be assigned to an object reference or to a conventional `objectid` variable safely. Thirdly, lexical references have the `objectid` type as their super-type; in this sense, expressions of lexical reference type are also allowed to be assigned on C++ variables, through bridge statement classes. When assigning an object instance, or a lexical reference to an `objectid` variable, the class information is lost. Hence, if from the program code it is not evident the original class (as it was with the example above), you may need to save class information together with the `objectid` value. The following example illustrates this issue.

```
struct ObjInfo [ objectid id; string class; ];
ObjInfo objects[MAX_OBJS];
int curr=0;
void InitObjects () [
      int i;
      for (i=0; i<MAX_OBJS; i++) objects[i]=nil;
]
void AddObject (objectid id, string class) [
      if (curr==MAX_OBJS-1) return;
      objects[curr].id=id;
      objects[curr++].class=class;
]
lexical(Xaw) ApplicationWindow win;
lexical(Xaw) Box box :parent={win};
```

```
lexical(Xaw) Command quit:parent={box};


AddObject({win}, ”Application Window”);
AddObject({box}, ”Box”);
AddObject({quit}, ”Command”);
void ProcessObjects() [
      int i;
      for (i=0; objects[i]!=nil; i++)
            case (objects[i].class) [
            “Application Window”: [
            ref lexical(Xaw) ApplicationWindow obj;
            obj=objects[i].id;
            ]
            “Box” : [
            ref lexical(Xaw) Box obj=objects[i].id;
            ]
            “Command”:[
            ref lexical(Xaw) Command obj=objects[i].id;
            ]
            else  printstr(“No served class.\n”);
            ]
]
```

In the previous example, `objectid` values are stored together with string values; the latter carry the class name, while the former carry the super-class instance pointer. We use a *case* statement (see Section 13 - Statements) to dispatch processing of object instances depending on their respective class (`case objects[i].class`); within each specific case block, a class-specific lexical reference (called `obj` in all cases) is declared through which

the particular lexical instance may be further manipulated.

9.6 Adding lexical classes within I-GET headers

Assume that you have defined a number of lexical classes, either serving as a toolkit interface specification for toolkit integration, or encompassing new interactive behaviours not met in an underlying toolkit. Then you will normally need to utilise such lexical classes in the context of other modules, for creating and manipulating instances. In order to do that, you will have to create an I-GET header file, in which it is minimally required to paste all the publicly accessible members which you need to reference, for each lexical class. We will clarify more this issue with examples.

As part of the I-GET software release, there are already two lexical sets of interaction elements, each including a number of lexical classes. These sets have been primarily employed for toolkit integration of the WINDOWS object library (WINDOWS 32-bits) and of the Xaw Athena widget set (UNIX variants). Regarding the WINDOWS integrated toolkit, the specification of the interaction elements is provided within the file `win.gdd` (see Volume 2 - Installation Guide, to identify exactly the directory in which it resides). An excerpt from this file follows, which provides the lightweight definition of a `Button` lexical class (i..e. real implementation realized by WINDOWS object library); notice that we have chosen to name the WINDOWS object library as `DeskTop` (any name can be given).

```
struct  Font [
      string          name;
      int             pointSize;
      bool            italics;
      bool            bold;
      bool            underline;
      bool            strikeout;
      int             orient;
];

struct Color [
      int red, green, blue;
```

```
];


enum ButtonType=[ ButtonOneState, ButtonTwoState ];
enum ButtonState=[
      ButtonStateOn, ButtonStateOff, ButtonStatePress
];
enum ButtonTextPosition=[
      AboveBitmap,
      BelowBitmap,
      LeftOfBitmap,
      RightOfBitmap,
      OnBitmap
];

lexical Button (DeskTop) [

      public:
      int x, y, width, height;
      bool isVisible;
      Font font;
      string label;
      bool isDefault;
      Color bkColor;
      Color textColor;
      ButtonTextPosition textPosition;
      ButtonType buttonType;
      ButtonState state;
      method Pressed;

      constructor []
      destructor []
]
```

The above code segment from `win.gdd` defines some data types (i.e. `struct Font`, and enumerated `ButtonTextPosition`, `ButtonType` and `ButtonState`) and a lexical class (i.e. `Button` for `DeskTop` toolkit name). In order to utilise these two constructs from another module than the original one in which they are defined, we have to follow the regulations of Section 8.3.2 - Constructing I-GET header files. According to those rules, in case of class categories, it suffices to copy the original class definition and surround it with header blocks. For data types, we have again to do the same thing.

We recall that for classes, another level of filtering could be applied, to make classes more readable by removing unnecessary detail; this is possible by

removing private members and all implementation-related constructs. You may notice that the previous `Button` lexical class definition already fulfils these requirements, hence, the class definition itself is also appropriate for header inclusion. This is natural for the case of lightweight lexical classes, since no implementation constructs are expected to be defined in the context of their I-GET definition. As a result, in the case of utilising the WINDOWS lexical classes defined within `win.gdd` (the same also holds for the Xaw integrated toolkit) which are all lightweight classes, from within a particular file `myfile.gdd`, it suffices to do the following:

```
// Source file myfile.gdd.

headers [
      extern ["win.hh"]
      #include "win.gdd"
]

// You may use from now on lexical classes
// from win.gdd.
```

Lets study what we have done in the previous example. According to the rules of creating header files, all lexical classes and user-defined data types must be enclosed with header blocks. In our example, instead of creating a header file, we have decided to include directly the original `win.gdd` file, since it contains lightweight lexical classes, hence, already appropriate for headers. Then, we have surrounded the `#include` directive with header blocks, as it is required. Finally, to resolve the include dependency among the code generated files, we have put the `extern ["win.hh"]`, which indicates that the header file generated for `myfile.gdd` will include the `win.hh` header file (which is the code generated header file for `win.gdd`). As you will notice, all the examples supplied with the I-GET tool, which utilise the integrated interaction elements for WINDOWS and Xaw, follow exactly this approach.

Chapter 10

10. Virtual interaction objects

The notion of virtual interaction objects is of key importance in the context of the I-GET language. Virtual objects are synonymous to abstract interaction objects, in the sense that they can be completely relieved from physical interaction matters. Virtual interaction objects are not hard-coded in the I-GET language; initially, the I-GET tool offers no virtual interaction object classes. This *tabula rasa* concept represents the central philosophy of the I-GET tool; as we have seen, the same also holds for lexical interaction objects. However, as part of the I-GET software release we have already done the work of constructing some useful virtual classes (as it was also the case for lexical classes). The I-GET language offers two mechanisms for manipulating virtual object classes:

- The *genesis mechanism*, for defining new classes of virtual interaction objects. Such newly created classes do not yet have any link to the physical interaction layer; hence, they do not exhibit any physical interaction behaviour.

- The *instantiation mechanism*, for mapping virtual interaction object classes to lexical interaction object classes. It is possible that a particular virtual class is mapped to lexical object classes via multiple alternative mapping schemes; this is called *lexical polymorphism*.

Developers may declare instances of virtual object classes in a manner similar to declaring lexical object instances. When virtual object instances are declared, developers may choose the particular physical mapping scheme

active for that virtual instance, hence, having full control of the physical instantiation style for virtual objects at the virtual instance level. Developers may choose to manipulate either the virtual instance, or the physical instance(-es) for virtual objects; in this case, the consistent mapping among the virtual and physical attributes can be easily certified.

Reference types for virtual object classes are supported, in a manner analogous to lexical object references. Virtual object classes support incremental development, by allowing instantiation schemes to be defined externally within other modules, without requiring recompilation of the original virtual object classes.

Finally, virtual object classes should not be viewed as typical abstract OOP classes, where ISA hierarchies are built and only derived classes may be instantiated; virtual instances in the I-GET language are separate programming object instances, which can be created even without a physical mapping. A virtual instance may have multiple concurrent physical instances active, as opposed to OOP classes where instances are *unary objects*, while the abstract instance is just a super-class pointer type casting.

10.1 Specification of virtual objects

The class definition body for virtual interaction object classes is defined as a block starting with an opening `[` and ending with a closing `]` square bracket. Similarly to lexical classes, the class definition body should encompass both definitions / prototypes and well as implementations for all class members. The header of a lexical class starts with an explicit listing of all issued toolkits (i.e. named in the I-GET language context) for which the particular virtual object class forms a valid abstraction. The class body which follows, should always end with constrcutor `constructor [...]` and a de-

structor `destructor [...]` blocks, exactly in this order (empty blocks `[]` are naturally allowed). A virtual class definition has the following syntax (we omit the details of the class body definition for clarity):

The header of a virtual class definition.

VirtualClass:

virtual ID $^{\text{class name}}$ **(IdList** $^{\text{toolkits}}$ **) VirtualClassBody**

IdList: ID (**,ID**)*

An example of the simplest virtual class definition.

```
lexical (Xaw);          // Athena widget set.
lexical (DeskTop);      // WINDOWS object library.
virtual SimplestVirtualClass (Xaw, DeskTop) [
      constructor []
      destructor []
]
```

Next we will provide the full list of all various constructs which can be declared within virtual object classes, and we will discuss each such construct in detail. The syntax for the class body of virtual objects follows.

VirtualClassBody:

[

(**VirtualClassConstruct**)*

constructor Compound

destructor Compound

]

VirtualClassConstruct:

VariableDeclaration *or*

MethodIssue *or*

ScopeQualification *or*

MethodImplementation *or*

Monitor *or*

Constraint *or*

FunctionPrototype *or*

FunctionImpl *or*

UserDefinedDataType

10.1.1 Variable declarations and method issue

Variables can be normally declared, supporting also initializers to be supplied. When variables are defined in the context of virtual classes, they are called *virtual attributes*. Semantically, you will define virtual attributes reflecting the abstract properties of a virtual interaction object class. Issuing methods in the context of virtual classes is similar to issuing methods in the context of lexical object classes; however, methods of virtual object classes should (i.e. *virtual methods*) reflect more abstract behavioural properties, in comparison to lexical methods. We will provide the definition of a real virtual class, included as part of the I-GET software release, an excerpt from file `virtual.gdd` (please refer to Volume 2 - Installation Guide for more details on where this file is located).

Excerpt from virtual.gdd, the library for virtual object classes supplied with the I-GET software release.

```
virtual Toggle (Xaw, DeskTop) [

      public:
      bool    Accessible=true;    // default.
      bool    State=true;         // default.
      method  Switched;           // state changed.

      constructor []
      destructor []
]
```

In the previous example, the virtual class is named `Toggle` (i.e. a boolean state interaction object abstraction), and is defined to be applicable as an abstraction for `Xaw` and `DeskTop` issued toolkits (these are the names we use as part of the supplied I-GET libraries). It is not allowed to supply an empty

list of toolkits names, since this would mean that you define a virtual object class which forms an abstraction for no toolkit. In case that you integrate within the I-GET tool a new toolkit *Y*, for which an already defined virtual class *C* constitutes a valid abstraction, you may wish to expand the list of toolkits for virtual class *C*, by adding *Y* as well; re-compilation will be also needed in this case.

In the body of the `Toggle` virtual class, two virtual attributes are declared, both of public scope: (i) `bool Accessible`, initially set to `true`, for controlling if interaction is enabled by a particular `Toggle` virtual instance; and (ii) `bool State`, initially set to `true`, which stores the state of the `Toggle` virtual object. Both attributes can be directly set by the developer. It is obvious that both attributes represent the abstract behavioural properties of the `Toggle` interaction category, without introducing any particular physical interaction characteristics. To demonstrate the abstraction capability offered by virtual objects, as a subtraction of physical features, we provide below the corresponding lexical classes for `Xaw` and `DeskTop` toolkits, as they are defined as part of the I-GET library for integrating Xt/Athena and WINDOWS object library toolkits.

```
// Excerpts from win.gdd
//
enum CheckBoxType=[TwoState, ThreeState];
enum CheckBoxState=[CheckOn, CheckOff, CheckGray];

lexical CheckBox (DeskTop) [

      public:
      int x, y, width, height;
      bool isVisible;
      string title;
      method StateChanged;
      CheckBoxType boxType;
      CheckBoxState state;
      bool leftText;
      Color  bkColor;
      Font  font;

      constructor []
      destructor []
```

```
]

enum RadioState=[RadioOn, RadioOff];

lexical RadioButton (DeskTop) [

      public:
      int x, y, width, height;
      bool isVisible;
      string title;
      Font font;
      method StateChanged;
      RadioState state;
      bool leftText;
      Color  bkColor;

      constructor []
      destructor []
]



// Excerpts from xaw.gdd
//

enum ShapeStyle=[
      Rectangle, Oval, Ellipse, RoundedRectangle
];

enum ResizeBehavior=[
      ChainTop,
      ChainLeft,
      ChainRight,
      ChainBottom,
      Rubber
];

lexical Toggle (Xaw) [

      public:
      objectid fromHoriz,fromVert;
      int     horizDistance, vertDistance;
      ResizeBehavior left, right, top, bottom;
      string  background, foreground;
      string  borderColor;
      int     borderWidth;
      int     x, y, width, height;
      bool    sensitive;

      bool            state;
      method          StateChanged;
      string          label;
      string          bitmap;
      string          font;
      ShapeStyle      shapeStyle;
```

```
        objectid        radioGroup;

        constructor []
        destructor []
]
```

In the above example, various data types and lexical classes are defined. The `Font` and `Color` data types are the ones defined under Section 9.6 of this Volume. The previous listing of code has been constructed by taking excerpts from `win.gdd` and `xaw.gdd` (as indicated on the left, with double-head arrows). You may observe that for `DeskTop` toolkit, there are two lexical classes which can be attributed as physically realizing the `Toggle` abstract behaviour, namely `CheckBox` and `RadioButton` classes, while for the `Xaw` toolkit there is only one lexical class similarly identified, named `Toggle`. It is evident from the definition of these lexical classes that there are many more attributes defined, reflecting the physical properties (either presentation or behaviour) of each particular lexical class, in comparison to the virtual `Toggle` class definition.

Keep in mind these lexical classes defined in our previous example, since we will recall them later on, under Section 10.2, when discussing the construction of lexical instantiation schemes for the `Toggle` virtual class defined before. Next we continue by discussing the specification of members within the virtual class body definition.

10.1.2 Scope qualification for class members

Two scope qualifiers, private: (the

As it is the case with lexical classes, there are two scope categories for members within virtual classes (the same scope categories also apply for agent classes, as it is discussed under Section 11): **public** scope, which allows access to member elements (those which can be referenced via an identifier) from outside the owner class, and **private** scope, which restricts access to

default) and public:.

members only from within the owner class. Public scope is defined through the `public:` scope qualifier, while private scope is defined through the `private:` qualifier; the default class scope is private (i.e. all class members, starting from the beginning of class definition, and until a scope qualifier is met, gain private scope. Exactly as it is the case with lexical classes, scope qualifiers are applied to the following member element categories:

- Variables.

Member categories on which scope qualification is applied.

- User-defined data types (user means programmer here).
- Functions.

```
virtual OneClass (MyToolkit, AndOnotherToolkit) [
      int a, b, c; // private attributes.
      c := a+b;    // a constraint local to the class.
      void F();    // private local function.
      public:
      string content;
      string name;
      void G();
      void F() []  // conflict with private prototype.
      method ContentChanged;
      method Destroyed;
      constructor [
            {me}.a =10; // use of 'me' to access 'a',
            b= a ;      // may also access 'a' directly.
      ]
      destructor [
            {me}->Destroyed; // notify Destroy method.
```

```
            {me}->F();          // call 'F' via {me}.
        ]
    ]
```

The previous example, apart from scope qualification illustrates some other interesting features of the I-GET language. Firstly, it is shown that functions will be checked for scope symphony among function prototypes (if supplied) and their respective implementations; as a result, a compile error is generated for function `F`, having firstly private (i.e. default) scope (prototype), and then public scope (implementation). Secondly, the use of `{me}` can be applied to access either variables (e.g. `{me}->a`) or to call functions (i.e. `{me}->F();`).

Finally, we can take advantage of the artificial method notification mechanism, which is also allowed for virtual methods, similarly to lexical methods. In our example, we enable developers to register methods which will be executed upon object destruction. We ensure that the method will be notified appropriately upon instance destruction, by generating explicitly a notification for the `Destroyed` method within the class destructor (i.e. `{me}->Destroyed;`).

10.2 Specification of polymorphic lexical instantiation relationships

Virtual object classes as such have no connection to the lexical layer of interaction. Even though instances may be freely declared, as it will be shown within the next Section, there is no corresponding lexical / physical instantiation caused by the creation of virtual instances. Hence, even though virtual instances are real objects, with instance identifiers and instantiated members, they do not yet exhibit any physical property, and have no link to the lexical

layer of interaction. In this Section we will present how virtual object classes can be linked to the lexical layer of interaction.

The notion of instantiation relationships.

The I-GET language provides a mechanism enabling the definition of *instantiaton relationships*, i.e. how virtual object classes map to particular lexical object classes. One and only one instantiation relationship (which is optional) may be defined for each named toolkit found within the virtual class definition header. Hence, considering our example of the `Toggle` virtual class, we may supply zero (i.e. none), one (i.e. either for `Xaw` or `DeskTop` toolkit), or two (i.e. for both `Xaw` and `DeskTop` toolkits) instantiation relationships.

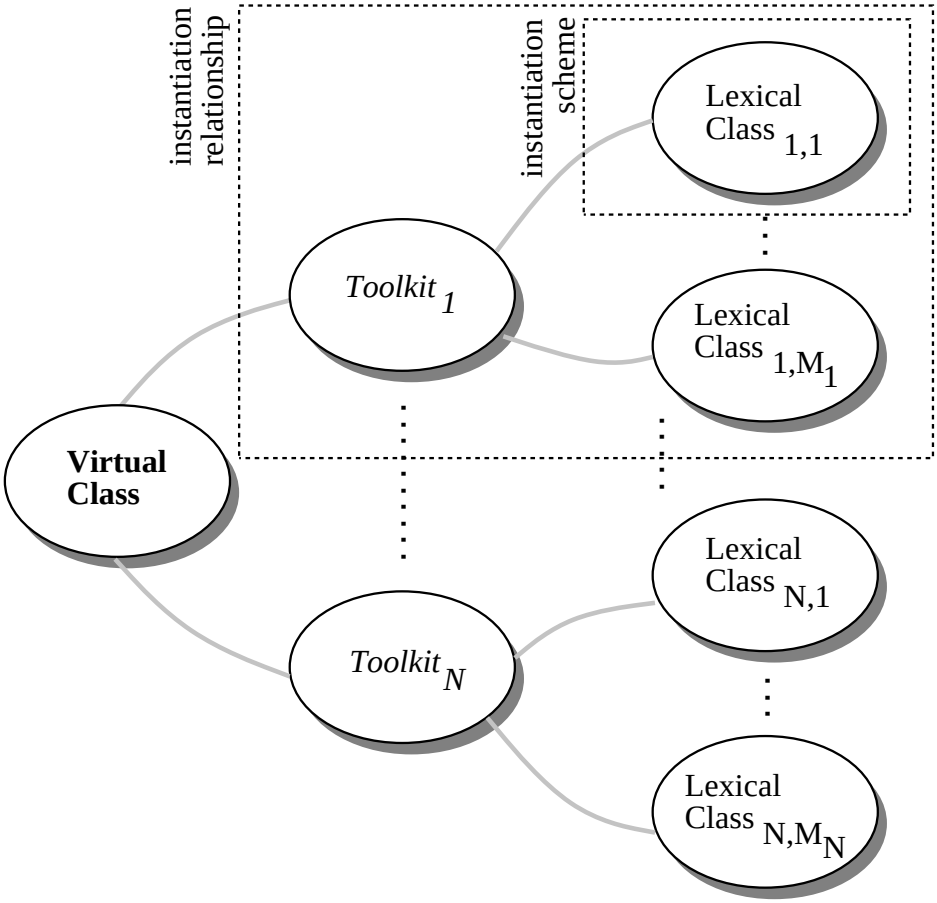


Figure 10.2–1: The polymorphic nature of virtual object classes through

instantiation relationships and instantiation schemes.

The notion of instantiation schemes.

What is an instantiation relationship ? It contains an arbitrary number of *instantiation schemes*. Each instantiation scheme, which must be uniquely named in the context of a particular instantiation relationship, maps the virtual object class to a particular lexical object class; hence, alternative instantiation schemes may map a virtual object class to alternative lexical object classes. Within the body of an instantiation scheme, all the necessary constructs are defined so that the virtual and the lexical class are consistently bridged. This may require constraining virtual and lexical attributes together, artificially notifying virtual methods when lexical methods are physical notified (i.e. due to user interaction), etc; more will be made clear from the real examples (i.e. part of the I-GET library) which will be given In Figure 10.2-1, the physical polymorphic nature of virtual interaction objects is illustarted. Next, we will continue by presenting the syntax for instantiation relationhips and instantiation schemes.

Grammar for instantiation relationships and instantiation schemes.

```
InstantiationRelationship:
        instantiation ID virtual class ( ID for toolkit ) [
        ( InstantiationScheme ) +
        default ID default scheme name ;
        ]
InstantiationScheme:
        ID scheme name : ID lexical class [
        ( InstantiationSchemeConstruct ) *
        constructor Compound
        destructor Compound
        ]
InstantiationSchemeConstruct:
        VariableDeclaration or
```

MethodImplementation *or*

Monitor *or*

Constraint *or*

FunctionPrototype *or*

FunctionImpl *or*

UserDefinedDataType

```
instantiation Toggle^Virtual Class (Xaw) [

      Toggle^Scheme Name : Toggle^Lexical Class [

            {me}Xaw.state := {me}.State;
            {me}Xaw.state : [
                  {me}.State={me}Xaw.state;
            ]

            {me}Xaw.sensitive:={me}.Accessible;
            {me}Xaw.sensitive: [
            {me}.Accessible={me}Xaw.sensitive;
            ]

            method {me}Xaw.StateChanged [
                  {me}->Switched;
            ]

            constructor []
            destructor []
      ]

      default Toggle^Default Scheme ;
]
```

In the previous example, we provide the definition of the instantiation relationship of the virtual `Toggle` class for toolkit `Xaw`. This definition is an excerpt from file `instxaw.gdd`, providing the instantiation relationships of the I-GET supplied library of virtual class for the Xaw toolkit. Please refer to Volume 2 - Installation Guide, for more information on locating this file within the I-GET software release.

Name the scheme with its corresponding lexical class

The previous instantiation relationship defines a single instantiation scheme named `Toggle`, mapping to the lexical `Toggle` class. You may choose the

name: that's a good policy.

same name for virtual classes, lexical classes and schemes, since they belong to different name spaces. In our case, we have purposefully selected to name schemes with the name of their corresponding lexical class, so that it is directly clear from the scheme name which is the target lexical class. The instantiation relationships requires an explicit definition of the default active instantiation scheme for a particular instantiation relationship. When you declare instances of a virtual object class, unless otherwise specified (will be clarified in the next Section), the I-GET tool will instantiate the virtual object for all toolkits which supply an instantiation relationship for this virtual class. The physical instantiation will take place according to the default defined instantiation scheme; hence, in the case of the `Toggle` virtual class, a `Toggle` lexical object instance will be created, due to the default `Toggle` scheme.

The role of the default instantiation scheme definition.

Now its time to study the body of the `Toggle` instantiation scheme, which defines how the `Toggle` virtual instance and the `Toggle` lexical instance are related. The `Toggle` virtual class has two virtual attributes, called `State` and `Accessible`. In the constraint defined locally within the `Toggle` instantiation scheme, the lexical attribute `state`, syntactically accessed via `{me}Xaw.state`, is constrained from virtual attribute `State`, syntactically accessed through `{me}.State`. This implies that by changing the `State` virtual attribute, the lexical instance (displayed to the user) will be accordingly affected, since its `state` attribute will change. Similarly, a monitor takes care of mapping a change of the lexical attribute `state` to the virtual attribute `State`. We will explain later on why we employ monitors for mapping updates of lexical attributes on virtual attributes. A similar constraint - monitor pair is defined for ensuring equality among the `Accessible` virtual attribute and the `sensitive` lexical attribute.

Defining how the virtual and lexical instances are related to each other.

What is of particular interest is the method implementation which follows the monitors and constraints in our previous examples. The implemtation concerns the lexical method `StateChanged` of the `Toggle` lexical in-

Defining how virtual method notification will be generated

upon lexical method notification.

stance. The role of this implementation is to ensure that each time the lexical instance is notified for state change, due to user interaction, then a notification will be also generated for the corresponding virtual method. This is accomplished by providing a method impementation which includes only one statement: an artificial method notification for the virtual instance. This is the general approach you will have to follow for mapping physical methods to virtual methods.

Explicit toolkit resolution to access lexical instances through virtual instances.

Notice that the `{me}` construct provides access to the virtual instance, when used within instantiation schemes. As a result, through `{me}`, virtual attributes and methods can be directly accessed; e.g. `{me}.State` (attribute) `{me}->Switched` (method), etc. However, in order to access members of the lexical instance, an explicit toolkit qualification is needed. This toolkit resolution is necessary, since when a virtual instance is created, multiple instantiation relationhips for multiple toolkits may be also instantiated, resulting in connecting a single virtual instance to multiple lexical instances. Hence, it is required to have correct syntactical and semantic access to each such lexical instance. In our exampe, the expression `{me}Xaw` provides access to the physical instance of the currently defined scheme for the `Xaw` instantiation relationship. Hence, lexical members can be accessed this way; e.g. `{me}Xaw.state` (attribute), `{me}Xaw.SateChanged` (method), etc.

A typical example in which you will have two concurrently active toolkits is when you employ a conventional windowing toolkit and a non-visual interaction toolkit. This combination is quite powerful for creating Dual User Interfaces, concurrently accessible by sighted and blind users. In this case you will have the two toolkits running in parallel, with their respective I-GET toolkit servers, while by creating virtual instances, a visual and a non-visual lexical instance will be automatically created, maintaining the relationships among their local members as defined within the instantiation

schemes.

10.2.1 Defining the mapping among virtual and lexical attributes

The case of maintaining a consistent mapping between corresponding virtual and lexical attributes is of particular importance. Usually, such mapping will mean preserving equality, possibly after some type conversions, if necessary. In the previous sub-section, we have seen that in a real instantiation scheme a monitor and a constraint have been employed to realize such mapping. Here, we will justify this approach and show why the use of constraints only cannot work safely.

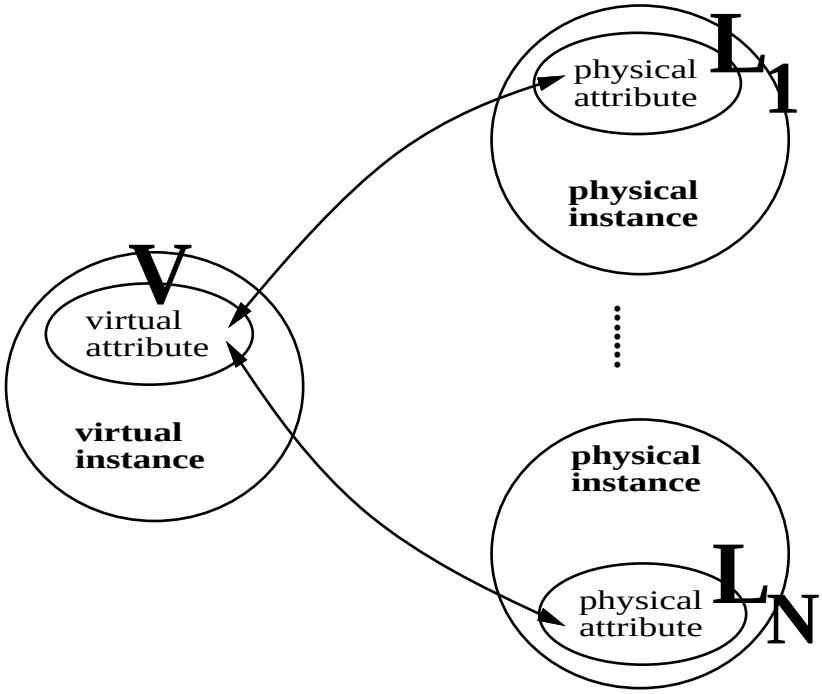
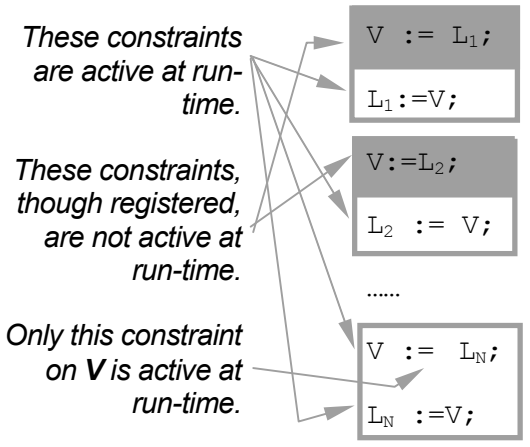


Figure 10.2–2: Mapping among a virtual attributes and its corresponding lexical attribute for all active lexical instances.

So, lets define the general case of having a virtual instance which at run-time corresponds to multiple active lexical instances. This run-time scenario will result by defining multiple instantiation relationships for a virtual class, corresponding to the various toolkits for which a virtual class forms a valid ab-

straction (as part of the virtual class definition header). Consider that there are **N** such active lexical instances, resulting from **N** active instantiation relationships. Now, lets assume that there is a virtual attribute **V**, which should consistently map to **N** lexical attributes, one from each active lexical instance. Lets name those attrbutes as $\mathbf{L_1},\ldots,\mathbf{L_N}$. This scenario is also illustrated under Figure 10.2-2. The *mapping logic* is defined within the corresponding instantiation schemes. Hence, within each of the **N** distinct instantiation schemes, one could define two constraints for preserving the equality between the **V** attribute and the $\mathbf{L_i}$ attribute as below.



What is problematic in defining the attribute mappings through constraints? Each such pair of constraints aims to constraint **V** and $\mathbf{L_i}$ together, and is defined locally within an instantiation scheme. Assume that at run-time, the lexical instance whose scheme owns the last constraint on the left, is also the last to be created. Then, clearly the last constraint on the left, will be also the last to be installed.

But then, the constraint `V := LN;` will be the active constraint for **V**. What is the effect of this ? If any $\mathbf{L_i}$, $\mathbf{i \neq N}$ changes (e.g. due to user interaction), then there is no chance that this change is also propagated on **V**, and consequently to all rest of $\mathbf{L_j}$, $\mathbf{j \neq i}$. This will leave the group of virtual and lexical attributes in an inconsistent state. To solve this problem, the solution is very simple, and has been introduced under Section 5 - Monitors. We can constraint the **V** attribute with each $\mathbf{L_i}$ attribute via a special purpose monitor; monitors do not exclude each other, even when for the same Lvalue. Hence, practically all such constraints (even though modelled via monitors) will be still active. The previous set of constraints is now redefined as follows:

All the constraints are active.

All the monitors are active.

```
L1 : [ V=L1; ]
L1:=V;

L2 : [ V=L2; ]
L2 := V;
……

LN : [ V=LN; ]
LN :=V;
```

∀ i: i=1,…,N, define:

```
Li : [ V=Li; ]
Li := V;
```

```
#define LV_MAP(l, v) \
      l : [v=l;] \
      l:=v;
```

You may wish to define a standard macro, such as `LV_MAP`, to be used in all cases where a virtual and a lexical attribute need to be "tight" together. In some cases, if the attributes are not of data types which can be directly assigned to each other, you may need to implement small conversion functions. This issue is illustrated in the following simple example.

Virtual and lexical attributes

Necessary type conversion function among the two attributes.

Monitor and constraint to preserve equality between lexical and virtual attributes.

```
bool state_as_bool;  // assume a virtual attribute.
enum State=[On,Off];
State state_as_enum; // assume a lexical attribute.
bool State2bool (State state) [ // conversion func.
      if (state==On) return true; else return false;
]
State bool2State (bool state) [ // conversion func.
      if (state) return On; else return Off;
]
state_as_enum : [
      state_as_bool=State2bool(state_as_enum);
]
state_as_enum := bool2State(state_as_bool);
```

10.3 Instance declaration and member access

Similarly to lexical classes, creating instances of virtual object classes is made in the form of declarations (i.e. instantiating virtual objects is not allowed directly via statements); instances may be only declared within agents and only at class scope (i.e. never in block scope). We will present the style for *declaring virtual object instances*, and we will show how members can be accessed via such instances.

Grammar for declaring virtual object instances.

VirtualObjectInstance:

virtual ID $^{\text{virtual class}}$ **ID** $^{\text{instance name}}$ (**OptionalParm**)* **;**

OptionalParm:

: parent (ID $^{\text{toolkit name}}$**) = Expression** $^{\text{objectid type}}$ *or*

: scheme (ID $^{\text{toolkit name}}$**) = ID** $^{\text{scheme name}}$

Before starting with some code examples, lets see how instantiation works for virtual objects in the I-GET language. The instantiation of a virtual object class takes place in five logical stages:

Five logical stages in the automatic process of instantiating virtual object classes.

1. Creating an object instance of the virtual class structure defined; all class members are instantiated, in their order of appearance (e.g. variables with their initializers, constraints are installed, call-back implementations are registered), and finally the constructor is called.

2. The instantiation relationships defined for the virtual class are activated. If none has been linked with the running interactive application (you may define an intsantiation relationship, but simply do not include in the files comprising a Dialogue Control component), then instantiation has been completed.

3. For each instantiation relationship defined, related to a particular named toolkit, do the following: if a scheme selection has been explicitly defined

for that toolkit, as part of the instance declaration, via the `scheme(<toolkit name>)=<scheme name>` parameter defininition, then activate the *<scheme name>* scheme, else activate the default defined scheme.

4. A scheme is activated as follows: (i) an instance of the lexical class associated with the scheme name, as part of the scheme header, is automatically created and associated with the virtual instance; (ii) all members local to the scheme body are also instantiated, according to their order of appearance; (iii) `{me}`, when used locally within the scheme body, accesses the virtual instance, while similarly `{me}<toolkit name>`, accesses the automatically created lexical instance.

5. After all instantiation relationships have been activated, the virtual class instantiation has been successfully completed.

In this example we declare both lexical and virtual instances.

```
lexical (Xaw) ApplicationWindow x_appwin;
lexical (DeskTop) FrameWindow w_frame;
virtual Toggle my_vtoggle
        : parent(Xaw)={x_appwin}
        : parent(DeskTop)={w_frame};
```

Programming with multiple toolkits in parallel: possible, though some execution limitations.

The previous example, though being very small, encompasses some very interesting properties. Firstly, we see that two integrated toolkits are utilised in parallel: the `Xaw` (i.e. Athena widget set - UNIX variants), and the `DeskTop` (WINDOWS object library - WINDOWS 32-bits). But is this possible ? the answer is yes and no. *Yes*, when it comes to manipulating object instances in the context of the I-GET language, but *no*, when trying to run such a Dialogue Control component utilising object classes from both toolkits (as the code segment of our previous example) with its respective toolkit servers. The problem at the execution phase is that each such toolkit

Toolkit servers must be available under the same OS platform to run concurrently.

is integrated via a toolkit server program, which must be initiated as an independent process. The `Xaw` toolkit server will have to run under UNIX, while the `DeskTop` toolkit server under WINDOWS. However, the current implementation of the I-GET run-time protocol library does not support toolkit servers to run across different OS platforms. Hence, you will be able to run such an example either under WINDOWS (with WINDOWS toolkit server), in which case you will "see" all WINDOWS instances on your display, or under UNIX and X (i.e. with Athena toolkit server), where you will "see" only Athena widget instances. In both cases, however, the Dialogue Control component need not change and will still behave correctly, irrespective of the fact that in both cases one toolkit physical layer will be missing.

Running a Dialogue Control component, employing multiple toolkits, is safe even on the absence of some or all of the toolkit servers.

Why you want to do something like that, i.e. employing objects from different toolkits in the same program, while it may run only with each of those toolkits separately ? The answer is "if you plan to make developments which need to be made available to several platforms". By employing virtual objects, you may construct a logical abstract structure of your interface, and then specialize as needed with the lexical classes provided. Off course you may employ a cross-platform toolkit to do that, or just create virtual classes which play the role of cross-platform object classes, instead of being pure abstractions, so that you can have portable interface implementation (we discuss this issue under the next Section - 10.4).

In any case, the I-GET tool supports both options in a an effective manner. Now , lets go back to our example. Two lexical object instances are declared, each of a different toolkit, but both being top-level container objects; an `ApplicationWindow` instance for `Xaw`, called `x_appwin`, and a `FrameWindow` instance for `DeskTop`, called `w_frame`. Then, a virtual instance is declared. In one of our previous examples, we have presented an excerpt from the I-GET specification library which demonstrated the instantiation relationship of the virtual `Toggle` class for `Xaw` toolkit. The I-

GET library encompasses a similar relationship for the `DeskTop` toolkit. As part of the virtual instance declaration, named `obj`, there are two categories of parameters enabling explicit scheme selection and explicit lexical parent definition, for each named toolkit. In our case, the `obj` virtual instance will be physically instantiated with the default schemes for both toolkits. For the `Xaw` toolkit, the parent object will be `{x_appwin}`, while for the `DeskTop` toolkit it will be `{w_frame}`. This is a significant feature, since it allows *object hierarchy differentiation* at the lexical level, even though common virtual objects are manipulated by the developer. You may even extend further this scheme by defining virtual object classes representing container objects so that all object instances declared are virtual instances (the I-GET specification library includes containers represented as virtual object classes).

The notion of object hierarchy differentiation, at the lexical level, when declaring virtual object instances.

10.3.1 Explicitly disabling instantiation for a particular toolkit

It is allowed to disable any particular instantiation relationship from being activated, for virtual object instances. This is made possible by supplying explictly `scheme(<toolkit name>)=nil`, where *<toolkit name>* is the name of the toolkit with which the instantiation relationship to be disabled has been associated. In the following example, an pure virtual instance is declared, for which no physical instantiation will be carried out.

```
virtual Toggle pure_virtual_instance
        : scheme(Xaw)=nil
        : scheme(DeskTop)=nil;
```

10.3.2 Accessing virtual / lexical attributes / methods

Virtual attributes are those which are defined locally within the virtual class body. Lexical attributes are those defined locally within the lexical classes of the particular active lexical instances of a virtual classes. To access any of those attributes from outside their owner class body, they must be of public scope. Accessing virtual attributes has a slightly differentiated syntax, in comparison to accessing lexical attributes via virtual instances. Similarly, accessing (i.e. impementing) virtual and lexical methods presents some small differences.

Syntax for accessing virtual / lexical attributes and virtual / lexical methods from virtual instances.

VirtualAttribute:

{ ObjRef *virtual instance* **} . ID** *attribte name*

LexicalAttributeFromVirtualObject:

{ ObjRef *virtual instance* **} ID** *toolkit name* **. ID** *attribte name*

VirtualMethodImpl:

method { ObjRef *virtual instance* **} . ID** *method name*

Compound

LexicalMethodImplFromVirtualObject:

method { ObjRef *virtual instance* **} ID** *toolkit name* **. ID** *method name*

Compound

Examples of accessig virtual and lexical attributes.

```
{obj}.State=false;
{obj}.Accessible=false;
{obj}Xaw.label="Auto Spelling";
{obj}Xaw.font="Times-Roman-I-12";
```

Examples of impementing virtual and lexical methods.

```
method {obj}.Switched [
      printstr("Virtual method called.\n");
]
method {obj}Xaw.StateChanged [
      printstr("Xaw: lexical method called.\n");
]
```

In the previous example, accessing virtual and lexical attributes is syntactically very simple. It should be noted that accessing members from a lexical instance of a virtual attribute is strongly checked by the I-GET compiler. This is possible since the I-GET language forces that full information on the class of the various lexical instances is provided at compile time. From where is this information gained ? form instantiation schemes, which uniquely define an association with a particular lexical class. Knowing the active schemes of a virtual object instance, for the various toolkits, means knowing the specific lexical classes of the active lexical instances. In the I-GET language, there are three possibilities for scheme selection for virtual interaction objects:

Compile-time checking of lexical member access via virtual instances.

- *The default scheme*, which is selected if no scheme is explicitly chosen as part of the virtual instance declaration.
- *A specific selected scheme*, which is defined via `a scheme(<toolkit name>)=<scheme name>` parameter definition.
- *No active scheme*, if `nil` is provided as the scheme name in the previous case. Trying to access a lexical instance is a compile error in this case.

In the last example, we notice the impementation of two methods: one implementation for the virtual `Switched` method, and another for the lexical `StateChanged` method. One question from this example is "what is the order of method execution when the lexical object is notified due to user in-

put ?". We can very easily answer this question by having in mind that virtual methods are artificially notified from within the instantiation schemes; an implementation of the lexical method is provided which only notifies the virtual object. Since this lexical method is the first to be installed (on the lexical instance), it will be also the first to be executed upon notification. But then, since it will notify directly the virtual class, the methods of the virtual class will start executing, before other lexical methods are executed. Hence, following our approach for artificial method notification, the virtual methods will always preceede in execution order the lexical methods registered.

Order of execution between lexical and virtual methods.

10.3.3 Calling member functions

Member functions of the lexical and virtual instances can be called, via the virtual instance, only if they have a pubic access scope. The syntax for calling functions in this manner follows.

Syntax for calling functions through the virtual instance.

CallFunctionOfVirtualInstance:

{ ObjRef $^{\text{virt instance}}$ } -> ID $^{\text{function}}$ (ExpressionList $^{\text{actual args}}$)

CallFunctionOfLexicalInstance:

{ ObjRef $^{\text{virt instance}}$ } ID $^{\text{toolkit}}$ -> ID $^{\text{function}}$ (ExpressionList $^{\text{actual args}}$)

```
// Assume x, y, z are virtual instances.
// Assume all function below are correctly defined.
{x}->F();        // F from virtual class.
{x}Xaw->F();     // F from lexical class for Xaw.
int c={y}->G();
{z}AnimationToolkit.Animate({y}, 100);
```

10.3.4 Instantiation schemes are automatically instantiated non-referenceable classes

Instantiation schemes syntactically look like classes: they are named, they have a constructor and a destructor block, and they may encompass local members typically met in lexical and virtual classes like variables, data types, functions, constraints and monitors.

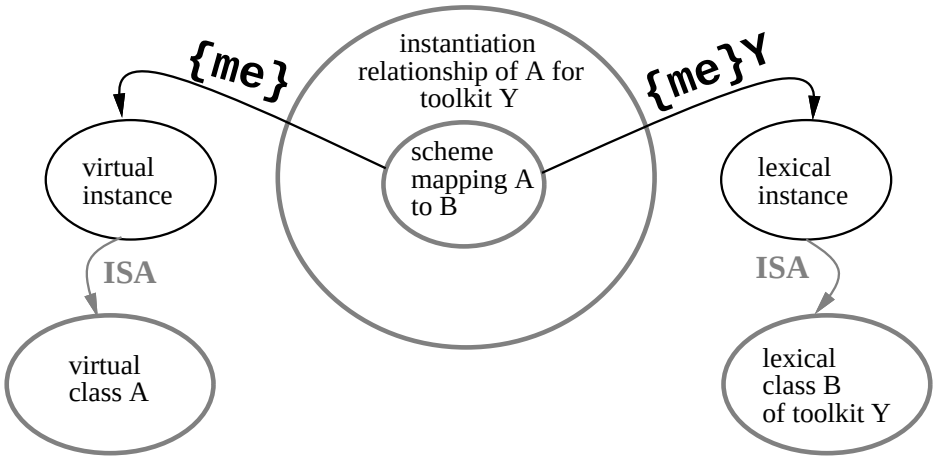


Figure 10.3–1: The role of instantiation schemes for bridging virtual and lexical object instances together.

However, instantiation schemes are not conventional classes in the sense that they could be manipulated by the developer. There is no facility for explicit instantiation, nor for referencing "instances" of instantiation schemes. For this reason, there are no scope qualifiers (i.e. public: and private:) allowed in instantiation schemes, since these are needed only if instance variables are supported, thus allowing external access to class members. The role of instantiation schemes is to bridge the virtual and lexical instances together, so that at run-time consistent mapping can be ensured. This issue is illustrated within Figure 10.3-1.

10.3.5 Inluding the necessary files when manipulating virtual object instances

When utilising virtual interaction objects, you will have normally to include the corresponding header files. This is required since, du to good development practice, virtual interaction object definition resides in separate modules. The same also holds for instantiation relationships, which enclose instantiation schemes. Here we will discuss two issues: (a) which header files are needed for the standard specification libraries which are provided by the I-GET software release; and (b) the structure of header files for virtual object classes and instantiation relationships.

Firstly, lets start with the case of the I-GET tool libraries. These libraries are written in the I-GET language, and concern mainly three categories of constructs: (i) lexical interaction elements, which include lexical object classes, for two integrated toolkits: the WINDOWS object library, integrated as `DeskTop` named toolkit, and Athena widget set, integrated as `Xaw` named toolkit; (ii) virtual object classes, which provide an important set of abstract interaction objects which you may utilise; and (iii) the instantiation relationships of the virtual library for the two integrated toolkits.

Assume that you are implementing a module in which you want to employ all of the above classes (i.e. virtual classes and instantiation for both integrated toolkits). In order to do that, you will have to manipulate:

- *Virtual object instances*, hence you will need to include headers for the virtual classes defined.

- *Instantiation schemes for the virtual object classes*, for both toolkits, hence you will need to include headers for the instantiation relationships defined.

- *Lexical instances of virtual instances*, as resulting from the instantiation relationships. Since the instantiation relationships concern both integrated toolkits, naturally the resulting lexical instances will be of classes from any of those two toolkits. Hence, you will have to include headers for lexical classes of these toolkits.

In order to be able to use all the above constructs, you will have to make the following inclusions in your code, as defined in the example which follows. In case that you are using only one of the two toolkits, you should remove only the files related to that particular toolkit (we indicate in the example the role for each file mentioned) - the order of inclusion is not important.

```
headers [
#include "virtual.gdd"     // virtual library
#include "xaw.gdd"         // Athena toolkit
#include "win.gdd"         // WINDOWS object library
#include "instxaw.gdd"     // Athena instantiation
#include "instwin.gdd"     // WINDOWS instantiation
extern [
      "virtual.hh",  // virtual library
      "xaw.hh",      // Athena
      "win.hh",      // WINDOWS
      "instxaw.hh",  // Athena
      "instwin.hh"   // WINDOWS
]
]
```

The files `virtual.gdd`, `xaw.gdd` and `win.gdd` include virtual classes, `Xaw` lexical classes and `DeskTop` lexical classes. Hence, following the rules for header files it only suffices to enclose the file inclusion directive within header blocks (as it is shown in our previous example). The files `in-`

`stxaw.gdd` and `instwin.gdd` provide the instantiation relationships of the virtual library for `Xaw` and `DeskTop` toolkits respectively. The rules are similar also for instantiation relationships, regarding their inclusion within header files. Hence, it also suffices to surround the original definitions with header blocks. You may also remove all local implementation constructs from instantiation relationships, and include them this way, without any problems. This may be a better approach, since it will hide information that developers utilising virtual objects do not need to know; this technique is illustrated in the example below, in which we provide the instantiation relationship for the `Toggle` virtual class for `Xaw` in a form complying to this idea.

Empty scheme implementation.

```
instantiation Toggle^Virtual Class (Xaw) [
        Toggle^Scheme Name : Toggle^Lexical Class [
                constructor [] destructor []
        ]
        default Toggle^Default Scheme ;
]
```

10.4 Defining cross-platform objects as virtual object classes

The term cross-platform or multi-platform has been attributed to interface toolkits preserving the same programming interface across different graphical environments. There are generally two approaches for building multi-platform toolkits:

- *The portable graphics layer approach*, according to which a basic highly portable graphics layer is built, providing typical rendering and device input facilities. This graphics library is re-implemented for various platforms. The emphasis is on making this library compact, efficient, extensible and expressive. Then, on top of this library, a complete interface toolkit is built, which is inherently portable to all installations of the underlying graphics layer. The result is a toolkit exhibiting a common look & feel across different platforms. The main disadvantage with this approach is the lack of a conformance to de facto industry standards for windowing interaction elements; usually, in order t address this problem, such toolkits try to *mimic* the look & feel of well accepted interaction objects.

- *The virtual toolkit layer approach*, in which a translation layer is built for translating calls / notifications to / from the underlying native toolkits. Following this approach, you may built a single interface with the cross-platform toolkit, and then compile & link it in different installations. The result is that the same interface will run with the native interaction elements of each particular platform. The approach is called as "virtual toolkit layer" due to the presence of a common toolkit layer, though never implementing directly all the supplied interaction elements.

The first approach requires considerable implementation efforts for building the dialogue of each interaction object from scratch. We will discuss in future documentation version how you may efficiently and effectively implement your own interaction objects through the I-GET tool. The second approach, as it will be shown, can be realized via the I-GET language facilities for virtual interaction objects and instantiation relationships; the implementation structure of such toolkits is shown in Figure 10.4-1.

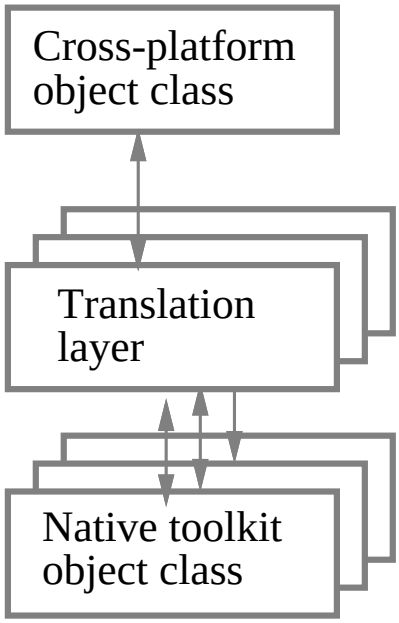


Figure 10.4–1: Cross-platform toolkits based on a virtual toolkit layer.

According to the virtual toolkit approach, a cross-platform object class maps, for each different target toolkit, to a specific native object class. We will outline the methodology for building virtual classes through the I-GET language, which exhibit physical properties and attributes, and are connected with both WINDOWS and Athena integrated toolkits. In order to do that, we will first provide a definition for cross-platform objects.

*A cross-platform object class Y, is a **generalization** of object classes from different toolkits, exhibiting similar "look & feel"; hence, as a generalization, it provides an **intersection** of properties met in the various native classes.*

We gave this definition just to make it clear that *abstractions*, providing a filter for physical properties are radically different from *generalizations*, providing a unification of physical properties. Hence, as it will be shown, virtual objects in the I-GET language are appropriate for implementing both abstractions, as well as generalizations. We will show that all the various characteristics of cross-platform classes are addressed by specific features of the I-GET language for virtual object classes; in the table which follows, we provide the various implementation requirements of cross-platform classes, and we also identify how they are met by virtual class definition constructs.

Cross-platform class properties	Virtual object class features
• Applicable for multiple toolkits.	• All target toolkits are listed in the virtual class definition header.
• For each toolkit, it maps to a specific native class.	• An instantiation relationship is defined for each toolkit, including a single instantiation scheme mapping to a specific native class.
• Mapping logic implementation is needed, including delegation schemes for function calls, type conversions, attribute associations, etc.	• Local scheme constructs allow delegation, constraining, monitoring and type conversions to be directly implemented.

As it will be made clear after reading Volume 4 - Integrating Toolkits, there is also another approach for creating cross-platform toolkits, through the toolkit integration mechanism. In our current discussion for virtual object classes which play the role of cross-platform classes, we assume that all target toolkit have already been integrated in the context of the I-GET tool. Since this may not be always the case, the toolkit integration mechanism may be employed, through which, the same toolkit interface specification may be used to integrate various toolkits. As a result, the developer is faced with a single programming view of interaction elements, although those are made retargetable (via the integration process) to multiple toolkit libraries. The details of toolkit integration are discussed within Volume 4.

10.5 Reference variables for virtual object classes

We have already discussed the support for reference variables on lexical objects instances under Section 9.5. In the I-GET language, there are similar variable categories supported for virtual object instances. Through such variables, it is possible to "assign" virtual instances on reference variables and manipulate those instance through the reference variable.

VirtualReferenceType:

ref virtual ID $^{\text{class name}}$ **(** **OptionalVirtRefParm**)*

OptionalVirtRefParm:

scheme(ID $^{\text{toolkit name}}$**)=ID** $^{\text{scheme name}}$

```
ref virtual Button bref1;

ref virtual Button : scheme(Xaw)=PushButton bref2;

type ref virtual Button : scheme(Xaw)=nil :
scheme(DeskTop)=nil : scheme(NonVisual)=nil
GenericButtonRef;

GenericButtonRef bref3=bref2; // ok.

bref1=bref3; // error, from general to specific.
```

Rules for assignment to virtual references.

It is evident that one important difference with lexical references is that there are some optional parameters which characterise the type of a virtual reference, apart from the class itself. More specifically, the specific lexical instantiation schemes which apply for each toolkit may define different reference types.

When a virtual reference is declared, the lexical instantiation schemes which apply follow the same rules as with the declaration of virtual in-

stances. Hence, a virtual reference represents a reference to a virtual class attributed with a particular lexical instantiation style. When assigning to a virtual reference variable *X*, the following should either apply, or else the assignment will generate a compile error:

- `nil` is assigned.
- An instace or a reference *Y* of the same virtual class is assigned, for which all active instantiation schemes of *X*, which are not `nil`, match the corresponding instantiaiton schemes of *Y*.

From the above two rules we may conlcude that it is allowed to assign any instance or reference variable of the same virtual class to those reference variables which define explicitly as `nil` all the instantiaiton schemes applicable for their respective class. One such reference type, named `GenericButtonRef`, has been declared in our previous example. Then, we have declared one variable of this reference type named `bref3`, to which we have directly assigned `bref2`; this is legal since, due to the fact that `bref3` is defined with actually no instantiations (i.e. `nil` schemes), it may be assigned with any reference variable, as far as it matches the virtual class. The opposite, however, is not possible, since, assigning `bref3` to something different from itself will fail, since the `nil` schemes may match only with `nil` itself. In conclusion, `nil` is more general, hence, we may assign something specialised to a general reference variable, but not the opposite. Below we provide one example which shows the use of virtual references for managing virtual object instances.

```
struct VirtualId [
      string class;
      GenericButtonRef button;
      GenericSelectorRef selector;
      GenericToggleRef toggle;
      GenericContainerRef container;
```

```
];

void RegisterVirtualInstance(VirtualId id) [
      // Add in registration lists.
]
void NotifyVirtualInstance(VirtualId id) [
      case (id.class) [
         "Button" : {id.button}->Pressed;
         "Selector" : {id.button}->Selected;
         "Toggle" : {id.button}->Switched;
         "Container" : ptintstr("No method.\n");
      ]
]

VirtualId id;
id.class="Button";
id.button={quit};
RegisterVirtualInstance(id);
NotifyVirtualInstance(id);
```

Take care of code in which references to object instances which have already been destroyed, are used.

You may maintain some references to virtual- or lexical- object instances which have been already destroyed, however, you may accidentally still access some members from those instances. This is naturally a run-time error, and it may cause your application to crash. To detect the exact source of such an error, i.e. which reference, at which point in your code, etc, you may re-compile your dialogue files with the **-dbg** option, so that the I-GET run-time error detection mechanisms may identify and locate the source of error.

Chapter 11

11. Agent classes

Up until now we have discussed about lexical or virtual interaction objects, and we have presented the syntax for instance declaration. However, we have not yet defined the *syntactic context* in which such instance declarations may be provided. In the I-GET language, this syntactic context is agents; more precisiely it is agent classes. Those can be seen as analogous to object classes in typical OOP languages, in the sense that an agent class is firsty defined, while various instances of an agent class may be created in a dialogue program during execution.

As we will show, the only "place" in which object instances may be declared is within the body of agent classes. One different with object classes as met in conventional pogramming lanaguages, is that, while in the latter the class definition and the class implementation may be physically split, i.e. may supply implementation of class members outside the class definition body, for the I-GET language, the imlementation of agent class members should be included within the agent class definition body. Now we will start explaining incrementally what you can do, and how, with agent classes.

11.1 Basic specification structure and examples

The syntax for agent classes. The main body of an agent class is defined between an opening square bracket **[** *just following the agent header and a closing* **]** *square bracket, just following the destructor block.*

AgentClass:

agent ID class name

{ **(FormalArguments)** *} or {* **create if (Precondition)** *}*

(**VariableDeclaration**)*

[**destroy if (Precondition)** *]* **[**

(**AgentMember** *)**

constructor Compound

destructor Compound

]

AgentMember:

VirtualObjectInstance *or*

LexicalObjectInstance *or*

MethodImplementation *or*

UserDefinedDataType *or*

VariableDeclaration *or*

FunctionPrototype *or*

FunctionImpl *or*

AgentPrototype *or*

AgentClass *or*

Constraint *or*

Monitor *or*

ScopeQualification *or*

EventHandler

Agent classes are actually dialogue control component classes. There are various types of members allowed within agent classes, as it is evident from the above syntax, and we will try to provide meaningful examples for all those consructs. The first syntactic construct of agent classes is the *agent class header*. This is composed by a preceeding **agent** keyword, followed by an agent class name. The agent class name should be unique for agent classes on the particular scope of the agent class definition (this is not always global scope, since, as it will be discussed latern on, embedded agent classes are allowed).

Following the agent class name, is the *instantiaiton style* definition. As it is shown, there are two alternative prefixes at this point, one being **(FormalArguments)**, and another being **create if (Precondition)**, both indicated as alternatives via the *{...} or {...}* grammar expression. The first prefix, with formal arguments, designates an agent class for which instances may be created in the dialogue program only via an explicit instatiation statement, while the second prefix, the one with the precondition, defines an agent classes for which instances may be created at run-time automatically, depending on whether the particular defined precondition evaluates to **true**. We will discuss the details of those two alternative agent instance creation approahces later on. Following the agent class instantiation style, is the agent class *destruction style*. This is defined via an optional **destroy if (Precondition)** construct, indicated via the *[...]* grammar expression, called the destruction precondition. When such a precondition is explicitly supplied for an agent class, instances of that class may be destroyed automatically if their respective destruction precondition evaluates to **true**. Alternatively, if no such precondiion is supplied, it is the responsibility of the developer to destroy instances, via the agent instance destruction statement, as needed. We will discuss the details of those two alternative agent instance destruction approaches later on. Now lets start with some simple examples.

```
agent SimpleAgent
```

```
create if (true) [
     constructor [
        printstr("SimpleAgent instance created!\n");
     ]
     destructor [
        printstr("SimpleAgent instance destroyed!\n");
     ]
]
```

Edit a file in which you will write the agent class defined above. Then save the file, and build it as a dialogue program; you will find details for building and running I-GET dialogue programs under Volume 1 of the I-GET public release documentation. Then, run this dialogue program. It should display the message `SimpleAgent instance created!` at command line (among other standard I-GET system messages); then, press Ctrl-C to terminate explicitly the execution of the I-GET program. Again, among other system messages, you will also see the `SimpleAgent instance destroyed!` message.

The above behaviour is explained as follows: the **SimpleAgent** is an agent which may be instantiated automatically on the basis of a precondition. The I-GET run-time system checks initially the precondition and finds it to be true; hence, it creates an initial instance, whose creation causes the constructor to be called and display the message as above. You may notice that the instantiation precondition, i.e. **(true)**, is constant. Hence, there is no chance that this precondition will be further changed during run-time, and as a result there is no chance for other instances to be created during run-time. The **(true)** precondition is a common trick in the I-GET language to be applied for the definition of such agent class that we want to be instantiated only once at start up.

Regarding destruction, the **SimpleAgent** class is not supplied with a destruc-

tion precondition. Hence, the only way to destroy instances of that class is via explicit destruction statements defined, as needed within dialogue code, by the dialogue developer. However, since in our program there are no such agent instance destruction calls, the single ceated agent instance will “live” as long as the dialogue program runs. By pressing Ctrl-C, the I-GET dialogue program terminates immediately, leading in the automatic destruction of all particularly present agent instances. Hence, at the point of program interruption, the single instance will be automatically destroyed, causing its destructor block to be called, and thus displaying the messsage.

11.2 Precondition-based and call-based instance creation policies

For the explanations of this Section, we wil use one of the examples already supplied as part of the I-GET public release. This example is the “Hello, world” program concept, programmed in the I-GET language. The example resides in the **EXAMPLES/HELLOWORD** directory for the UNIX version, and an in the **EXAMPLES\HELLOWOR** directory for the Windows version. Refre to Volume 1, Section 9.1.2, in which this specific example is used to explain the I-GET compilation, building and running phases. Also, in Volume 1, Section 10, you will find information on building the I-GET example programs. There are two versions of the “Hello, world” program, one for UNIX, employing the Xaw/Athena widget set, and one for Windows, employign the Windows object library. The differences among them are only with respect to the specific lexical object instances declared. We will work on the Windows example, while most of the changes and extensions we will introduce may be mapped directly for the UNIX version (if this is not the case, we will explicitly mention what should be different).

```
// Hello1.gdd.
// I-GET version of the "Hello, world" program concept.
// Anthony Savidis, January 1998.
```

```
// Windows version.
//

headers [
extern ["igetwin.hh"]
#include "win.gdd"
]

agent HelloWorld create if (true) [

        lexical(DeskTop) FrameWindow win;
        lexical(DeskTop) StaticControl label: parent={win};
        lexical(DeskTop) Button quit: parent={win};

        method {quit}.Pressed [ terminate; ]

        constructor [
                printstr("Hello, world! from command line.\n");
                {win}.x={win}.y=0;
                {win}.width=300;
                {win}.height=200;
                {win}.title="Hello, world application";

                Font font=[
                        "Arial",
                        14,
                        false,
                        false,
                        false,
                        false,
                        0
                ];

                {label}.x=5;
                {label}.y=10;
                {label}.width=90;
                {label}.label="Hello, world!";
                {label}.font=font;

                {quit}.x=5;
                {quit}.y=50;
                {quit}.width=90;
                {quit}.height=40;
                {quit}.label="Quit";
                {quit}.font=font;

                out(DeskTop).RealizeObject({win});
        ]
        destructor [
                printstr("Good bye world! from command line.\n");
        ]
]
```

The above code listing is **Hello1.gdd**, in **EXAMPLES\HELLOWOR** directory for the Windows version. The corresponding file for the UNIX version is, as mentioned before, in **EXAMPLES/HELLOWORD**, and is named as **Hello1.gdd**. In the above code listing, there are details which are not all of them important for our discussion, hence, in the upgrades we will introduce on this example program, we will use a simplified listing, where many of the code details shown above will be missing.

The first upgrade we will make on this program is to move from a precondition which is constant, i.e. **(true)**, to one that is not contstant, i.e. engaging some variables. But we will still make sure that only a single instance of the **HelloWorld** agent class will be created. This change is illustrated below.

```
bool makeInstance=true;

agent HellowWorld create if (makeInstance==true) [

// Agent body as before here.

]
```

The difference here is that we have moved from a constant precondition to a non-contstant one. This precondition still initially evaluates to **true**, hence, the I-GET run-time system will make one instance automatically. When running the program, the program still behaves as before. However, in the modified program, each time the precondition is re-evalued by the I-GET run-time system, and is found to be **true**, one new agent instance will be created. But when are preconditions re-evalued at run-time ? The rule is very simpe and follows below.

Rule for re-evaluating instantiation or destruction preconditions.

Each time an engaged variable is assigned with a particular value, either directly (i.e. via assignment), or indirctly (i.e. via pointers), the precondition is re-evalued. Such a re-evaluation will happen no matter if the old value of the variable is the same as the newly assigned value. If the precondition evaluates to true, then its associated action (i.e. instantiation / destruction for instantiation / destruction preconditions) will be performed.

Now, we will expand more this basic example, showing how multiple agent instances may be created. We will introduce one more *precondition-based agent class* (meaning an agent class with an instantiation precondition), while we will add some code in the original **HelloWorld** class to enable the user create instances of this agent by the press of a button. For the purposes of this example, we will only provide code for creating such agent instances,

but not for destroying them as well. In other words, the user will be able to cause the creation of some replicas of simple windows, however, he will not be given the means to also close them down separetaly (i.e. one by one). We will add such facilities in the next Section, for destruction policies. So, back to our example, lets see below what new we have to add.

```
bool makeExtra=false;
int extraSerial=1;
agent ExtraDialogue create if (makeExtra==true) [
      lexical(DeskTop) FrameWindow win;
      lexical(DeskTop) StaticControl label: parent={win};
      constructor [
            {win}.x={win}.y=extraSerial*10;
            {win}.width={win}.heigth=100;
            {label}.x={label}.y=5;
            {label}.width={label}.height=95;
            {label}.label="Extra #"+extraSerial++;
            out(DeskTop).RealizeObject({win});
      ]
      destructor []
]
...
agent HelloWorld ... [
      ...
      lexical(DeskTop) Button doExtra: parent={win};
      method {doExtra}.Pressed [
      makeExtra=true;
      ]
      ...
```

```
]
```

In the above code segments, the "..." represents parts of code which should be as in the previous version, while underlined segments indicate lexical layers and object class names which will normally be different for the UNIX version. The rest may not be changed. Also, statements for setting some lexical attributes, such as position, width and height, for the **doExtra** button, have been omitted for clarity.

The new code which has been inserted has the following effect: each time the button which is named **doExtra** in code is pressed by the user, the supplied method implementation will be called, due to notification of the **Pressed** method. As a result, the value **true** will be assigned to the **makeExtra** boolean variable. This will cause re-evaluation of **the (makeExtra==true)** instantiation precondition of the **ExtraDialogue** agent class; the precondition will be found to be **true**, hence, an instance will be created. The number of the agent instances which will be created is equal to the number of subsequent presses of the **doExtra** button by the user.

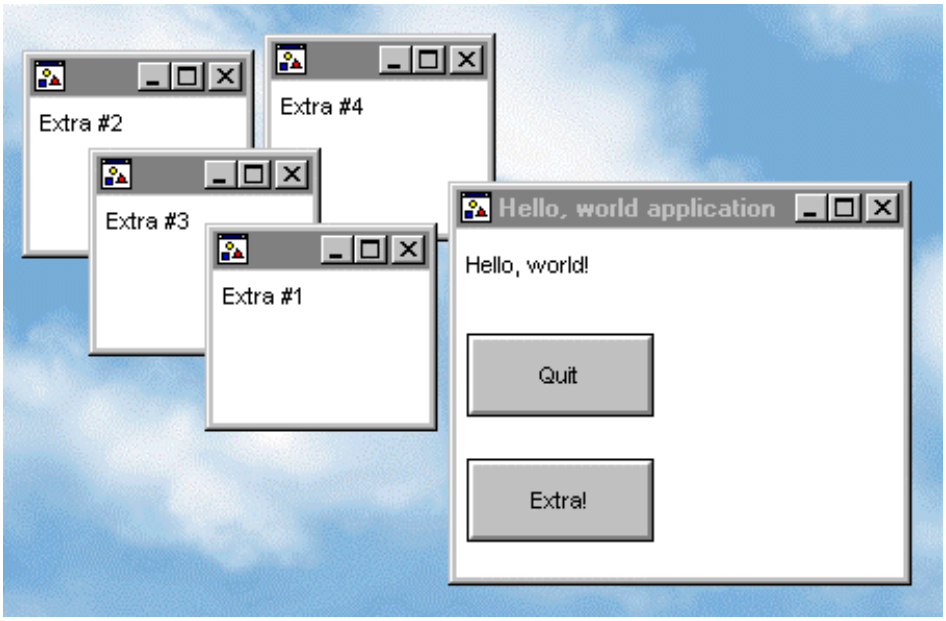


Figure 11.2–1: Screen snapshot of the modified "Hello, world" example.

In Figure 11.2-1, a screen dump of the modified example is shown. The *Extra!* labelled button has been pressed four times, hence, the creation of four windows by the four instances of the **ExtraDialogue** agent class. We will now modify further our example to demonstrate the use of parameterised agent classes, i.e. agent classes which may be instantiated via instantiation statements.

```
agent ExtraDialogue (int serial) [
    ...
    constructor [
    {label}.label="Extra #"+serial;
    ...
    ]
]
...
agent HelloWorld ...
    ...
    method {doExtra}.Pressed [
    create ExtraDialogue(extraSerial++);
    ]
    ...
]
```

On the previous version, the following changes have been introduced: (i) the **ExtraDialogue** agent class became a parameterised agent class, having one formal parameter named **serial**; and (ii) the impementation of the **Pressed** method for the **doExtra** button has been changed to explicitly call and agent instantiation statement, i.e**. create ExtraDialogue(extraSerial++)**. These two changes suffice for the new version, while the program still behaves (for

the user) as the old version. As we will show in the next Section, the **create** statement returns a value which is of a generic agent instance reference type, which may be used for either destroying the instance at a later point, or for accessing local agent members, in the same manner that class members may be accessed via object instances in OOP languages.

11.3 Precondition-based and call-based instance destruction policies

There are two alternative ways for destroying agent instances in the I-GET language: (i) via preconditions, meaning that in their respective agent classes, destruction preconditions are explicitly supplied, which may cause the agent instance to be automatically destroyed, if its respective destruction precondition is satisfied; and (ii) via statements, which take as an argument an agent instance identifier type. Instances of a precondition-based agent class, may still be destroyed via a statement, even though a precondition is explicitly supplied.

Firstly, we will start be presenting destruction preconditions. Again we will build upon the previous "Hello, world" examples, taking the most recent code listing we have provided. The first modification we are going to introduce, is the addition of an explicit destruction precondition for the **HelloWord** agent class.

```
bool closeHelloWorld=false;

agent HelloWorld ...

destroy if (closeHelloWorld==true) [

    ...

    method {quit}.Pressed [ closeHelloWorld=true; ]

    ...
```

```
    destructor [ out(DeskTop).DestroyObject({win}); ]
]
```

In the modified version, we have made the following changes: (a) we have declared a global boolean variable named **closeHelloWorld** with an initial **false** value; (b) we have added a destruction precondition which is satisifed only if the **closeHelloWorld** variable gains the **true** value; and (c) we have changed the implementation of the **Pressed** method for the **quit** lexical object instance, being of **Button** class for **DeskTop** lexical layer, to set explicitly the value of **closeHelloWorld** to **true** (the previous implementation made a call to the **terminate** built-in statement). Now, when the user presses the quit button, the **Pressed** method will be notified, leading to the execution of the supplied implementation block. Hence, the **closeHelloWorld** variable will be assigned the **true** value. This will cause the re-evaluation of the destruction precondition for the only **HelloWorld** agent instance present. Since the value of the precondition will be found to be **true**, the **HelloWorld** agent instance will be destroyed. Try to run this modified example. You will notice two things being different with respect to the previous version:

Figure 11.3–1: Screen snapshot of 2nd modification of the Hello, word", example application.

(a) when pressing the **quit** button, even though the main-window disappears, the I-GET application does not exit; and (b) if you have pressed the **doExtra** button a number of times, the ceated windows still remain on the display and are not affected. In Figure 11.3-1, this scenario is shown in a screen dump from a real execution of the newest version. One important difference with the previous version, is that we have added in the destructor of the **HelloWorld** agent class an explicit toolkit-specific call for

destroying the main window created by this agent. This is necessary, since, destruction of lexical objects at the I-GET language layer, which is done automatically when the own agent instance of instantiated lexical objects is destroyed, does not generate any destruction messages to the toolkit server. Hence, you have to explicitly call the proper output events to "command" the toolkit server destroy the object hierarchies. In our example, we have used the **DestroyObject** output event of the Windows imported toolkit (i.e. **DeskTop** lexical layer) for this purpose, supplying the top-level window of the **HelloWold** interface component as a parameter. When the Windows server will dispatch this output event, it will destroy the whole object hierarchy starting from **{win}** object instance which was given as a parameter. You may find this output event in the **win.gdd** file, in the **WIN\SPEC** directory of the I-GET software release.

For the UNIX version, there is a similar output event for object destruction at the toolkit server side, called also **DestroyObject**, and taking similarly a single lexical object instance parameter. You may view its definition in **xaw.gdd** file, within the **XAW/SPEC** directory of the I-GET software release.

One might wonder why such an explicit call to the output event for physical object destruction was not dictated before. The answer is that, in the previous versions, the call to the **terminate** statement caused an automatic shut-down of both the dialogue program, as well as the running toolkit servers. During shut-down, toolkit servers destroy automatically any instantiated objects, and as a result, without the need of explicit destruction, the various created objects were guaranteed to be implicitly destroyed on the shut-down actions of the server. We may switch back to the old behaviour, i.e. closing everything, by making only a versy simple change, on the newst version we have just discussed: substitute the call to the **DestroyObject** output event with a call to the **terminate** statement. This will initiate a shut-down procedure in the destructor of the **HelloWorld** agent class, causing the whole dia-

logue application to terminate. We provide below the complete program listing.

```
headers [
extern ["igetwin.hh"]
#include "win.gdd"
]

bool makeExtra=false;
int extraSerial=1;

Font font=[
    "Arial",
    14,
    false,
    false,
    false,
    false,
    0
];

agent ExtraDialogue create if (makeExtra==true) [
        lexical(DeskTop) FrameWindow win;
        lexical(DeskTop) StaticControl label: parent={win};

        constructor [
                {win}.x={win}.y=extraSerial*10;
                {win}.width={win}.height=100;
                {label}.x={label}.y=5;
                {label}.width={label}.height=95;
                {label}.label="Extra #"+extraSerial++;
                {label}.font=font;

                out(DeskTop).RealizeObject({win});
        ]
        destructor []
]

bool closeHelloWorld=false;

agent HelloWorld create if (true)
destroy if (closeHelloWorld==true) [

        lexical(DeskTop) FrameWindow win;
        lexical(DeskTop) StaticControl label: parent={win};
        lexical(DeskTop) Button quit: parent={win};

        lexical(DeskTop) Button doExtra: parent={win};
        method {doExtra}.Pressed [ makeExtra=true; ]

        method {quit}.Pressed [ closeHelloWorld=true; ]

        constructor [
                printstr("Hello, world! from command line.\n");
                {win}.x={win}.y=0;
                {win}.width=300;
                {win}.height=250;
                {win}.title="Hello, world application";


                {label}.x=5;
                {label}.y=10;
                {label}.width=90;
                {label}.label="Hello, world!";
                {label}.font=font;
```

```
                {quit}.x=5;
                {quit}.y=50;
                {quit}.width=90;
                {quit}.height=40;
                {quit}.label="Quit";
                {quit}.font=font;

                {doExtra}.x=5;
                {doExtra}.y=110;
                {doExtra}.width=90;
                {doExtra}.height=40;
                {doExtra}.label="Extra!";
                {doExtra}.font=font;


                out(DeskTop).RealizeObject({win});
        ]
        destructor [
                printstr("Good bye world! from command line.\n");
                terminate;
        ]
]
```

Now we will show the alternative agent instance destruction technique, being destruction via a statement. We wish to add the following behaviour to our most recent program version: add a push button in each separate extra window created, which will close that window when pressed. In order to do that, we take two main steps: (i) we add a **Button** lexical object instance in the **ExtraDialogue** agent class; and (ii) we add a **Pressed** method implementation for this object class, which causes the respective agent instance to be destroyed. Letrs try to accompish this goal with the following code changes, and test its behaviour:

```
bool closeExtra=false;

agent ExtraDialogue ...

destroy if (closeExtra==true) [

    lexical(DeskTop) Button quit: parent={win};

    method {quit}.Pressed [ closeExtra=true; ]

    destructor [ out(DeskTop).DestroyObject({win}); ]

]
```

You will also have to add some code for physical attributes, such as position, labels, width and height, which we omit here for clarity. When running this

example, try to press more than one times the *Extra!* labelled button, before doing anything else. The effect will be the creation of multipe overlapping windows, as in Figure 11.3-1, which now have one additional button labelled *Close*. Press this button for any of those windows. Surprisingly, all of the windows now disappear at once. Clearly, this is not the desirable behaviour. Lets see, then, why did that happen. When the **quit** button is pressed for any of the **ExtraDialogue** agent instances, its respective method implementation will be called, thus setting the value of **closeExtra** to **true**. This will cause re-evaluation of the destruction preconditions, engaging the **closeExtra** boolean variable, of all **ExtraDialogue** agent instances. Since all those destruction preconditions will evaluate to **true**, the I-GET run-time system will automatically destroy their respective agent instances, hence, all the **ExtraDialogue** instances. We have used this example to show that global variables may be good for instantiation preconditions, but to allow separate destruction (i.e. not massive), they are not approrpiate. We will later show a specific technique for defining destruction preconditions which may be used for non-massive destruction. For the time being, we will try to change the example so that when the **quit** button is pressed, only the owner agent instance is destroyed. The following code acomplishes this goal.

```
/* bool closeExtra=false; */
agent ExtraDialogue ...
/* destroy if (closeExtra==true) */ [
     method {quit}.Pressed [ destroy {myagent}; ]
]
```

As it is shown, the code for the destruction precondition is no longer needed. What we have done is to change the implementation of the **Pressed** method for the **quit** button to call an agent destruction statement for the owner agent instance. The destruction statement is **destroy**, and accepts an agent identifier type, while the owner agent instance, i.e. similar to the **this** pointer in C++, is gained via the **{myagent}** expression. Below we provide the updated

ExtraDialogue agent class definition.

```
agent ExtraDialogue
create if (makeExtra==true) [

        lexical(DeskTop) FrameWindow win;
        lexical(DeskTop) StaticControl label: parent={win};
        lexical(DeskTop) Button quit: parent={win};

        method {quit}.Pressed [ destroy {myagent}; ]


        constructor [
                {win}.x={win}.y=extraSerial*10;
                {win}.width={win}.height=100;
                {label}.x={label}.y=5;
                {label}.width={label}.height=40;
                {label}.label="Extra #"+extraSerial++;
                {label}.font=font;
                {quit}.label="Close";
                {quit}.x=5;
                {quit}.y=50;
                {quit}.width={quit}.height=40;

                out(DeskTop).RealizeObject({win});
        ]
        destructor [ out(DeskTop).DestroyObject({win}); ]
]
```

11.4 Preconditions based on API communication events

The I-GET tool provides an Application Interfacing (API) method which supports both a shared space of typed objects, as well as message channels. The API space is specified in the I-GET language by the definition of the type of objects which may be communicated either in the shared space or in the message channels; such a specification is part of the dialogue implementation code. The role of the API space is to bridge the dialogue implementation program, written in the I-GET language, with the Application Component, which is to be written in C++. During run-time, a special-purpose program, called the API component, which is supplied as part of the I-GET software release, will establish and manage the API commnication space. In the dialogue implementatoin, the developer is given two sets of language constructs, relevant to API communication:

- *API statements*, which may be used as normal programming statements, for creating / modifying / reading / destroying shared objects, and for sending messages through message channels. Those statements are discussed under Section 13.6 of this Volume.

- *API events*, which provide notification for creation / modification / destruction of shared objects, as well as for incoming messages within message channels. API events may be handled via agents, through a special category of preconditions called API-based preconditions. Such preconditions, as it is expected, refer to the presence of particular API events during run-time.

Below we provide the syntax for the definition of the API space with examples, in which shared data types as well as message channels are defined, and the syntax for API-based preoconditions with examples, through which various types of events in the API communication space may be captured.

Grammar syntax for definition of shared data types, and channels of typed messages.

SharedDataType:

shared ID type name **;**

ChanelDefinition:

channel ID channel name **of ID** type name **;**

```
struct Employee [
    string name, address, role, task;
];
shared Employee;
struct LoginRequest [ string userid, host, group; ];
```

```
channel Login of LoginRequest;
```

Grammar syntax for API events which capture creation, modification and destuction of shared data types.

SharedCreateEvent:

shcreate(ID type name **ID** place data var**, ID** place shid var**, ID** place tid var**)**

SharedUpdateEvent:

shupdate(ID type name **ID** place data var**, Shid, Tid)**

Shid : ID place shid var *or* **? ID** match shid var

Tid : ID place shid var *or* **? ID** match tid var

SharedDestroyEvent:

shdestroy(Shid, Tid)

MessageEvent:

message(ID type name **ID** place data var**, ID** tid**)**

The complete grammar syntax for instantiation / destruction preconditions of agent classes.

Precondition:

(SharedCreateEvent) *or*

(SharedUpdateEvent) *or*

(SharedDestroyEvent) *or*

(MessageEvent) *or*

(Expression)

```
int empSerial=1;
agent DisplayEmployee
create if (shcreate(Employee emp, empid, tid_1))
destroy if (shdestroy(?empid, tid_2)) [
      lexical(DeskTop) FrameWindow win;
      lexical(DeskTop) StaticControl text: parent={win};
#define ETEXT(e) \
e.name+'\n'+e.role+'\n'+e.task+'\n'+e.address
      agent OnUpdateEmployee
      create if (shupdate(Employee newemp,?empid, tid_3)) [
           constructor [
                {text}.label=ETEXT(newemp);
                destroy {myagent};
           ]
           destructor []
      ]
      constructor [
           {win}.x={win}.y=empSerial*10;
           {win}.width={win}.height=100;
           {text}.label=ETEXT(emp);
           out(DeskTop).RealizeObject({win});
      ]
      destructor [ out(DeskTop).DestroyObject({win}); ]
]
agent CloseDown
create if (message(string msg,tid)) [
      constructor [ if (msg=="Exit") terminate; ]
      destructor []
```

]

In the above example, we show how in the dialogue control implementation we can handle the following development scenario: the Application Component exposes in the shared space **Employee** records as shared objects; the content of those objects may be subject to change, while the Appication Component may also destroy shared objects; the dialogue is required to display directly each created **Employee** record in a separate window, map automatically any updates made on the display, and remove such windows when their respective shared **Employe** record is destroyed. Additionally, the dialogue will terminate when a string type message is received with the content *"Exit"*.

The agent class **DisplayEmployee** has a **shcreate** precondition, ensuring that each time an **Employee** type shared object is created, one agent instance will be also created. The **emp** identifier in the precondition is the name of an **Employee** variable, in which the I-GET run-time system will store the particular content of the newly created **Employee** object; this variable can be accessed normally from within the **DisplayEmployee** agent class body. Within the agent class, a few lexical objects instances are declard for displaying the **emp** variable content. In the constructor of the **DisplayEmployee** agent class, the assignment of **ETEXT(emp)** to **{text}.label** ensures that upon creation, the content of each **Employee** object will be mapped on the display. However, this does not suffice for subsequent updates of previously created shared objects. For this purpose, the **OnUpdateEmployee** agent class is defined, embedded within the **DisplayEmployee** agent class, which has a precondition to grasp updates on the created shared object instances. The **?empid** expression *means match with the* **empid** *shared object identifier*, where **empid** is a variable of the owner agent instance. Hence, for each different **DisplayEmployee** agent instance, being associated with one particular shared object identifier stored in **empid**, instances of the **OnUpdateEmployee** class locally created, will only concern updates on that

particular shared object instance. The **OnUpdateEmployee** instance will simply refresh the updated content on the display and directly "kill" itself. In all API-based preconditions, there is also an identifier supplied mentiond as "tid", being an abreviation of "transaction identifier". You may supply a variable name in which those values are simply placed when the event is detected, as we have done in the example above, or you may wish to match through a specific value via the **?** matching operator. Each event in the API-space is associated with an integer transaction identifier, which is supplied by the event originator, not the I-GET systen itself (in contrast to the case of shared identifiers). The event originator will be either the Application Component or the dialogue program; you will notice that all API-specific statements have one parameter being the transaction identifier you wish to supply; when the statement will be executed, it will generate an event in the shared space, carrying this specific transaction identifier value.

You may employ transaction identifiers to add an extra matching parameter in filtering event detection as needed. Lets study a realistic scenario in which this would be needed: you supply, through instances of the same agent class, multiple windows to the user, each providing a dialogue box for querying records from a data base. The user will be facilitated to edit queries, and then submit them by the press of a "submit" button, while returned results should be displayed in the same window that its respective query has been edited. The user may have multiple such search dialogue boxes concurrently, and may also have parallel sumbitted queries being executed. To do that, it is required that each agent instance, providing one separate query dialogue, should filter somehow the returned results, so that only those returned due to the particular edited query may be only displayed; this can be done via the transaction identifier easily, as we sketch below:

```
int agentSerial=1;
channel Query of string;
agent QueryDialogue ... [
       lexical(DeskTop) FrameWindow win;
       lexical(DeskTop) Button sumbit: parent={win};
       lexical(DeskTop) EditControl query: parent={win};
       lexical(DeskTop) StaticControl results: parent={win};
       int mySerial=agentSerial++;
       method {submit}.Pressed [
            Query<-{query}.text, mySerial;
       ]
       agent DisplayResults
       create if (message(Query results, ?mySerial) [
            constructor [
                 {results}.label=results;
                 destroy {myagent};
            ]
            destructor []
       ]
       constructor [...] destructor [...]
]
```

In the above example, each created agent instance is given a unique serial number, via the global **agentSerial** integer variable, locally stored at the **mySerial** variable, which is used as the transaction identifier, when each particular query is sent to the Application Component via the **Query** channel of string message packets. This serial number is used to match the results received via the **Query** channel, so as to ensure that the received results, through the **DiplayResults** agent instantiation, in each **QueryDialogue** agent instance, match the particular user query made through that instance. This is a typical scenario which cannot be handled unless there is a matching parameter.

11.5 More on local constructs allowed within agent classes - examples

Variable declrations allowed at the agent class definition header, before the main block.

Within agent classes, there are various categories of items which may be defined. You may review the grammar rules for agent class definition syntax to see which types of constructs may be "locally" defined. You may notice from the agent class header, that variable declarations are allowed at a point before the opening square bracket **[**, which actually indicates the beginning of the agent class body. Variables at that point are local to the agent class, however, they may be also accessed in the destruction precondition. This, as we will show in the example which follows, allows the definition of destruction preconditions which affect only the particular agent instance (since the precondition still accesses local variables).

```
agent DisplayData (Data data)
bool close=false;
destroy if (close==true) [
      lexical(DeskTop) FrameWindow win;
      lexical(DeskTop) Button quit: parent={win};
      method {quit}.Pressed [ close=true; ]
      ...
]
```

In the above example, we define an agent class which is intended to be used to display information of a hypothetical **Data** type. The agent class may be instantiated only through a call, while in its formal arguments the specific **Data** information instance to be displayed is supplied. Multiple instantiations are naturally allowed, hence, multiple sub-dialogues due to the various instances may be concurrently available to the user. Each such sub-dialogue contains a **quit** button, through which it may be closed by the user. The agent class encompasses a destruction precondition which accesses the local **close** variable, thus ensuring that each agent instance maintains a separate copy. As a result, when the **quit** button is pressed for any of the agent in-

stances available at run-time, the local **Pressed** method implementation will be called, thus assigning to its local **close** boolean variable the **true** value. This will make the destruction precondition of the agent instance to evaluate to **true**, thus leading in the automatic destruction of that particular agent instance by the I-GET run-time system.

Monitors and constraints may be defined locally within the body of agent classes.

Up until now, we have seen the declaration of lexical object instances, and variables, as well as the implementaton of methods. Methods relate to the so called "call-back" model, in which you register code to be executed when particular notifications are detected at run-time. In that sense, methods can be characterised as more declarative dialogue control constructs, since you define mainly "what" is to be done, without bothering a lot about "how" the detection of the relevant conditions (i.e. notification) takes place. Apart from methods, which concern virtual or lexical object instances, we have seen preconditions being a pure declarative construct, allowing the instantiation of agent classes only on the basis of automatically evaluated conditions or particular detected API events. Within agent classes, the definition of monitors and constraints is also facilitated. Each created agent instance will have a local copy of the monitors and / or preconditions defined within its agent class body. Below we provide some examples from monitors and preconditions.

```
agent LineEquation (real *a, real* b, real* y, real* x) [
      *y := *a * *x + *b;
      constructor [] destructor []
]

Data applicationData;
agent DataView_1 create if (true) [
    void Display(Data data);
    applicationData: [ Display(applicationData); ]
    constructor [Display(applicationData); ] destructor []
]
agent DataView_2 create if (true) [
```

```
    void Display(Data data);
    applicationData: [ Display(applicationData); ]
    constructor [ Display(applicationData); ] destructor []
]
```

The above example aims to demonstrate the practical value of constraints and monitors, when used in conjunction with agent classes; more information on the syntactic or semantic aspects of these two declarative constructs are supplied under Section 5 (monitors) and Section 6 (constraints) of this Volume. The agent class **LineEquation** is an agent class which may be instaniated via a call. It will apply a line-equation constraint to its ***y** formal argument, while all enagged parameters are supplied as pointer vaiables to ensure that the original variables, not copies of values, will be actually engaged.

Following, two agent classes, namely **DataView_1** and **DataView_2**, with evident similarities, are defined, for which, one instance is at run-time created (since a constant **true** instantiaition precondition is supplied). The role of these two agent instances is the provision of two alternative displayed views of the same internal information pieces (i.e. **applicationData** of a hypothetical **Data** type). To achieve this goal, each of the agent classes has various object instances locally declared, omitted from our example for clarity, as well as an appropiate **Display()** function. In each agent class, a monitor is defined locally for the **applicationData** variable, which makes a call to the local **Display()** function; hence, each time the **applicationData** content is modified, it is guaranteed that the change will be mapped to both available user views. The example can be generalised for multiple views. Next, we will briefly show how user defined data-types and functions may be be defined localy within agents. Scope rules apply for user-defined data types in the same manner that they apply for variables and functions; hence, you may define a data type locally having the same name with a global data type.

User-defined data types and functions.

```
struct A [ int a; ];
enum B=[one, two, three];
```

```
int F() [ return -1; ]
A a;
B b;

agent MyAgent create if (true) [
    void F() []
    struct A [ string a; ];
    struct B [ string b; ];

    int i=F();          // Error. Local F is 'void'.
    int j=::F();        // Ok. Global F is 'int'.
    A a=::a;            // Error. Different types.
    ::A a=::a;          // Ok. Same types.
    int G();            // Ok. Function prototype.
    int q=G()+::F();    // Ok. Types match.
    constructor []
    destrcutor []
]
```

In the above example, we show how data types can be declared both globally as well as locally, with the same name, however, still enabling resolution via the global scope resolution operator **::**. Also, when variables are of structure data types, assignement is allowed only if both are of the same structure type. Function prototypes are also allowed within agents; the global scope resolution operator applies to function calls as well, apart from variables.

About externally referenceable local members, and scope qualification.

Data types, functions and variables are a special category of local items in agent classes called *externally referenceable local members*. As we will show in Section 11.6 o this Volume, agent instance variables are allowed, called agent references, which are similar to class instances in C++. Via such instance variables, externally referenceable local members may be accessed, in a way analogous to accessing class members via instance variables in C++. For this purpose, scope qualification is supported within agent classes, to define which externally referenceable members may be accessed from

outside the body of agent classes. In this context, there are two scope categories upported: public scope, allowing members to be accessed from outside the agent class body, via instance variables, and private scope, disabling such access. It is important to mention that other local items, such as constraints, monitors or method implementation, are not referenceable constructs, i.e. are not meant to be referenced or manipulated, outside their specification block. Below, we provide examples for scope qualification, while more examples will be given in the following Section, when discussing instance variables.

```
agent ConfirmBox ...[
   public:
   string okMsg="Ok",cancelMsg="Cancel", noMsg="No";
   string mainMsg="MESSAGE GOES HERE";
   void Close();
   void Hide();
   void Show();
   private:
   void Close() [  // Error. Prototype was 'public'.
   ]
   enum Dialoge=[Synchronous,Asynchronous];
   public:
   Dialogue style; // Error. Public var, private type.
   ...
]
```

11.6 Instance variables for agent classes and forward agent class definition

Instances variables are supported for all types of agent classes, independently of the instantiation style supported (i.e. precondition-based or call-based). There is a built-in agent instance type supported in the I-GET language, which is applicable for all agent classes, i.e. something like a super-

class of all instance variable types, called **daid**.

```
daid agentId=nil;
agent FirstAgent create if (true) [
    constructor [ agentId={myagent}; ]
    destructor [ terminate; ]
]

agent SecondAgent create if (true) [
    constructor [ destroy agentId; ]
    destructor []
]
```

The above program will behave as follows: Firstly, the **agentId** variable will be assigned with the **nil** value. Then, an instance of the **FirstAgent** class will be created, whose instance reference value will be assigned (when the constructor block is called) to the **agentId** variable. Following, an instance of the **SecondAgent** class will be created; however, by calling its constructor block, the **FirstAgent** instance will be destroyed, due to the execution of the **destroy agentId** statement. As a result, the destructor of the **FirstAgent** instance is called, inherently making a call to the **terminate** statement, which shuts-down the dialogue appplication immediately.

Now we will present the specifics of the agent instance variables, including syntax, semantics and various examples. Instance variables may be declared at any point that normal variables are allowed (the same also holds for reference variables on virtual or lexical object instances). The syntax is similar to variable declarations, differing only on the type specifier, with respect to normal variables where it is the name of a built-in or user-defined data type.

`

AgentInstanceVarTypeSpecifier:

ref agent ID agent class

```
agent A () [
   public:
   int a, b, c;
   void F();
   private:
   int i,j, k;
   string G();
   public:
   struct A [ int a; ];
   private:
   enum B=[cannot, access, from, outside];

   constructor [] destructor []
]

agent B() [ constructor[] destructor[] ]

daid agentId=nil;;
ref agent A A_id1=nil, A_id2=nil;

agent StartUp create if (true) [
constructor [
   agentId=create A();      // Ok. Superclass.
   A_id1=create B();        // Error. Class mismatch.
   A_id1=create A();        // Ok. Same classes.
   A_id2=agentId;           // Ok. Super-class.
   {A_id1}->F();            // Ok. 'F' is public.
   {A_id1}->G();            // Error. 'G' is private.
   {A_id1}->a={A_id1}->i;   // Error. 'i' is private.
   agent::A::A my_a;        // Ok. 'A' is public.
   agent::A::B my_b;        // Error. 'B' is private.
]
destructor []
]
```

In the above example, we show the use of agent instance variables for ac-

cessing agent members, as well as the type matching regulations which apply. The **create** statement returns a value which can be assigned either to a variable of **daid** type, or to an instance variable of the same agent class that is passed in the **create** statement. For the **destroy** statement, a **daid** variable, or any type of agent instance variable may be supplied. We also demonstrate the way in which public data types defined locally within the agent class body may be accessed from outside, something that is supported with a similar syntax for both virtual as well as lexical object classes.

Data types may be accessed without the need of an instance variable, just by qualifying with the class name. Hence, in our previous example, the public data type **A**, of agent class **A**, could be accessed from outside as **agent::A::A**, where the first **A** designates the owner agent class, and the second the data type. Hence, the general syntax is:

AgentQualifiedTypeName:

agent::ID agent class **::ID** type name

11.7 Embedded agent classes

The I-GET language supports the specification of agent classes within other agent classes. This feature is syntactically similar to the support of embedded classes in C++, however, only syntactically. The definition *of embedded agent classes* results in the creation of agent class hierarchies. Since agents are dialogue *control* constructs, the specification of an agent class X within another agent class Y defines a controlling relationship among instances of these two classes, as opposed to a part-of relationship which would result in the case of compositional constructs:

> *The X child agent class is syntactically visible only within the body of the parent agent class Y. At run-time, every created*

instance of the X agent class (i.e. controlled) is automatically associated with one specific instance of the Y agent class (i.e. controller) ; we will explain later on how the controller agent instance is determined during run-time.

Next, we will start with some simple specification examples. Then, we will discuss the run-time dependencies among the controller and controlled agent instances; such dependencies result when instances of agent classes engaged in agent hierarchies are created.

11.7.1 Specification examples

We will update slighly the example of Section 11.3, in which we have added an **ExtraDialogue** agent class in our basic "Hello,world" example. In the modified version we sketch below, the **ExtraDialogue** agent class is defined inside the **HelloWorld** agent class:

```
agent HelloWorld ... [
      bool makeExtra=false;
      agent ExtraDialogue ... [
      ]
      ...
      method {doExtra}.Pressed [ makeExtra=true; ]
      ...
]
```

In the previous example, the embedded agent class has been defined to be instantiated via a precondition. It is naturally allowed to define agent classes instantiated by call within other agent classes. The idea of embedded agent classes, apart from supporting the principle of "information hiding" in OOP traditions, allows the more natural mapping of hierarchical dialogue control schemes to coding patterns in the I-GET language. More specifically, consider the following development scenario:

An interactive document processor application is to be developed. The application provides a main window which encompasses the text-editing area and some icons to activate various panels / windows for facilities such as: font style manipulation, spell checking, drawing, etc. This is similar to facilities supplied by word-processors like MS Word™.

Next, we provide a sketch of the impementation which shows how the dialogue will be organised arround a hierarchy of agent classes. Many details are ommited for clarity, since the impementation pattern is more important in this case:

```
agent DocumentProcessor create if (true) [

    // Here we use some virtual Toggle buttons. Check the
    // file virtual/virtual.gdd and the relevant Section
    // in this Volume. We leave empty the parent instance,
    // indicated with ... , since this depends on the
    // particular toolkit being used.

    virtual Toggle fontTool ... ;
    virtual Toggle spellTool ... ;
    virtual Toggle draw Tool ... ;

    agent FontTool
    create if ({fontTool}.State==true)
    destroy if ({fontTool}.State==false) [ ... ]

    agent SpellTool
    create if ({spellTool}.State==true)
    destroy if ({spellTool}.State==false) [ ... ]

    agent DrawTool
    create if ({drawTool}.State==true)
    destroy if ({drawTool}.State==false) [ ... ]
```

```
] // DocumentProcess agent class
```

One migt notice that in the above example the specific sub-dialogues are unnecessarily destroyed and re-created upon user unselection and reselection. This is not actually the case. The way each agent class is implemented is open to the developer. One may choose to have an embedded agent class inside each of those agent classes, which would create only once the interface objects (i.e. upon the first selection) for the three dialogue components; then, upon unselection, only the main window(-s) of the component(-s) would be hidden, while upon reselection, they would be switched back to the visible state again.

11.7.2 Run-time dependencies and links

As mentioned, control relationships are automatically established during run-time, upon the creation of new agent instances. Each new agent instance will be assigned to a specific parent instance, depending on the execution context which caused its creation. In the I-GET language, there are two possible execution contexts which may cause agent instantiation: (a) the instantiation preconditions as such, which are handled automatically by the I-GET run-time system; and (b) the agent instantiation statements, which may be called by the dialogue developer in various points within the code. According to each execution context, which may cause the creation of a new agent instance, its parent instance is defined as follows:

- If the execution context is a precondition, then the parent instance is the one which has installed that particular precondition at run-time.

- If the execution context is a statement, the parent instance is the one for which the specific code block, which includes the instantiation

statement recently executed, has been called.

The way in which preconditions are installed / cancelled in / from the global precondition table is defined by specific regulations defined in the Table which follows (those rules denote some of the standard actions that all agents perform upon instantiation and destruction). It should be noted that those details may be closer to the run-time mechanisms, rather than to the I-GET language features. However, it is believed that this information is important for dialogue developers, since it helps understand better how agents are created and destroyed upon execution; such a level of detail is also provided in mainstream programming languages like C++, since it allows developers to predict, or to justify, the effects of their implementation blocks.

1	Upon start-up all instantiation preconditions of toplevel agent classes are installed.
2	When an instance of an agent class is created, the instantiation preconditions for all of its child agent classes are installed. Its destruction precondition, is also installed.
3	When an agent instance is destroyed, the instantiation preconditions for all of its agent classes installed upon instantiation, are now cancelled. Its destruction precondition is also cancelled.
4	When an agent instance is to be destroyed, all its child agent instances are destroyed first.

11.7.3 Scope rules

Child agent classes inherit the scope of their ancestor agent class, and thus have access to all construts that can be directly referenced by any ancestor agent class. For instance, consider the example below (preconditions, parameters, constructors and destructors have been removed for clarity):

```
1: agent X [
2:     int i;
3:     agent Y [
4:         int j;
5:         agent Z []
       ]
```

```
]
```

Within agent **Y**, the local variable **i** of agent **X** (line 2) can be accessed. Similarly, variable **j** (line 4) can be accessed from agent **Z**, and also variable **i**, since **i** is accessible from the parent of **Z** (i.e. **Y**).

11.7.4 Scope resolution operator

At run-time, multiple instances of a child agent class may exist, resluting in corresponding multiple indepependent instances of the their respective local items. Due to agent hierarchies, name collisions may result. In this case, priority is naturally given to local items. For resolution of non-local items, the scope resolution operator may be applied. The following example shows how scope resolution applies to different types of items which may be defined within agent classes (varius details are ommited for clarity).

```
1:    agent X [
2:          virtual Button a; int i;
3:          void f() [ i++; ]
4:          agent Y [
5:                virtual Button a; int i;
6:                method {X::a}.Pressed []
7:                method {a}.Pressed []
8:                constructor [
9:                      i=X::i;
10:                     X::f();
11:                     f();
12:               ]
13:         ]
14:         constructor [ if (X::i==i) printstr ("!"); ]
15:   ]
```

Scope resolution has the syntax <*agent class*>*::*<*item name*>, while in case of global items, it is suffices to write *::*<*item name*>. The name of the agent class used in scope resolution, should belong to the set of ancestor agent classes for that agent class, in the body of which, the scope resolution operator is used. As a result, **X::i** (line 10), **X::f()** (line 11), and **X::a** (line 7), when used within agent class **X**, refer always to the **i** (line 2), **f** (line 3) and **a** (line 2) of agent class **X**. Finally, due to line 14, every new instance of the **X** agent class will print an exclamation mark ! in the standard terminal output, since **X::i** and **i**, at that syntactic context, are the same integer object.

Chapter 12

12. Event handlers

Event handlers are one of the earliest models for implementing dialogue control, and is currently supported in all known toolkits. The basic model is based on the definition of "reaction code" upon user input. What is different among different toolkits is the way handling of events is programmed, i.e. the event handling programming model. Irrespective of the way event handling is supported, there are three fundamental constructs:

- Event class, for which handling is to be defined;
- Code to be executed upon detection of events belonging to the the specific event class;
- Interaction object in the physical context of which (i.e. screen area) handling of the specific event class is to be installed.

The various input event classes which are on the disposal of dialogue developers for event handling, are those which are defined in the context of toolkit interface specification. Their specification begins with the keyword **inputevent**, and it looks very similar to the definition of structure data types. We will discuss below the various event categories supported in the I-GET language.

12.1 Input and Output events

In the I-GET language there are two categories of event classes, which are directly linked to the toolkit level:

- Input event classes, which primarily concern device-level user-input, however, may model other types of events such as toolkit notifications (e.g. window exposure, move, minimisation / maximisation);
- Output event classes, which model toolkit facilities in a form which looks similar to function calls; output events address all types of toolkit functionality which is to be supplied to dialogue developers.

Only input events can be associated with event handlers, while output events can be "called" in a way similar to conventional function calls. Both input and output event classes which are to be supported for an imported toolkit are defined as part of the toolkit interface specification. Below we provide some excerpts from the Windows toolkit interface specification, showing some input and output event classes:

File win\spec\win.gdd, line 426.

```
inputevent WindowPosChanging (DeskTop) [
        int x,y,width,height;
]

inputevent KeyDown (DeskTop) [ // non system key
        string key;
        int repeatCount;
]
```

File win\spec\win.gdd, line 445.

```
#define MODKEYS         \
        bool control;   \
        bool lButton;   \
        bool mButton;   \
        bool rButton;   \
        bool shift;

inputevent LeftButtonPressed (DeskTop) [
        int x,y;
        MODKEYS
```

```
]

inputevent LeftButtonUp (DeskTop) [
        int x,y;
        MODKEYS
]
```

File win\spec\win.gdd, line 558.

```
outputevent AllocateGraphics (DeskTop) [
        out:
        in:
        int     gid;
]

outputevent DrawText (DeskTop) [
        out:
        int             gid;
        objectid        obj;
        string          text;
        int             x, y;
        int             width, height;
        in:
]
```

File win\spec\win.gdd, line 70.

```
// Additional services for the ListBox object class.
//
outputevent AddString (DeskTop) [
        out:
        objectid theList;
        string item;
        in:
        int pos; // new item's index, -1 if error
]

outputevent InsertString (DeskTop) [
        out:
        objectid theList;
        string item;
        int index;      // if -1 append at the end
        in:
        int pos;
]
```

As it is naturally expected, input event classes start with the keywrod **inputevent**, while output event classes start with the keyword **outputevent**. Then, the name of the event class follows, together with the name of the particular imported toolkit, in parenthesis **()**, with which it is to be associated. For example, the prefix **inputevent KeyDown (DeskTop)** defines an input event class named **KeyDown** belonging to the impored

toolkit named **DeskTop** (which, in the I-GET public release, corresponds to the Windows toolkit library).

12.2 Basic specification structure

Syntax for the specification of event handlers.

EventHanlder:

eventhandler ObjRef *lexical object* **[EventBlocks]**

EventBlocks :

EventBlocks EventBlock *or*

EventBlock

EventBlock:

(**VariableDeclaration** *)**

(Expression) ID *event class* **Compound**

An event handler starts with the keyword **eventhandler** and continues with the definition of a lexical object instance with which the specific event handler is to be attached. Then, within an event handler block, multiple event blocks may be defined. An event block handles a specific event class. Examples are given below. It should be noted, that event handlers may only be defined in the body of agent classes; however, in our examples we may omit for clarity the particular agent class syntactical context.

```
lexical (DeskTop) FrameWindow win;

eventhandler {win} [

(true) KeyDown [
      printstr(KeyDown.key);
]
```

```
(true) Moved [
      pintstr("Moved at ("+Moved.x+","+Moved.y+").\n");
]

]
```

As it is shown from the example above, every event block starts with a boolean expression surrounded by parenthesis (those are always required). At run-time, for each event found in the event queue, an event bock defined for its event class will be called only if the associated expression (to the event block) evaluates to **true**. By defining this expression to be **(true)** for an event block, we ensure that the associated event block will be unconditionally called, when an event matching its handled event class is present in the queue. So, for the above example, each time an event of the **KeyDown** or **Moved** event class is present on the head of the event queue, the defined event blocks will be called. We will show later on the advanced use of such an expression which supports conditional event processing.

From within the event block, the event parameters may be syntactically accessed in a form similar to the access of fields in structure variables. Hence, **KeyDown.key** accesses the **key** parameter of the **KeyDown** event class.

12.3 Event preconditions for conditional event processing

Event preconditions have been originally introduced by Ralph Hill, as part of his theoretical Event Response System (ERS) model, which has been supported practically by his Sassafras UIMS and the Event Response Language (ERL); more information may be found in his article "Supporting Concurrency, Communication and Synchronization in Human-Computer Interactin - The Sassafras UIMS", in ACM Trans. Gr. 5, 3 (July 1986), pp 179-210. In the original ERS model, event preconditions are boolean flags. In the I-GET language, abitrary expressions are supported (inluding pointer variables as

well). Lets explain firstly which are the implications of such preconditions on event processing, and then lets study a scenario in which preconditional event processing helps reduce implementation comlexity.

Consider an event block for an event class **X**, having a precondition **P**. Then, during run-time, on the presence of an **X** event instance in the event queue, the I-GET run-time system will locate all event blocks engaging **X** event class, including the above event block as well. If **P** is **true** at the time the **X** event instance appears on the queue, the corresponding event block will be executed, and the event will be removed from the queue. In general, if at least one event block for **X** event class is found to have a **true** valued pre-condition, the event will be removed after the necessary event block(-s) are called. Else, it will stay in the queue for the next event processing cycle.

Hence, preconditions allow to postpone processing of an event until some particular conditions are met. One might argue that this type of functional behaviour may be modelled easily by conventional programming constructs; e.g. checking with an "if" statement a precondition within an event block. What is wrong with this approach ? the testing takes place within the event block, which means that after exiting the block, the event would have already been consumed (i.e. removed from the event queue), clearly failing to support postponed processing.

This type of postponed processing is needed in all cases that we want to synchronise easily some actions together. For instance, we may want to allow processing of an event only after a notification is received that the system has completed some internal operations. One example is the so called "type ahead" facility: a terminal window is opened, the user may type in directly, however, the keys are only processed when the underlying OS terminal connection has been established. With a conventional event handling model, all keys typed prior to "terminal connection establishment" have to be stored by

the programmer in a queue, and then, after the connection, they have to be pocessed alltogether. In our event model, there is no need for two queues:

```
bool connectionEstablished=false;
eventhandler {terminal} [
      (connectionEstablished) KeyDown [
            // Process key here.
      ]
]

agent MakeOSTerminalConnection() [
      // Agent body here.
      constructor [
            // Connection code here.
            connectionEstablished=true;
      ]
]
```

As it is shown from the above code segment, the key events are normally queued, however, they are only processed as far as the boolean variable **connectionEstablishment** is set to **true**. It should be noted that in the I-GET run-time system, events which are not processed due to a **false** precondition do not block subsequent events waiting in the queue from processing (if their precondition is **true**).

12.4 Defining multiple event blocks for the same event class

It is allowed to have multiple event blocks associated with a particular event class. In such case, it is possible that multiple event blocks may succeed to be called at run-time. The order of execution is the order of installation of event blocks at run-time. Event blocks which appear in the same event handler have as order of installation the order of appearence in the event han-

dler. Event handlers within the same agent instance are installed with their relative order of appearence within the corresponding agent class. Finally, agent instances install event handlers upon construction.

So, in the following code, if the user presses a key within object **terminal**, the expected output will be: *Blok1(E1) ok.Block2(E1)ok.Block1(E2)ok.*

```
eventhandler {terminal} [
      (true) KeyDown [ prinstr("Block1(E1)ok."); ]
      (true) KeyDown [ prinstr("Block2(E1)ok."); ]
]
eventhandler {terminal} [
      (true) KeyDown [ prinstr("Block1(E2)ok."); ]
]
```

12.5 Artificial Event Broadcasting

Artificial event broadcasting is the insertion of event instances in the system event queue by the developer in an intuitive way. For instance, the developer may wish to generate artificiallly a mouse click in a particular object, while such an event would be treated as if it would be real mouse click. Artificial event broadcasting in the I-GET UIMS is very powerful, while it is very easy to manage. There is a dedicated statement class for this purpose, which has the following syntax:

ArtificialEventBroadcasting:

in(ID$^{\text{lexical layer}}$).ID$^{\text{event class}}$(ObjRef$^{\text{lexical object}}$, ExpressionList);

```
in(DeskTop).KeyDown({terminal}, "Y",1);
in(DeskTop).WindowPosChanging({terminal},0,0,100,100);
```

```
in(DeskTop).Paint({terminal},true,0,0,100,100);
```

Chapter 13

13. Statements

The I-GET language supports a plethora of statements, some of which are general purpose programming statements, while most of them are specific to the interface building facilities of the I-GET UIMS (e.g. handling agents, manipulating events, accessing the application interfacing space). Below we provide the detailed description of each statement category, by providing firstly an explanation, followed by the syntax and representative examples.

13.1 Expression statements

In this category fall statements which are related to the basic programming expressions, by adding the statement **;** (semicolon) termination symbol. The various categories of expressions are described under Section 4 of this Volume. The syntax for expression statements is very simple and is defined below, while some examples also follow.

ExpressionStmt:

Expression ;

```
i++;
f(a->b,10);
(a || b) && PI*Theta>35;
--p;
“Hello, world”;
```

13.2 Conditional statements

These are the traditional “if”, and “if”…”else” statements. The syntax looks similar to C, while for the “if” condition, expressions of the following types may be used:

- boolean expressions;
- arithmetic expressions;
- pointer expressions.

ConditionalStmt:

if (Expression) Stmt *or*

if (Expression) Stmt else Stmt

```
sharedid id1,ids[10];
int cont, *p, i;
string exitCode;
if (id1==nil || ids[3]==id1)
      p++;
else
      p--;
if (exitCode==”OK” || exitCode==”NORMAL”)
      return 1;
if (i || p)
      p=p+i;
if (id1)  // illegal, id1 not in allowed expressions
      id1=ids[0];
if (!p || p==p-1)
      new(p,10);   // make a dynamic array of 10 int
```

13.3 Loop statements

There are two classes of loop statements, the "while" statement and the "for" statement. Both engage an iteration condition, for which the same type regulations, as those for the conditional expression of the "if" statement, apply.

13.3.1 *while* statement

In this statement, the "while" iteration condition is evaluated. If it is found to be **true**, the "while" statement is executed; then, the execution round starts again. If the condition fails, the statement following the "while" statement is executed.

WhileStmt:

while (Expression) Stmt

```
bool process=true;
while (process) [
      // Necessary code here.
      if (ShouldStop())
            process=false;
]
```

13.3.2 *for* statement

This is another well known statement, in which the programmer defines some initial statements, an iteration condition, and some statements to be executed after the end of each iteration. The "for" statement is executed only if the iteration condition is found to be **true**.

ForStmt:

for (ExpressionList; Expression; ExpressionList) Stmt

```
int i, arr10[10];
for (i=0; i<10; i++)
      if (i!=9)
            arr10[i]=I;
      else
            arr10[i]=-1;
int *p;

for (p=arr10;   *p!=-1;  p++)
      printstr("*p="+*p+"\n|);

p=arr10;     // we can do the same with a "while"
while (*p!=-1) [
      printstr("*p="+*p+"\n");
      p++;
]
```

13.4 *case* statement

The case statement is similar to the "switch" statement in C and the "case" in Pascal, though it is more powerful and flexible in the I-GET language. There is a matching expression provided (the "case" expression), which is checked against lists of expressions, each list being associated with a code block. Then syntax of the I-GET case statement follows.

CaseStmt:

case (Expression) [*(* CaseBlocks *)*+ CaseElse]

CaseBlock:

ExpressionList : Stmt

CaseElse:

else Stmt

Next we define some semantic properties of this statement which show clearly the differences with the C/Pascal version:

- The Expression as well as ExpressionList may engage all types of expressions, except structures and arrays (i.e. aggregate types).
- The expressions engaged in the selection list need not be constant, but may engage variables as well. The expressions will be evaluated when the case statement is to be executed, and if no matching has been found on the preceeding (in their order of declaration) list of expressions.
- The expressions are all evaluated one after the other, from one selection list, towards the next, and if a matching is found, the rest of the expressions in that selection list are not evaluated (i.e. short circuit evaluation from left to right), while the associated statement is executed.
- Upon execution of the corresponding statement of a matching selection list, the control jumps directly to checking the next selection list, which will be executed only if it matches the case expression. This is different from C / C++ where the control directly flows to the next statement unconditionally, until a "break" statement is met; here, no "break" statement is required.
- If more than one matches are detected, belonging to different selection lists, all the corresponding statements are executed in the order of definition of their corresponding selection lists.
- If no expression from any of the selection lists matches, the "else" statement (if any, since it is optional), is executed.

Examples are provided below, which help clarify the execution semantics of the I-GET "case" statement.

```
string day=readstr();
#define WORKING_DAYS \
      "Monday","Tuesday","Wednesday","Thursday","Friday"

#define WEEKEND \
      "Saturday","Sunday

case (day) [
      WORKING_DAYS : printstr("Hmmm! You have to work!\n");
      WEEKEND : printstr("Have a nice weekend!\n");
      else
            printstr("Is "+day+" applicable in...Mars?\n");
]
// If 4 is entered, [1...4][4...8] is printed.
//
case (readint()) [
      1,2,3,4: printstr("[1...4]");
      4,5,5,6,7,8: printstr("[4...8]");
]

// In the following code, and within the first selection list
// the i is increased by one, hence, when *p+i is evaluated,
// the increased i is engaged.
//
int i,arr10[10],*p;
case (readint()) [
      i++, arr[1],arr[2],arr[3],*p+i: [ /* code here */ ]
      i : [ printstr("This 'i' has been increased by 1!\n");
]
```

13.5 Variable declaration statement

The declaration of variables is a statement in the I-GET language, in a manner similar to C++. Hence, variable decalarations may be mixed with programming statements. It is recommended that local variables are declarated at the smaller distance with the part of code in which they are to be used, to avoid any conflicts and to make code more readable. The syntax for variable declarations, as well as many examples, are provided within Section 2 of this Volume.

13.6 *API* statements

Application Interfacing (API) statements are a special category of statements which provide the means to generate events in the communication layer with the Application Component. Since this "layer" consists of a shared space and message channels, the API statements allow to: (a) send messages though message channels; and (b) create, modify, destroy and read typed objects from the shared object space. Below we discuss those statement categories.

In order to send messages or to manipulate shared objects, it is required that either the referenced message channels or the various shared object types have been already defined. Below we provide a list of the builtin message channels as well as the various builtin shared data types; you cannot redefine those. In each column, the names of the channel and shared types at the Application Component side are mentioned first (**CHANNEL_** and **SHARED_** prefixes respectively), followed by the type names as they may be used at the Dialogue Implementation (e.g. int, word, lognint).

Builtin channels		Builtin shared types	
CHANNEL_int	int	SHARED_int	int
CHANNEL_word	word	SHARED_word	word
CHANNEL_longint	longint	SHARED_longint	longint
CHANNEL_longword	longword	SHARED_longword	longword
CHANNEL_real	real	SHARED_real	real
CHANNEL_longreal	longreal	SHARED_longreal	longreal
CHANNEL_char	char	SHARED_char	char
CHANNEL_string	string	SHARED_string	string
CHANNEL_sharedid	sharedid	SHARED_sharedid	sharedid
CHANNEL_bool	bool	SHARED_bool	bool

13.6.1 *shcreate* statement

With this statement, an instance of a typed object may be created in the shared space. Each such statement will allocate appropriate memory at the shared space for placing the typed object content, while the shared space will generate a unique identifier of type **sharedid** and will return it. This identifier will have to be used when the shared object is to be manipulated by modification, destruction or read statements. The syntax follows:

SharedCreateStmt:

shcreate (Type Expression, Expression, Expression) ;

Type:

ID *or* **BuiltInForAPI**

In the above grammar rule, the *Type* non-terminal corresponds to the shared data type, which can be either a user-defined data-type (i.e. *ID*) or a built-in data-type allowed for the API space (i.e. *BuiltinForAPI*, the types which have been listed in the previous table). The second expression corresponds to the object value which is to be exposed in the shared space, and should conform in data type with the *Type* supplied. Then, the third expression is where to place the returned identifier, and should be a pointer to a **sharedid** type.

Finally, the last expression is an integer number which can be used for matching purposes as explained below:

> When a **shecreate** statement is successfully completed, the I-GET run-time system, at the API process, enqueues an appropriate event for those parties interested for a **shecreate** event (either the API or the DC components). If at the dialogue side there is at least one agent class interested for the specific event category, the DC may be both the "originator" as well as an "acceptor" of this **shcreate** event. The returned event, apart from exposed data and the shared integer identifier, encompasses this integer number you pass. You may check Section 11.4 for agent preconditions based on API events.

```
sharedid myId;
shcreate(int 3.14, &myId, 0);

struct Employee [ string name, address, role; ];
shared Employee;
sharedid empdId;
Employee myEmp=["A. Savidis","Homeless","Useless"];
shcreate(Employee myEmp, &empId, 23);
shcreate(int myEmp,&empId, 23); // error, type mismatch
```

13.6.2 *shdestroy* statement

As it is expected, this statement releases an object entry in the shared space, while further references to it will generate a run-time error (the I-GET run-time shared space manager will display an appropriate message at the console, explaining the source of such an error). The syntax is simple:

ShdestroyStmt:

shdestroy (Expression , Expression);

Here the first *Expression* should be of a **sharedid** type, and actually correspond to a value which has been returned by a previous **shcreate** statement, while the second *Expression* is the integer to be returned (i.e. for matching purposes) when the shared space will post a **shdestroy** event to the interested parties, due to this destruction statement. Examples follow below.

```
shdestroy(empId,0);     // ok.
shdestroy(empId,0);     // error, no such shared object.
shdestroy(nil,0);       // error, no such shared object.
```

13.6.3 *shupdate* statement

Via the **shupdate** statement, it is possible to supply a new content, of the same data type, for an already existing object at the shared space. This statement requires: (a) the data type name, so that type checking can be done at compile time (i.e. an **ID**); (b) the actual data content (i.e. an *Expression* which should be of the supplied **ID** type); (c) the shared identifier of the object to be modified (i.e. an *Expression* of **sharedid** type); and (d) an integer *Expression* for matching purposes (again pass any integer number, e.g. 0, if value is not important). The syntax looks very similar to the **shecreate** statement.

SharedUpdateStmt:

SHUPDATE (Type Expression, Expression, Expression) ;

Type:

ID *or* **BuiltInForAPI**

Again, if the **sharedid** *Expression* supplied is invalid, a run-time error will be generated by the shared space manager at the console. Also, an appropriate **shupdate** event will be posted by the shared space manager to the inter-

ested parties (i.e. the AC and / or the DC components). Examples follow.

```
myEmp.name="John Smith";
myEmp.address="Oxford Str. 23, London, UK";
myEmp.role="Manager";
shupdate(Employee myEmp, &empId, 10);
sharedid domainId;
shcreate(string "csi.forth.gr", &domainId, 10);
shupdate(string "netscape.com", &domainId, 10);
shupdate(string empId, &domainId, 10); // type error.
```

13.6.4 *shread* statement

The **shread** statement provides a way to read directly the content of a shared object within a dialogue variable. This statement requires: (a) the type of the shared object, again as an **ID**; (b) the **sharedid** of the object to be read, as an *Expression*; and (c) a place to store locally the data content, as an *Expression* being a pointer to the supplied **ID** type. Normal compile-time type conformance checking (of the **ID** with the type of the pointer *Expression*), as well as run-time checking (i.e. the shared identifier is valid, and the stored object should be of the **ID** supplied type) take place. The **shread** statement generates no event. The syntax follows.

ShreadStmt:

shread (ID Expression, Exprssion) ;

```
Employee thisEmp;
shread(Employee &thisEmp, empId);
string whichDomain;
shread(string &whichDomain, domainId);
int a,*p=&a;
shread(int p, myId); // if myId is ok, stores in 'a'.
```

13.7 Jump statements

Jump statements concern four types of functionality: (a) controlling the continuation of loop execution; (b) exiting from functions and possibly returning values; (c) terminating pogram execution; and (d) exiting from "case" statements. For these purposes, there are the following simple statements: "break" (loops and "case") and "continue" (loops), "return" (for functions), and "terminate" for program execution termination.

13.7.1 *break* statement

The "break" statement will cause direct exit from the inner-most loop (either a "while" or "foor" loop), and from the inner-most "case" Notice, that for the "case" statement, in the I-GET language the "break" may be used to enforce that no other matching for the "case" expression is to be checked (while in C it ensures that only the code associated to a particular 'case' value is to be executed). The syntax is simple, while if the "break" is not provided only within one of the above statement contexts, a compile error will be generated.

BreakStmt:

```
while (true) [
      if (readstr()=="exit")
            break; // Jump at Position.2.
      int j;
      for (j=0; j<readint(); j++)
            if (j==13)
                  break; // Jump at Position.1.
            else
                  printstr("good luck "+j);
      // Position-1.
```

```
]
// Position.2.
case (readstr()) [
      "f","F" : File();
      "T","t" : Tools(); break;
      userCmmd=readstr(): User(userCmmd); break;
      "t","T" : Telnet(); // This code is unreachable.
]
```

13.7.2 *continue* statement

The "continue" statement is used within 'for" and "while" loops, and causes a jump to the header of the loop statement, for performing another cycle. In the case of "for" statement, the expression list supplied after the loop condition is also evaluated. The syntax follows, with examples.

ContinueStmt:

continue ;

```
for (j=0; j<N; j++) [
      if (j==currIndx) [
            A[currIndx]=-1;
            continue;
      ]
      // Other type of processing here.
]
bool done=false;
int curr=-1;
while (!done) [
      if ({radioObjsAtr[++curr]}.state==RadioOn)
            continue;
      // Do something with 'off' radio buttons.
]
```

13.7.3 *return* statement

The "return" statement causes direct exit from a function. In case of non-void functions, it returns also an expression. If a function is non-void and you forget to return a value within a normal path of execution for the statements of this function, this may cause a run-time error. Use the I-GET automatic debug facilities for the detection of such erroneous situaitons. In the case of non-void functions, type checking is performed for conformance with the function type.

ReturnStmt:

return ; *or*

return Expression ;

```
void Inc1 (int* p) [
      if (!p)
            return;
      (*p)++;
]
int Inc2 (int j) [ return j+1; ]
int k=10;
Inc1(&k);  // k becomes 11.
k=Inc2(k); // k becomes 12.
type int arr10[10];
type arr10* arr10ptr;
arr10ptr Init(arr10 arr, int initValue) [
      int j;
      for (j=0; j<10; j++)
            arr[j]=initValue;
      return &arr;
]
```

13.7.4 *terminate* statement

The "terminate" statement causes direct termination of the running I-GET application. The effect of this statement is that: (a) the running toolkit servers will be shut-down; (b) the running Application Component will be shut-down; (c) the running Application Interfacing process will be shut-down; and (d) after all those components terminate, the Dialogue Component also terminates. The "terminate" statement is the only statement you should use to terminate the dialogue from the Dialogue Control (in other Volumes you will also notice that the Application Component or any toolkit server may initiate application termination as well).

TerminateStmt:

terminate;

```
agent ExitConfirmBox(string msg) [

      lexical(Xaw) PopupWindow win;
      lexical(Xaw) Box box: parent={win};
      lexical(Xaw) Label msg: parent={box};
      lexical(Xaw) Command yes: parent={box};
      lexical(Xaw) Command no: parent={box};

      method {no}.Activated [
            destroy {myagent};
      ]
      method {yes}.Activated [ terminate; ]

      constructor [
            {msg}.label=msg;
            out(Xaw).PopUp({win},GrabExclusive);
      ]


      destructor [
```

```
                out(Xaw).DestroyObject({win});
            ]
        ]
```

13.8 Memory management statements

Memory management statements provide two categories of functionality: (a) allocating memory dynamically (i.e. the "new" statement); and (b) destroying memory allocated previously (i.e. the "release" statement). Memory management statements concern pointer variables.

13.8.1 *new* statement

NewStmt:

new (Lvalue , Expression) ;

The **new** statement has the form of a function call, accepting two arguments. The first is a pointer variable, of an Lvalue type, on which the allocated memory will be assigned. The second is the number of objects to be allocated. Those objects will conform to the type «pointed by» the supplied pointer variable. Examples which clarify the way this works follow.

```
int* arr, i;
new arr(10);
for (i=0; i<10; i++)
      arr[i]=0;
struct Employee [ string name, address, role; ];
Employee* employees;
extern int totalEmployees;
new(employees, totalEmployees);
int** pp;
```

```
new(pp,3);
new(pp[0],10);  // ok.
new(pp[1],20);  // ok.
new(*(pp+2), 30); // Parse error. Use pp[2] instead.
```

13.8.2 *release* statement

The **release** statement will dispose memory previously allocated via a call to the new statement. In case that a bad pointer is supplied to a release statement a run-time error will be caused. In order to identify the cause as well as the source of the error in your program, compile your dialogue files with the *-dbg flag* (for automatic error detection).

ReleaseStmt:

release (Lvalue) ;

```
string* names;
new (names,10);
release(names);   // ok.
release(names);   // run time error.
```

13.9 Bridge / hook statements

Hooks and bridges are an advanced technique offerred by the I-GET language for mixing I-GET and C++ code at global / statement level, allowing even data interchange at the statement level. Those facilities are described in detail under Section 7 of this Volume.

13.10 Miscellaneous statements

Here we will discuss various statements which are provided to help manipulate some special classes of I-GET language constructs, like agent classes, and input / output events. Most of them have already been discussed within previous Sections, so we will not readdress their semantic / syntactic properties here, but we will link to the appropriate Section in this Volume were they are described in detail.

13.10.1 *create* agent statement

The agent creation statement is reduced from the agent creation expression which returns an agent identifier instance (of **daid** type), for the recently instantiated class. The agent **create** statement applies only to parameterised agents, i.e. agent classes which may be instantiated by call. Agent classes are discussed in detail under Section 11.

AgentCreateExpression:

create ID agent class **(ExpressionList** actual arguments **)**

AgentCreateStmt:

AgentCreateExpression ;

```
// An example of how to make reusable dialogue boxes
// via parameterised agent classes.
//
agent ConfirmBox (
      bool* ok, bool* no, bool* cancel, string* msg
      );
```

```
bool ok=false, no=false, cancel=false;
create ConfirmBox(
      &ok, &no, &cancel, «Really want to quit ?»
      );
```

13.10.2 *destroy* agent statement

The agent **destroy** statement is to the **create** statement, what the **release** statement is to the **new** statement. It requires a valid agent identifier instance (of type **daid**) to be passed. In case that an invalid agent identifier is provided, a run-time error will be generated (gain, use the *-dbg* compile option, to activate the I-GET run-time error detector). The syntax and examples follow.

DestroyAgentStmt:

destroy Expression *agent instance identifier* **;**

```
daid cBox=create ConfirmBox(...);
destroy cBox;
destroy cBox;  // 2nd destruction is a run-time error.
destroy nil;   // definitely a run-time error.
```

13.10.3 Artificial input event broadcasting statement

Artificial event broadcasting is the insertion of particular event instances within the system event queue, by the developer, in an intuitive programming way. This facility is discussed in detail under Section 12.5.

13.10.4 Output event statement

The output event statement broadcasts an output event to the toolkit server. The notion of output events is discussed under Section 12.1; output event classes model toolkit facilities in a form which looks similar to function calls. Output events address all types of toolkit functionality which is to be supplied to dialogue developers. The syntax and examples for calling output events follows.

OutputEventStmt:

out (ID lexical layer**).ID** output event class **(ExpressionList** parameter values**);**

```
int gid;
out(DeskTop).AllocateGraphics(&gid);
out(DeskTop).DrawText(
      gid,
      {drawObject},
      «Hello, output event»,
      10, 10, 100,100
];
```

13.10.5 Artificial method notification statement

Artificial method notification statements allow the developer to cause a notification for a particular method of a particular object. The capability for artificial method notification is introduced under Section 9.3.5. for lexical interaction objects, while it applies to virtual objects as well. Artificial method notification is applied at the object instance level.

Chapter 14

14. Built-in name spaces and rules for name collision

The definition of name spaces.

In the I-GET language, a name space concerns language constructs (e.g. variables, functions, data types), for which same identifiers are not allowed when declaring instances of any such construct (e.g. a specific variable, a specific function, a specific data type), within the same scope. In other words, within any given scope, only one construct instance from those in the same name space may appear with a particular name.

Eight (8) distinct scope spaces are allowed in the I-GET language.

In the I-GET language there are eight (8) distinct built-in name spaces, which allow constructs within the same scope to appear even with the same name, only if they belong to different scope spaces. In Figure 14-1, the name spaces of the I-GET language are illustrated. The particular scope categories applicable to each language construct, for each name space, are also indicated. It should be noted that apart from scope rules which when broken may cause compile errors, there are other scope rules which must be preserved, which, even though not causing compile errors, will cause linking errors. For instance, different agent classes should not be used with the same name across a whole interactive application (i.e. *project* scope). The same holds for virtual object classes. Lexical elements (i.e. lexical object classes, input events and output events) should preserve name uniqueness for their associated lexical layer name. One example showing the flexibility of the eight scope spaces in the I-GET language follows:

Apart from scope rules which can be checked at compile-time, there are global rules which if broken will lead to linking problems.

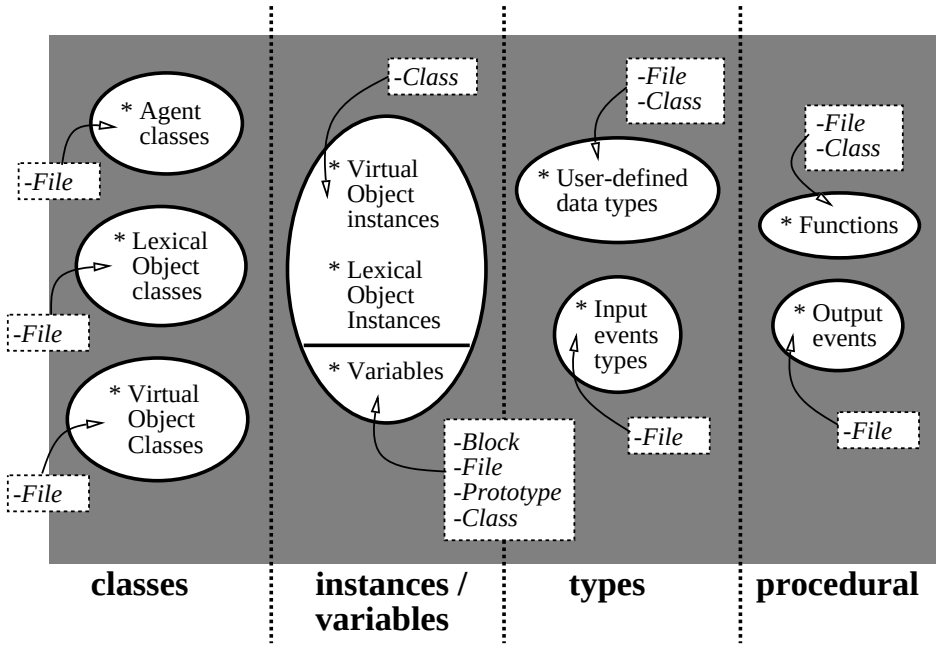


Figure 14–1: The eight name spaces supported in the I-GET language, with their respective construct categories, indicating the scope categories applicable for each construct (arrows).

Supporting construct recognition, versus syntax compactness: multiple explicit name spaces versus a global name space.

It is possible to define in file scope a function and a variable having the same name. Similarly, it is possible to define an agent class and a data-type in file scope with the same name. We have decided to make explicit name spaces in the I-GET language, by supporting a different syntax for manipulating constructs from different categories, in order to increase the readability of I-GET programs (i.e. construct recognition). This, however, has its disadvantages since it requires that developers learn more keywords and operators, than those typically required in programming languages supporting a single scope space, and inherently a simpler syntax.

An example follows which shows how name collisions are avoided in the I-GET language, even though defined constructs share identifiers.

```
int CommonId;

void CommonId () []

struct CommonId [ ];

lexical (Y);

virtual CommonId (Y) [ constructor[] destructor[] ]

lexical CommonId (Y) [ constructor[] destructor[] ]

agent CommonId;
```

Index