National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Object-Oriented Tools for Visualization of Multi-Sensor Data

by

Catalin Andronic

A THESIS

submitted to the School of Graduate Studies and Research

in partial fulfillment of the requirements

for the degree

Master of Applied Sciences

in

Electrical Engineering

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering

Faculty of Engineering

University of Ottawa

OTTAWA, ONTARIO K1N 6N5

ISBN 0-315-82588-X

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

To my parents ...

Acknowledgments

I would like to take this opportunity to thank my supervisor, Dr. Dan Ionescu, for his invaluable support and guidance throughout my research.

Thanks to my co-supervisor at the Canadian Center for Remote Sensing, Dr. David Goodenough, for providing me with expertise and direction.

A special thanks to my parents and my brother who, although far away, provided moral support during my studies.

Abstract

This research addresses the problem of generating source code for an object oriented image processing application by means of graphic manipulation.

The visual symbols (called icons) manipulated by the user on the screen are linked through data- and/or control-connections. The acyclic graphical structure (whose nodes are visual symbols and whose edges are data- and control-connections) build by the user on the screen is parsed and the corresponding source code is generated.

The programming environment developed in this research can be used to generate source code for any type of object-oriented application. An example has been implemented and demonstrated in the field of multi-sensor data visualization. The source code generated by means of graphic manipulation was cross-compiled, down-loaded into a massively parallel Single Instruction Multiple Data computer and executed.

Contents

List of Figures

Chapter 1

Introduction

1.1 Terms Of Reference

The aim of this research is to create a visual programming environment for the development of object oriented image processing applications. Although the programming environment implemented in this thesis is general, the particular purpose of designing such a tool is related to the development of various programs on a single instruction multiple data (SIMD) computer.

The visual programming environment designed and implemented in this research is an object oriented tool that supports simultaneous program composition/editing in both textual and graphical representations. The correspondence between the graphical view and the textual view is automatically maintained by the system.

The final result of user's actions is the creation of the source code for an image processing application by means of graphic manipulation. An example is presented and validated using an object oriented programming language, Objective-C.

The term visual programming language [20] is used to describe the languages supporting visual interaction and languages for programming with visual expressions.

Visual programming languages make a departure from the traditional programming

languages. This new field of research is stimulated by the following premises:

- People, in general, prefer pictures over words.
- Pictures are more powerful than words as a means of communication. They can convey more meaning in a more concise unit of expression.
- Pictures do not have the language barriers that natural languages have. They are understood by people regardless of what language they speak.

Studies [2] have shown that the human visual system and human visual information processing are clearly optimized for multi-dimensional data. Computer programs, however, are conventionally presented in a one-dimensional textual form, not utilizing the full power of the brain. Two-dimensional displays for programs, such as flowcharts or even the indenting of block structured programs have been known to be helpful aids in program understanding. Graphical programming uses information in a format that is closer to the user's mental representation of the problem, and allows data to be processed in a format closer to the way objects are manipulated in the real world [16].

Another motivation for using graphics is that it tends to describe the desired action at a higher level, often de-emphasizing issues of syntax and providing a higher level of abstraction, and may therefore make the programming task easier even for the professional programmers.

In the past, programming language syntax has been designed primarily for the benefit of compiler or interpreter design. Since these language translators view programs as stream of characters, a number of special symbols must be added in order to make the concrete syntax parsable. Unfortunately, these extra symbols have no semantic meaning and tend to interfere with the programming task. To overcome this problem programmers use coding tricks such as indentation and naming conventions. Unfortunately, these coding practices have no impact on computers' view of the program; a program may have one meaning based on this visual cues, yet the translator will find a different meaning based on the concrete syntax.

Graphical languages give a programmer, or even a user, the ability to perceive an apparent real physical universe and manipulate it naturally through the use of icons and symbols. The general notion of graphical or visual programming is that an arrangement of graphical symbols in two (or more) dimensions represent a program [21]. Since the pictures thus created must be translated into instructions to the computer, there would have to be a precise mapping of the meanings of the symbols into syntactically correct formal structure then into efficient computer code. On the other hand, it may be possible to endow a compiler for such a language with inferential capabilities so it can "reason" about the symbols, then map the result into machine code.

The primary goal of a visual programming environment is to present to the user a visual syntax that matches the user's conceptual model of the constructs in the language. Thus, the program should not be cluttered with symbols that are only used for machine understanding. The programmer should be unaware of the underlying language, just as he/she is unaware of the machine instructions that are ultimately executed. As Glinert points out [7] which language constructs are needed and what their visual forms should be is still an open question; the answer certainly depends on the model of computation presented by the language and the target application area.

1.2 Statement of the problem

This thesis presents the design and development of an object oriented tool that generates source code for applications aimed at visualizing multi-sensor data.

At the present time this implementation represents the first attempt of such a programming tool, the literature inspected does not record similar environments, and from this point of view it is original in its entirety.

The aerial images (multi-sensor data) acquired from satellites are processed on AIS-3500

a highly parallel, single-instruction, multiple-data (SIMD) computer designed and produced primarily for image processing application. Due to the internal memory limitations of the SIMD computer the image processing applications are developed on a SUN platform, cross-compiled and then downloaded into the AIS-3500. The executable code contains vision function calls that are recognized and executed by the operating system running on the SIMD machine.

An object-oriented application for image processing consists of interface objects and action objects that provide actions and define the interface. One of the action objects defines the application and it is called application object. The interface objects communicate with the action objects (the application object is an action object) using methods that read and write parameter values, trigger actions, etc. The action objects are the clients of all the interface objects.

The creation of an object implies the creation of the corresponding source code that describes its behavior. The source code can be created using a text editor; this implies a good knowledge of image processing techniques and a broad experience in working with object-oriented languages.

The visual programming environment designed and implemented in this research (called Automatic Code Generator) gives the user the possibility to build an image processing application by graphically manipulating visual symbols (called icons) on the screen.

The acyclic graphical structure (whose nodes are visual symbols and whose edges are data- and control-connections) build by the user on the screen is parsed by the ACG which then generates the corresponding source code in Objective-C.

The object-oriented approach increases code reusability and the visual programming environment makes the application development more efficient.

1.3 Research Contributions

Although the design and implementation of the Automatic Code Generator is completely original, the contributions brought by this research can be summarized as follows:

- ◇ An object oriented visual programming environment that can be used for the development of an object oriented application, using Objective-C as the underlying language, has been designed and implemented
- ◇ The object oriented visual programming environment can modify its own structure and customize it according to the application to be implemented
- ◇ A novel approach in developing image processing applications for parallel machines has been developed

An object oriented application for visualizing multi-sensor data has been developed using this tool.

1.3.1 Advantages and Disadvantages of ACG

Advantages

The iconic method proposed in this thesis is an alternative to the normal method of textual programming. The advantages of this method are:

- it relieves the designer of the problems concerning the intricacies of the underlying programming language
- it facilitates a rapid visualization of the control and data flow of the application
- it allows the enhancement of the standard object and vision-function libraries with new elements through the open architecture of the design
- it reduces the time required to modify an image processing application for a massively parallel computer
- it facilitates a rapid modification of the source code through graphic manipulation

- it supports code reuse through cut-and-paste operations

Disadvantages

Despite the advantages presented above, there are some disadvantages of using the ACG such as:

- the visual representation of an application takes more space than its textual form, making the displaying of large programs difficult
- the time required to implement some parts of an application may be longer when using graphic manipulation as opposed to textual composition

However, the advantages offered by the ACG outweigh the disadvantages increasing the efficiency of image processing application development.

1.3.2 Thesis Overview

In chapter 2 a broad overview of related research will be presented. Chapter 3 describes the design requirements for an object oriented tool which automatically generates a source code file from a connected graph of icons. Chapter 4 takes the generalized architecture developed according to the design requirements and produces a specific implementation. Chapter 5 presents an experiment with the ACG in the development of an object oriented application for multi-sensor data visualization and validates the implementation. Finally chapter 6 includes the conclusions drawn and the potential directions for continuing this research.

Chapter 2

Trends in Visual Programming

This chapter gives a broader context to this research and presents arguments in support of visual programming.

2.1 Introduction

The process of human-computer communication was for technical reasons constrained to be uni-dimensional. Commands to the operating systems, programs, were all linear text strings. In the past decade environments rather than languages became the focus of attention, where the term environment refers to a programming language within the context of an integrated set of software support tools and appropriate hardware, especially I/O devices [7]. The two new styles of human computer interaction which resulted may collectively be called nontextual.

In visual environments, graphical elements play a prominent role alongside text; in iconic environments, user interacts with the machine primarily by pointing at, manipulating, and juxtaposing small images.

According to Glinert [7] an environment will be termed visual, as opposed to textual, if at least one of the following conditions holds:

- High-level graphical entities are available as atoms which users can manipulate when programming.

- High-level graphical entities are available as atoms which users can manipulate at run time.
- High-level graphical entities form an integral and essential part of the display generated by the system during the programming phase or at the runtime.

A visual environment will be called iconic if the following conditions both hold:

- Graphical elements predominate when programming; use of text is minimal
- The programming process is essentially one of selecting (by pointing) and/or composing and/or placing in proper juxtaposition with one another on the screen) small images (which are commonly called icons).

An icon is usually a predefined flat pictorial symbol representing an "object" - physical or abstract (e.g., a fact, an action, an idea, a concept). In a computer environment, the object represented by the icon is usually limited to a command, an operator, a data type, a text file, an image or a collection of objects of the same type.

Icons, such as push button commands and menus of symbols representing simple actions on files, directories, etc. are currently used with the object-oriented programming environments and object-oriented interfaces to some personal computers derived from Smalltalk.

Icons, pictures, and symbolic graphics are sometimes considered as "words" or "blocks" which are placed in a two- or three-dimensional space following some predefined construction rules to create expressions and procedures in an iconic (or graphical) programming language.

The term "visual language" means different things to different researchers [2]. To some, it means that the objects handled by the language are visual. To others, it means that the language itself is visual. To the first group "visual language" means "language for processing visual information" or "visual information processing language". To the second group "visual language" means "language for programming with visual expressions", or "visual programming language".

In visual information processing language, the objects to be dealt with usually have an inherent visual representation. On the other hand the languages themselves may not have a visual representation. These languages are usually based upon traditional "linear" languages.

In visual programming languages, the objects to be dealt with usually do not have an inherent visual representation. They include traditional data types (such as arrays , stacks, queues) and application oriented data types (such as documents, databases).

Visual Programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two-dimensional since the compilers or interpreters process them as long, one dimensional streams. Visual Programming includes graphical programming languages and using conventional flow charts to create programs. It does not include systems that use conventional (linear) programming languages to define pictures, such as Macintosh Toolkit or X-11 Window Manager Toolkit [20].

Based on the principles of design, most of the visual programming languages reported in the open literature fall into three broad categories:

- On one extreme, graphics are deliberately designed to play the central role in programming. Lakin uses for these languages the expression "executable graphics" [17]. Some of the works belonging to this category include D. Smith's Pygmalion [32] and Glinert and Tanimoto's PICT [9].

- In the middle, the graphic representations are designed as an integral part of a language. However, unlike the icons in the first category, they are not the "super stars" of the language; and unlike the graphical extensions of the third category, the language cannot function without the graphic representations. Many of the table-based and form-based languages belong to this category: Luo and Yao's "Form operation by example - A language for office information processing" [19], Shu's FORMAL [31] and Yao's FORMANAGER [35].

- On another extreme, graphics are incorporated into the programming environment as an extension to the conventional programming languages. One of the works belonging to this category is Pong and Ng's PIGS [25].

A brief presentation of the most significant systems will provide a clearer illustration of the criteria presented above.

A highly graphical system, developed at the university of Washington, was reported by Glinert and Tanimoto [9]. Unlike Star which uses the office as its operational metaphor, PICT is designed to aid program implementation. With PICT, users sit in front of a color graphic display and communicate with the system through all phases of their work by pointing to icon menus in a menu tree. At execution time PICT uses simple forms of animation to make the drawings "come to life". As a programming language PICT is at "a language level similar to that of BASIC or simple PASCAL". User programs may be recursive and contain arbitrary chains of subroutine calls. Its capabilities, however, are very limited: it allows the user to compose programs that do simple, numeric calculations. Both the type and the number of variables are quite restricted.

Pong and Ng have described an experimental system for Programming with Interactive Graphical Support named PIGS [25]. Like PICT, the system has been designed with the aim of supporting program development and testing in the same graphical environment.

The approaches taken by the systems differ significantly. In PICT, icons are the essential language elements and play a central role. Programming process is essentially to select and/or compose icons, to place them in proper juxtaposition on the screen, and to connect the icons by paths to indicate the desired flow of control. PIGS on the other hand, is a graphically extension to a conventional programming language. Nassi-Shneiderman diagrams (NSD) are incorporated into PASCAL as the structured control constructs of the logic flow. PIGS can interpret a program in the NSD chart form, and the execution sequence of the NSD is displayed at a graphical terminal. PIGS also provide interactive debugging and testing aids to support program development.

Cuniff [3] reported that FPL (First Programming Language), which is a flowchart programming language (uses flowcharts to represent the structure of a program [6]) might be particularly well suited to helping novices learn programming because it eliminates syntactical errors.

Another way to present program constructs is using tiles that look like jigsaw pieces, and

will only fit together in ways that form legal programs. One version of this is Proc-BLOX [10]. Programming environments based on the BLOX methodology are termed worlds. BLOX tiles are at once real and imaginary, for unlike their counterparts in the real world, they can hide miniature or lower-level substructures which they encapsulate. BLOX tiles are also dynamic rather than static, in that their visible features (e.g., size, color, and edge contour or shape) may all change under appropriate circumstances, say, when elements are repositioned on the screen. Users compose BLOX programs by building structures which consist of one or more tiles, according to the usual jigsaw puzzle lock and key metaphor in which protrusions are plugged into correspondingly shaped indentations so that the two juxtaposed tiles interlock.

PROGRAPH [24] is an interpretative visual programming system that supports a functional data flow oriented language, expressed graphically in the form of pictographs. Pictographs are created by computer graphics and are directly executable. PROGRAPH attempts to overcome some of the problems of this type of language by using a graphical representation that is structured.

Pygmalion [32] was one of the seminal visual programming systems. It provides an iconic method for programming: concrete display images for data and programs, called icons, are manipulated to create programs. The emphasis is on "doing" pictorially, rather than "telling". With the aid of a mouse, the Pygmalion programmer demonstrates the steps of a computation using sample data values. Pygmalion then replays the exact sequence of operations, to perform the desired computation on other values.

A general approach for designing icon-oriented software systems on a LISP machine is described by Clarisse and Chang [1]. VICON is designed based upon the concept of generalized icons. This icon oriented approach is ideally suited for the design of high level structured graphic interfaces. Applications include image database design, computer aided design of VLSI circuits, robotics, etc. Moreover, icon-oriented software systems need not be restricted to the design of efficient user interface. Since such systems could include distributed databases, an iconic language may be transformed into high-level visual programming lan-

guages operating in a distributed processing environment.

HI-VISUAL [13] is an iconic programming language supporting visual interaction in programming. In this framework, icons represent real objects or the concepts already established in a target application environment, whereas icons representing functions are not provided. A function is represented by the combination of two different icons. Each icon can take an active or a passive role against the other. The role sharing is determined dynamically depending on the environment in which the icon was activated.

Peridot [20] is a tool for creating user interfaces by demonstration without programming. The user draws a picture of the desired interface and the system generalizes the picture to produce a parameterized procedure. The user gives example values for any parameters so the system can display a concrete instance of the user interface. Peridot allows a non-programmer to create menus, scroll bars, buttons, sliders, etc., and it can create most of the interaction techniques in the Apple Macintosh Toolbox.

C2 [15] is a visual programming environment for a subset of the C language. Both conventional textual code entry and editing, and program composition by means of an experimental hybrid textual/graphical method, are supported and coexist side by side on the screen at all times.

Hybrid textual/graphical program composition is facilitated by a BLOX-type environment in which graphical icons represent program structures and text in the icons represents user supplied parameters attached to those structures. The two representations are coupled, so that modifications entered using either one automatically generate the appropriate update in the other.

Textual files that contain C programs serve as input and output. Graphical representation serve merely as internally generated aids to the programmer, and are not stored between runs.

CUBE [22] is a three-dimensional, visual, statically typed, higher-order programming

language. According to the authors CUBE will eventually be embedded into a virtual-reality-based programming environment that allows a user to manipulate a CUBE program simply by grabbing, placing and moving its components.

The GRClass [26] combines Graphics, Relations and Classes to provide a visual interface for programming graph data structures within an object-oriented framework.

This is done by extending the object-oriented model with inter-object relations. These relations are then used to directly implement the conceptual model of the graph data structures.

Within the GRClass framework, data structures are objects that maintain relation tables. These relations and the objects participating in the relations constitute the form of the data structure. A graphical notation is used to specify the possible relations and to manipulate the relation graph. The foundation of the GRClass environment is its Class programming language.

Class is an object-oriented language based on C. As such, it is somewhat similar to C++. The GRClass provides a graphical interface for programming graph data structures. The output of the GRClass system is source code for the class compiler. This source code is used together with classes programmed using other tools to create a complete application.

VPL [18] is a visual programming language and environment for image processing. For processing the constructed programs VPL makes use of a C++ back-end image processing library called V-Sugar.

VPL is a fully live system which executes as several UNIX processes. Using a combination of synchronous and asynchronous requests in its communication protocol the system runs as a set of interacting, loosely coupled services. The VPL front-end provides a visual programming language interface in which data-flow diagrams are constructed. The nodes in these diagrams are components - functions and interactive devices - which, linked together, form a set of possibly disjoint, continually active program graphs.

As soon as a component is dragged to the workspace it becomes active, and the evaluation engine (running as a separate process) examines the newly modified graph looking for

parts that might now require evaluation. VPL takes the view that for visual programming to be successful, it must take into account user-interaction issues. In particular "liveness", tool integration, and support for exploratory programming must be provided.

The "liveness" notion is defined by Tanimoto [33] as a degree to which a program presents "live" feedback to the programmer. The first level is an "informative" level which is not used by the computer but used only by the user as an aid to documenting or understanding the program. The second level, "informative and significant" level, a visual representation is the actual specification to the computer of what computation is to be performed. A visual representation at the third level of liveness is one for which any edit operation by the user triggers computation by the system. At the fourth level the system is continually active.

The software development environment ObjectWorld [23] combines object-oriented programming and visual programming to enable software reuse.

A programmer builds new objects by directly manipulating prefabricated visual objects.

G [14] is a strongly typed, structured data flow programming language. The elementary data types are numeric (integer and floating point), string, and Boolean, and there are two aggregators - array and cluster (similar to a Pascal record or C struct).

A module in G is called a Virtual Instrument (VI). A VI consists of an icon, a panel, and a diagram. A panel contains controls and indicators that define the data types of the inputs and outputs of the VI; for example, a slide control represents a numeric input, and a plot represents an output that is an array of points, where a point is a cluster of two numbers. An icon contains terminals that are in one-to-one correspondence with a subset of the panel controls and indicators.

A diagram is a directed acyclic graph containing nodes, interconnecting signals, and source and sink terminals which correspond to the panel controls and indicators respectively. Nodes have zero or more input terminals (sinks) and zero or more output terminals (sources). A signal connects a single source terminal to one or more sink terminals. Nodes are built-in primitive functions, icons of other VI s representing calls to the VI (recursion is not supported), or structures.

Structures contain one or more sub-diagrams; sink terminals on the structure are source terminals in the sub-diagram and vice versa. Execution of a diagram or sub-diagram begins by placing tokens on all the isolated source terminals, which correspond to panel controls or sink terminals of a structure. The tokens propagate along the signals, duplicating themselves if the signal splits.

Cantata [36] is a graphically expressed, data flow-oriented language which provides a visual programming environment within the KHOROS system. The user builds a Cantata application program by connecting processing nodes to form a data flow graph. Nodes are selected from an application specific library. Control nodes and a parser extend the functionality of the underlying data flow methodology. Using graphical elements single pipelines can be constructed to perform fairly powerful information processing tasks.

2.2 Conclusions

In this chapter a number of visual programming systems have been presented. Although there is a great deal of excitement about visual programming and a number of working systems, there is still a lot of skepticism about the success of the field [2].

The visual programming systems presented in this chapter are aimed at programmers. Most of them do not generate source code for an application. The icons are used as visual representations of executable programs and the connections between icons represent the pipe-lineing of data through these executables [36].

There are a number of visual programming systems that implement the generation of source code, but the code generated is straight C [15] or concerns very narrow domains, as for example Graphic User Intefaces.

The system designed and implemented in this research proves that for carefully chosen domains the object oriented visual programming environments can increase the efficiency of software development and reuse.

Chapter 3

The Design Principles of the Automatic Code Generator

In this chapter the designing principles and the general architecture of an object oriented visual programming environment that generates source code for an image processing application is discussed.

Section 3.1 introduces the symbols used in the Visual Programming Environment, the rules of interconnecting the graphical symbols and the two steps implemented by the graphical parser. Section 3.2 discusses the programming environment used to develop the Automatic Code Generator (ACG) and shows why the object oriented design methodology was chosen. Section 3.3 describes the requirements for the ACG and the minimum set of graphic symbols (icons) necessary to create an application for the parallel processor. It also shows how the programming environment presented in section 3.2 is used to implement the graphic symbols. The features of the icons are listed and the visual representation of an icon is inferred from them.

3.1 Visual Programming Language Symbols

For creating an application using the Visual Programming Environment, six graphical symbols (icons) were found to be necessary. These icons correspond to the following entities:

- Classes (**C**)
- Functions (**F**)
- Control Structures (**S**) — (i.e. `while`, `if-then-else`)
- Message Expressions (**M**)
- Variables (**V**)
- Statements (**I**)

The bold letters represent the textual notation used for the icons.

The icons can be interconnected through three types of links:

- data connections (symbol: —)
- control connections (symbol: ===)
- client-server connections (symbol: __)

The starting point of a data connection is named: data output arrow. The ending point of a data connection is named: data input arrow. Similarly there are control input arrows and control output arrows, and also client-server input points and client server output points.

The Visual Programming Environment can be compared with a formal language leading to the notion of Visual Programming Languages. Any formal language is defined in terms of an "alphabet" and a "grammar". An alphabet is a finite set of symbols; the grammar determines the ways is which the symbols of the alphabet may be combined into sentences. The ACG is a hybrid programming environment in which the images play an important role alongside text. As opposed to a "standard" alphabet the icons manipulated by the ACG do not exist as entities before being created by the user. Because the user actions leads to the creation of the icons, the "alphabet " of the ACG Visual Programming Environment consists of the six graphical symbols presented above and also of a set of "actions". The alphabet contains also the following symbols:

- create an instance of a class (**Cc**)

- send a message to an instance of a class (**Cm**)
- create an instance of a Function(**Fc**)
- create an instance of a Control Structures (**Sc**)
- create an instance of a Message Expressions (**Mc**)
- create an instance of a Variables (**Vc**)
- create an instance of a Statements (**Ic**)
- select the root (**R**)

The "alphabet" of the Visual Programming Environment is the set: {**C, F, S, M, V, I, Cc, Cm, Fc, Sc, Mc, Vc, Ic, R,** —, ===, ___}.

The grammar is a scheme for generating sentences from elements of the alphabet, and a particular grammar is specified by describing the following four components:

1. an alphabet of terminal symbols that may appear in sentences of the language; for the ACG the set is {**C, F, S, M, V, I,** , —, ===, ___}.
2. an alpabet of nonterminal symbols that may appear in partially-derived sentences but may not appear in actual sentences of the language: {**Cc, Cm, Fc, Sc, Mc, Vc, Ic, R**}.
3. a start symbol, a specified member of the nonterminal alphabet {**R**}.
4. a finite set of productions, each of which consists of a left hand side string and a right side string

The "productions" for the Visual Programming Environment are presented below:

1. an icon has to be first created and then linked in the graphical structure; therefore, for example, the **Vc** always precedes the **V** symbol in the structure of a sentence
2. the instance of a class has to be first created and then sent an instance method; therefore the **Cc** symbol always precedes the **C** symbol

3. the icon of a Variable can be linked only through a data connection with the input (arrow) of a Function's or a Control's Structure icon

4. the icon of a Class can be linked only through a client-server connection with another icon of a Class

5. the icon of a Message Expression can be linked through two types of connections:

   - control connections with the control input arrow of a Function, Control Structure, Statement or another Message Expression
   - data connections with the data input arrow of a Function or of a Control Structure

6. the icon of a Function or a Control Structure can be linked through two types of connections:

   - control connections with the control input arrow of a Statement, Message Expression or another Function or Control Structure, respectively
   - data connections with the data input arrow of a Function or of a Control Structure

7. the icon of a Statement can be linked only through a control connection with the control input arrow of a Function, Control Structure, Message Expression or another Statement

8. the types of the starting and ending points of a data connection have to be identical or compatible

9. a Class has only input and output client-server points

10. a Function, a Message Expression or a Control Structure have both data and control input and output arrows

11. a Variable has only data input and output arrows

12. a Statement has only control input and output arrows

According to the above presented rules an intermediary sentence like:

R==[Mc Fc Cc Cm->]M === F

is valid as a partially derived sentence; the square brackets are used to suggest that the root symbol **R** is connected with the Message Expression **M** only after the actions in the square brackets take place. The arrow (->) is used to show that **Cm** modifies **M**.

A "textual" description of the sentence is: create a message expression, create the icon of a function, create the instance of a class, send a message to the instance object and modify in this way the Message Expression created by Mc then link the Message Expression with the function. The sentence is correct only if **F** has no input parameters because no variable was created and no data link connection is established. The final statement in the above presented example is

R==M==F.

A final statement containing any nonterminal symbol is incorrect. Also incorrect and tagged as error is the following final statement:

R==V==C.

It breaks two rules, because the Variable or Class icons cannot have control connections.

The correctness of the graphical structure build by the user in the workspace is verified by the ACG "parser" in two steps.

The first verification id done at the moment of icon interconnection. There are a number of rules that the ACG is verifying each time a data, control or client-server interconnection is realized. The rules are part of the productions presented above.

Production 8 deserves a more detailed presentation. Each data input or output arrow has a type associated with it. Examples of types are: integers, characters, frames, pointers, etc.

If the user attempts to create a link that does not obey the above presented rules the connection is refused. The last rule deserves a more detailed explanation. There are four possibilities in the case of a data interconnection.

- the types of the two icons being interconnected are the same, hence the connection is permitted

- the types of the two icons being interconnected are different, hence the connection is refused
- one of the icons has a type associated with it and the other doesn't; in this case the icon with no associated type takes the type of the other icon
- none of the icons has an associated type; in this case the connection is permitted and the compatibility verification is done in the second step.

In the second step the graphical parser analyzes the workspace along the data, control and client server interconnections.

The errors detected in this phase are:

- different data types associated with the entities linked by a data connection.
- missing inputs or outputs to or from the icons of Functions or Control Structures

The minor errors are corrected automatically by the parser, for example permitted data conversions (i.e. from `int` to `float`). Major errors require user's intervention. For example a missing variable or data type is required from the user through a dialog box.

After the verification phase is done the source code generation is initiated and the corresponding code is saved in a file.

3.2 The Programming Environment Used for the Development of the ACG

Objective-C, the programming environment used to develop the Automatic Code Generator, is a superset of C. This has the advantage that object oriented principles can be followed, but when efficiency is required the code can be written in C.

Traditional programming separated data from the operators which manipulated the data. Object oriented programming uses "encapsulation" to combine both the data and operators into a unit called an "object". To manipulate the data inside an object, a "message" (i.e. some instruction) is sent to the object. Message expressions are allowed wherever C

functions are allowed. Objects which respond in the same manner to the same messages are members of what is called a "class". When a class is compiled, the "factory object" for that class is created. The factory object is most often used to create new instances of a class. Object-oriented terminology often blurs the distinction between the source file of a class and its factory object by calling both entities "classes".

The classes defined in Objective-C have a complex structure; they may contain function definitions and global variables which can be accessed directly by methods in other classes (see Fig. 3.1).

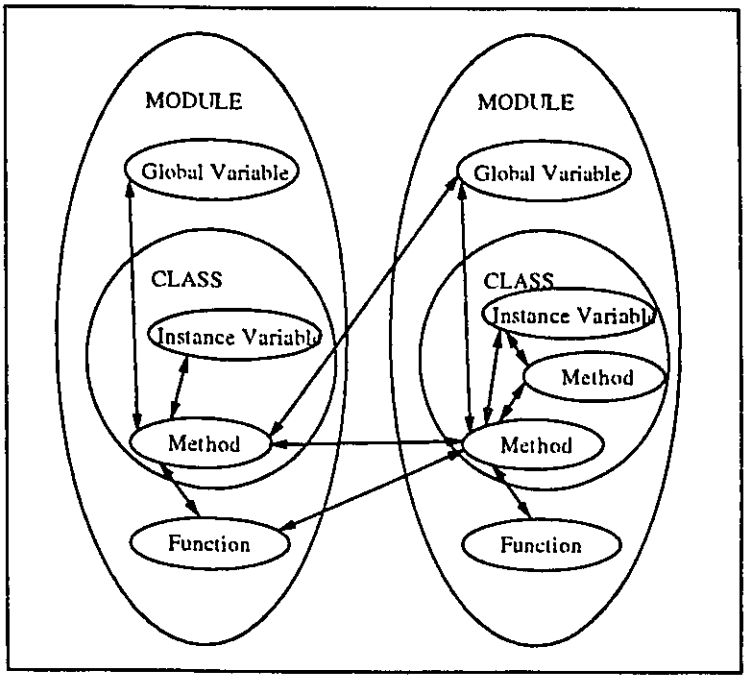

Fig. 3.1. An Objective-C library item

Methods use other methods, of either the same or other class, as part of their processing. Instance variables can be read or written by methods in the class. In addition global variables and functions can be accessed by methods.

Methods define the behavior of an object. To invoke one of these methods, a "message" is sent to an instance of a class. Each method has a unique name, called its "selector". The object that receives the message is called the "receiver". Methods are similar to conventional C functions in that they have formal arguments and return a value to their caller.

However there are several important differences:

- Method names need not be, and often are not, unique across different classes.

- Unlike functions, methods are not called directly by name but indirectly by the object's selection mechanism.

Another basic concept in object-oriented programming is "inheritance".

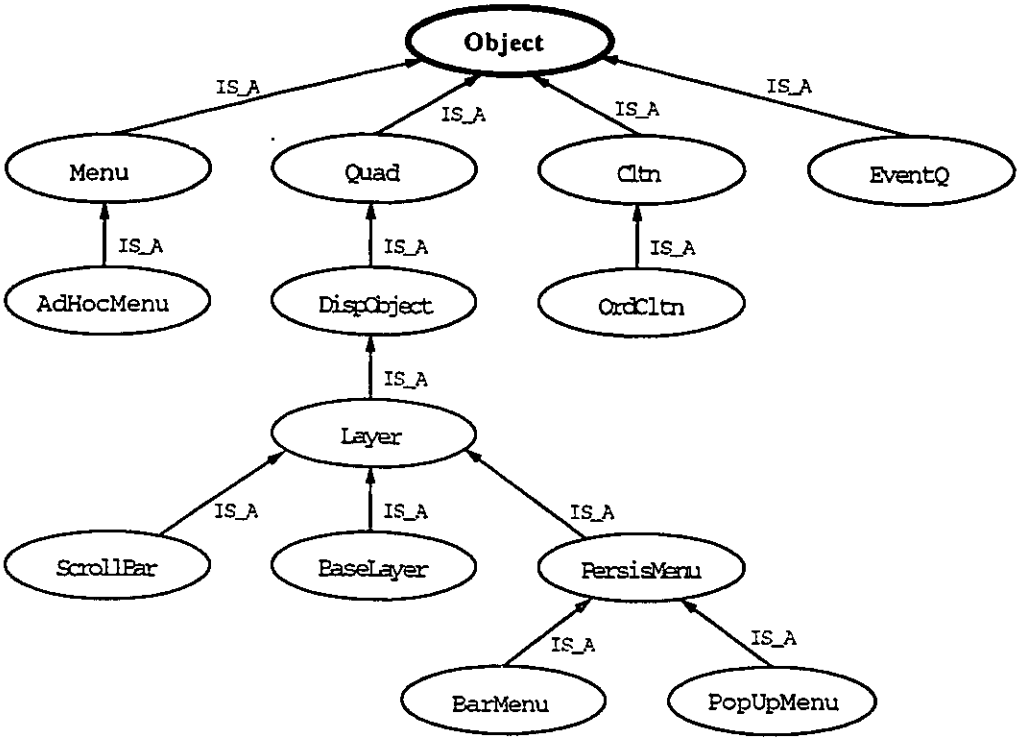

Fig. 3.2. The inheritance hierarchy

The descendants of a class inherit the instance variables and methods of their ancestor classes. In this way, message behavior of each class does not need to be explicitly defined in a class description file. The only thing to be specified is how the class being defined differs from its ancestor class. The rest is inherited automatically.

All classes are arranged into a tree called the IS_A or inheritance hierarchy, with generic, non-specialized classes such as Object at the root and very specialized classes at the leaves. Figure 3.2 shows a part of the library used to implement the ACG. There are two ways to alter the inheritance of a new class from its superclass. Instance variables and methods can simply be added, or an inherited method can be "overridden". The ACG itself is an object oriented application whose implementation is based on a number of classes, called ACG classes; a part of these classes are described below.

3.2.1 Display Objects and Display Mediums

Display objects are objects which display on display mediums. Examples of display objects are text, lines, colors and patterns. Examples of display mediums are screens, bitmaps and layers.

3.2.2 Layers

Layers are display mediums on which to display objects. Each instance of Layer has a backLayer, that is the parent layer, and a group of frontLayers, which are its child layers in the layer hierarchy. Layers will always appear on top, or in front of their parent layer and below or underneath their child layers. A layer may be attached to another layer. The root of the layer hierarchy should be an instance of BaseLayer or one of BaseLayer's subclasses. BaseLayer is the class which manages the event processing loop and the interface to the screen.

BorderLayer is a Layer with a border and an opaque inside color. A BorderLayer differs from a Layer in that it is not transparent. Layers also provide the capability to handle events or "user input actions".

3.2.3 Menus

A menu consists of two parts:

- the logical menu, which describes the items in the menu and the actions for each item
- the physical menu, which formats the logical menu into a visible and usable menu

The Menu class is used to build logical menus. Physical menus may be activated causing them to become visible and accessible. Each instance of Menu has three attributes: an image, an action, and a collection of children. The image specifies how this item appears in a physical menu, for example as a text string or as an icon. The action of a menu item specifies what happens when this item is chosen by a user. An action is specified by a selector-receiver pair; the selector is sent to the receiver when the item is chosen. When an

instance of Menu is formatted into a physical menu, the Menu's image becomes the title of the physical menu and its children become the items in the physical menu.

The AdHocMenu class is used to build logical menus. An AdHocMenu will calculate its children each time the children are requested (a Menu has fixed children). AdHocMenu provides the same capabilities as Menu, with the added functionality of creating its sub-menu immediately after the user asks for the sub-menu.

The PopUpMenu class is a physical menu. It appears in response to a user action, typically the pressing of a mouse button.

The BarMenu class is a physical menu formatted horizontally with no title.

3.3 The Design of ACG's Graphical Symbols

The minimum set of elements in the AIS software environment (called AIS entities) necessary to implement an image processing application for the AISI-3500 parallel machine are:

- classes (see Appendix A)
- functions (see Appendix B)
- control structures such as `if-then-else, for, while`
- message expressions
- identifiers
- statements

Each element from the above presented set is implemented by an ACG class. The icons are visual representations of the ACG-classes, hence they represent elements in the AIS set.

These AIS entities are sufficient to build any image processing application. By creating visual representations for these language entities and by manipulating them in the work-space any image processing application can be implemented by means of graphic manipulation.

The structure of an icon corresponding to an AIS entity reflects the actions that can be performed with or upon it.

The ACG assists the user in the process of building an object-oriented application for multi-sensor data visualization.

3.3.1 Graphic Symbols Used for Functions

An application for multi-sensor data visualization uses a large number of image processing functions, which all run on the parallel processor and enable the user to manipulate image data.

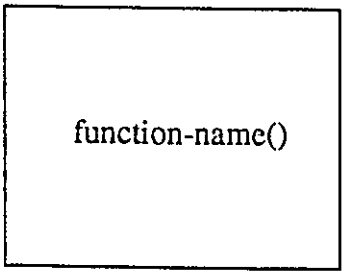

Fig. 3.3. The Icon of a Function—General Characteristics

The AIS parallel processor system libraries contain image processing functions and methods (implemented in classes) facilitating the creation of very structured image processing applications which eventually will independently run on the parallel machine. Mathematical computations, matrix calculus or neural networks can also run on the AIS parallel computers.

The icon of a function is represented in the ACG as a rectangular box containing the name of the function inside it.

3.3.1.1 General Properties of a Function's Icon

The ACG-class which implements a function of the AIS environment has a number of features that increase the user's efficiency in developing image processing applications. The user should be able to:

- move the icon of a function anywhere inside the work-space
- visualize the names and types of the input and output parameters
- connect the function's icon with other icons in a meaningful way

The implementation of the first two features doesn't require any modifications to the function's icon suggested in Figure 3.3. The third feature cannot be implemented without modifying the function's icon.

3.3.1.2 Data/Control Connections

There are two types of connections that are significant for a function:

- data connections (graphically represented as a single line)
- control connections (graphically represented as a double line)

Data connections are used in the ACG visual programming environment to provide information about the input/output parameters of the function.

The common features of the data arrow icons are:

- they are permanently attached to the function's icon
- they cannot be destroyed unless the function's icon is destroyed
- they cannot be moved inside the function's icon; they have a fixed position

3.3.1.3 Input Data Connections

There are a number of data input parameters that have to be graphically provided for the great majority of functions. The data connections are used to link the input parameters' icons with the function's icon. In Figure 3.3 there is no way of distinguishing among the input data connections; moreover the order in which the data connections are created shouldn't be important.

In order to implement these requirements a number of data input arrows equal with the number of input parameters of the function are attached to the left side of the function's icon. The icon of a function with data input arrows is represented below.

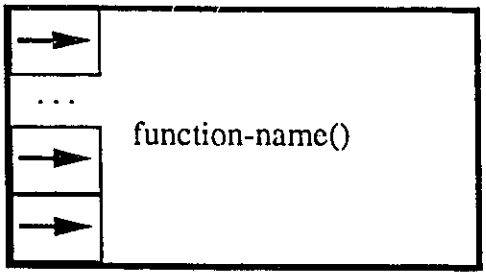

Fig. 3.4. The Icon of a Function—Data Input Connections

The icon of a data input arrow is the visual representation of an ACG-class with its own features. Other features of a data input arrow icon are:

- it represents the end point of a data-connection
- it is transparent to the events generated by the mouse buttons.

3.3.1.4 Output Data Connections

A function either returns a value or modifies one or more of its input parameters. The value returned by the function may represent the input parameter for another function. The data connection in this case has as the starting point the function's icon and as the ending point the data input arrow of another function.

There are some functions that are modifying more than one input parameter, thus generating multiple output values.

To implement the starting point of a data connection a data output arrow is attached to the right side of the function's icon. The icon of a data output arrow is the visual representation of another ACG-class; other features are:

- it represents the starting point of a data-connection
- it is not transparent to the events generated by the left and right mouse buttons; the user creates a data connection pressing the left mouse button with the cursor inside the icon of an output arrow and then releasing the button when the cursor is inside the corresponding input arrow. A data connection is destroyed by doing the same action with the right button of the mouse.

The functions in the AIS system library can be divided into two broad categories. The first group comprises the classes that return only one value and do not modify any input parameter. This type of functions have only one data output arrow attached at the middle of the right side of the function's icon. The icon of such a function is represented below.

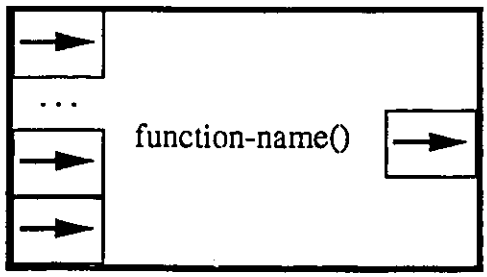

Fig. 3.5. The Icon of a Function— Single Data Output

The other group of functions comprises the functions that modify one or more input parameters. The icon of such a function has more data output arrows each of them corresponding to the input parameter that the function modifies.

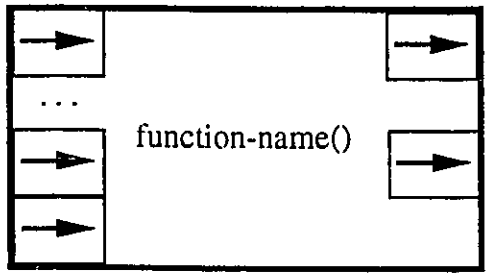

Fig. 3.6. The Icon of a Function —Multiple Data Outputs

3.3.1.5 Control Connections

A function's icon is linked in the acyclic graph representing the structure of a method through control connections. The graphical structure built by the user on the screen by creating and interconnecting various icons is analyzed by the ACG beginning with a "first" icon called the "root" of the graph. The order in which the icons are analyzed is given by their relative position in the principal control-flow chain.

There is only one principal control-flow chain (the ACG doesn't implement parallel algorithms) and none or more secondary control-flow chains. A secondary control-flow chain is created by a control structure such as `if-then-else` (as presented in section 3.2.2). The first icon in the principal control-flow chain is the root icon.

The common features of the control arrow icons are:

- they are permanently attached to the function's icon
- they cannot be destroyed unless the function's icon is destroyed
- they cannot be moved inside the function's icon; they have a fixed position

3.3.1.6 Input Control Connections

To connect the icon of a function in a control-flow chain the user creates a control connection which has as the starting point an icon that is already part of the control-flow chain, and as the ending point the current icon. The end point of the control flow connection is called a "from-arrow" and is implemented by an ACG-class. The icon of a from-arrow is a rectangular box with an arrow pointing downwards inside it, and is attached to the middle of the top side of the function's icon.

Other features of a from-arrow icon are:

- it represents the ending point of only one control-connection
- it is transparent to the events generated by the mouse buttons

3.3.1.7 Output Control Connections

The icon of a function can also be the starting point of a control connection. This feature is implemented by a "to-arrow", the visual representation of an ACG-class, attached to the middle of the bottom side of the function's icon.

Other features of a to-arrow icon are:

- it represents the only starting point of a control-connection
- it is not transparent to the events generated by the left and right mouse buttons;

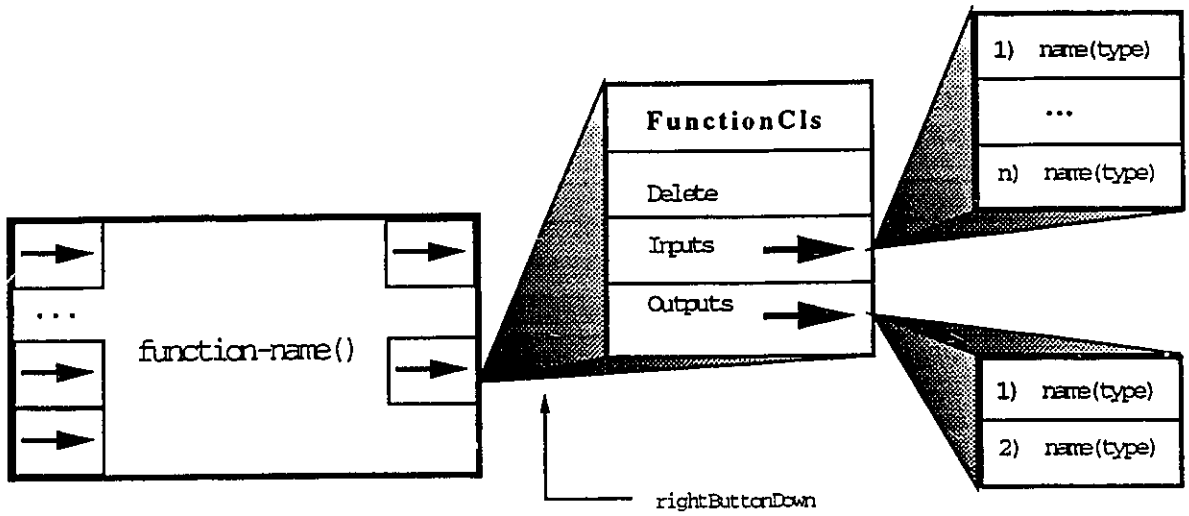

Fig. 3.7. The Icon of a Function

The user creates a control-connection pressing the left mouse button with the cursor inside the icon of a to-arrow and then releasing the button when the cursor is inside the corresponding from-arrow. A control connection can be destroyed by repeating the action with the right button of the mouse or by creating a control-connection with a different starting point.

A control-flow chain represents a finite number of icons linked through control connections. There is a first icon, called the root, which has only a control connection (beginning in its to-arrow), a number of intermediary icons which have two control-connections, one to the previous icon and one to the next icon, and a last icon which has no control connection starting from its to-arrow.

A secondary control-flow chain has as the starting point the icon of a conditional structure (such as, if-then-else, for, or while) and represents the alternative path(s) taken by the execution of a program.

The icon of a function is not transparent to the events generated by the pressing of the mouse buttons.

The middle button of the mouse is used to move the icon anywhere inside the work space.

The right button of the mouse is used to display information about the name and type of the input and output parameters of the function (see Fig. 3.7) When pressed by the user, the right button triggers the generation of a PopUpMenu with three options.

- the option "Delete" is used to remove the icon from the work space
- the option "Inputs" is used to display the names and types of the input parameters
- the option "Outputs" is used to display the names and types of the output parameters

The selection of an input item allows the user to assign textually a value to an input parameter and represents an alternative to a data connection.

3.3.2 Graphic Symbols used for Control-Flow Functions

The group of control-flow structures comprises: "If()", "For()" and "While()" all of them implemented by an ACG-class.

These functions are used to create secondary control-flow chains. A secondary control-flow chain is an alternative execution path. The "If()" function is used to create one or two secondary control-flow chains.

The icon of the if-then-else structure is a rectangular box with the text "If()" inside it; it has the from- and the to-arrows described in the previous chapter. The only data connection is represented by the condition to be tested by the "If()" function. Because there is only one data connection the icon of the "If()" function has only one data input arrow attached to the middle of the left side of the function's icon. Usually the starting point of this data connection is the data output arrow of a logic function or the icon of a variable.

The "If()" function has no data output arrow, but has two control output arrows. If the condition to be tested by the "If()" function is true, then the code associated with the icons connected to the first control output arrow is executed; if the value is false then the code associated with the icons connected to the second control output arrow is executed. The starting point of a secondary control-flow chain is a control output arrow — a to-arrow.

The icon of the "If()" control flow function is represented below.

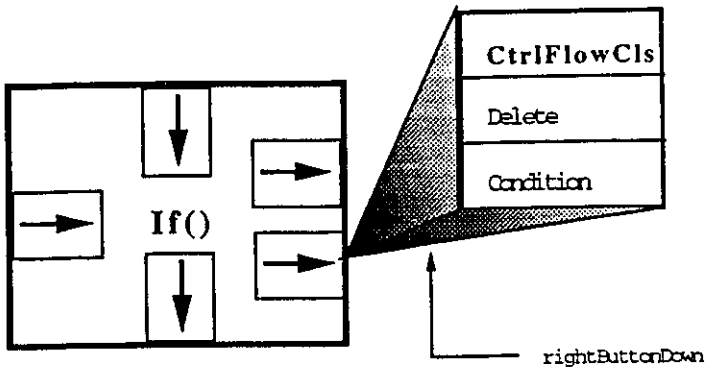

Fig. 3.8. The Icon of the if-then-else Control Structure

The icon of a control flow structure/function is created by attaching to a rectangular box containing the name of the function inside it a number of "arrows". The icon of any control flow structure has one data input arrow, one from-arrow (control input arrow), one to-arrow (control output arrow) attached to the bottom side of the icon box and one ("While()" and "For()") or two ("If()") to-arrows attached to the right side of the icon box.

The condition to be tested by a control flow structure can be sometimes very complex and the user might prefer to create it in a "traditional" manner using a text editing facility. Pressing the right button of the mouse inside an icon of a control flow structure generates a PopUpMenu with two options: "Condition" and "Delete"(see Fig. 3.8).

The option "Delete" is used to remove the icon from the work space. The selection of the option "Condition" allows the user to create textually the condition to be tested by the function.

3.3.3 Graphic Symbols Used for Message Expressions

A message expression is represented by the name of a method send to the name of a class, for example: ``[AisObject new]''.

The graphical representation of a message expression is a rectangular box containing the text of the message expression inside it.

A message expression is connected in the graphical structure of a method through data and/or control connections. Two control arrows are attached to the icon of a message expression: a from-arrow and a to-arrow, in order to make possible the connection of the icon in a control-flow chain.

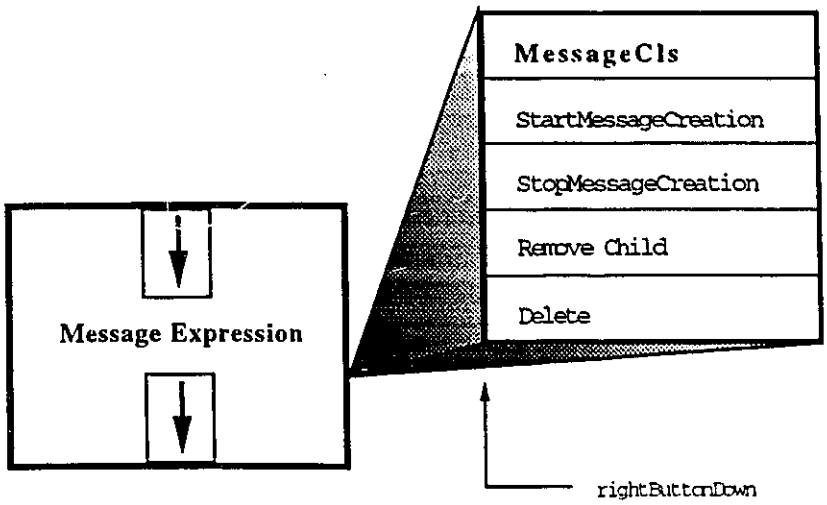

Fig. 3.9. The Icon of a Message Expression

The icon of a message expression can be the starting point of a data connection which is created by pressing the left mouse button inside the icon of a message expression. To simplify the graphical representation the starting point of the data connection is the icon itself and not an output arrow.

The right button of the mouse is used to generate a PopUpMenu (see Fig. 3.9) with four options: "StartMessageExpression", "StopMessageExpression", "RemoveClient" and "Delete".

The selection of the first two menu options has no visible effect; these options are used to create the text of a message expression and require the use of an object's icon.

The option "RemoveClient" is used to cancel a client/server connection and is presented in Chapter 4.

3.3.4 Graphic Symbols Used for AIS Objects

The graphical representation of an AIS object is a rectangular box containing the name of the object inside it.

The methods implemented by an AIS object are accessed through a PopUpMenu generated by pressing the right button of the mouse. The PopUpMenu has three options: "Delete", "RemoveClient", and "Methods". The third option allows the user to select one of the methods implemented by the current object or by one of its superclasses and to create a message expression.

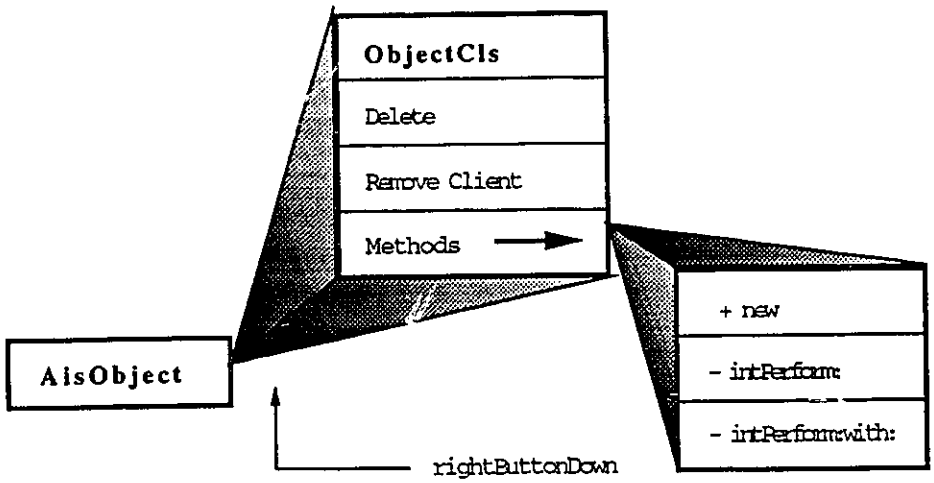

Fig. 3.10. The Icon of an AIS Object

The instance of an object is connected in the graphical structure of a method through a client/server connection (represented as a triple line).

Some methods take a number of formal parameters. These parameters are given textually by the user, because usually they are numbers or method names. Sometimes formal parameters are represented on the screen as icons and the user may create graphically the connection between a method and its parameter. This connection is called a client/server connection.

A client/server connection is created by pressing the left mouse button inside the instance object's icon; the starting point is the object's icon itself. A client/server connection can be canceled by selecting the option "RemoveClient" from the PopUpMenu generated with the right button of the mouse.

3.3.5 Graphic Symbols Used for Identifiers

The graphical representation of an identifier is a rectangular box with the text "Variable" inside it.

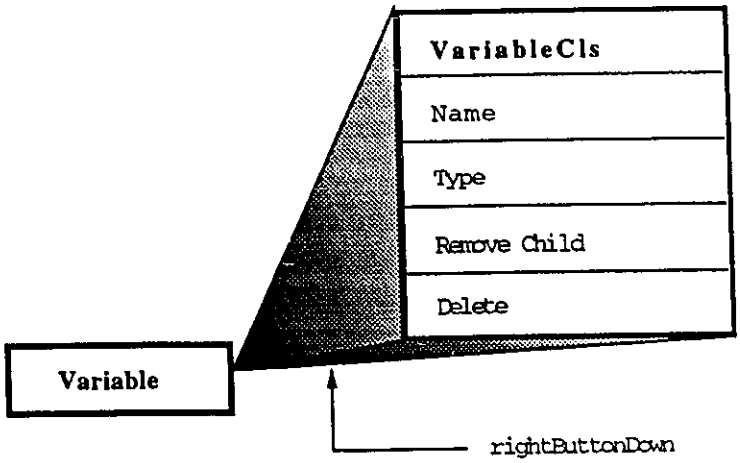

Fig. 3.11. The icon of an Identifier

The icon of an identifier is connected in the graphical structure of a method only through data connections. A data-connection is created by pressing the left mouse button inside the icon of the identifier.

A PopUpMenu with four options is generated by pressing the right mouse button inside the icon. The options are: "Name" (to change the name of the identifier), "Type" (to assign a type to the identifier), "RemoveChild" (to cancel a data connection) and "Delete" (to remove the icon from the work space).

3.3.6 Graphic Symbols Used for Statements

The graphical representation of a statement is a rectangular box containing the text of the statement inside it. Initially the text inside the icon is "Instruction".

The visual representation of a statement is connected in the graphical structure of a method only through control-connections using the two control arrows.

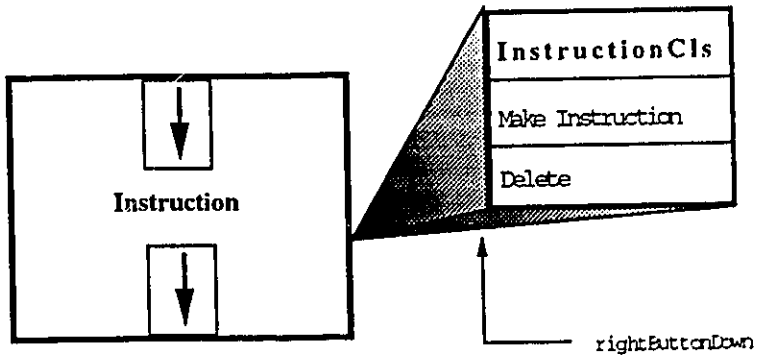

Fig. 3.12. The Icon of a Statement

The PopUpMenu generated with the right mouse button has two options: "MakeInstruction" and "Delete". The first menu option allows the user to create textually a statement that will be inserted in the graphical representation of a method.

3.3.7 Other Interface Items for Controlling the Operation Flow in ACG

In order to create an object oriented application in the ACG programming environment the user manipulates graphically the icons corresponding to the AIS entities. The icons are interconnected through data, control and client/server links to create the graphical representation of a method or a function.

The generation of the corresponding source code requires from the user a number of "commands", which are implemented as options in the lower menu-bar, present at the top of the work-space.

There are two categories of commands:

- commands used to provide information and actions directly related to the graphical structure on the screen; these commands are grouped under the option "Icon Manipulation"
- commands used to provide information and actions related to the files and source code to be generated; these commands are grouped under the option "File Options"

There are also a number of additional features not explicitly implemented by the ACG-classes but rather inherited from their super classes.

3.3.7.1 Icon Manipulation Commands

The options in the "Icon Manipulation" sub-menu are presented in Fig. 3.13.

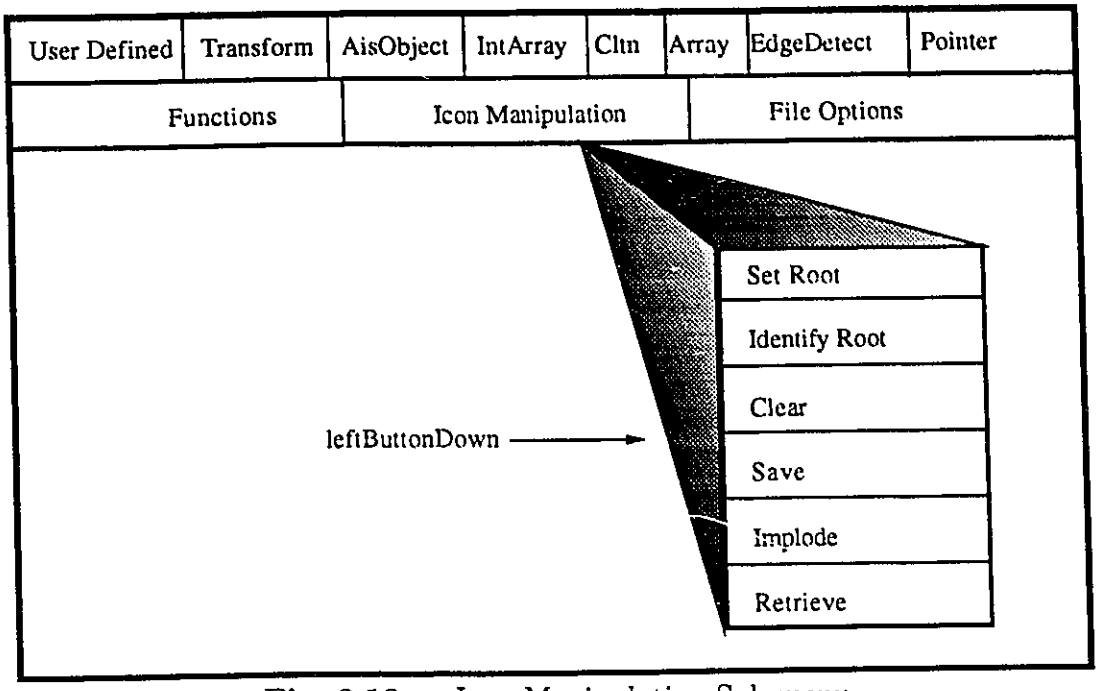

Fig. 3.13. Icon Manipulation Sub-menu

- "Set Root" is used to select the root icon, which is the starting point for the parsing process
- "Identify Root" flashes the root, if it exists
- "Clear" allows the user to delete the icons present in the work-space
- "Save" allows the user to save the graphical structure created by means of graphic manipulation in a file (called graphic-file)
- "Implode" allows the user to create large methods by collapsing a graphical structure in a rectangular box and by clearing the work-space
- "Retrieve" allows the user to bring in the work-space from a graphic-file the graphical structure created with the ACG in a previous session

3.3.7.2 File Options Commands

The options in the "File Options" sub-menu are presented in Fig. 3.14.

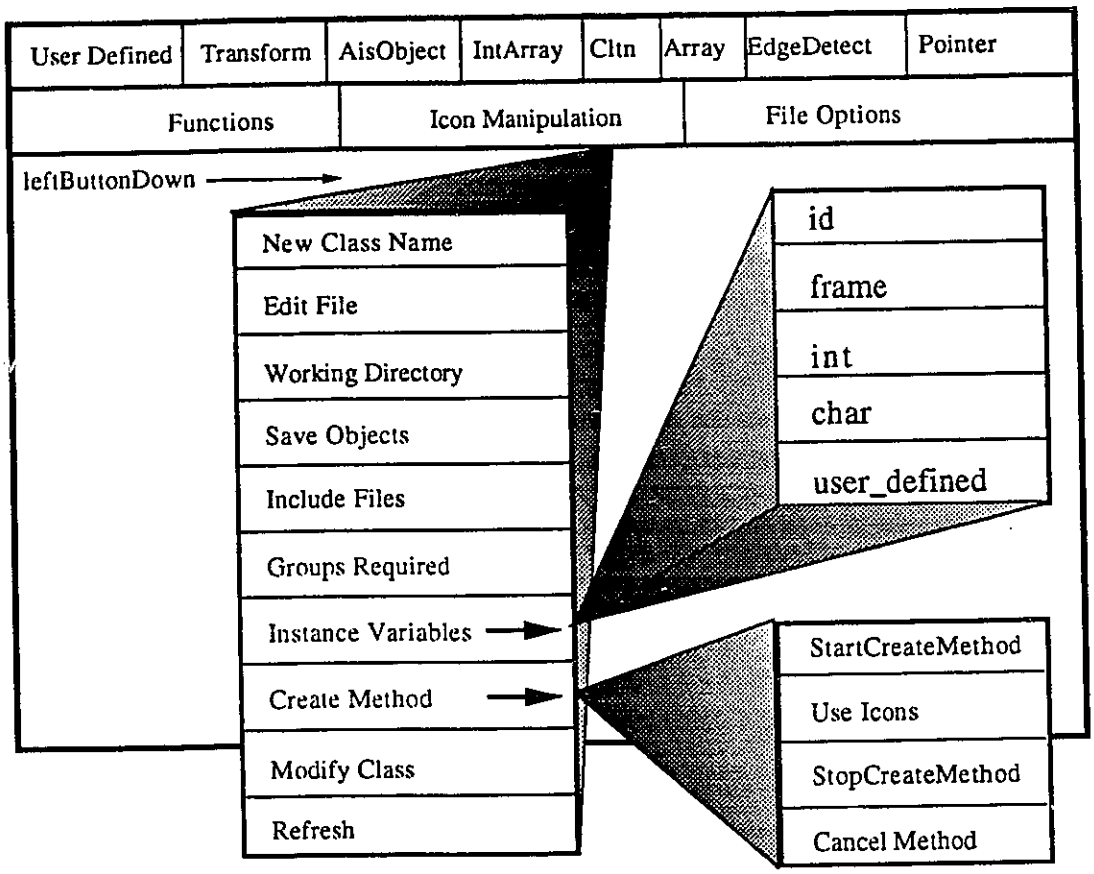

Fig. 3.14. File Options Sub-menu

- "New Class Name" is used to introduce the name of the class to be implemented in the visual programming environment

- "Edit File" allows the user to modify a file using a text editor instead of graphical manipulation

- "Working Directory" allows the user to change the name of the working directory. At the beginning of its execution, the ACG extracts the value of the current directory and assigns it to a global variable. The files containing the source code for the object oriented application created with the ACG are stored in the working directory

- "Save Objects" allows the user to save the information about the classes created with the ACG for future reuse

- "Include Files" allows the user to include in the file containing the source code the names of the header files required by a specific application

- "Requires" allows the user to include in the file containing the source code the names of the message groups (the libraries needed for class implementation) required by a specific class

- "Instance Variables" is a menu option that implements five alternatives which allow the user to create an instance variable of that type. The five data types are:

  1. id
  2. int
  3. frame
  4. char
  5. user_defined; this option allows the user to create instance variables with other data types (i.e. char*, frame*, char[], etc.)

- "Create Method" implements a sub-menu with four options that allow the user to generate the source code corresponding to a graphical structure:

  1. "StartCreateMethod" is used to initiate the creation of a method in a class; it allows the user to enter the method's name
  2. "Use Icons" is used to initiate the parsing process; the graphical structure created on the screen is analyzed starting from the root icon and the source code corresponding to it is inserted in a temporary file called **newMethod**
  3. "StopMethodCreation" is used to end the creation of a new method and triggers the insertion of the source code corresponding to the method in the file text corresponding to its class
  4. "Cancel Method" is used to abort the method creation

- "Refresh" is used to insure the consistency between the data stored in the instance variables of **objectDataBase** and the information in the ObjectBaseFile (see 4.2.1). The information in the ObjectBaseFile is changed whenever a method is deleted from the source file because it is simultaneously deleted from the ObjectBaseFile. After

initialization the instance variable of **objectDataBase** reflect the information in the ObjectBaseFile but the consistency is not preserved automatically when the structure of the ObjectBaseFile changes. The option "Refresh" has to be selected by the user in order to insure this consistency.

3.3.7.3 Inherited Features

The options in the PopUpMenu generated by pressing the right button of the mouse inside the work space are presented in Fig. 3.15.

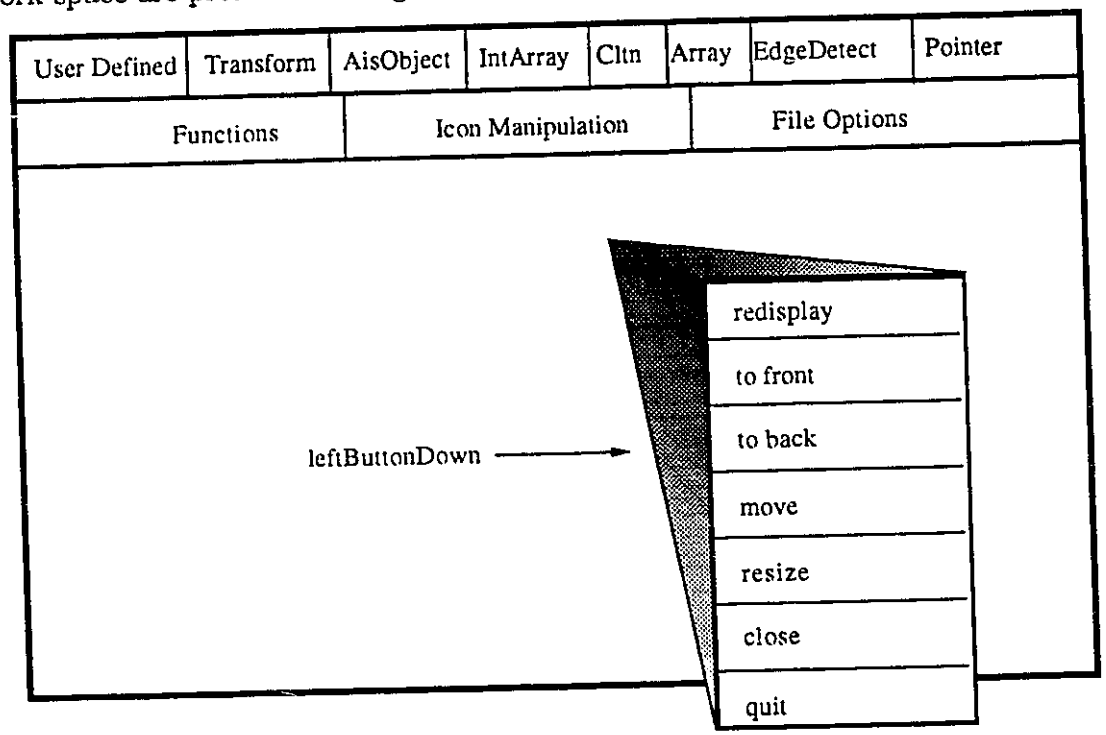

Fig. 3.15. Inherited Features

- "redisplay" is used to redisplay everything in the work space
- "to front" places the ACG window on top of all the sibling layers
- "to back" places the ACG window under all the sibling layers
- "move" interactively moves the application layer
- "close" closes the ACG into an icon
- "quit" quits the ACG

The above options are used in the process of implementing the object oriented application, allowing the user to interact with a GUI instead of a command interpreter.

3.4 Conclusions

In this chapter the design requirements for a software tool which automatically generates a source code file from a connected graph of icons, was given.

The detailed structure of a visual entity (icon) manipulated in the work-space was inferred from the functions which the icon has to perform.

The graphical representation was created to be suggestive for the user. Text was also used when the entity represented by an icon was too abstract to be visualized in a self-explanatory way.

The icons implement also features not represented visually (i.e. the methods to which a class responds). These features were implemented as options in the PopUpMenus generated by pressing the mouse buttons.

The icons are connected through different types of links. The creation of a link is another important feature implemented by an icon and is a part of its architectural design.

The whole design of an icon was performed according to the functions that particular icon has to implement. A part of the features of the image processing application's building blocks were given visually (i.e. the name of a class or function, the number of input or output parameters); other features were implemented as actions triggered by pressing the mouse buttons.

Chapter 4

The Automatic Code Generator, Structure and Implementation

4.1 Introduction

The real power of an object-oriented approach to problem solving is that analysis methods map directly into design methods, which map directly into implementation methods.

The fundamental component of an object-oriented software is the object. Thus a key step in the design of such a software is to define the objects. Objects are characterized by their attributes and the messages to which they respond, which leads to another key step - defining the attributes and the messages for each kind of object. The objects in a software system have relationships among themselves that are represented by their dependencies; therefore, a third key step in designing a software system is to identify a structure that accurately depicts these relationships. The implementation of these three steps represents, in fact, the definition and the development of hierarchies of classes.

4.2 The General Architecture of the ACG

The general architecture of the ACG is presented in Fig 4.1.

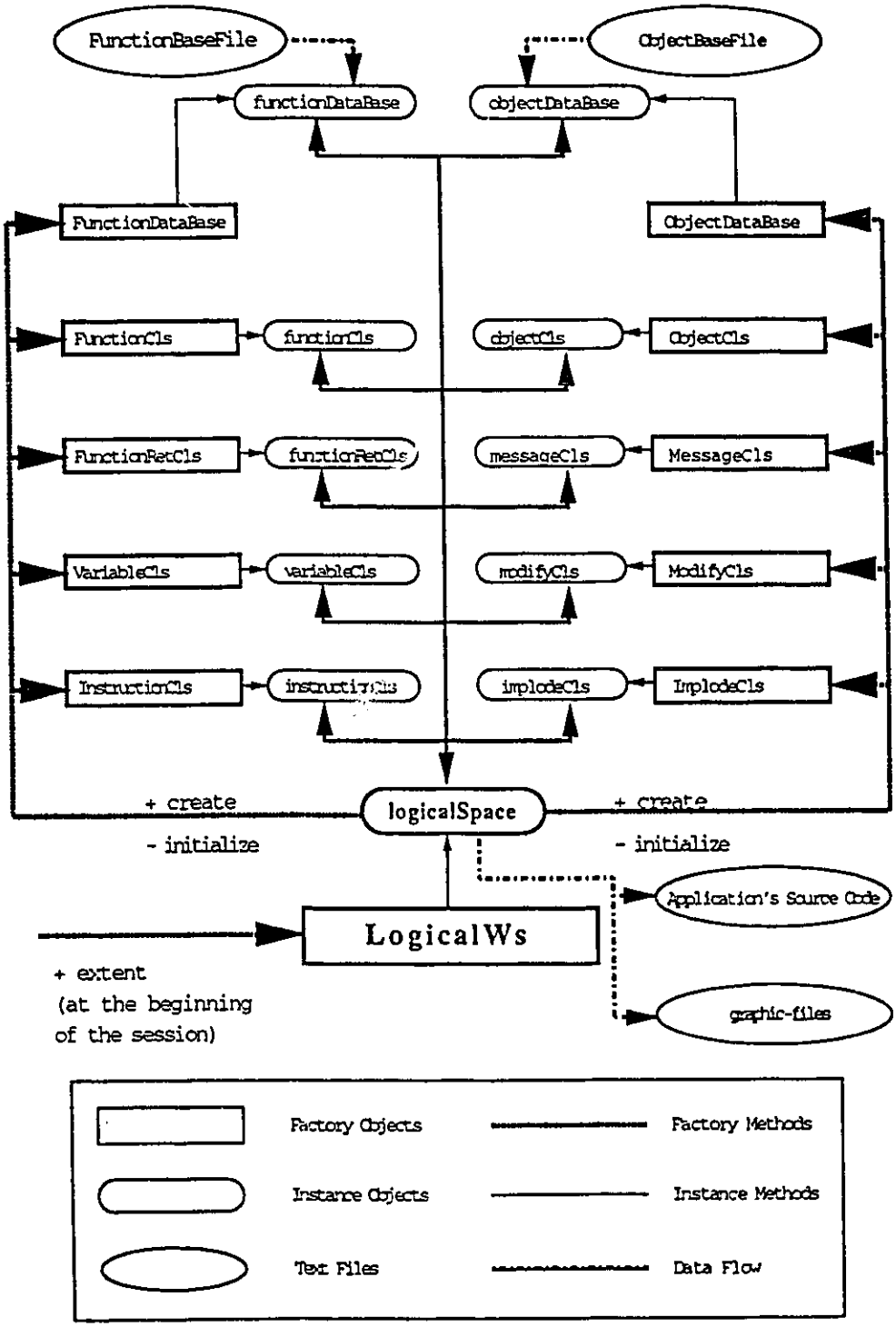

Fig. 4.1. The General Architecture of the ACG

The ACG visual programming environment is itself an object oriented application. The

building blocks are the objects in Fig. 4.1. A graphic user interface allows the programmer to activate graphical symbols through a menu selection. The graphical symbols are grouped in classes, methods, message expressions, functions, identifiers and statements. A data base of library items allows the user to use and add, if needed, other items in the library as soon as they are created.

4.2.1 The LogicalWS Class

LogicalWS is the main object of the ACG. It creates both the physical workspace and the logical environment in which all the actions take place.

The tasks carried out by **logicalSpace**, the only instance of the LogicalWS class, are:

- Creates and initializes **objectDataBase** - the only instance of the ObjectDataBase class. **objectDataBase** keeps in it's instance variables the information about the AIS objects (names of classes, names of superclasses, names of methods, etc.). The ACG-class ObjectCls is used to implement the objects in the AIS hierarchy according to the information stored in a text file called "ObjectBaseFile" (see Appendix A).

- Creates and initializes **functionDataBase** - the only instance of the FunctionDataBase class. The instance object **functionDataBase** keeps in its instance variables the information about the functions in the AIS system library. Two ACG classes, FunctionCls and FunctionRetCls, are used to implement the functions in the AIS system library according to the information stored in two text files called "FunctionBaseFile" and "FunctionRetBaseFile" (see Appendix B).

- Creates the graphical interface that allows the user to build a program using graphic manipulation (see Figure 4.2).

- Creates the instances of the horizontal and vertical scroll bars used to move the user's window in the much larger workspace inside which the graphic symbols are manipulated.

- Creates the two menu-bars used to build the application for the parallel computer.

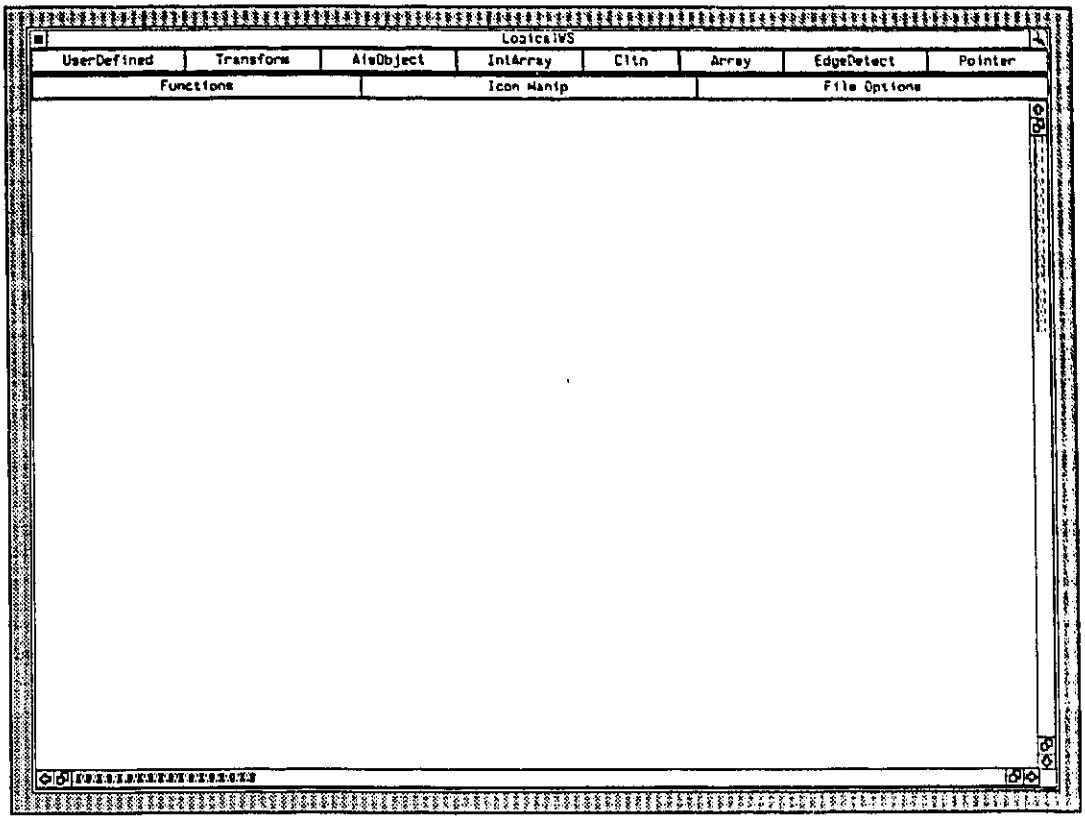

Figure 4.2: The Graphic User Interface

The ACG workspace consists of a large blank canvas upon which the application program is assembled. To aid the user in creating large program graphs, the entire space is several times larger than the window visible on the screen and is accessed by scroll bars on the window frame. The user can place interactively the icons on the screen, and once placed, the icons can be dragged anywhere inside the workspace. Except for the specification of certain numeric and string parameters for the objects or functions, all interaction with ACG is accomplished through mouse movements and button clicks.

User's actions are reflected in the creation of new objects, methods, functions, etc., and in the continuous modification of the values stored in the instance variables of **objectDataBase.**

In order to reuse the code and to enhance the original AIS objects hierarchy the ACG modifies the file ObjectBaseFile and creates also new files. One or more of them contain the source code of the newly created application (called application-files) and one or more of

them contain information about the graphical layout of the source code (called graphic-files). The graphic-files are used as inputs for the ACG at the beginning of a new session. The ACG scans the graphic-files and displays in the workspace the structure of the previously created objects, methods or functions.

The **logicalSpace** instance object interacts continuously with the other building blocks of the ACG.

4.2.2 The ObjectCls Class

The ACG-class ObjectCls is used to implement the objects in the AIS hierarchy. The initial AIS hierarchy is presented in Appendix A. The root of the hierarchy is the class Object, a very general class which implements only the basic behavior of the objects in the hierarchy (all classes are descendants of the Object class).

An instance of an AIS object is created by selecting the name of the object from the upper menu-bar.

One of the most important features of the ACG is the possibility to enhance the original AIS objects hierarchy. Every time a new object is created it is added to the collection of objects in the **objectDataBase**.

After a session with the ACG, the information in **objectDataBase** is saved in a text file called ObjectBaseFile (see Fig 4.1). Initially, this text contains only the information about the standard AIS objects. After an ACG session it also contains information about the newly created objects.

The names of the standard AIS objects are represented as options in the upper menu-bar. A non-standard AIS object (implemented in a previous session with the ACG) can be selected with the "New Object" option in the "User Defined" sub-menu.

4.2.3 The FunctionCls and FunctionRetCls Classes

An application for multi-sensor data processing implemented with the ACG makes use of the functions in the AIS system library (see Appendix B).

In order to use a function in the ACG visual programming environment, the function's

name is selected from the lower menu-bar of the ACG interface. The function's icon is created according to the information obtained from **functionDataBase**. The number and types of the input parameters, the type of output parameter (if existent) are obtained by "interrogating" **functionDataBase**. The "interrogations" are realized through message expressions.

The icon of a function can be connected through data and/or control links with the other icons in the work space. Type compatibility verifications are done automatically whenever a data or control connection is being established.

4.2.4 The CtrlFlowCls Class

The ACG-class CtrlFlowCls is used to implement the control structures: if-then-else, for and while.

The icon of a control structure is created by selecting an option from the "Control Flow Function" submenu in the "Functions" menu.

The icon corresponding to a control-structure is used to create alternative control-flow paths.

4.2.5 The ArithLogCls Class

The ArithLogCls ACG-class implements the logic and arithmetic functions. These functions can be unary functions (i.e. take one input parameter, for example the logic negation "!") or binary functions (i.e. take two input parameters, for example the four arithmetic operations and the majority of logic functions).

The icons corresponding to the unary and binary logic and arithmetic functions are created by selecting "Unary" or "Binary" from the "Arithmetic/Logic" sub-menu in the "Functions" menu.

4.2.6 The MessageCls Class

The ACG-class MessageCls is used to implement a message-expression. The icons corresponding to a message-expression is created by selecting "Message Expression" from the

"User Defined" sub-menu.

4.2.7 The VariableCls Class

The instances of the ACG-class VariableCls have – in the visual programming environment – the same role as the identifiers in a programming language. The VariableCls class instance is created by selecting the option "Variable" from the "User Defined" sub-menu.

4.2.8 The InstructionCls Class

The InstructionCls ACG-class facilitates the textual implementation of a statement. The icon of an instance of the InstructionCls class is created by selecting "Instruction" from the "User Defined" sub-menu.

The instances of the InstructionCls class are used for a mechanical translation of the graphical structure into source code.

4.2.9 The Arrow Classes

The "arrow" ACG-classes (InArrow, OutArrow, ToArrow and FromArrow) are used to implement the starting and ending points of data and control connections, respectively.

They are created by the ACG environment and the user cannot create or destroy any of their instances. They can be considered as auxiliary classes, that do not have a direct correspondent in the source code generated by the ACG.

4.2.10 The ImplodeCls Class

One of the limitations imposed by a visual programming environment is that the graphical representation of the source code takes more space than the textual representation. Some methods implemented in an object-oriented application for multi-sensor data processing are very big and the fragment of the visible **applicationLayer** is too small for the graphical representation of these methods.

In order to solve this problem the whole graphical structure build on the screen is collapsed in a box. The implementation of a method in the work space is continued taking

as root for the control flow path the icon of the ImplodeCls class. With the left button of the mouse, the user creates a control-connection having as the starting point the icon of the ImplodeCls class and as ending point the icon inside which the mouse button is released — a from-arrow or another icon of the ImplodeCls class.

Although on the screen the starting point of the control-connection is the icon of the ImplodeCls class, in fact the starting point is the to-arrow of the last icon in the principal control-flow chain of the graphical structure that was collapsed. Similarly, on the screen an icon of the ImplodeCls class can be the ending point of a control-connection, but in fact the ending point is the from-arrow of the first icon (the root) in the principal control-flow chain of the graphical structure that was collapsed.

The ImplodeCls class allows the user to build the parts of a large method by graphic manipulation and then connect them together.

4.2.11 The ModifyCls Class

The ACG-class ModifyCls allows the user to modify the structure of a class previously implemented with the ACG, by modifying its source file and the corresponding graphic-files. The ObjectCls class allows the user only to access the information in these files but not to modify them.

At the end of one session with the ACG visual programming environment one or more classes are added to the AIS objects hierarchy. Corresponding to each class a number of text files are created in the working directory. One of these files contains the source code (with the extension ".m") and the rest of the files are the graphic-files containing information about the graphical layout of the methods implemented in the new class (a graphic-file for each method).

In order to modify a method, the user first deletes it and then re-creates it by altering the graphical structure of the deleted method and saving it as a new method.

The creation of a new method leads to the modification of:

- the file containing the source code
- the file ObjectBaseFile

- the graphic-file corresponding to the method
- the information stored in the instance variables of the **objectDataBase**

4.2.12 The ObjectDataBase Class

The instance of the ObjectDataBase ACG-class is used to store the information about the standard AIS objects and about the objects created by the user during the sessions with the ACG.

The only instance of the ObjectDataBase class (called **objectDataBase**) is created at the beginning of an ACG session. The information about the AIS objects is stored between the sessions with the ACG in a text file called ObjectBaseFile (initially this text file contains only information about the standard AIS objects). At the beginning of a session with the ACG the text file ObjectBaseFile is scanned and the data extracted from it is assigned to the **objectDataBase**'s instance variables. During a session with the ACG the instance variables of **objectDataBase** are continuously modified (new classes are created, old classes are modified) and at the end of the session all the information in these instance variables is saved back in the ObjectBaseFile text file.

The **objectDataBase**'s instance variables are implemented as ordered collections (tree-like structures) and store all the necessary information about the objects in the AIS hierarchy.

The methods implemented in the ObjectDataBase class allow the user to find out the superclass of a selected class, the number of factory and instance methods implemented, the name and type of a formal input parameter, etc.

4.2.13 The FunctionDataBase Class

The instance of the FunctionDataBase class is used to store the information about the functions in the AIS system library. The only instance of the FunctionDataBase class (called **functionDataBase**) is created at the beginning of a session with the ACG.

At the beginning of a session with the ACG the text file FunctionBaseFile is scanned and all the data is assigned to the **functionDataBase**'s instance variables. The methods

implemented in the FunctionDataBase class allow the user to find out the name and type of an input parameter of a function, name and type of an output parameter, etc.

The text file FunctionBaseFile is not modified by the user's actions.

4.2.14 Other Interface Items for Controlling the Operation Flow in ACG

There are a number of features essential to the ACG, although they don't have a visual representation and/or are not manipulated as icons by the programmer during the implementation of the object oriented application. They are implemented by the **logicalSpace** instance object.

They represent user commands designed to initiate or terminate different phases in the generation of the source code for the object oriented application. They were presented in 3.3.7 and their implementation is presented in 4.4.14.

4.3 The Implementation of the ACG

4.3.1 The LogicalWS Class

The LogicalWS class is implemented in the files: LogicalWS.h and LogicalWS.m. The LogicalWS class is instantiated (see Fig 4.2) by the message:

```
(1) logicalSpace = [LogicalWS extent:  pt(1000,750)];
```

LogicalWS receives the factory message `+extent:` that determines the creation of its only instance object called **logicalSpace**. Message (1) determines the initialization of the instance of the LogicalWS class and the creation of the physical workspace, which is a very large blank canvas upon which the application is assembled:

```
(2) aBorder = [BorderLayer extent:  pt(5000,5000)];
(3) [aBorder insideColor:  colors.White];
(4) [ applicationLayer addFrontLayer:  aBorder];
```

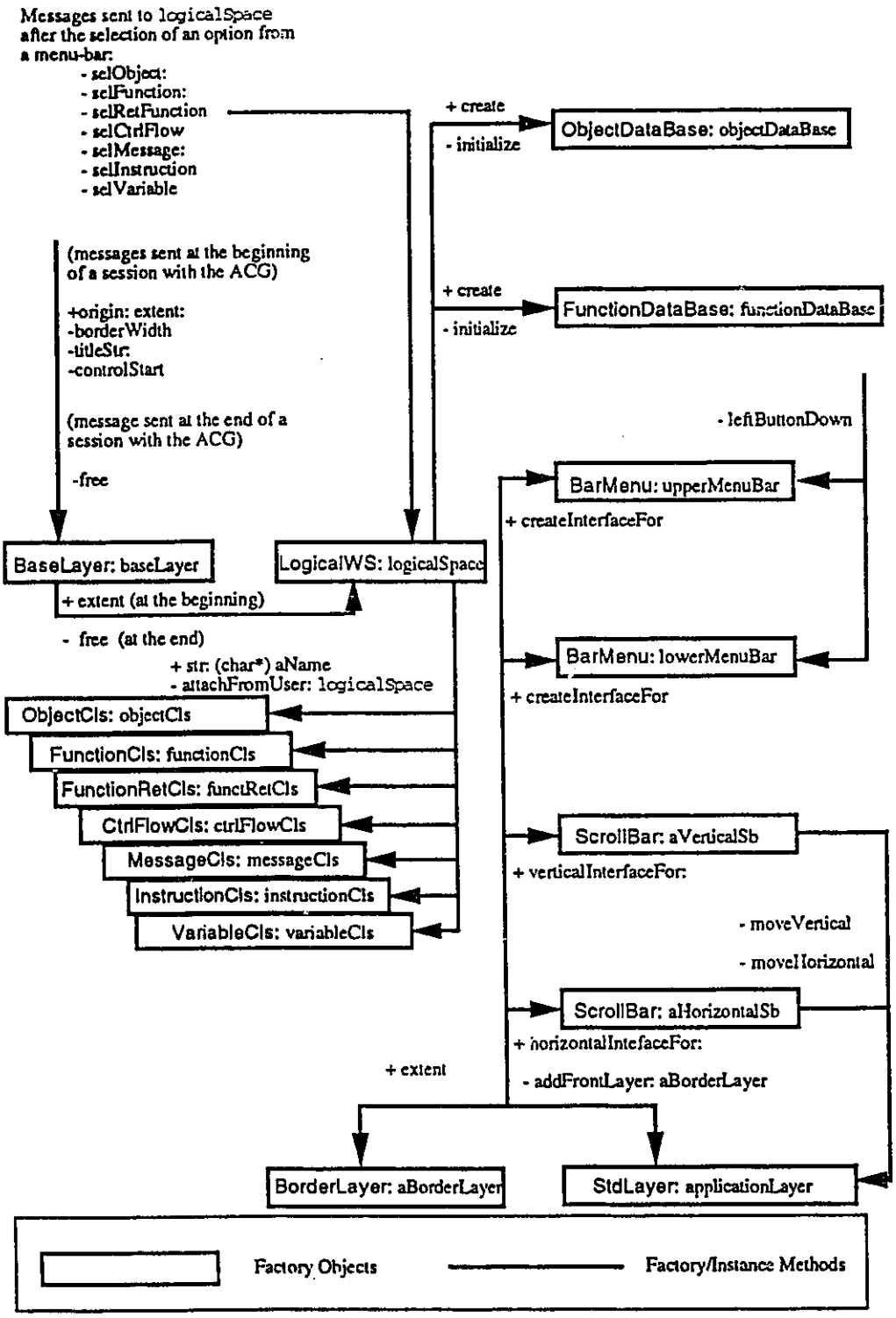

Fig. 4.3. LogicalWS Instantiation

programmer to assemble graphically the application. Message (4) allows the user to access the visible part (1000 pixels by 750 pixels) of the workspace (1).

The implementation of the most important tasks carried out by the **logicalSpace** instance object is presented below:

- Creates and initializes (5) **objectDataBase** (see Fig 4.1 and 4.2).

```
(5) objectDataBase = [[ObjectDataBase create] initialize];
```

- Creates and initializes (6) **functionDataBase** (see Fig 4.1 and 4.2).

```
(6) functDataBase = [[FunctionDataBase create] initialize];
```

- Creates the instances (7-8) of the horizontal and vertical scroll bars (see Fig 4.2 and 4.3).

```
(7) aVertSb = [ScrollBar verticalInterfaceFor:  applicationLayer];
(8) aHorizSb = [ScrollBar horizontalInterfaceFor:  applicationLayer];
```

- Creates the two menu-bars (9-10) used to build the application for the parallel computer (see Fig 4.2 and 4.3).

```
(9) myFunctionMenu = [[BarMenu createInterfaceFor:  functionMenu]
                      receiver:  self];
(10)myObjectMenu =   [[BarMenu createInterfaceFor:  objectMenu]
                      receiver:  self];
```

After the initialization phase is over the graphic interface is presented to the user (see Fig 4.3).

The creation of an icon represents, in fact, the instantiation of a factory object. The event generated by the selection of a menu option is interpreted by the **logicalSpace** object (see Fig 4.2). The request for a new object or function determines an initialization message to be send to the appropriate class.

For example, the icon of the AIS class "AisObject" is created by selecting the name of that class from the upper menu-bar. The factory method `selObject:` is sent to **logicalSpace** and a new instance of that class is created (see Fig 4.1 and 4.2). The method

`selector: receiver:` is used to sent a message to an instance of a class that is determined only at runtime.

```
(11)[aMenu append: [Menu str: "AisObject"
          selector: @selector(selObject:) receiver: nil]];
```

4.3.2 The ObjectCls Class

The ObjectCls class is implemented in the files: ObjectCls.h and ObjectCls.m. The initial AIS hierarchy is presented (in part) in Figure 4.4. The message `selObject:` sent to **logicalSpace** (see Figure 4.2) determines the creation of a new instance of the ObjectCls class called **anAisObject** (see Figure 4.5). This instance object is than attached to the workspace using the message `addItemFromUser:`. The actions generated by the `selObject:` message are presented in Figure 4.5, and the source code is given below:

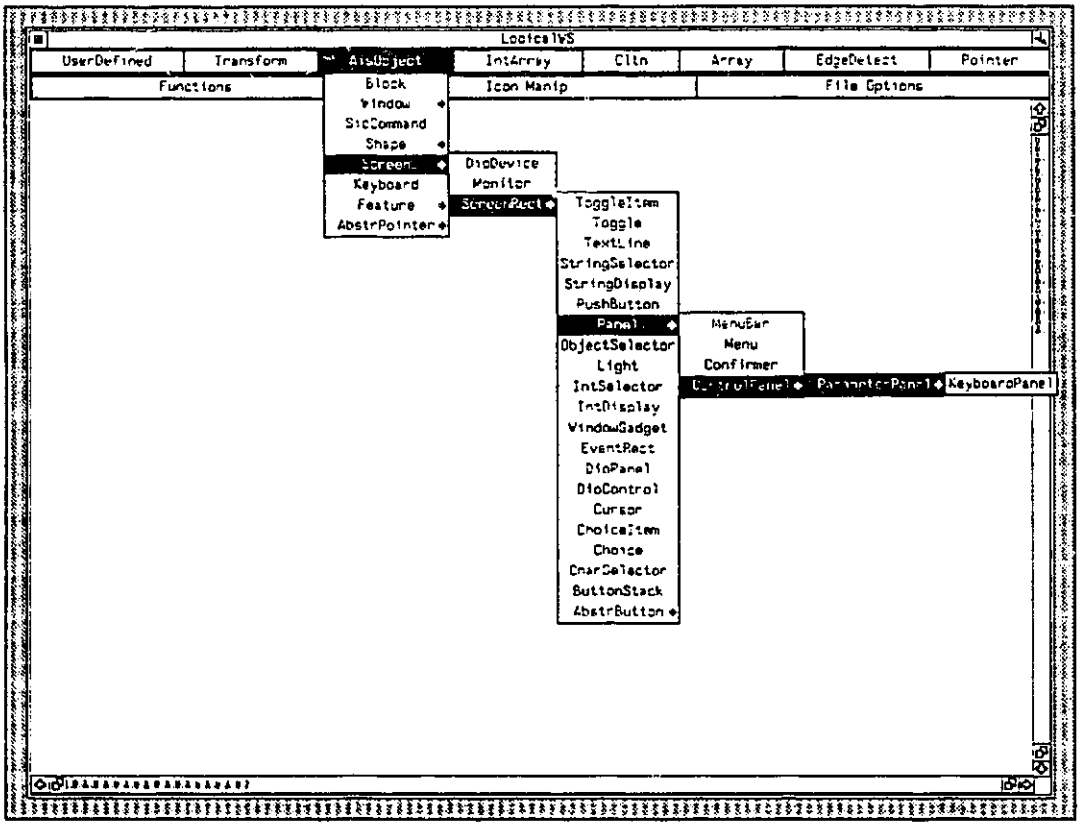

Figure 4.4: A Part of the AIS Objects Hierarchy

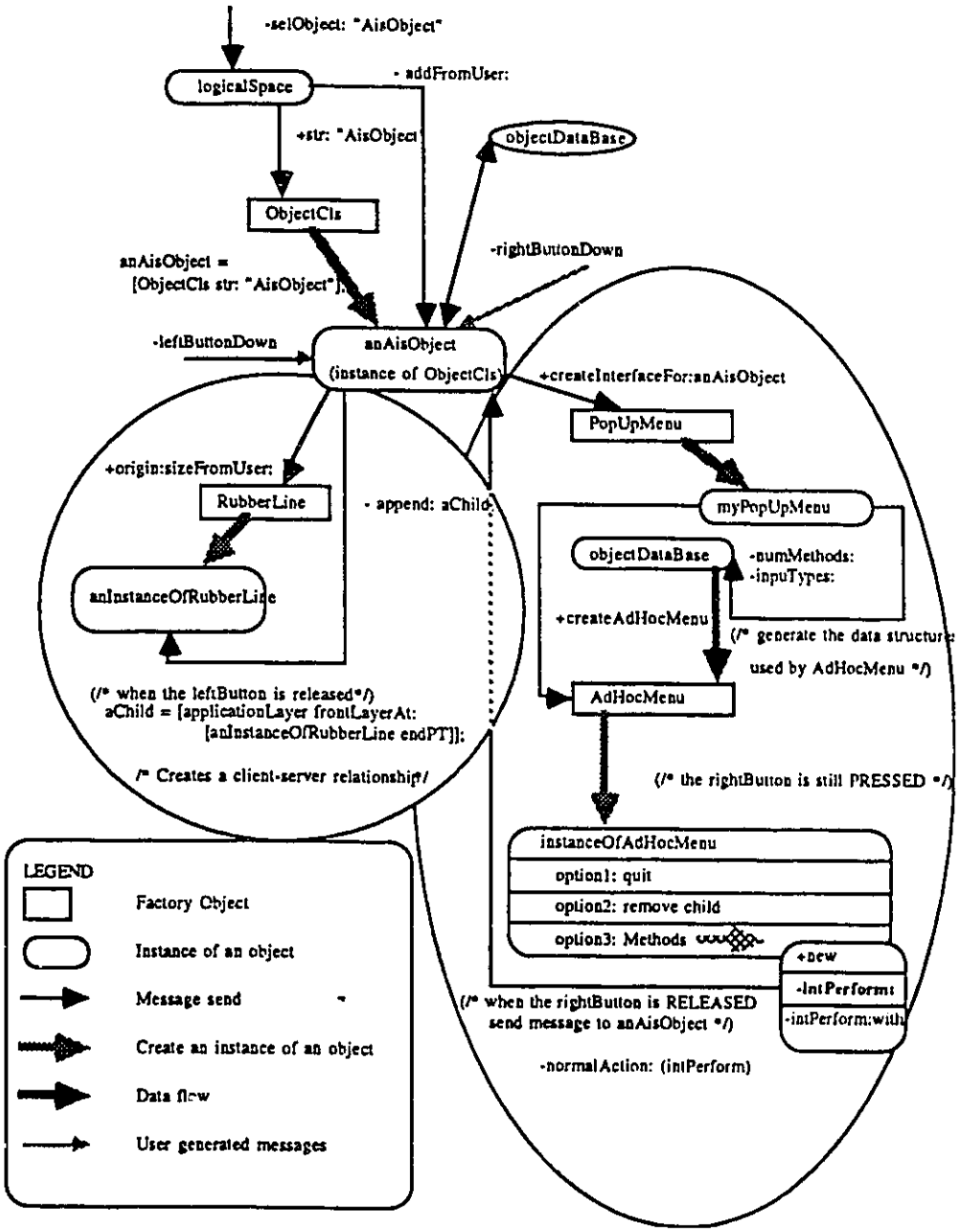

Fig. 4.5. The Generation of an Instance Object

```
-selObject:  aName {
             id    item;
(12)         item = [ObjectCls str:  [[aName menu] str]];
(13)         [self addItemFromUser:  item]
```

The right button generates a PopUpMenu with three options (see Fig 4.6).

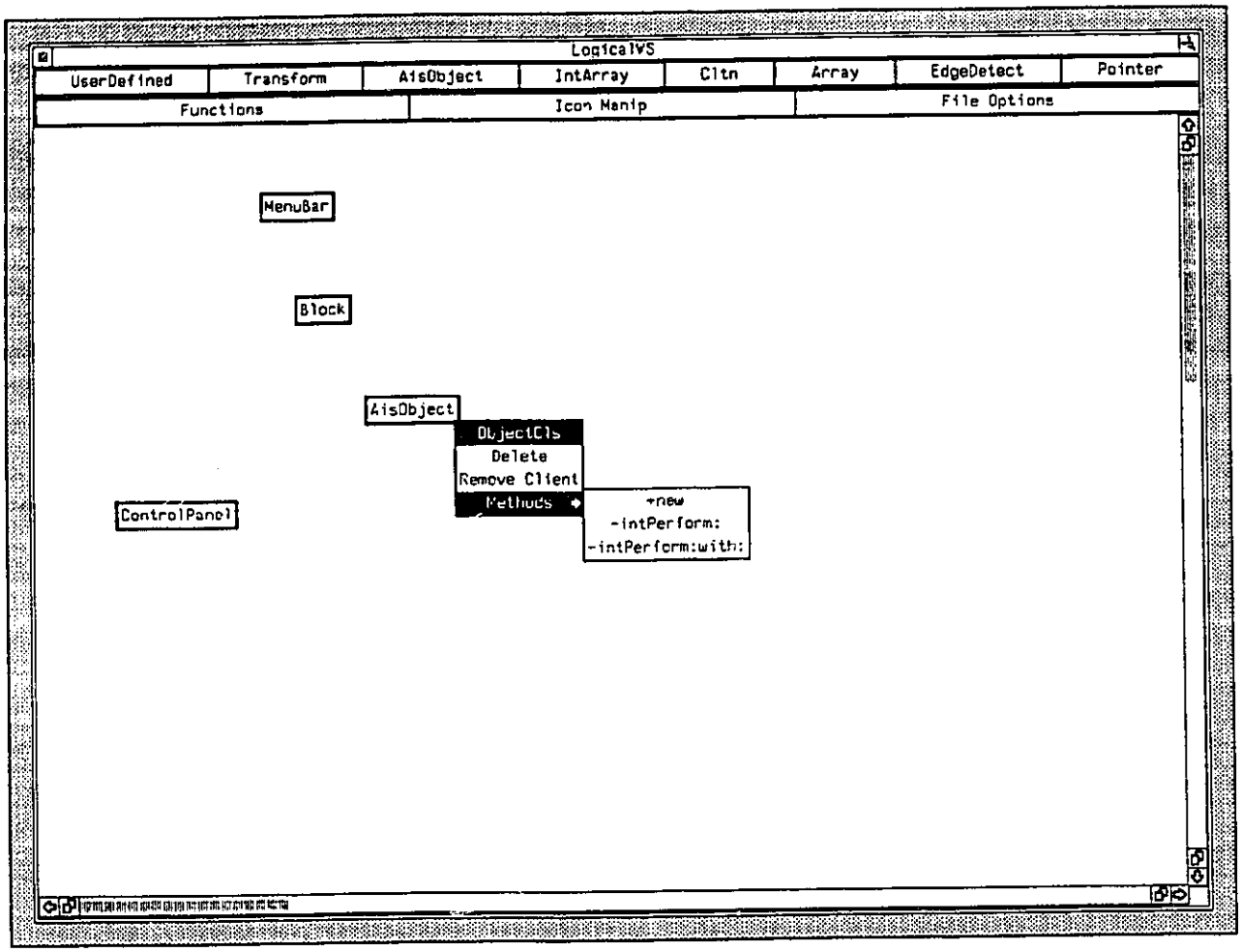

Figure 4.6: The ObjectCls Class

A part of the source code that implements the menu items is presented below:

```
(14) [aMenu append:  [Menu str:  "Delete"
            selector:  @selector(delete) receiver:  nil]];
     [aMenu append:  [Menu str:  "Remove Client"
            selector:  @selector(removeFromUser) receiver:  nil]];
(15) if(!strcmp(nameArray,"New Object"))
     {[aMenu append:  [Menu str:  "Name"
            selector:  @selector(assignSTRFromUser) receiver:  nil]];
     return aMenu;}
```

The option "Methods" is used to generate an AdHocMenu (18) containing all the methods to which this instance object reacts.

The information about the methods implemented by an instance object are extracted from **objectDataBase** and copied in previously allocated memory buffers (17).

The selection of a method from the "Methods" AdHocMenu has no visible outcome, but determines the creation of the source code for a message-expression. For example, selecting the menu option `+new` implemented by the class "AisObject", determines the source code `[AisObject new];` to be written in a memory buffer.

The left button of the mouse is used to create a Client/Server connection. The actions generated by the left button of the mouse, when pressed, are presented in Fig. 4.5, and the source code is given below:

```
-appendFromUser {
(16)line = [[RubberLine origin:  [self bottomCenter] sizeFromUser];
(17)child = [[backLayer backLayer] frontLayerAt:[line endPt]];}
```

In (16) it is presented the creation of a rubber-line, with the origin in the middle of the bottom side of the object's icon and with the end point attached to the cursor. When the mouse button is released (see Figure 4.5), the absolute position of the cursor is obtained using `[line endPt]`.

The graphic entity that is "in front" of all the other layers at the position given by the end of the rubber-line (found with the method `frontLayerAt:`) is the Client of the object's instance (a client/server connection is represented by a triple line).

4.3.2.1 New Objects in the AIS Objects Hierarchy

The selection of the option "New Object" from the "User Defined" sub-menu triggers the creation of an instance of the ObjectCls class called "New Object"(see Figure 4.7).

The menu generated by the right mouse button, when pressed, has three options: "Delete", "RemoveClient" and "Name".

Let's assume that a new class called "Threshold" has been implemented in a previous session with the ACG. The name of this class doesn't appear as a menu option in the upper

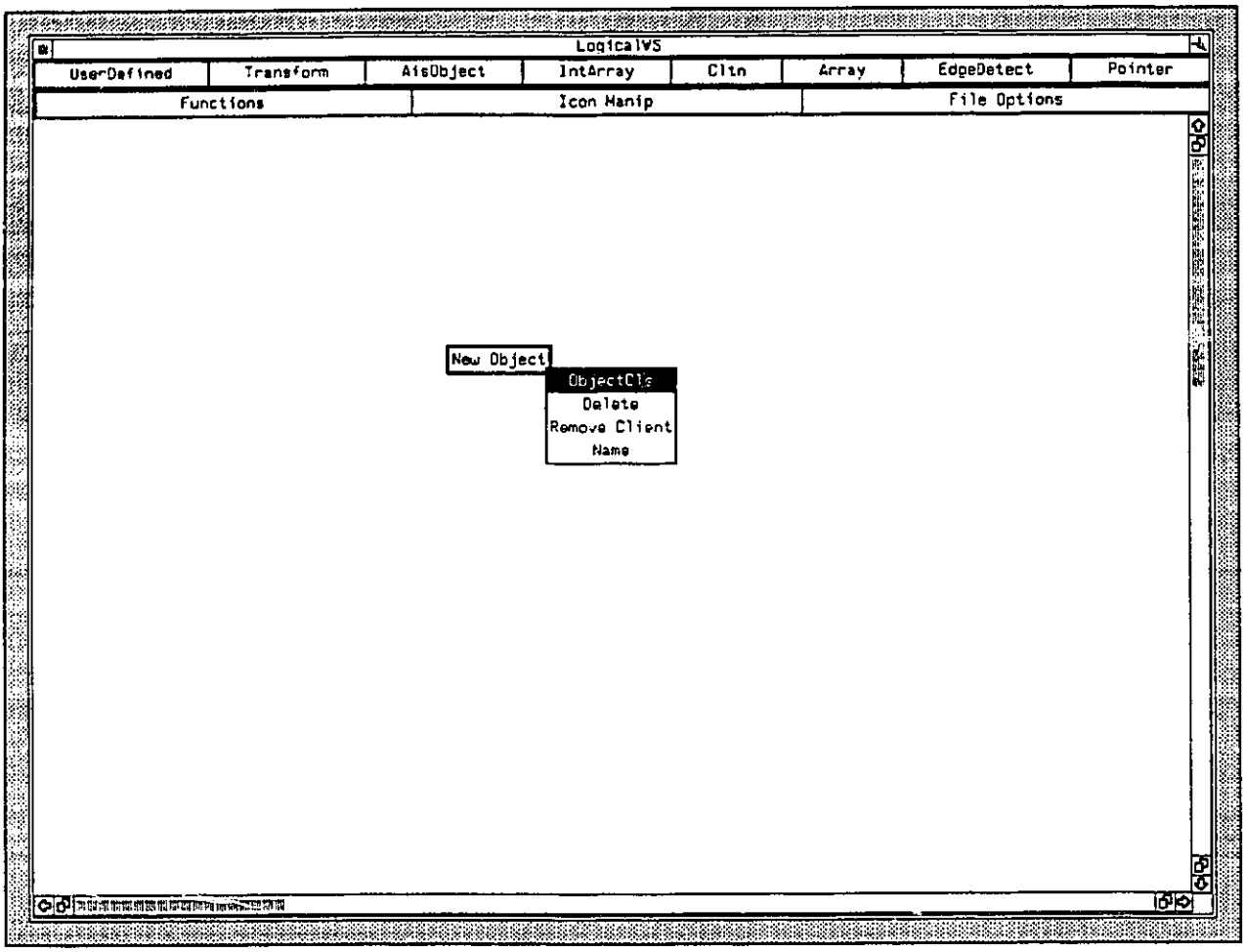

Figure 4.7: The menu options of a "New Object"

menu-bar, but all the information about this class exists in the **objectDataBase**'s instance variables. In order to access this information the user selects the option "Name" of "New Object" and introduces the name "Threshold" in the text-input box displayed on the screen.

The text displayed inside the icon is changed from "New Object" to "Threshold" (see Figure 4.8) and the menu generated by the right mouse button, when pressed, contains the option "Methods" instead of "Delete". Through the options implemented by the "Method" sub-menu the user has access to the methods implemented by the "Threshold" class.

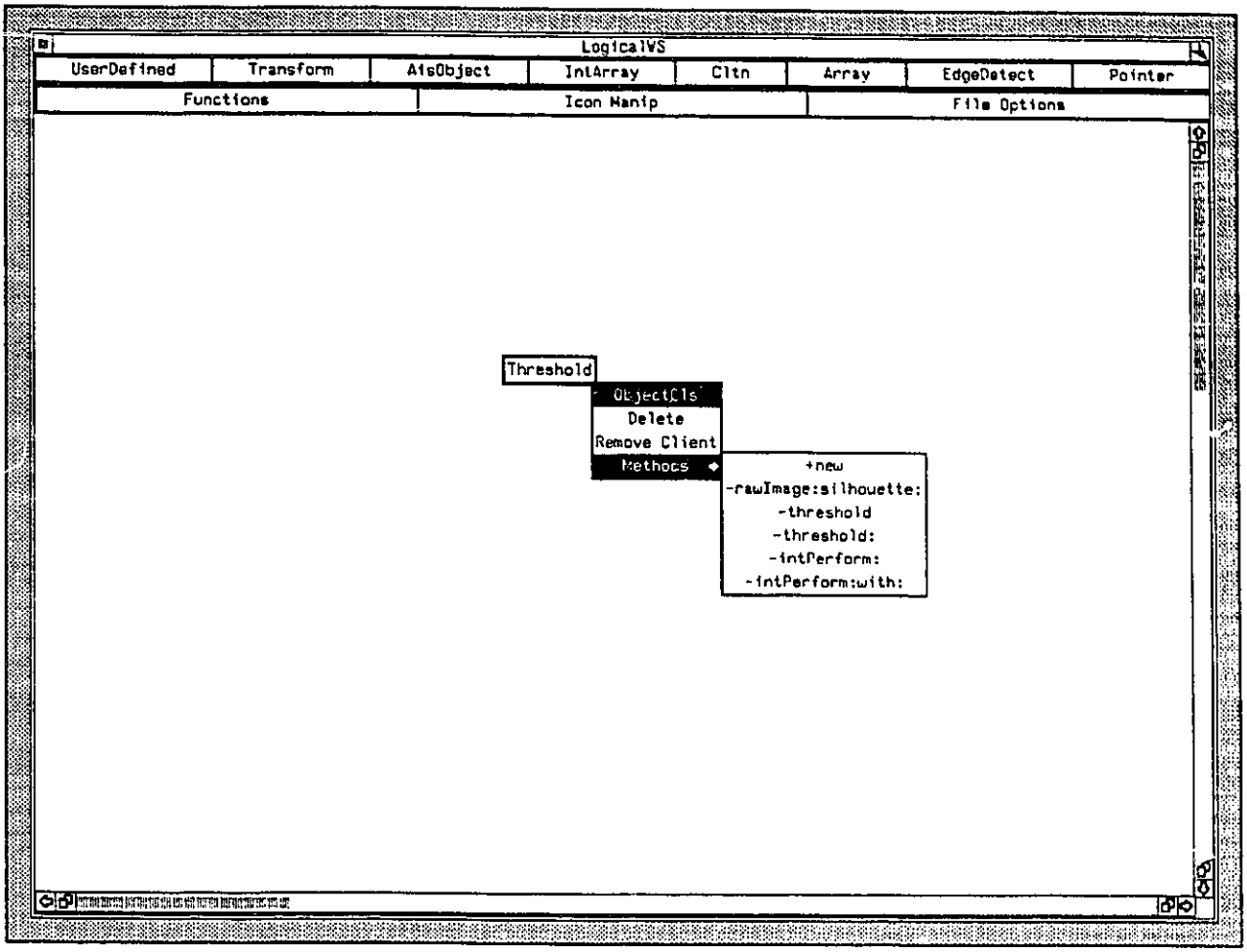

Figure 4.8: Accessing a class previously implemented with the ACG

4.3.3 The FunctionCls and FunctionRetCls Classes

The FunctionCls class is implemented in the files: FunctionCls.h and FunctionCls.m. The FunctionRetCls class is implemented in the files: FunctionRetCls.h and FunctionRetCls.m. The functions in the AIS system library are presented in Fig 4.9.

The icons presented in Fig 4.10 correspond to the AIS vision functions "addkx()" and "getpix()"

The function "addkx()" is implemented by the ACG-class FunctionCls. The corresponding source code is presented below:

```
(18) -selFunct:  aName {
                 item = [FunctionCls str:  [[aName menu] str]];
```

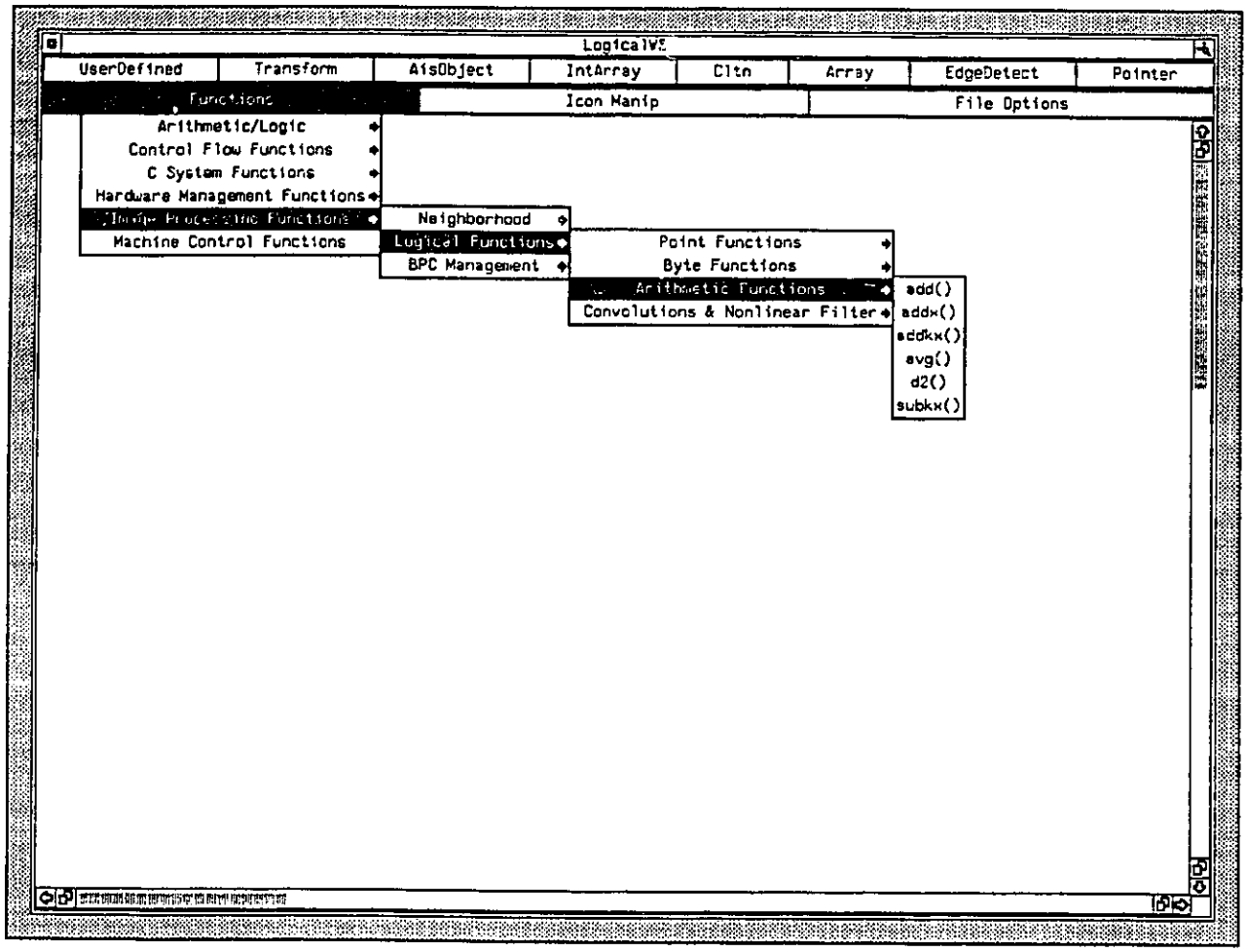

Figure 4.9: The AIS Vision Functions

```
        [self addItemFromUser:  item];
        return self;}
```

The creation of an instance of the FunctionCls class is more complex because of the arrows that have to be created and attached.

```
(19) inArrow= [InArrow displImage:  [BitMap extent:
       pt(LEFTAWIDTH,LEFTAHEIGHT) imageData:  rightArrowBits depth:1]];
(20)        [inArrow attachTo:self at:  pt(0,(22*(i-1)))];
(21)        [self appendI:inArrow];
```

In (19) an instance of the InArrow class is created and attached to the icon-box of the

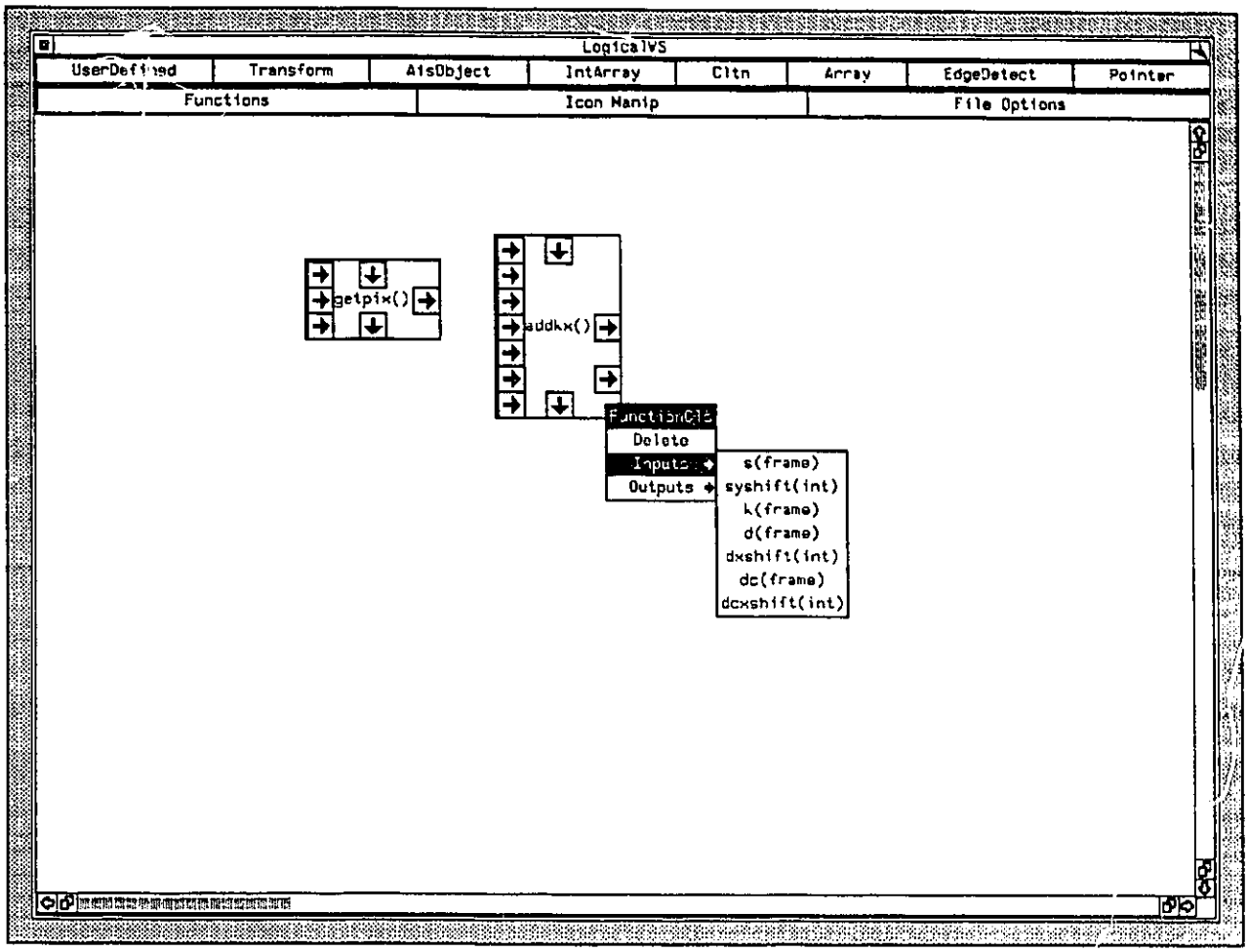

Figure 4.10: AIS vision functions: examples

function in (20). The position of the arrow and the dimensions of the icon-box representing the function are calculated according to the total number of inputs.

Each input arrow id is stored in an instance variable (an ordered collection) of the FunctionCls or FunctionRetCls class instances. A new input arrow is added to the ordered collection in (21). Instances of the OutArrow, FromArrow and ToArrow classes are created in the same manner. The FunctionRetCls ACG-class implements the functions that are returning a value, for example "getpix()". The function "getpix(s,x,y)" returns the n-bit value of the pixel with coordinates (x,y) in the n-bit frame s.

The method `-makeFunction:` is used to create source code for the object-oriented application executed on the parallel machine. The ACG modifies the **newMethod** temporary file, which is given as a parameter to the `-makeFunction:` method.

```
-makeFunction:(FILE*)pfileFunct
{ for(i=0;i<inputsNum;i++){
(22)          anInput = [inputs at:  i];
(23)          theParent = [anInput parent];
(24)          if([theParent isKindOf:VariableCls])
                    fprintf(pfileFunct,''%s'',[theParent menuSTR]);}
};
```

A data-connection is always linking a parent (the start point of the data-connection) and a child (the end point of the data-connection). The child is obtained in (22) and the parent in (23). The **newMethod** is then modified according to the parent's type in (24).

If the parent is an instance of the VariableCls (24), then the name and type of the parent are already known and the graphical parsing is over. If the parent is the output arrow of a different function the graphical parsing continues until the name and type of the parent is determined.

In order to save its graphical structure, each ACG-class implements the methods `storeInFile:` and `retrieveFromFile:`.

The data stored in the graphic-file represents:

- the type of the icon (for example, "FunctionCls")
- the name of the instance object (for example: "addkx()")
- the position of the icon in the workspace in.

The `storeInFile:` message is sent also to the parents of the input arrows. The control connections are scanned after all the data dependencies are analyzed and the `storeInFile:` message is sent to the next icon in the control flow chain.

4.3.3.1 The Assign Operation

The "Assign" operation is implemented by the FunctionRetCls class (see Figure 4.11).

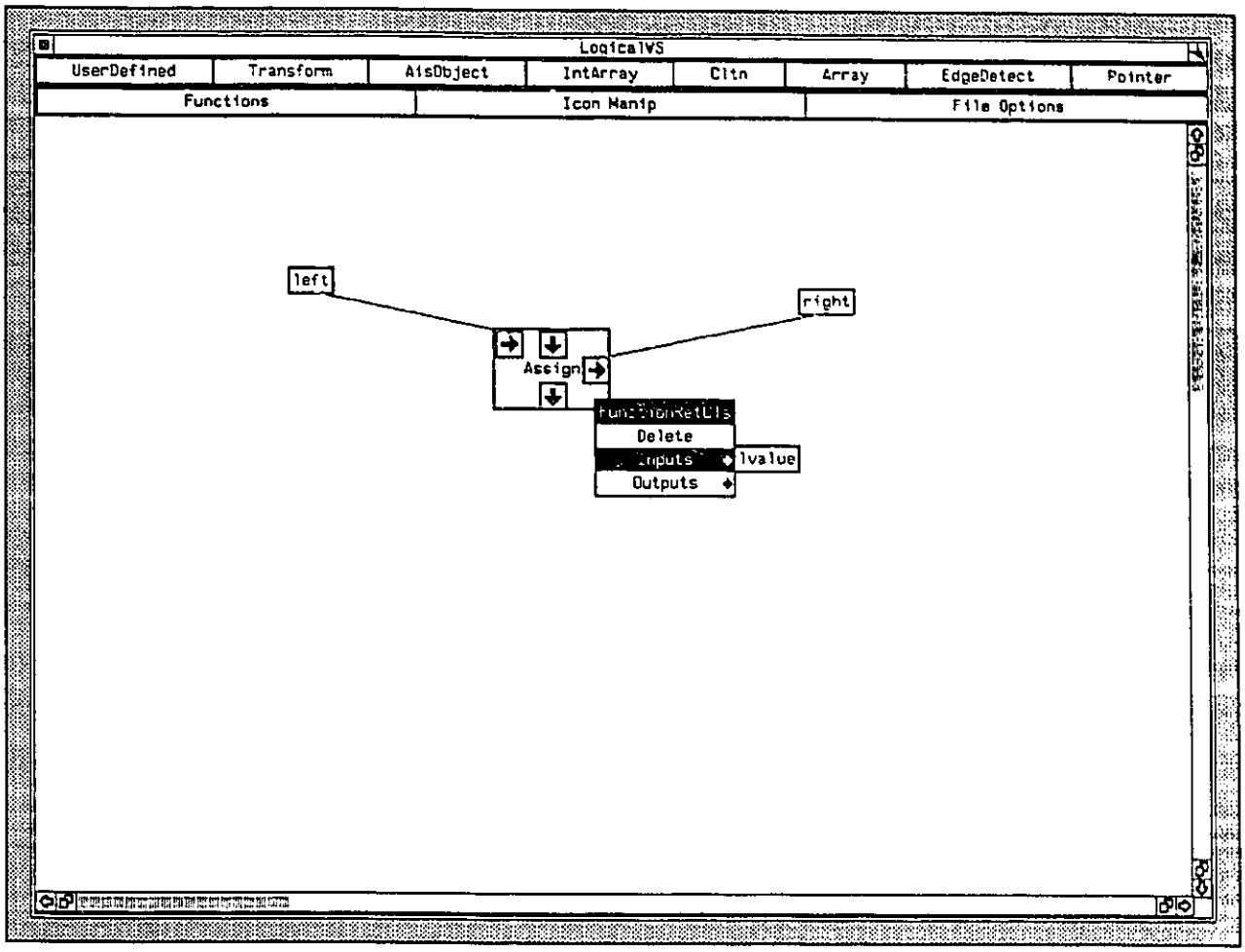

Figure 4.11: The Assign Operation

The arrow attached to the right side of the rectangular box is an input arrow. Figure 4.11 suggests that the value represented by the icon "left" goes into the value represented by the icon "right" through the "Assign" icon

4.3.4 The CtrlFlowCls Class

The CtrlFlowCls class is implemented in the files: CtrlFlowCls.h and CtrlFlowCls.m. The icons of the control structures implemented by the ACG are presented in Figure 4.12.

The icon of the "If()" function has an input arrow attached to the left side of the icon, a from-arrow attached to the top of the icon, a to-arrow attached at the bottom of the icon and two more to-arrows attached to the right side of the icon.

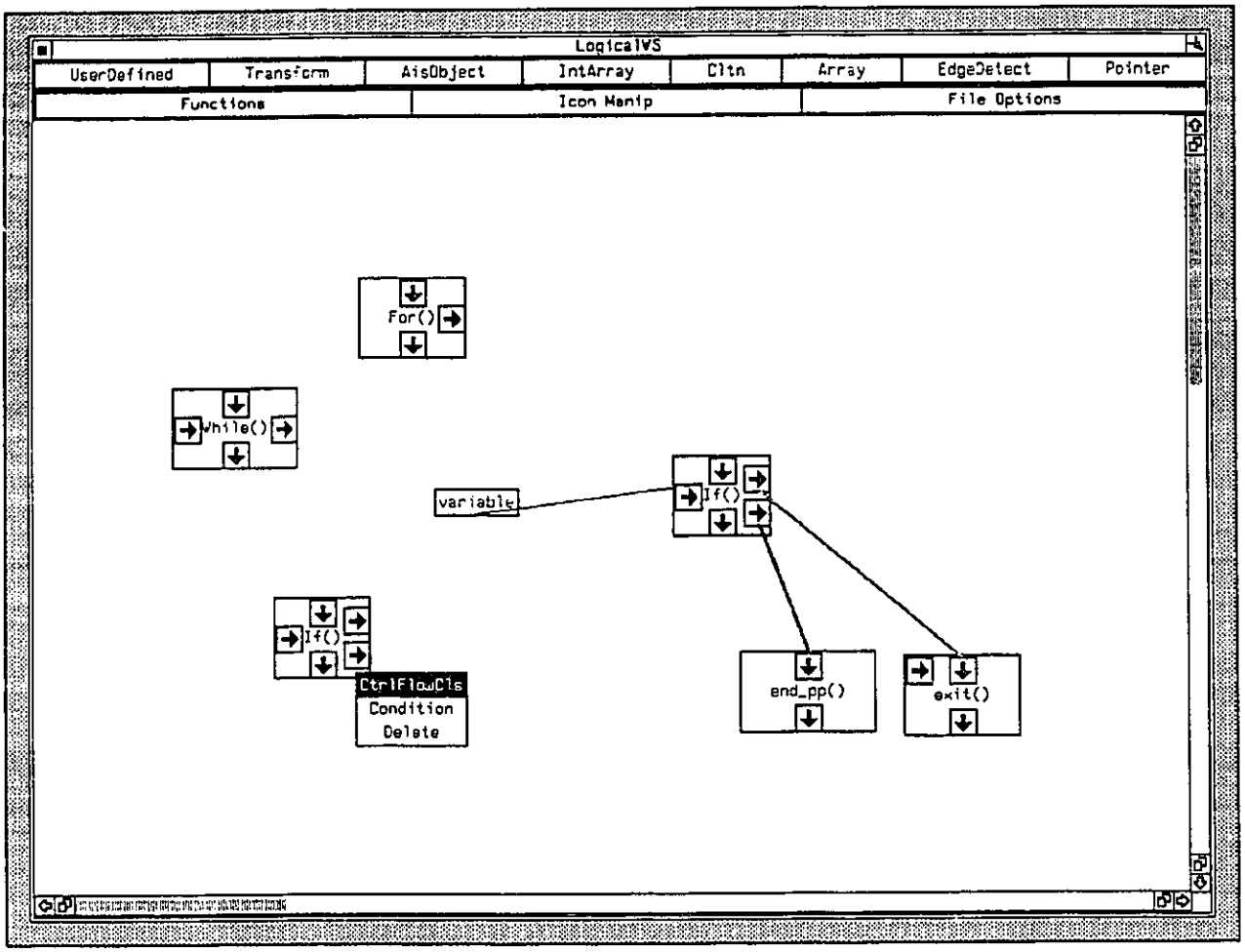

Figure 4.12: The ControlFlowCls Class

The two to-arrows attached to the right side of the icon are the starting points of secondary control-flow chains. The icons connected to the top to-arrow represent the functions that are executed if the condition evaluated by the function is true, and the icons connected to the bottom to-arrow represent the functions that are executed if the condition is false.

The `makeFunct:` method implemented by the CtrlFlowCls class is invoking one of three methods according to the instance being analyzed.

```
(25) -makeFunct:  (FILE*) pfileFunct {
                  if(!strcmp(iconName,''If()''))
                          [self makeIf:pfileFunct];
                  else if(!strcmp(iconName,''For()''))
```

```
                [self makeFor:pfileFunct];
        else if(!strcmp(iconName,"While()"))
                [self makeWhile:pfileFunct];
        return self;}
```

A fragment of the `makeIf:` method is presented below:

```
(26) -makeIf:(FILE*) pfileFunct{
(27)    firstAlt = [[output children] at:  0];
        if([firstAlt isKindOf:  MethodCls])
                fprintf(pfileFunct,"[%s];",[firstAlt giveMetBuffer]);
        else
                [[firstAlt backLayer] makeFunct:pfileFunct];
        if([secondAlt isKindOf:  MethodCls])
                fprintf(pfileFunct,"[%s];",[secondAlt giveMetBuffer]);
        else
                [[secondAlt backLayer] makeFunct:pfileFunct];
    ...
(28)    if(toNextFunction = [[nextFunction children] at:0])
                [[toNextFunction backLayer] makeFunct:  pfileFunct];
        return self;}
```

The variable `firstAlt` (first alternative) corresponds to the top to-arrow which creates a secondary control-flow chain (27). The process described for `firstAlt` is repeated for `secondAlt`, the bottom to-arrow. The `makeFunct:` method is recursive; it takes as parameter only the file-descriptor of the newMethod file.

4.3.5 The ArithLogCls Class

The ArithLogCls class is implemented in the files: ArithLogCls.h and ArithLogCls.m. The icons corresponding to the unary and binary logic and arithmetic functions are created by selecting "Unary" or "Binary" from the "Arithmetic/Logic" sub-menu in the "Functions"

menu. The only difference between the icon of a unary and that of a binary function is the number of data input arrows (see Fig. 4.13).

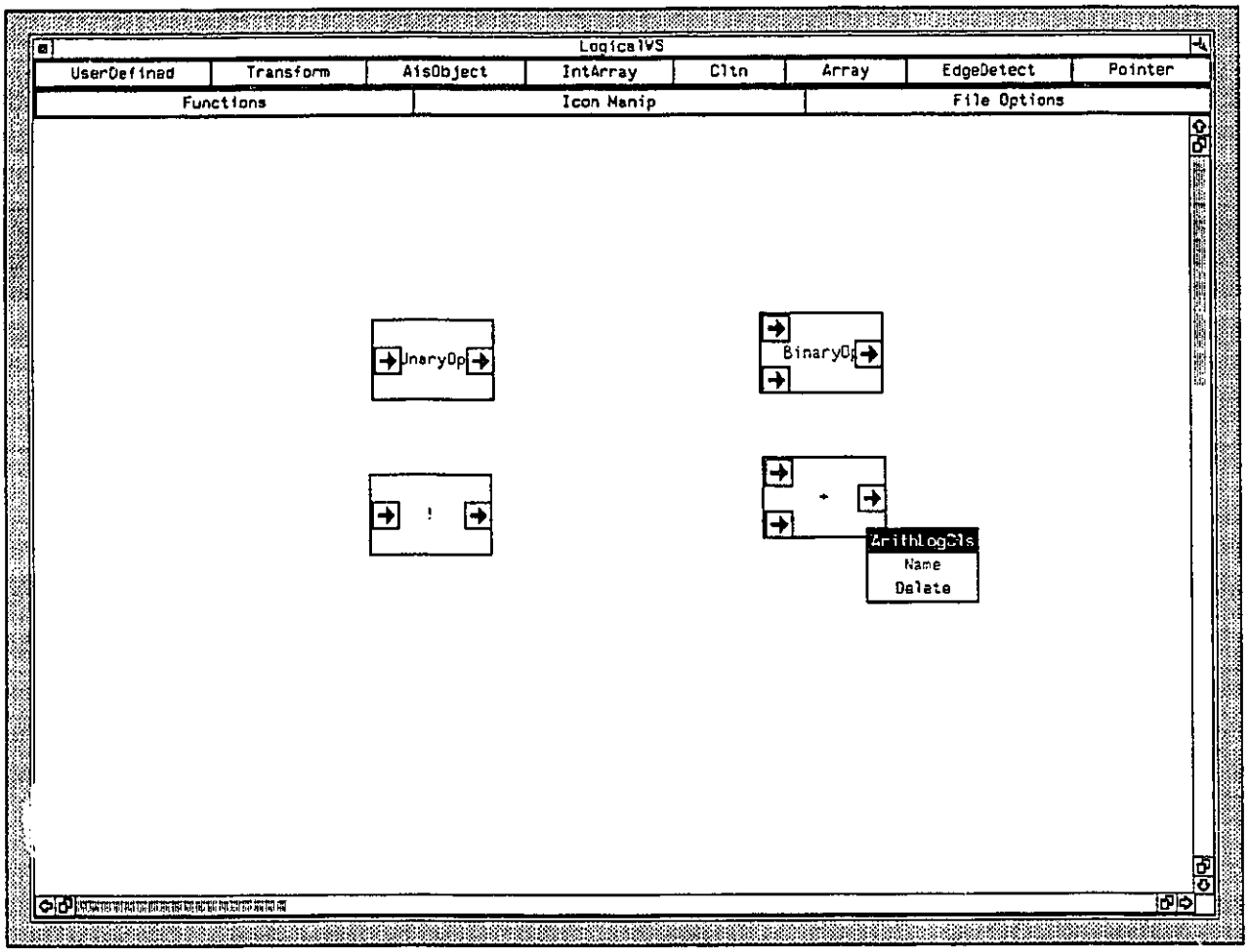

Figure 4.13: The ArithLogCls Class

An icon of the ArithLogCls class is connected in the graphical structure built by the user only through data connections (hence, it has only input and output arrows).

The PopUpMenu generated by pressing the right button of the mouse has two options: "Name" and "Delete". The icon of an arithmetic or logic function is created by introducing the name of the function (for example the plus sign ("+") for addition) in the dialog box displayed on the screen as a result of selecting the option "Name" from the PopUpMenu generated by the right mouse button.

The implementation is very similar to that of the FunctionCls class. The only difference is that the ArithLogCls class instance doesn't have control-arrows. This fact simplifies the

implementation and reduces the number of instance variables defined for each instance of the class.

4.3.6 The MessageCls Class

The MessageCls class is implemented in the files: MessageCls.h and MessageCls.m. The following example is used to demonstrate the concept behind the MessageCls class.

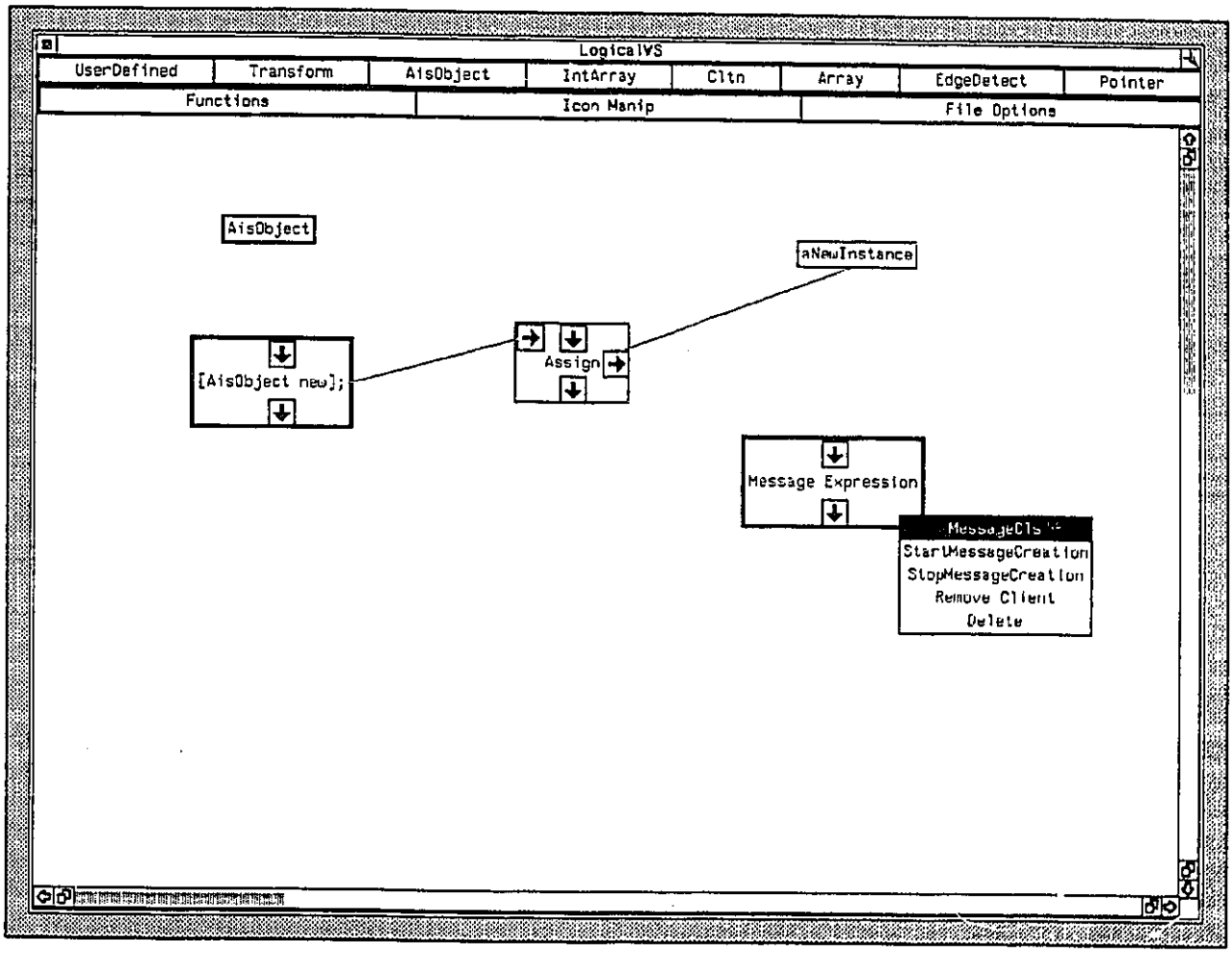

Figure 4.14: The MessageCls Class

```
aNewInstance = [AisObject new];
```

In this example the `id` variable `aNewInstance` takes the value returned by the message `+new` sent to the `AisObject` class. The MessageCls class is used to implement the message-expression (i.e. the right hand side of the assign operation (see Fig. 4.14). The left button

of the mouse is used to create a data connection (see (20) in subsection 4.3) having as the starting point the icon of the MessageCls class.

The right button of the mouse generates a PopUpMenu with four options: "StartMessageCreation", "StopMessageCreation", "Remove Child" and "Delete".

The selection of the first option - "StartMessageCreation" - creates a memory buffer and assigns the buffer's address to a global variable called **baseMethodBuffer**.

The source code corresponding to the message-expression is created by selecting the name of a method implemented in one of the AIS classes (see subsection 4.3) and is stored in the memory buffer **baseMethodBuffer**.

The second option - "StopMessageCreation" - changes the text "Message-expression" with the text stored in **baseMethodBuffer**. In the present example, the text inside the icon box is changed from "Message-expression" to `[AisObject new];`.

The MessageCls class can be used to create nested message-expressions, for example `[[AisObject new] performAction]`. In order to do this the actions described above are performed twice: once for the `+new` method and once for the `-performAction` method.

4.3.7 The VariableCls Class

The VariableCls class is implemented in the files: VariableCls.h and VariableCls.m.

The menu options of the PopUpMenu activated by the right button of the mouse when pressed inside the icon of an instance of the class VariableCls are implemented in the method `-defaultMenu` in the VariableCls.m file.

```
(29) -defaultMenu {
     aMenu = [Menu str:  [self name]];
     [aMenu append:  [Menu str:  ''Name''
          selector:  @selector(assignSTRFromUser) receiver:  nil]];
     ...
     return aMenu;}
```

The option "Name" in the PopUpMenu generated with the right button of the mouse allows the user to customize the instance object. Instances of the VariableCls class are used

to create the graphical structure corresponding to an assign operation (see Figure 4.14). The graphical structure corresponding to the assign operation `aVar = 125;` is presented

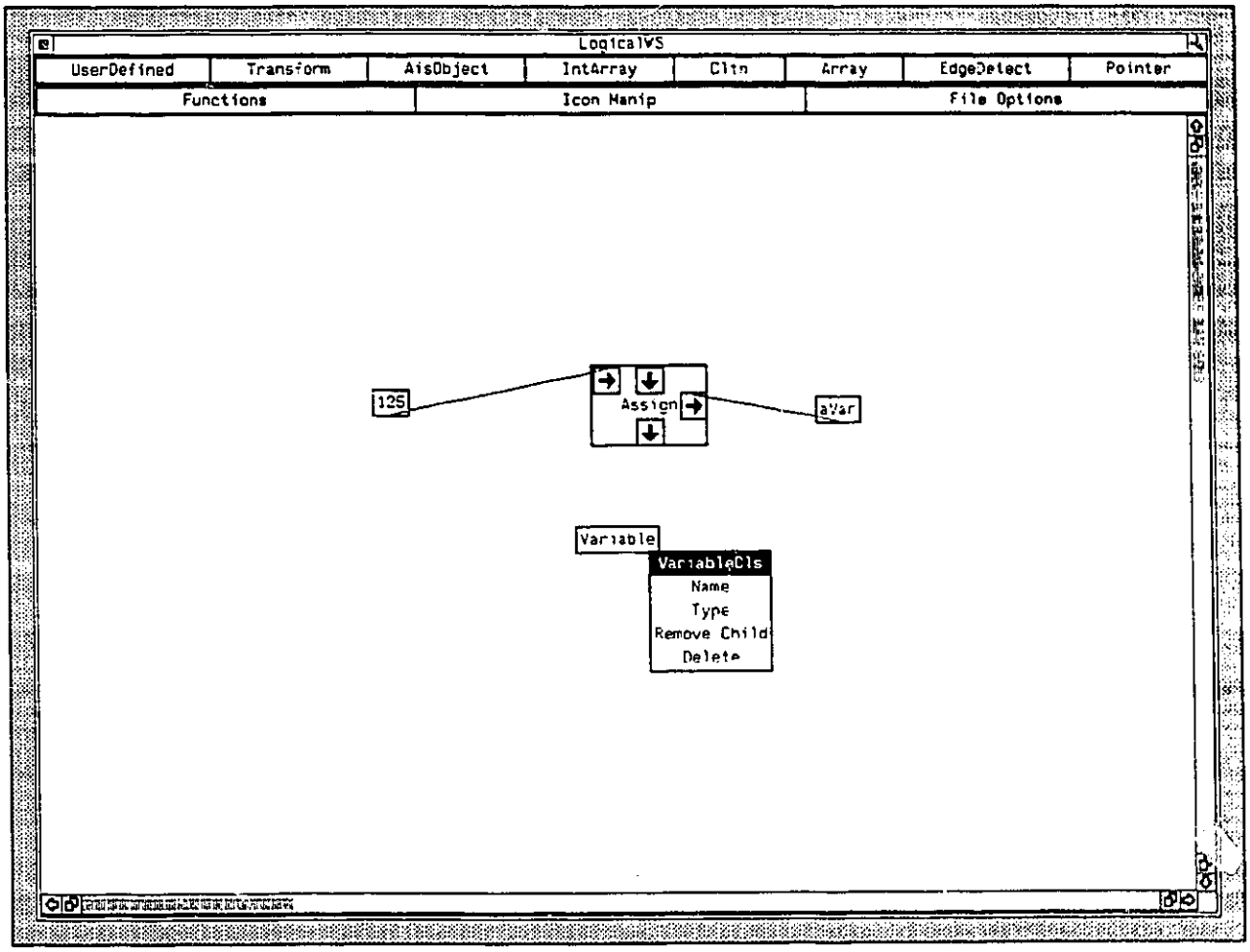

Figure 4.15: The Variable Class

in Figure 4.15; it contains two instances of the VariableCls class and one instance of the FunctionRetCls class (i.e. "Assign").

The instance which implements "aVar" corresponds to a variable in the programming language and is created by introducing the text "aVar" after selecting the option "Name" from the PopUpMenu generated with the right mouse button. The text displayed inside the icon changes from "Variable" to "aVar".

The type is assigned to an instance of the VariableCls class by selecting the option "Type", and introducing the corresponding key-word (for example: int, char, frame, etc.). The type can be omitted when a new instance of the VariableCls class is created; it is

requested from the user only if it cannot be determined during the graphical parsing phase.

The other instance of the VariableCls class implements a constant. The name is given by the user as being "125" after selecting of the option "Name". The type "int" is automatically associated with the instance of the VariableCls class because in this case the name contains only digits.

An important feature implemented by the VariableCls class is the type verification. This feature is activated when the icon of a VariableCls class instance is connected to another icon through a data connection. First the validity of the selection is verified: the ending point of the data-connection has to be an input arrow. If the connection is valid, a type verification is performed. The type of the instance of the VariableCls class has to be the same as the type associated with the input arrow selected as the ending point of the data connection. If the two icons don't have the same type, a warning message is issued and the connection is refused.

4.3.8 The InstructionCls Class

The InstructionCls class is implemented in the files: InstructionCls.h and InstructionCls.m. The icon of an instance of the InstructionCls class is almost identically with the icon of a MessageCls class. The only difference is that the name inside the icon is "Instruction" instead of "Message-expression". The user modifies the text inside the icon of the InstructionCls class by selecting the "MakeInstruction" option of the PopUpMenu generated by the right button of the mouse when pressed. The "MakeInstruction" option creates on the screen a confirmer box with two options (see Figure 4.17) Selecting "yes" changes the text inside the icon box to `return self;`. The "no" selection determines the creation of a dialog box on the screen (30) that requests the user to enter the statement.

```
-makeInstruction {
   if ([Confirmer askSTR: "return self ?"])
(30) strcpy(instrBuffer,"return self;");
   else
     prompt = [[LabelValue labelSTR:"Enter instruction:  "] maxValueSize:  60];}
```

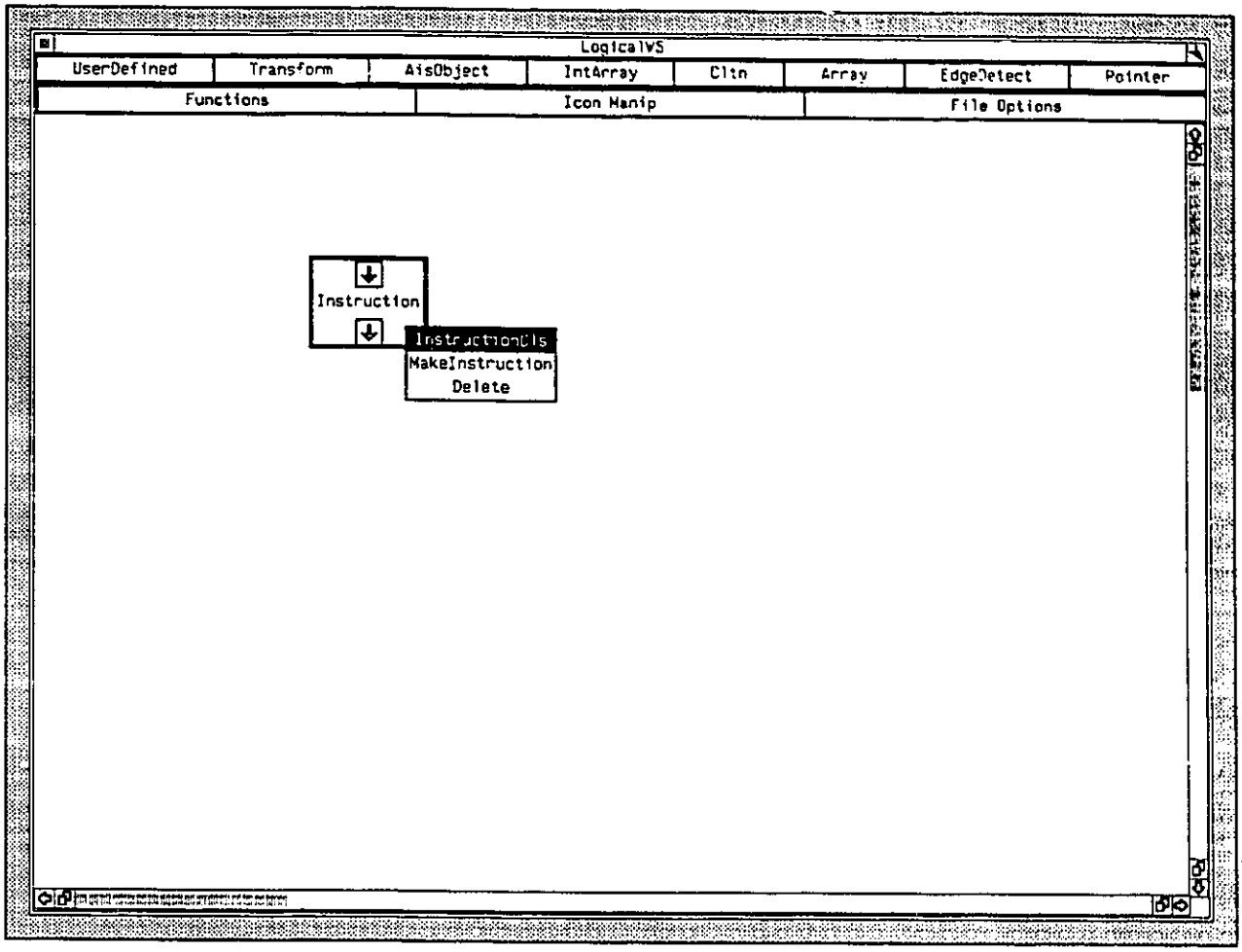

Figure 4.16: The InstructionCls class

The text inside the icon of the InstructionCls class instance changes to "`return self;`" or to the text introduced by the user.

The InstructionCls class is used to create a statement using text-edit facilities when it is too complicated to create it graphically.

4.3.9 The Arrow Classes

The majority of ACG-classes' icons contain two or more instances of the "arrow" classes .

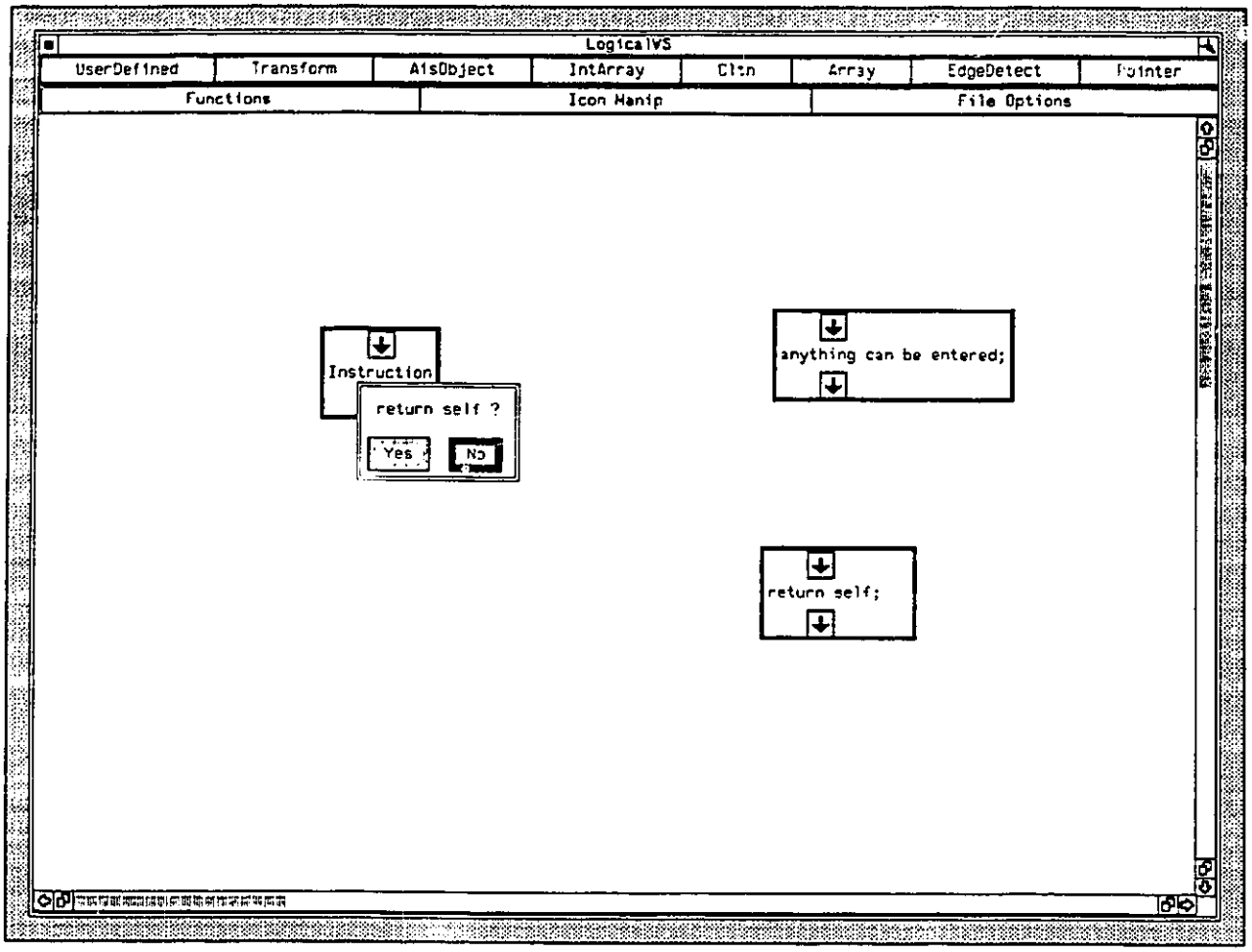

Figure 4.17: Confirmer box created by the InstructionCls

4.3.9.1 The InArrow Class

The InArrow class is implemented in the files: InArrow.h and InArrow.m. The icon of the InArrow class (a sub-class of the BorderLayer class) is a square box with an arrow pointing to the right inside it. An instance of the InArrow class is created by the following code:

```
(31) inArrow = [InArrow dispImage:  [BitMap extent:
     pt(LEFTAWIDTH,LEFTAHEIGHT) imageData:  rightArrowBits depth:1]];
```

An input arrow is used to implement the notion of data input parameter. The id-s of the input arrows are stored in an instance variable called "inputs"implemented as an ordered collection.

4.3.9.2 The OutArrow Class

The OutArrow class is implemented in the files: OutArrow.h and OutArrow.m. The icon of the OutArrow class is identical with the icon of the InArrow class but has different properties. An output arrow is used to implement the notion of data output parameter.

The OutArrows's instance variable "children" (32) is the root of the ordered collection that contains all the id-s of the input-arrows linked through a data connection to an instance of the OutArrow class.

```
(32) children = [OrdCltn new];
```

The instance variable "children" is used in the parsing process to identify the icons that form the graphical structure of a method.

4.3.9.3 The FromArrow and ToArrow Classes

The ToArrow class is implemented in the files: ToArrow.h and ToArrow.m. The FromArrow class is implemented in the files: FromArrow.h and FromArrow.m. The icons of the ACG-classes ToArrow and FromArrow – a square box with an arrow pointing downwards inside it – are used to implement the starting and respectively the ending point of a control connection.

A from-arrow (attached to the top of an icon box) can be connected only with a to-arrow (attached to the bottom of an icon box). The instance of the ToArrow class is the parent and the instance of the FromArrow class is the child. A to-arrow can have only one child and a from-arrow can have only one parent.

4.3.10 The ImplodeCls Class

The ImplodeCls class is implemented in the files: ImplodeCls.h and ImplodeCls.m.

The creation of an instance of the ImplodeCls class is triggered by the selection of the option "Implode" from the "Icon Manipulation" sub-menu (33).

```
(33) [iconManip append:  [[Menu str:  "Implode"]
     selector:  @selector(implodeMenu) receiver:  nil]];
```

When the "Implode" option is selected, the message `implodeMenu` is send to an instance of the ImplodeCls class whose **id** is determined at run-time.

```
-implodeMenu {
// If no root, better get one
(34)        if (!root) [self rootFromUser];
// Change the cursor to indicate something is happening
(35)        [Cursor push:  cursors.Wait];
// Save the graphical structure
(36)        [self saveMenu:  root on:  fileName];
(37)        item = [ImplodeCls str:  iconName ];
(38)        [self addItemFromUser:  item];
// Restore cursor
            [Cursor pop];
            return self;}
```

The actions triggered by the message `implodeMenu` are:

- the existence of the root is checked in (34) (the icon from where the graphical structure is saved)
- if the root is not yet selected the shape of the cursor is changed and the user is requested to select a root (35)
- the graphical structure is saved in a file (36) and deleted from the **applicationLayer**
- an icon of the ImplodeCls class is created (37) and attached to the screen (38)

The factory message `+str:` sent to the ImplodeCls class creates an instance object whose icon is a rectangular box having the same name as the file in which the graphical structure is saved.

The "Explode" menu option in the PopUpMenu generated by the right button of the mouse is sending the`retrieveFromFile` method to **logicalSpace**.

The actions triggered by the `-explode` method are:

- a file with the same name as the text inside the icon's box is opened
- the message `retrieveFromFile:` is sent to the instance of the LogicalWS class. According to the information in the specified file, the graphical structure is restored on the screen (39)
- the temporary file in which the information was stored is deleted
- the ImplodeCls's icon is deleted

4.3.11 The ModifyCls Class

The ModifyCls class is implemented in the files: ModifyCls.h and ModifyCls.m. The selection

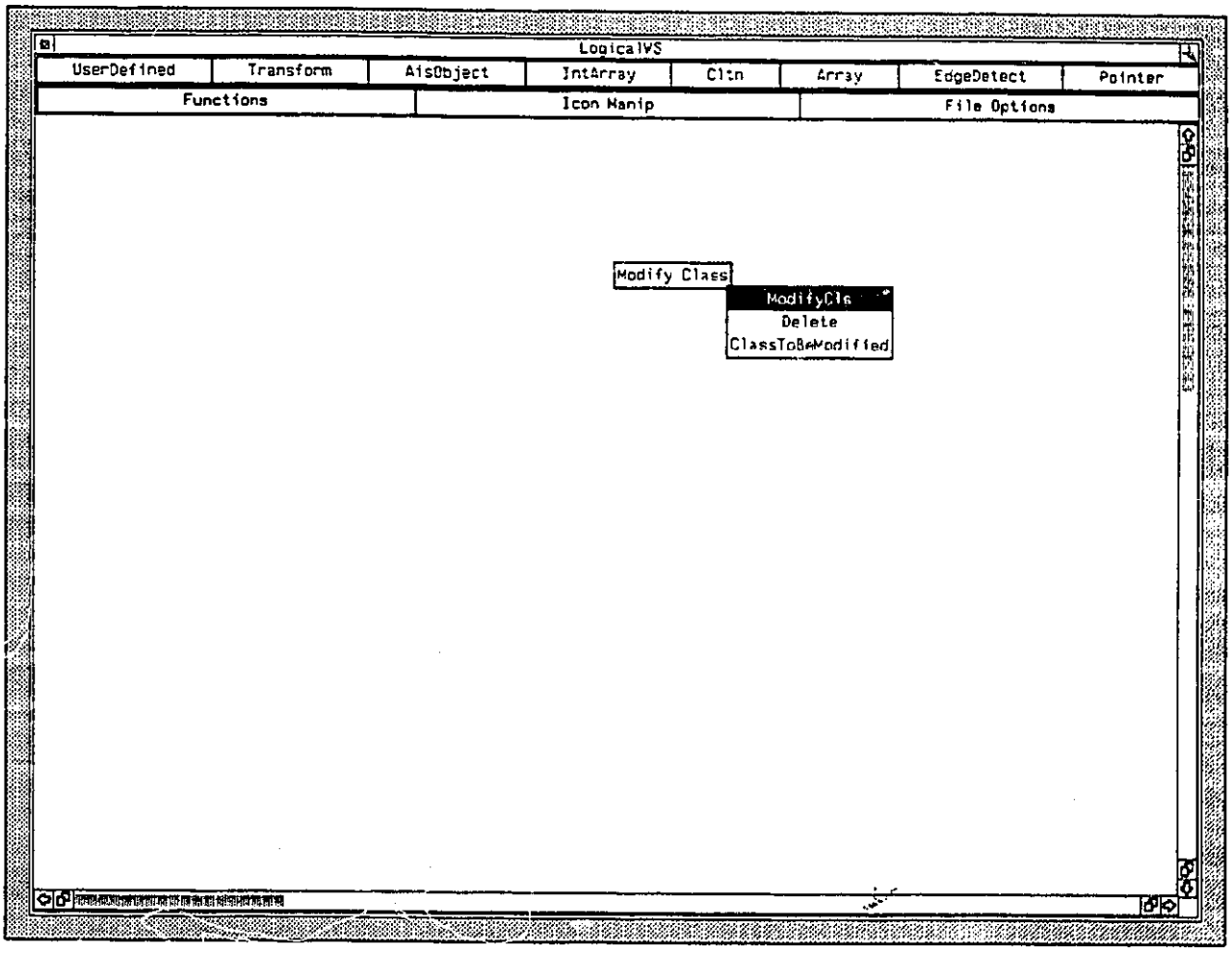

Figure 4.18: The ModifyCls Class

of the "Modify Class" option from the "Application File" sub-menu triggers the creation of an instance of the ModifyCls class.

The user modifies the structure of a class previously implemented with the ACG by selecting the option "ClassToBeModified" from the PopUpMenu generated by the left mouse button and introducing the name of the class. The text inside the icon changes from "Mod-

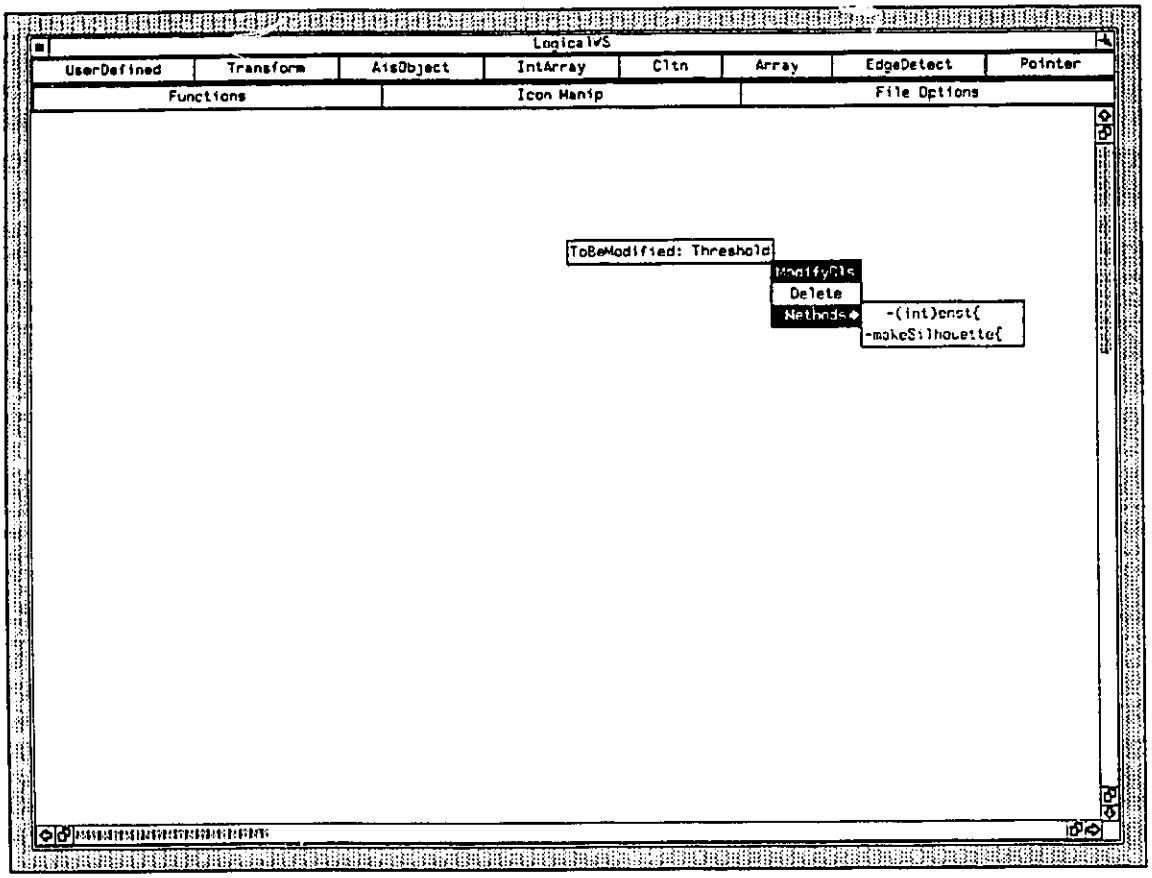

Figure 4.19: The ModifyCls Class

ifyCls" to "ToBeModified: Name", where Name is the name of the class to be modified.

If the name is recognized as valid (the file "Name.m" exists) the PopUpMenu generated by pressing the right button of the mouse contains the options: "Delete" and "Methods".

The selection of an option from the AdHocMenu (i.e. the name of a method implemented in the class being modified) triggers the deletion of the method from the source file and also from ObjectBaseFile (the text file containing information about the objects hierarchy that can be used by the ACG).

After this action there is a discrepancy between the structure of the ObjectBaseFile and the information contained in **objectDataBase** , which should reflect the structure of the ObjectBaseFile.

The consistency is maintained by selecting the option "Refresh" from the "Application File" sub-menu. This selection triggers the re-initialization of the **objectDataBase**. The old instance of the **objectDataBase** class is deleted and a new one is created according to the current information in the ObjectBaseFile.

4.3.12 The ObjectDataBase Class

The ObjectDataBase class is implemented in the files: ObjectDataBase.h and ObjectDataBase.m. The initialization of the **objectDataBase** class instance is done by scanning the ObjectBaseFile. A fragment of the ObjectBaseFile is presented in Appendix A and the structure of the tree-like instance variables of **objectDataBase** are presented in Figure 4.20. The instance variable `nameOfObject` is the root of an ordered collection that contains the names of the classes in the AIS hierarchy (standard classes and user defined classes).

The names of the classes are read from the ObjectBaseFile (the lines with the "#" symbol at the beginning) and added one by one to the collection. The relative position of a class name is called "offset" and is an important parameter used to preserve the consistency of the information stored in the instance variables of **objectDataBase** instance object.

The instance variable `nameOfSuperClass` is the root of an ordered collection that contains the names of the super-classes for the classes whose id-s are in `nameOfObject`. The names of the super classes are read from the ObjectBaseFile (the lines with the "$" symbol at the beginning) and are added to the collection in the same order as the names of the objects (the offset for the name of a class and for the name of its superclass has to be the same).

The `nameOfMethod` instance variable the root of an ordered collection that stores the names of the methods corresponding to a specified class. Because a class usually has more than one method it is not possible to keep all this information in a simple ordered collection. Each element of the `nameOfMethod` is itself the root of an ordered collection containing the names of the implemented methods. The names of the methods are read from the

ObjectBaseFile (the lines with the "+" or "-" symbol at the beginning).

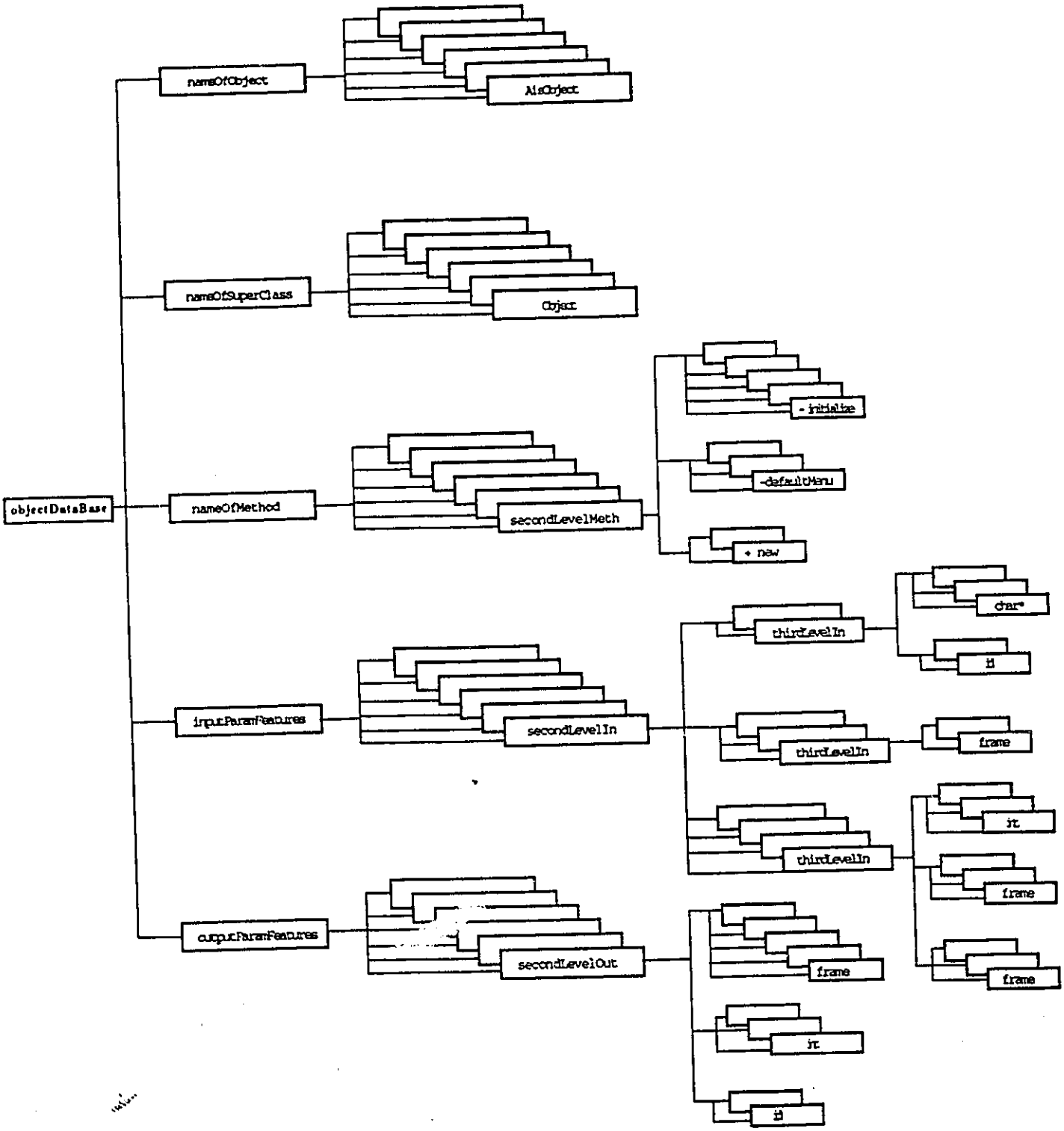

Figure 4.20: The Instance Variables of the objectDataBase

For example, the second element in the ordered collection `nameOfMethod` is the root of an

ordered collection for the class with the name stored in the second element of `nameOfObject`.

The instance variable `secondLevelMeth` is used to implement the second level of ordered collection (see Figure 4.20).

A method can take none or more input parameters. The names of the methods are stored in an ordered collection of ordered collections, a two-level tree-like structure. The types of the input parameters for a methods are stored in a three-level tree-like structure of ordered collections.

As presented in Figure 4.20 the `inputParamFeatures` is an ordered collection in which each element corresponds to a class name and is also an ordered collection (implemented by the instance variable `secondLevelIn`) initialized in. Each element of the `secondLevelIn` ordered collection corresponds to a method implemented in the respective class and is also an ordered collection (implemented by the instance variable `thirdLevelIn`). Each element of the `thirdLevelIn` ordered collection corresponds to an input parameter type.

The number of parameters doesn't require a separate instance variable because it can be deduced from the number of elements of the thirdLevelIn ordered collection. The types of the input parameters are read from the ObjectBaseFile (the lines with the "?" symbol at the beginning).

Each method returns only one value; the type of this value is implemented by the instance variable `outputParamFeature` as an ordered collection where each element of the collection corresponds to a class name and is implemented as an ordered collection of character strings. The instance variable that implements the second level of the tree-like structure for `outputParamFeature` is called `secondLevelOut`.

The methods implemented in the ObjectDataBase class are used either to modify the content of the ordered collection contained in the instance variables or to get information about specific classes, methods, input parameter types, etc.

Some methods implemented in the ObjectDataBase class are:

```
-forObject:(char*) anObject addMethod:(char*) aMethod
```

used to add the name of a method when the name of its class is known;

```
-forObject:(char*) anObject forMethod:(char*) aMethod
```

```
addInpPar:(char*) aPar
```

used to add the type of an input parameter when the name of the class (anObject) and the name of the method (aMethod) are known;

```
-forObject:(char*) anObject forMethod:(char*) aMethod
                           addOutput:(char*) anOut
```

used to add the type of an output parameter when the name of the class (anObject) and the name of the method (aMethod) are known;

```
-(char*)forObject:(char*)
                  anObject returnOutForMethod:(char*) aMethod
```

used to get the type of an output parameter when the name of the class (anObject) and the name of the method (aMethod) are known;

```
-(char*) nameObject:  (char*) aString
                      positionMethod:  (int) aNumber
```

used to get the name of a method when the name of the class (anObject) and the offset of the desired method are known;

```
-(int) numberMethods:  (char*) aString
```

used to get the number of methods for a class when the name of the class (anObject) is known;

```
-(int) numInputsFor:(char*) aObject nameMethod:  (char*) aMethod
```

used to get the number of input parameters of a method for a class when the name of the class (anObject) and the name of the method (aMethod) are known;

```
-(char*) nameObject:(char*) aObject nameMethod:  (char*) aMethod
                                    inputParameter:  (int) position
```

used to get the type of an input parameter of a method for a class when the name of the class (anObject), the name of the method (aMethod) and the offset of the input parameter are known.

The method `-saveStructure` is used by **logicalSpace** when the option "Save Objects" is selected by the user from the "Application File" sub-menu at the end of one session with the ACG.

It allows the user to save all the modification done to **objectDataBase**. The `-saveStructure` method scans the **objectDataBase**'s instance variables and according to the information extracted creates a text file called ObjectBaseFile. If the user has not created any new classes then the ObjectBaseFile won't be changed. Usually the size of the ObjectBaseFile increases, because it contains information about the classes created in the current session.

4.3.13 The FunctionDataBase Class

The FunctionDataBase class is implemented in the files: FunctionDataBase.h and FunctionDataBase.m. A fragment of the FunctionBaseFile is presented in Appendix B and the structure of the tree-like instance variables of FunctionDataBase is presented in Figure 4.21.

The FunctionDataBase class is instantiated and initialized at the beginning of the ACG session shortly after the initialization of the LogicalWS class.

The instance variable `nameOfFunction` is the root of an ordered collection that contains the names of the functions used by the ACG. The names of the function are read from the FunctionBaseFile (the line with the '#' symbol at the beginning) and added one by one to the collection. The relative position of a function name is called offset and is an important parameter used to preserve the consistency of the information stored in the instance variables of functionDataBase.

The names of the functions are stored in an ordered collection, a one-level tree-like structure. The types of the input parameters for a function are stored in a two-level tree-like structure of ordered collections. The `inputParamFeatures` is an ordered collection in which each element is corresponding to a function name and is implemented as an ordered collection (implemented by the instance variable `secondLevelIn`. The number of parameters doesn't require a separate instance variable because it can be deduced from the number of elements

of the secondLevelIn ordered collection.

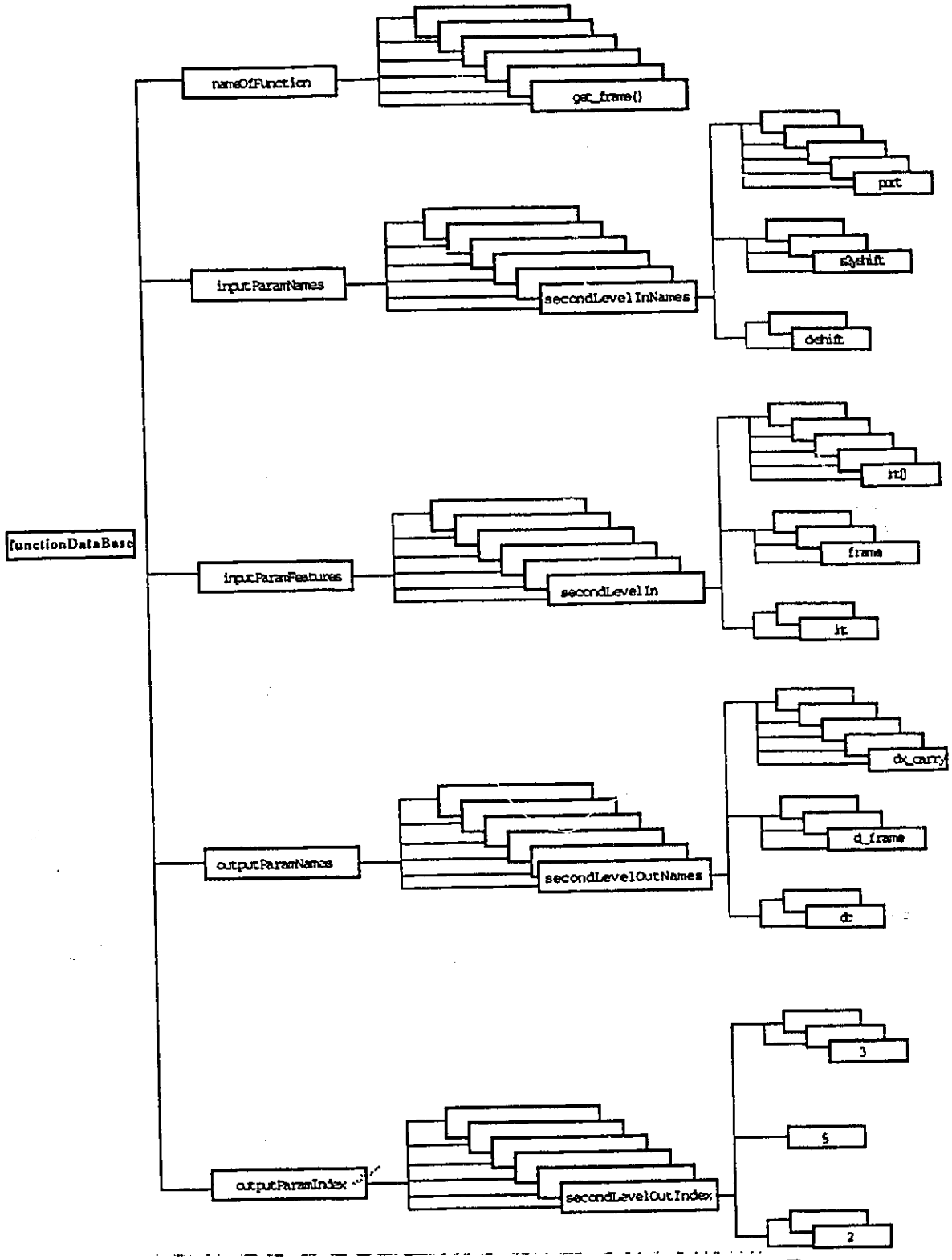

Fig. 4.21. The Instance Variables of the functionDataBase

The types of the input parameters are read from the FunctionBaseFile (the line with the

'$' symbol at the beginning). A similar implementation is used for the names of the input parameters; `inputParamNames` is an ordered collection in which each element is corresponding to a function name and is implemented as an ordered collection implemented by the instance variable `secondLevelInNames`. The names of the input parameters are read from the FunctionBaseFile (the line with the "<" symbol at the beginning). Similar structures are used for the names and types of the output parameters.

An instance variable that has not appeared in the implementation of the ObjectDataBase class is `outputParamIndex`. There are some functions that don't return a value, but instead modify one or more input parameter. Because the number of the input parameters modified by the function can be greater than one it is not possible to implement the position of the input parameters modified as an ordered collection (one-level tree structure); instead it is used a two-level tree-like structure. Each element in the `outputParamIndex` ordered collection corresponds to a function name and is implemented as an ordered collection using `secondLevelOutIndex`. The elements of the second level are id-s corresponding to the positions of the modified input parameters.

The methods implemented in the FunctionDataBase class are used to get information about specific function, input parameter names, types, etc. Some methods implemented in the FunctionDataBase class are:

```
-(char*) findForFunctName:  (char*) aString
                            inputParType:  (int) aNumber
```

used to find the type of an input parameter when the name of its function (aString) and the position of the input parameter are known ;

```
-(char*) findOutTypeForFunct:  (char*) aString
                               indexOut:  (int) aNumber
```

used to get the type of an output parameter when the name of the function (aString) and the position of the input parameter modified (and considered somehow an output parameter) are known.

4.3.14 Other Interface Items for Controlling the Operation Flow in ACG

The GUI provided by the ACG has a number of menu options that allow the user to control the visual programming environment and to initiate or terminate different actions.

The elements used as control options are implemented in the LogicalWS.m file as menu options. A part of this implementation is presented below.

```
[aMenu append:iconManip = [Menu str:  "Icon Manip"]];
        [iconManip append:  [[Menu str:  "Set Root"]
                selector:  @selector(rootFromUser) receiver:  nil]];
        [iconManip append:[[Menu str:  "Identify Root"]
                selector:  @selector(flashRoot) receiver:  nil]];
        [iconManip append:  [[Menu str:  "Clear"]
                selector:  @selector(clear) receiver:  nil]];
        [iconManip append:  [[Menu str:  "Save"]
                selector:  @selector(saveMenu) receiver:  nil]];
        [iconManip append:  [[Menu str:  "Implode"]
                selector:  @selector(implodeMenu) receiver:  nil]];
        [iconManip append:  [[Menu str:  "Retrieve"]
                selector:  @selector(retrieveMenu) receiver:nil]];
```

In the above source code each menu option is implemented as an instance of the class Menu. The method activated by the selection of a menu option is designated by `@selector`. For example, the message "clear" is send to the receiver whenever the option "Save" is chosen from the lower menu-bar.

4.3.15 Conclusions

This chapter presents the building blocks of the ACG visual programming environment. The ACG (an object oriented application itself) is a tool which assists the user in generating/modifying source code for, theoretically, any kind of object oriented application including itself.

The ACG has a modular structure. Each module is implemented as a class, called an ACG-class to make the distinction from the AIS-classes (the classes used by the image processing application).

The behavior of the ACG is dictated by the information about the classes and objects to be used in the visual programming environment. These information are stored in text files called ObjectBaseFile, FunctionBaseFile and FunctionRetBaseFile.

The ACG-classes can be used to implement any kind of object oriented application if the information in these files is changed. In the present example the ACG-classes are used to implement the AIS hierarchy of objects, but they can be used as well to implement any hierarchy of objects.

The basic concept of the object-oriented design and implementation presented in this chapter is the use of the classes already implemented. Choosing the set of messages for the new classes is an important step in the design of the ACG.

The design of the object oriented visual programming environment starts with the classes in the class library that are "closest" to the desired behavior and structure. The new classes are constructed from the old by changing the few things required to adapt them to the application to be designed or by adding the necessary structure and behavior.

Any class is implemented by two source files: the first one has the extension ".h" and is called the interface file; the second one has the extension ".m" and is called the implementation file. Usually (but not necessarily) the two files have the same name as the class name, but the appropriate extensions.

Chapter 5

Programming with the Automatic Code Generator a Remote Sensing Application

In this chapter the ACG assists the user in the process of building an object-oriented application for multi-sensor data visualization.

The aim of this research, as stated in the first chapter, is to create a programming environment, a tool, that assists the user in creating object oriented applications. An example has been implemented and demonstrated in the field of multi-sensor data visualization. The number and type of objects and functions used in the ACG programming environment is given by the information stored in the text files: ObjectBaseFile, FunctionBaseFile and FunctionRetBaseFile. If the information in these files is modified the whole programming environment changes.

Therefore, depending on the functions and objects described in the data base files mentioned above, the ACG can be used in various fields of software development where libraries of C and Objective C, C++ or other object oriented environment are extensively used for developing applications. Examples of other domains might easily be given from the data-base domain, expert systems, etc.

The methodology of constructing object oriented applications by means of graphic ma-

nipulation remains the same but the building blocks can be changed according to the application's domain.

5.1 AISI Vision Computers

The AIS-3500 (from the AISI family of machine vision computers) is a massively parallel, Single-Instruction, Multiple-Data (SIMD) linear array computer. The parallel processor in the AISI computers is not oriented toward any particular type of image processing or vision algorithm or class of algorithms but, instead, is programmed to perform a wide variety of vision operations at very high speeds. Operations such as convolutions, filtering, morphology, feature extraction, and many others are performed by the general purpose parallel processor.

The AIS-3500 vision computer is a complete system which, in addition to the parallel processor, contains a Motorola 68000 host CPU, and video I/O sub-system in compact, self-contained, cost effective configurations suitable for industrial environments. Serial and digital I/O as well as SCSI peripheral interfaces are supported.

When used for development purposes, the AIS-3500 computer is connected to a Sun workstation with either an RS-232/ RS-422 serial communications line or through an Ethernet connection. Other devices, such as a mouse, industrial terminal etc., can also be connected to the AIS-3500 computer.

The applications for the parallel computer are developed on a Sun workstation. The Sun software includes:

- the UNIX operating system and utilities
- the Sun system C compiler
- the SunTools system software, including SunView and SunWindows
- the Intermetrics-C Cross Development tool. This tools is a C compiler and linker that produces machine code that runs on the AIS-3500 parallel processors

The AISI System Software is the programming environment used to implement an image processing application that runs on the parallel machine and consists of:

- the AIS parallel processor operating system kernel
- the AIS parallel processor system libraries
- the AIS parallel processor shell
- the workstation interface programs

There are two programming environments, both of them object oriented, one running on the Sun workstation (called the ACG environment) and the other running on the AIS parallel processor (called the AIS environment). There are classes with the same name but with different features in the two environments, for example Layer. The ACG classes are used to implement the elements of the AIS programming environment.

5.2 Building an AIS Application

An object-oriented learning system responsible for automated road detection in aerial images was studied and implemented in the image processing laboratory at the University of Ottawa. The automatic road extraction is accomplished through the help of various tools responsible for processing the aerial image in an interactive fashion. Each tool or module takes the image from the previous step and performs a transformation. The output is then passed on to the next stage. Thus, imagery available from the Canadian Center for Remote Sensing (CCRS) in Ottawa is processed for the purpose of road network extraction.

This thesis demonstrates that a system for automatic road extraction can be implemented more efficiently by means of graphical manipulation.

The process of developing an application on the AIS-3500 computer consists of:

- phase 1: the user creates the Objective-C source code for the image processing application on a Sun workstation.
- phase 2: the Intermetrics-C Cross Development Tools are used to generate the machine code that runs on the AIS parallel processors
- phase 3: the file containing the machine code is down-loaded into the parallel machine and executed.

The ACG is used in this chapter for phase 1 of the development process.

The architecture of the Automated Road Extraction System comprises the following building blocks:

- Preprocessing
- Processing
- Knowledge extraction
- Feature identification using apriori information

The aerial images are stored in an image database. This database contains windows (512 x 512 pixels) of the aerial image. Image processing is responsible for organizing the information available in the aerial images in a meaningful way such that the interpretation step can identify and correctly extract the significant features. The processing step consists of two objects: Calculate and Filter.

The Calculate object is responsible for combining different channels of the same image. The Landsat TM data is available in seven channels. Each channel contains information specific to a particular bandwidth of the visual spectrum. By combining different channels the information available can be used to further strengthen the signature of the elongated features. The Calculate object has three methods:

- `-addChannels:source2:dest:`
- `-subtractChannels:source2:dest:`
- `-multiplyChannels:source2:dest:`

One of the above methods - `-addChannels:source2:dest:` - is implemented with the ACG and the corresponding source code is inserted in the file CalculatePanel.m.

5.3 Action Objects and Interface Objects

One of the most time consuming tasks in developing working machine vision applications that are easy to use and install is creating the interfaces - interfaces to both humans and

machines. In order to simplify this problem AIS has implemented a set of interface objects which enable the user to construct graphical user interfaces or machine interfaces. These objects are presented in Appendix A.

The interface objects provide the user with the tools to organize and control the input to the application. Using the interface objects the user can create applications with interactive, graphic-based interfaces for user input through a pointer device, such as a mouse, and provide immediate feedback to the user of results.

The AREX (Automatic Road Extraction) program contains two different types of objects: interface objects and action objects that provide actions and define the interface. One of the action objects defines the application and it is called application object.

The interface objects communicate with the action objects using methods that read and write parameter values, trigger actions, etc. The action objects are the clients of the interface objects.

In the implementation of AREX, the action objects are: AerialAppl, CalculatePanel, ColorPanel, CriteriaPanel, FeaturePanel, FilePanel, FilterPanel, InterfacePanel, MapPanel, SegmentPanel, TestPanel.

AerialAppl is the application object. When an application is simple, all the actions can be grouped into one single application object. For more complex applications, such as AREX, the actions are divided into separate independent action objects.

The interface objects connect to the action objects via a client/method relationship just like they connect to the application object. The action objects become the clients of the interface objects.

The application object, in this case AerialAppl, creates all the interface objects. In addition, it creates instances for all the action objects and provides the collection of methods that form the primary behavior of the application.

5.4 Implementation of AREX: A Program for Extracting Road Networks From Aerial Images

The source code implementing the AREX application is contained in 12 files: AerialAppl.m, CalculatePanel.m, ColorPanel.m, CriteriaPanel.m, FeaturePanel.m, FilePanel.m, FilterPanel.m, InterfacePanel.m, MapPanel.m, SegmentPanel.m and TestPanel.m. The file implemented using the ACG is CalculatePanel.m.

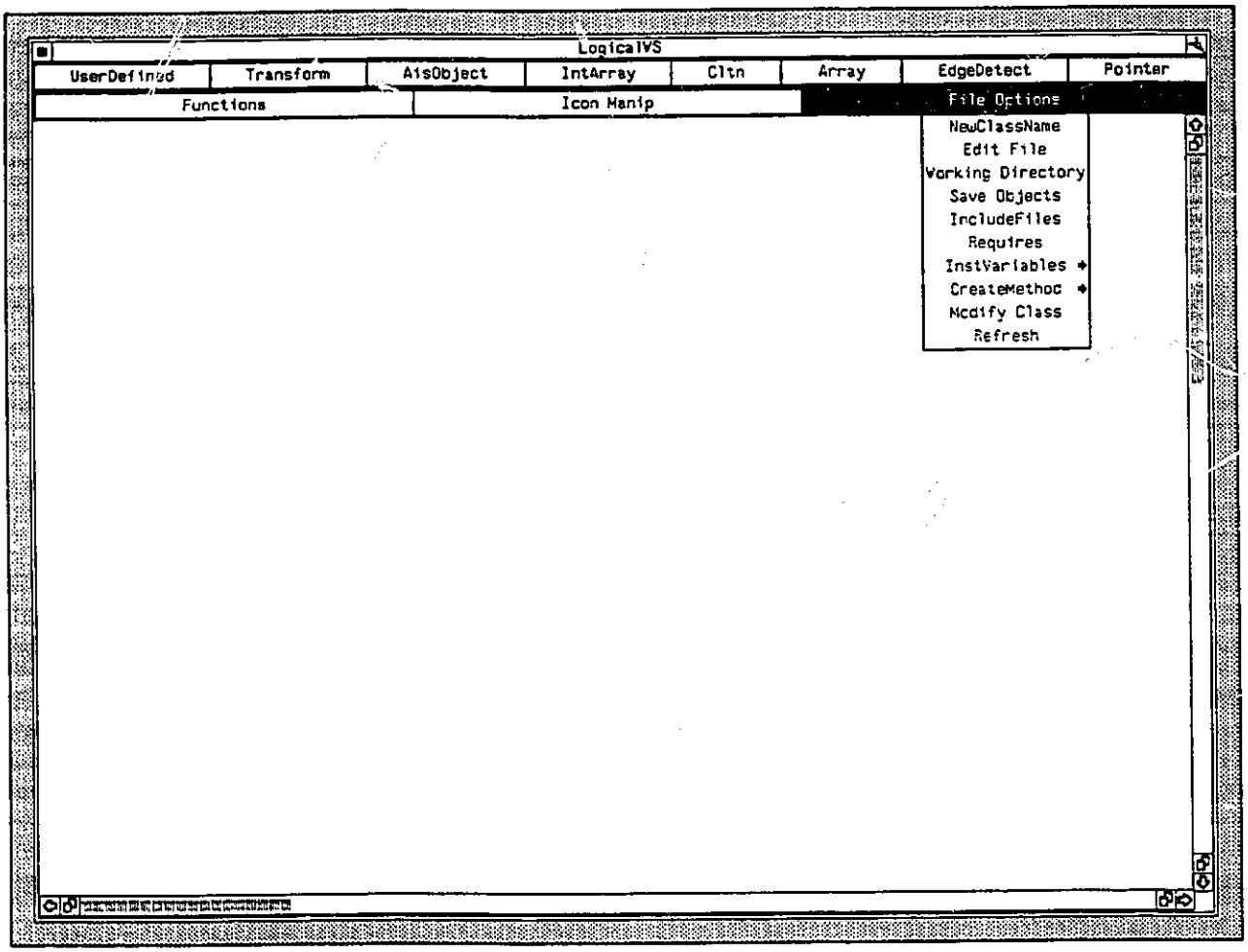

Figure 5.1: The Options in the "File Options" sub-menu

The option "NewClassName" in the "Control Options" sub-menu is used to start the implementation of a new class. A number of three dialog boxes are displayed consecutively on the screen requesting information from the user about:

- the name of the class to be implemented (in this example the name is "CalculatePanel")
- the name of the super class (in this example the super class is "ControlPanel")
- the name of the message groups (in this example they are "Aerial, Interface, Vision and Primitive")

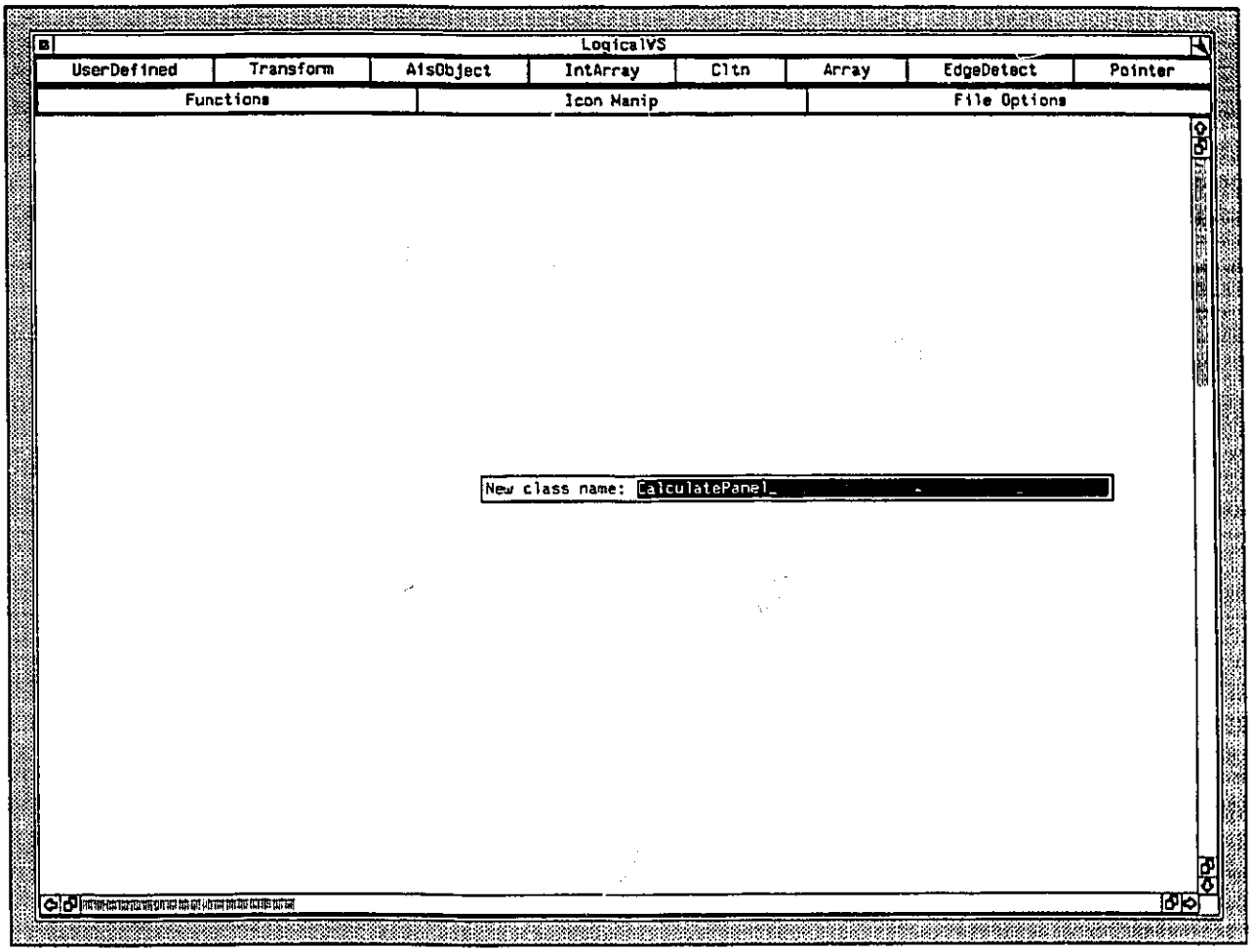

Figure 5.2: The Implementation of a New Class

The ACG creates a text file whose name is identical with the name of the class being implemented and whose extension is ".m" (in this example the name of the file is "CalculatePanel.m").

```
#include <aistypes.h>
#include <objc.h>
```

```
#include <condition.h>
//reqRef
//extRef
//applRef
=CalculatePanel:  AisObject(Aerial, Interface, Vision, Primitive)
{
//varRef
}
//methRef
```

The header files included are always used by the image processing applications. The commented lines (starting with //) are used as reference points for the inserting operations performed by the ACG. For example, the source code corresponding to a new method is always inserted after the line `//methRef`.

An instance variable is created by selecting one of the alternatives implemented by the "InstanceVariables" option in the "Control Options" sub-menu.

To implement an instance variable with the type "frame" the user selects the "frame" option (see Figure 5.3) and introduces the name of the instance variable at the request of the dialog box. A number of six instance variables of different types are created using this menu option and the corresponding source code is inserted after the line `//varRef`. . The file "CalculatePanel.m" is modified accordingly.

```
#include <aistypes.h>
#include <objc.h>
#include <condition.h>
//reqRef
//extRef
//applRef
=CalculatePanel:  AisObject(Aerial, Interface, Vision, Primitive)
{
//varRef
```

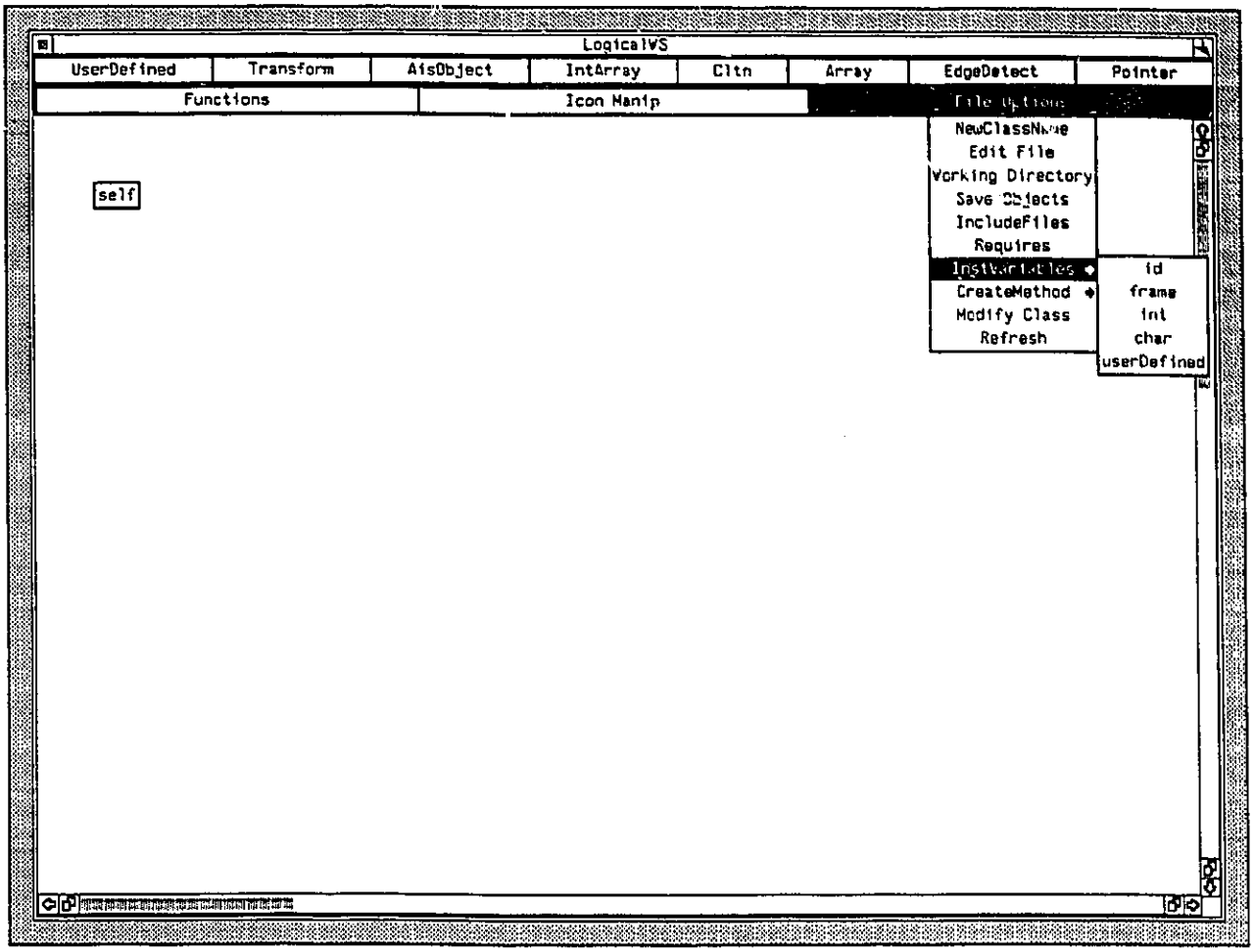

Figure 5.3: The Creation of an Instance Variable

```
frame chan0;
frame chan1;
frame chan2;
int channelOption1;
int DYofset;
id calculChoice;
}
//methRef
```

There is an important aspect that the user has to be aware of before starting the implementation of any method. The icons of MessageCls class are the visual representations

of message expressions. A message expression is formed with a receiver and a selector. The receiver is either the name of a class or the identifier corresponding to an instance of a class. The selector is the name of a method implemented by the receiver. If the receiver is a standard class or a class previously implemented by the user with the ACG then the creation of a message-expression is straightforward.

The receiver in a message expression can be "self"; that means that the selector is the name of a method implemented either in the class currently being defined or in one of its parents.

In order to create a message expression for "self", the user needs in the work-space an instance of the class being defined, in this example "CalculatePanel". The "NewObject" option from the "UserDefined" sub-menu allows the user to access the methods implemented by the class Calculate Panel (see 4.2.2 and Fig. 5.4).

The methods to which an instance of the CalculatePanel class reacts are inherited from its parent because no method has been implemented locally with the ACG. The factory method `+willBeOn:` (see final source code for CalculatePanel class in Appendix E) contains the message expression: `[self calChoice: -1];` - where `calChoice:` is the name of an instance method implemented also in the class CalculatePanel.

In order to be able to implement the method `+willBeOn:` the user needs to access the method `-calChoice:`. Hence, the order of implementing the methods with the ACG is not random. In this case the method `-calChoice:` has to be implemented before the method `+willBeOn:`.

5.4.1 The source code generation for the method `-calChoice:`

To create the method `-calChoice:` the user selects the option "StartCreateMethod" from the "Create Method" sub-menu in "Control Option". The name of a method begins with a "+" or with a "-" sign, otherwise the ACG cancels the method creation and requests a new name from the user.

The method `-calChoice:` takes one formal parameter; the type of this formal parameter is requested by the ACG trough a dialog box. The type of the formal parameter is preceded by a ">" or a "<" sign. Generally, an input parameter is either given by the user in a textual

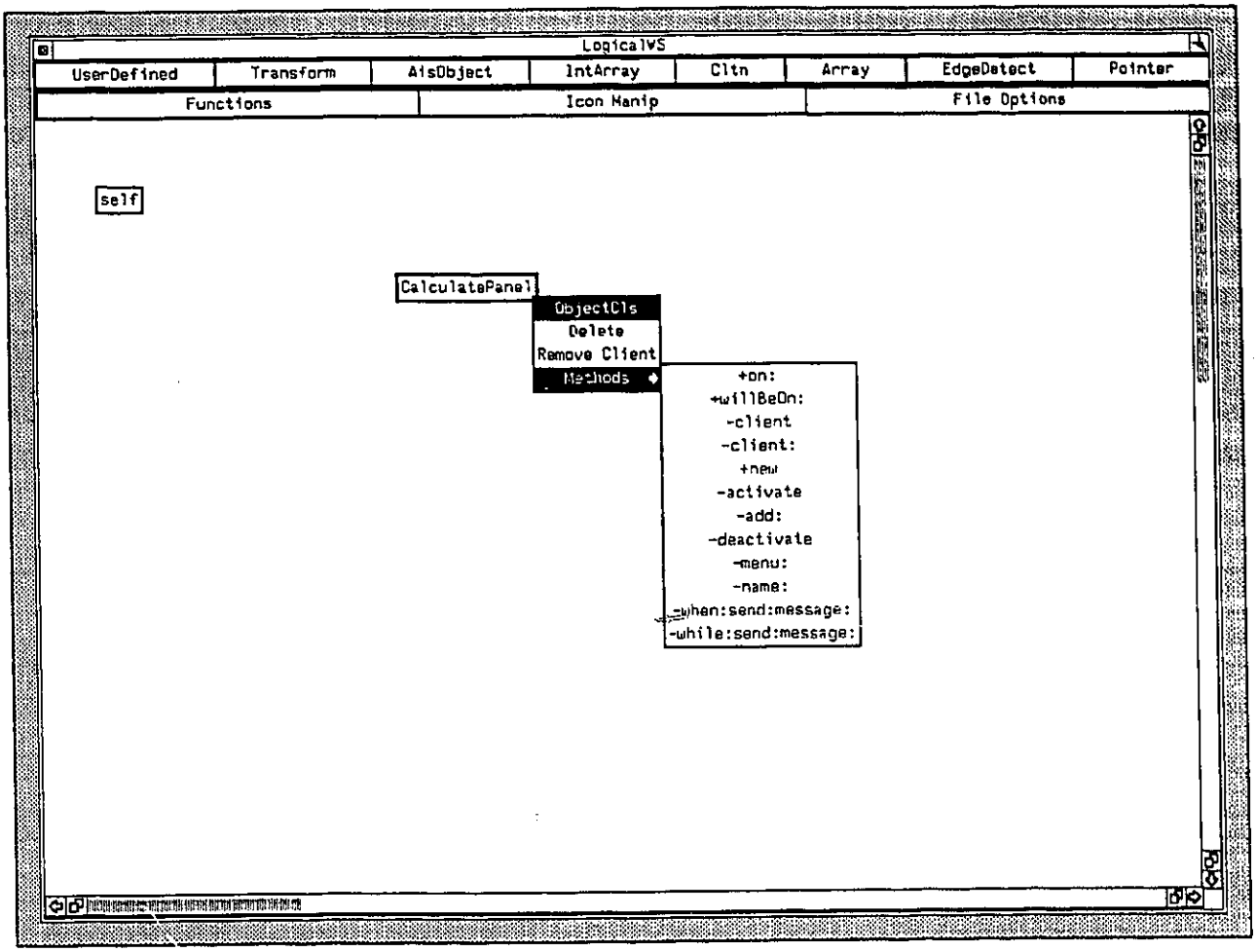

Figure 5.4: The Methods Implemented by the CalculatePanel Class

form (in this case the type of the parameter should begin with a "<") or can be deduced in the parsing process from a graphic connection (in this case the type of the parameter should begin with a ">").

In the present example, the name of the input parameter is "cC" and the type is "<int". The method returns "self" unless otherwise specified.

The method `-calChoice:` contains only two "graphic statements" (see Fig. 5.5).

The first one corresponds to an assign operation. On the right side of the Assign function there is the identifier "calChoice" previously defined as an instance variable of the class CalculatePanel. The "Assign" icon is the visual representation of an instance of the FunctionRetCls class.

On the left hand side of Assign is the icon corresponding to an instance of the Func-

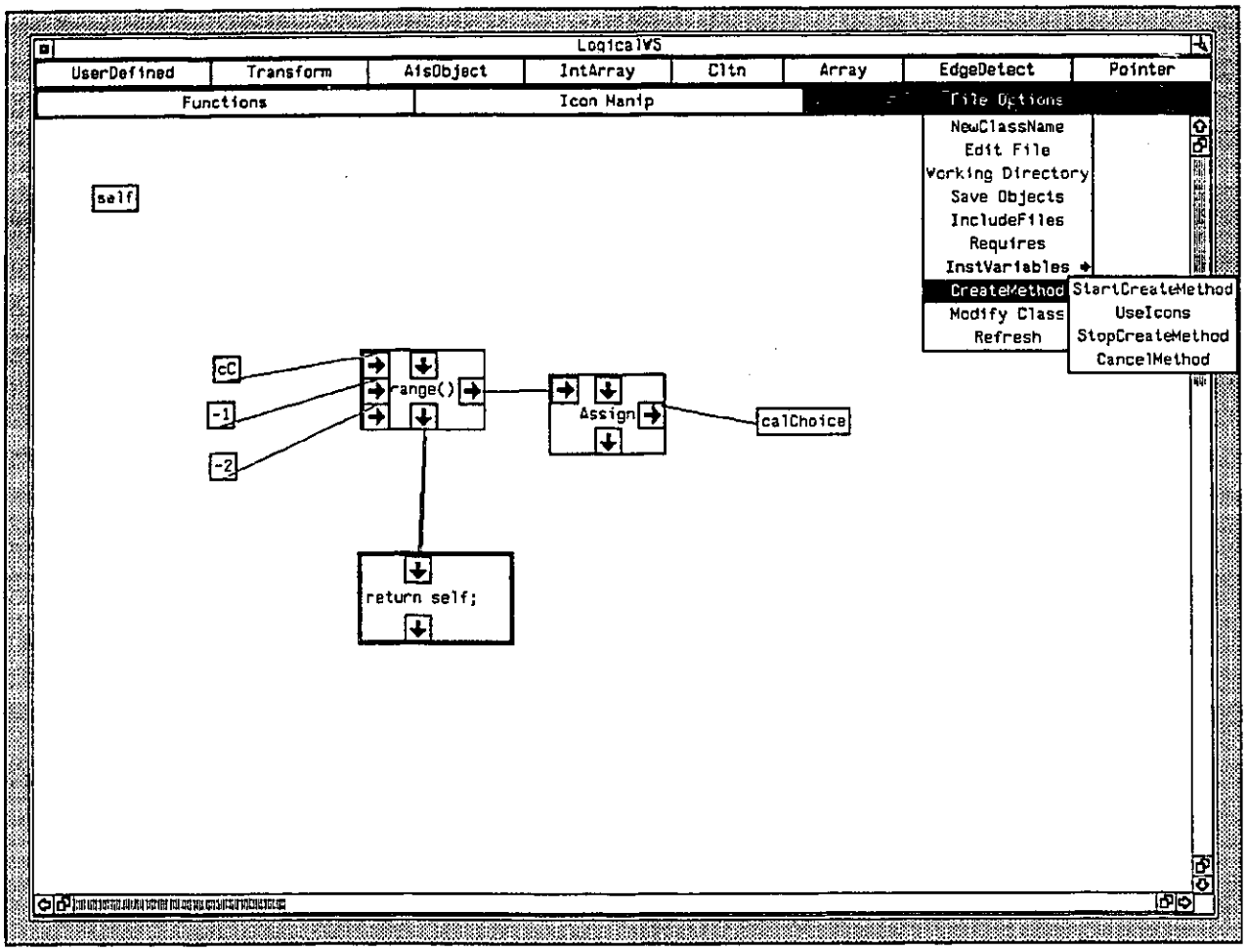

Figure 5.5: The "StartCreateMethod" menu option

tionCls class called "range()". This function takes three input variables one of them being the formal parameter previously introduced, namely "cC". The three inputs, the icons "cC", "-1", "-2" are instances of the VariableCls. The types of the three input variables is automatically assigned by the ACG.

This graphical representation suggests that the value returned by `range(cC,-1,-2)` is assigned to the variable `calChoice`. The method returns the "self" **id**. The return statement is implemented by an instance of the InstructionCls class.

The to-arrow of the Assign function is connected to the from-arrow of the InstructionCls instance. The `return self;` instruction is created by selecting the option "MakeInstruction" from the PopUpMenu generated by pressing the right button of the mouse inside the icon of the InstructionCls instance object.

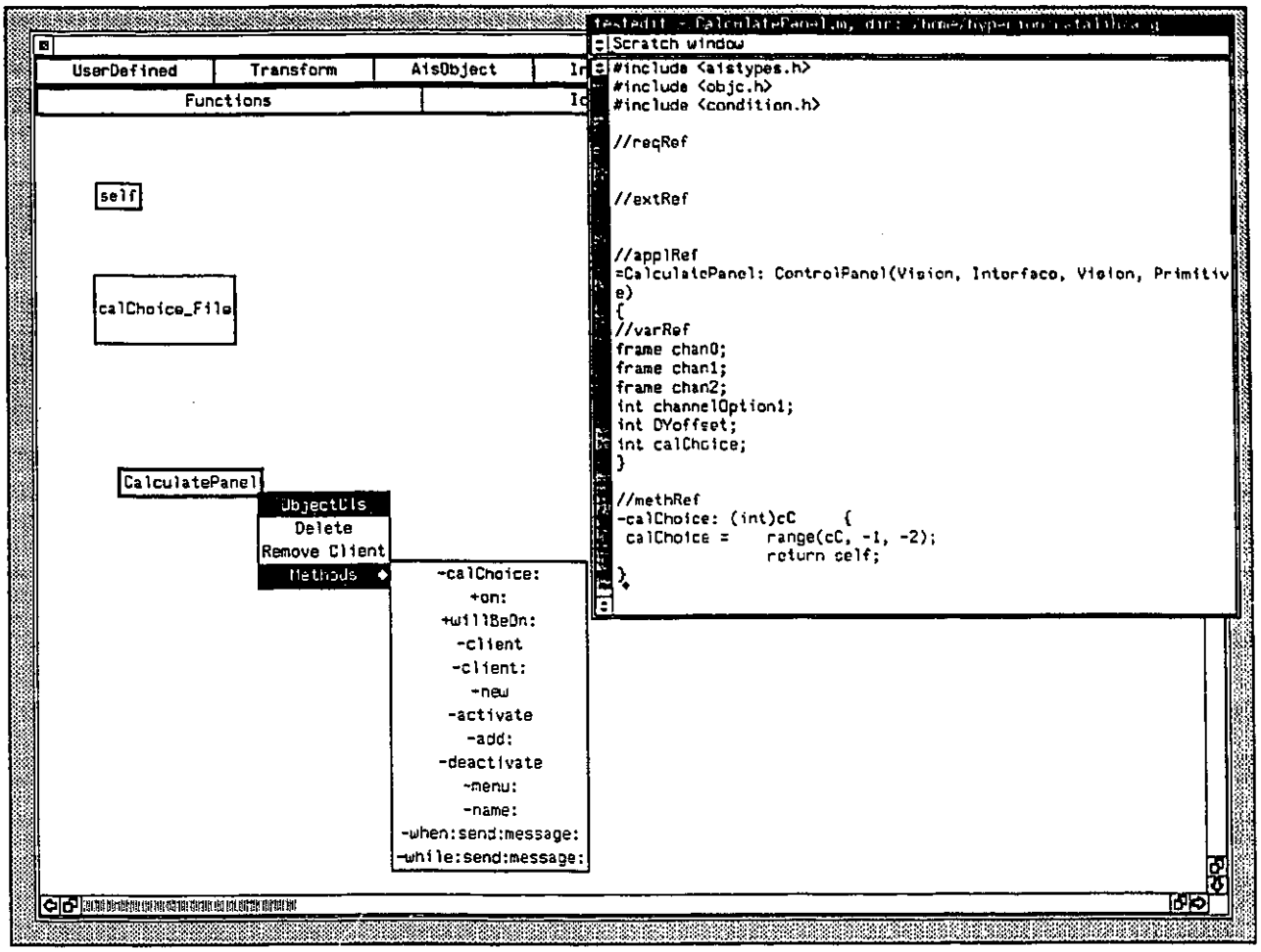

Figure 5.6: The Graphical Structure of the Method `-calChoice:`

The parsing of the graphical structure built by the user starts from an icon called "root"selected by the user. In this example the selected root is the Assign icon.

The source code generation is triggered by the selection of the option "Use Icons" in the "Create Method" sub-menu (see Fig 5.5). The corresponding source code is presented in the textedit-window (see Figure 5.6).

The temporary file called **newMethod** contains the source code corresponding to the icons on the screen:

```
calChoice = range(cC,-1,-2);
return self;
```

The last command in the implementation of a new method is "StopMethodCreation".

The newMethod file is inserted in the CalculatePanel.m file after the line //refMet:

```
#include <aistypes.h>
#include <objc.h>
#include <condition.h>
//reqRef
//extRef
//applRef
=CalculatePanel:  AisObject(Aerial, Interface, Vision, Primitive)
{
//varRef
frame chan0;
frame chan1;
frame chan2;
int channelOption1;
int DYofset;
id calculChoice;
}
-calChoice:  (int) cC
{
calChoice = range(cC,-1,-2);
return self;
};
//methRef
```

The number of methods implemented by the class CalculatePanel has increased with one (see Figure 5.4 and Figure 5.6).

5.4.2 The source code generation for the method `addChannels: source2: dest:`

The graphical structure for the first part of the method `addChannels: source2: dest:` is presented in Figure 5.8.

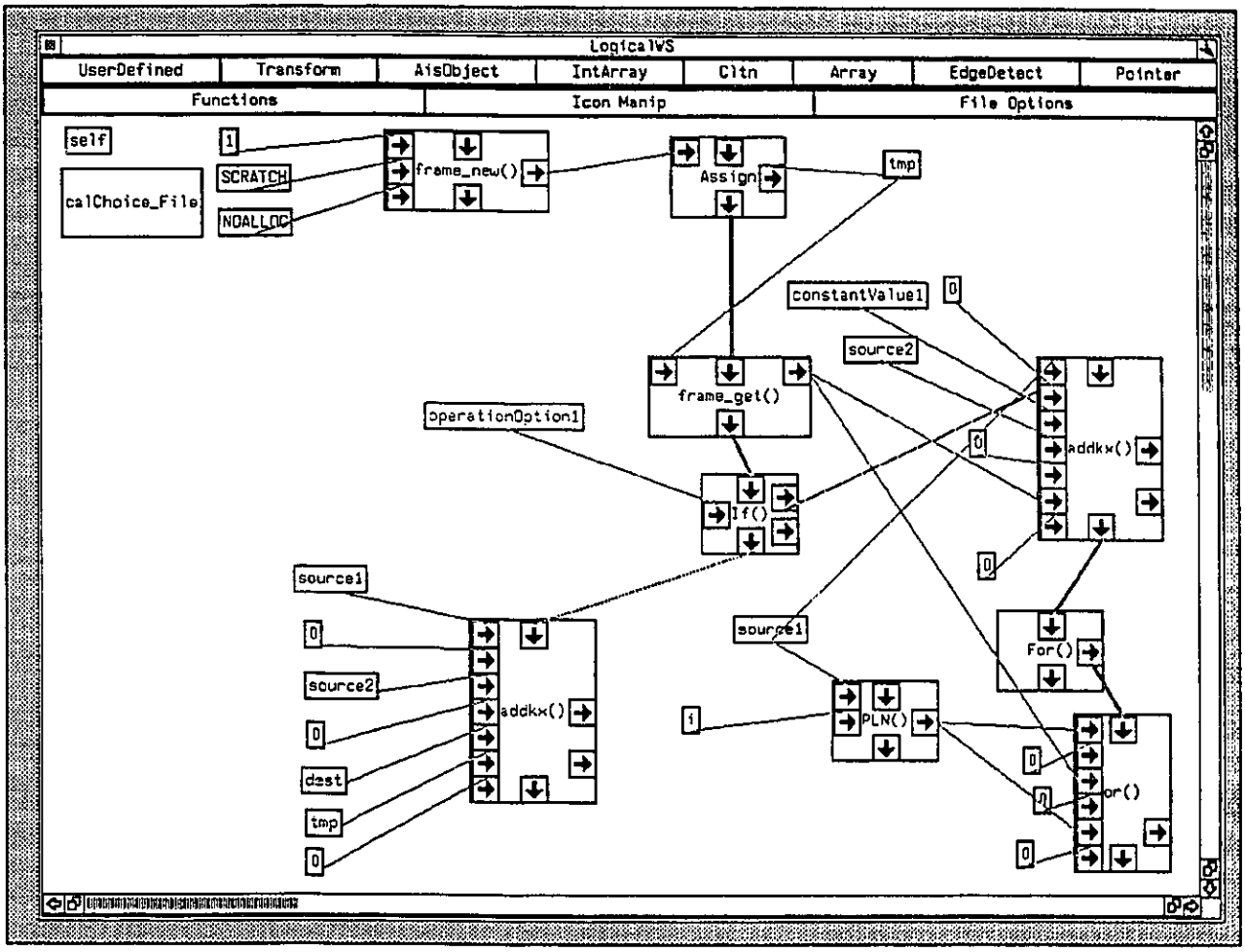

Figure 5.7: The Method `addChannels: source2: dest:`

The ACG analyzes the name given by the user for the method to be implemented; the number of input parameters is given by the number of double colons contained in the name of the method; for the method `addChannels: source2: dest:` there are three input parameters.

Because the method is large it is not possible to display the whole graphical structure of the method on the screen. The first part of the graphical structure (one screen) is collapsed

in a box (an instance of the ImplodeCls class), named in this example "addChanelFirst".

This instance of the ImplodeCls class is connected through a control connection with the next icon in the graphical structure of the method, allowing large methods to be created in successive steps (see Fig. 5.9).

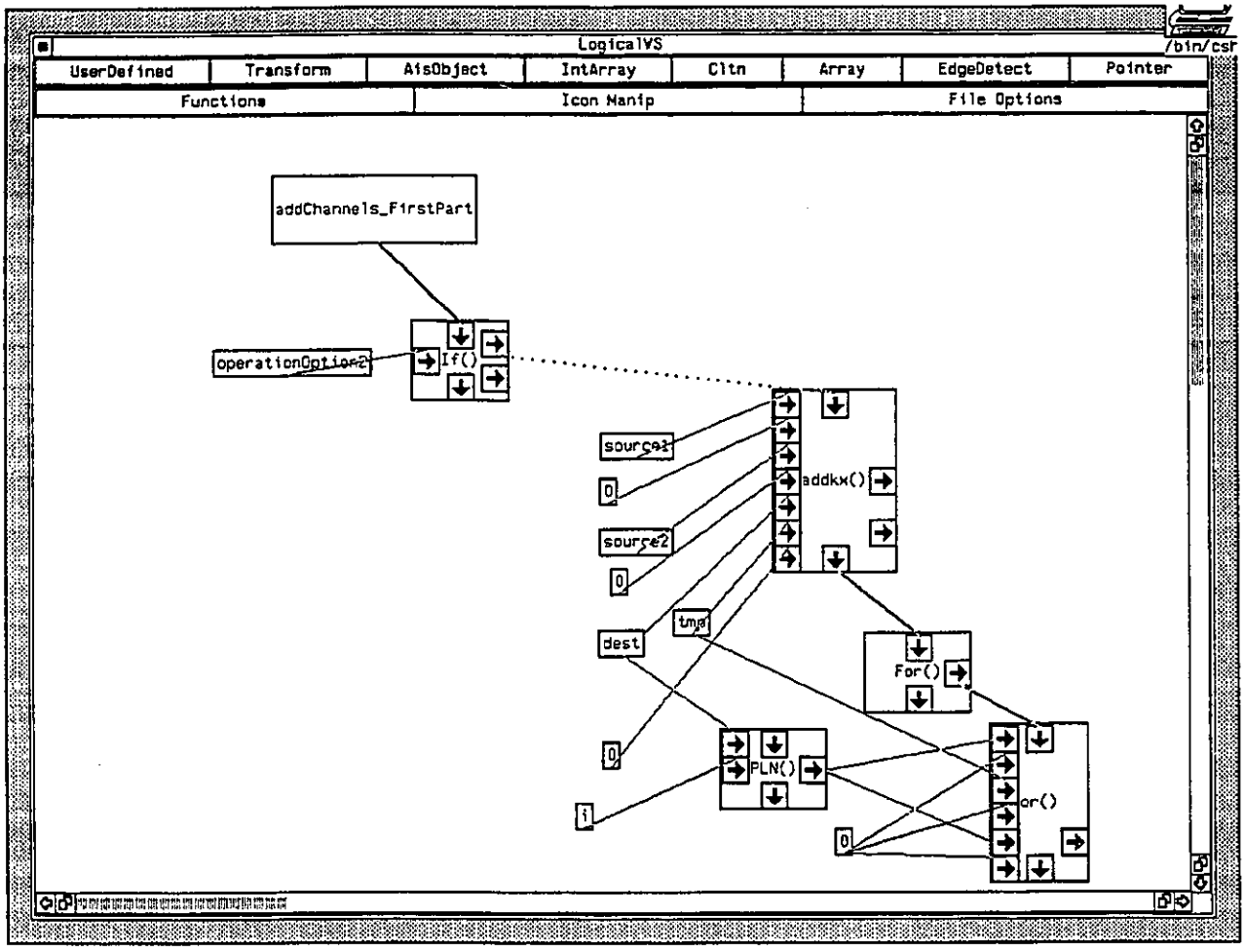

Figure 5.8: The Use of ImplodeCls Class in Methods Implementation

After the creation of the graphical structure is finished the root icon is selected and the parsing process is started.

The first statement is an assign operation. The variable "tmp" takes the value returned by the vision function "frame_new()". During the parsing process a confirmer box is displayed on the screen requesting information from the user about the variable "tmp" which was not declared as an instance variable and is considered by the ACG as being locally defined

The code source corresponding to the graphical structure is generated and inserted in

the file CalculatePanel.m.

```
#include <aistypes.h>
#include <objc.h>
#include <condition.h>
//reqRef
//extRef
//applRef
=CalculatePanel:  AisObject(Aerial, Interface, Vision, Primitive)
{
//varRef
frame chan0;
frame chan1;
frame chan2;
int channelOption1;
int DYofset;
id calculChoice;
}
-addChannels:(frame)source1 source2:(frame) source2 dest:(frame) dest
{
int i;
frame tmp;
            tmp = frame_new(1, SCRATCH, NOALLOC);
            frame_get(tmp);
            if(operationOption1){
                  addkx(source1,0,constanctValue1,source1,0,tmp,0);
                  for(i=0;i<8;i++)
                  or(PLN(source1,i),0,tmp,0,PLN(source1,i),0);
                  }
            else {
                  subkx(source1,0,constantValue1,source1,0,tmp,0);
```

```
                not(tmp,0,tmp,0);
                for(i=0;i<8;i++)
                and(PLN(source1,i),0,tmp,0,PLN(source1,i),0);
                }
        addkx(source1,0,source2,0,dest,0,tmp,0);
                for(i=0;i<8;i++)
                or(PLN(dest,i),0,tmp,0,PLN(dest,i),0);
        if(operationOption2){
                addkx(dest,0,constanctValue2,dest,0,tmp,0);
                for(i=0;i<8;i++)
                or(PLN(dest,i),0,tmp,0,PLN(dest,i),0);
                }
        else {
                subkx(dest,0,constantValue2,dest,0,tmp,0);
                not(tmp,0,tmp,0);
                for(i=0;i<8;i++)
                and(PLN(dest,i),0,tmp,0,PLN(dest,i),0);
                }
        frame_free(tmp);
        return self;
}
-calChoice:  (int) cC
{
calChoice = range(cC,-1,-2);
return self;
};
//methRef
```

The graphical structure of the method is saved in a graphic-file by selecting the option "Save" from the "User Defined" sub-menu.

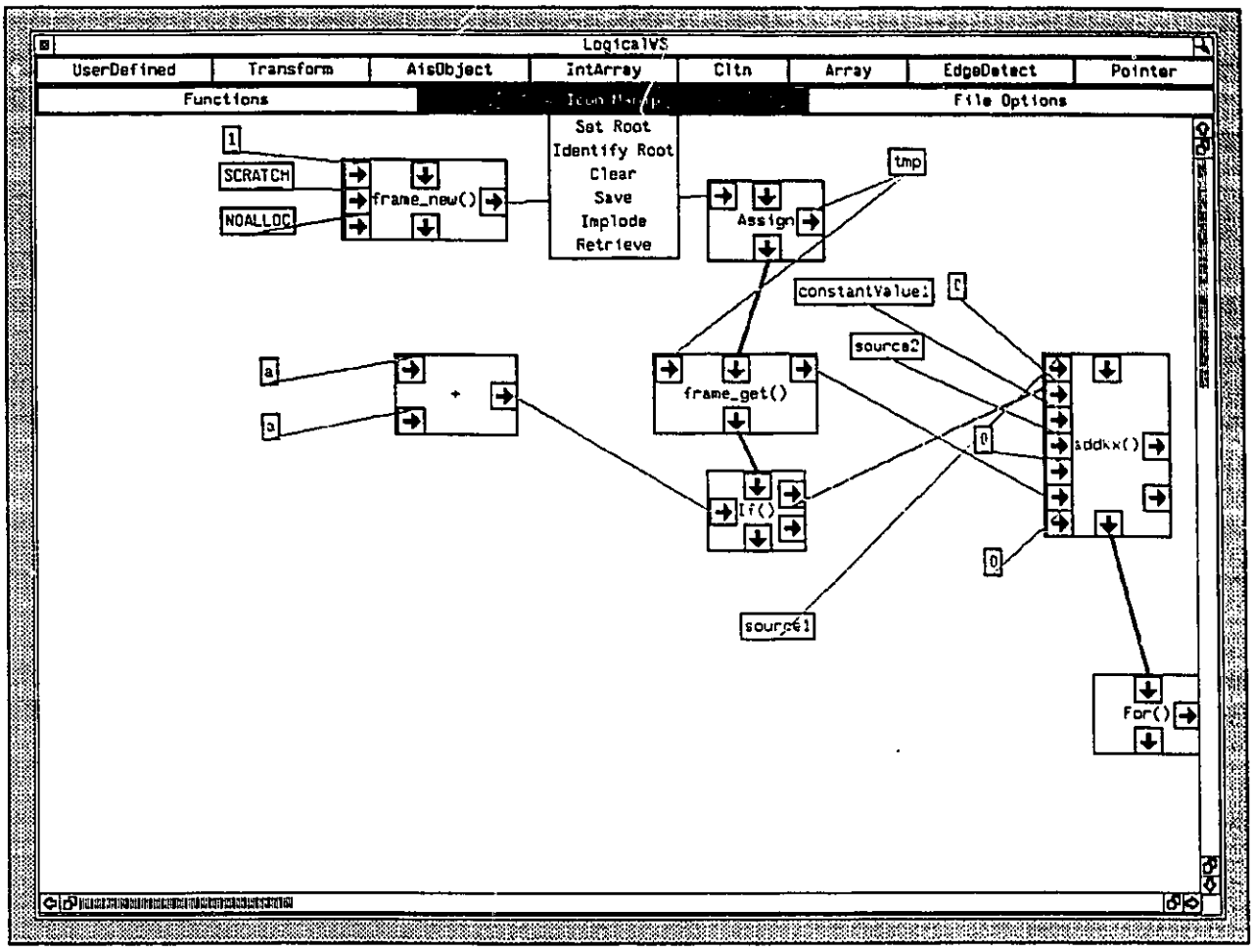

Figure 5.9: The Creation of a "graphic-file"

The ACG requests a name for the graphic-file and stores in it information about the position and the type of the icons that are used to implement the method. A part of the "graphic-file" is presented below.

A graphical structure containing mostly message expressions is implemented in the method `willBeOn:`. The first "graphical statement" is an assign operation. The "self" instance of the VariableCls class takes the value returned by the message expression: `[[super willBeOn: panel] name: ''Calculate''];`.

The message expression `[[super willBeOn: panel] name: ''Calculate''];` is equivalent with `[[ControlPanel willBeOn: panel] name: ''Calculate''];` and is created by a double access of the methods implemented by the ControlPanel class, the parent of CalculatePanel class.

The steps required to create the message expression are:

- create an instance of the ControlPanel class
- create an instance of the MessageCls class
- select the option "StartMessageCreation" from the PopUpMenu of the MessageCls class (see Figure 5.5)
- select the method `+willBeOn:` from the PopUpMenu of the ControlPanel class and enter the input parameter when requested by the ACG (in this case the id "panel")

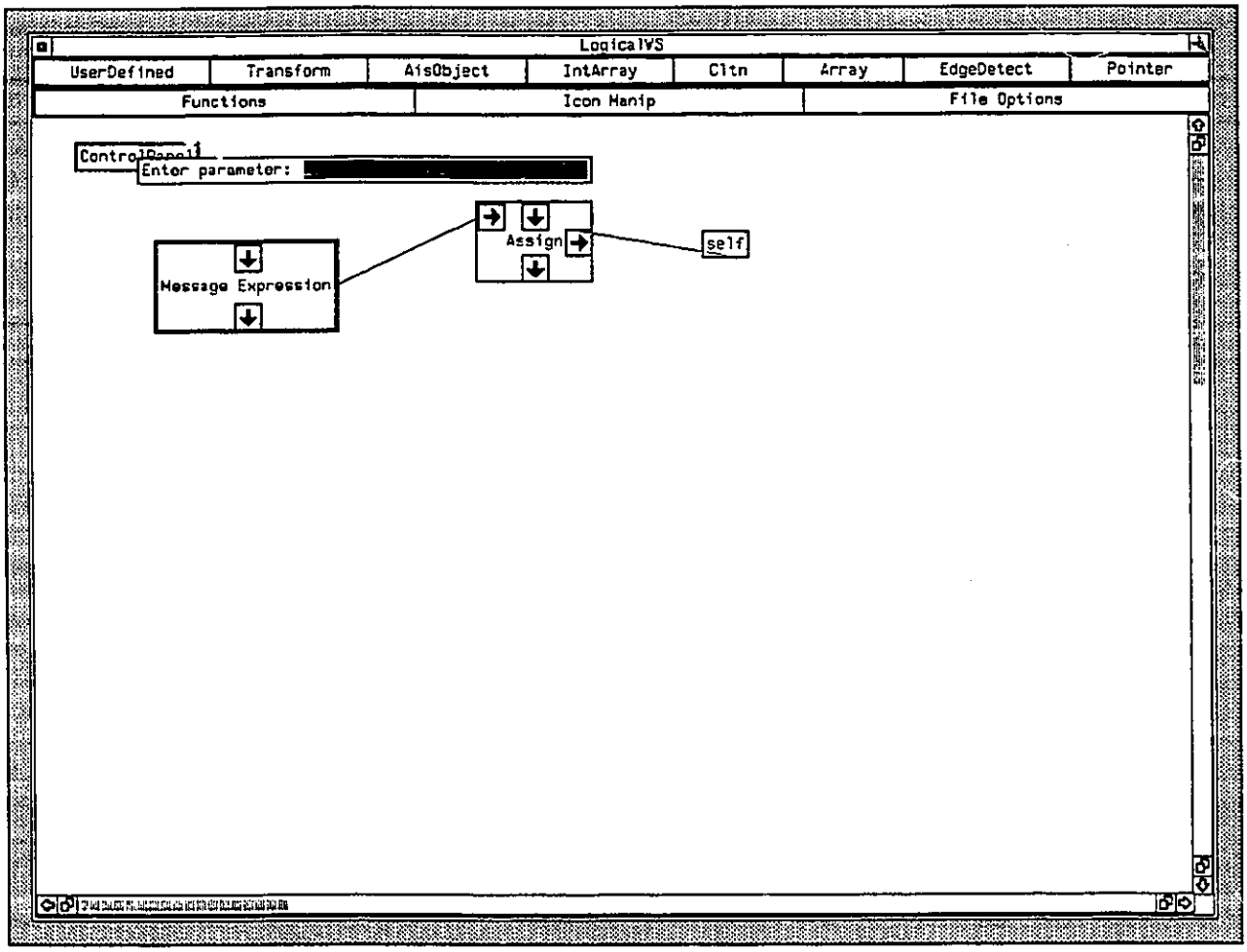

Figure 5.10: Dialog Box Requesting an Input Parameter

- select the option "StopMessageCreation" from the PopUpMenu of the MessageCls class; at this moment the text inside the icon changes from "Message Expression"

to [ControlPanel willBeOn: panel];

The "self" icon and the icon corresponding to the message expression are then connected to the two input-arrows of the Assign function.

The the source code corresponding to the method +willBeOn: is inserted in the file CalculatePanel.m by selecting the option "StopCreateMethod".

The graphical structure corresponding to the method +willBeOn: is saved in a graphic-file called "CalculatePanel_willBeOn" for future reuse or modification.

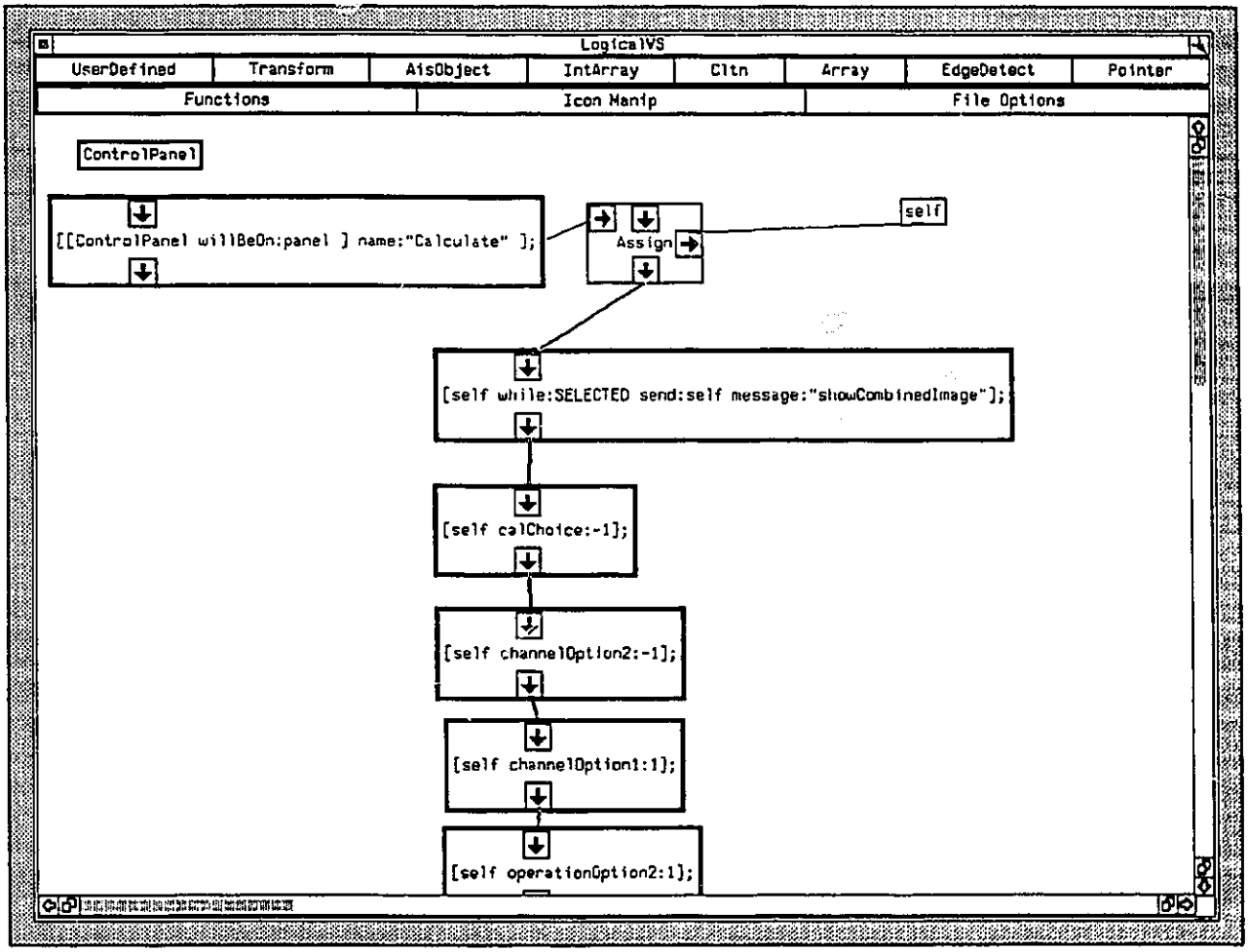

Figure 5.11: The Method willBeOn:

5.5 The Modification of a Class with the ACG

Code reusability allows the user to implement more efficiently object oriented applications for image processing. The graphical structure of a method can be easily retrieved from a graphic-file and can be modified using cut-and-paste operations.

The modification of a method implemented in the class CalculatePanel starts with the creation of an instance of the ModifyCls class (see 4.2.5). The selection of the option "Name" from the PopUpMenu generated with the right button (see Figure 4.18) of the mouse allows the user to access enter the name of the class to be modified: CalculatePanel.m.

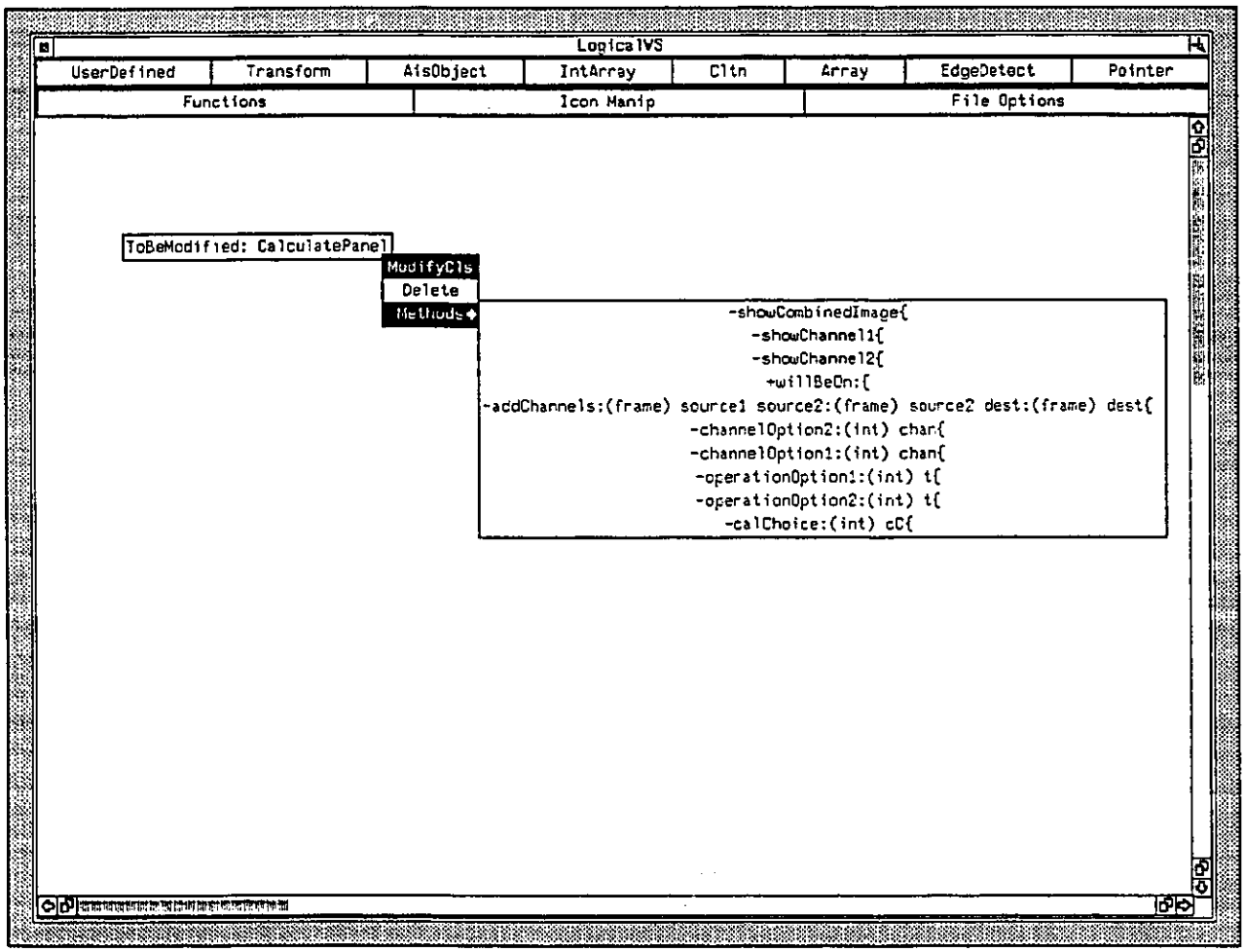

Figure 5.12: The CalculatePanel Class before the Method Deletion

The methods implemented in this file are presented as options in the PopUpMenu generated by pressing the right button of the mouse inside the icon of the ModifyCls class.

The selection of the method `-calChoice:` determines the deletion of the source code corresponding to this method from the file "CalculatePanel.m" and also the disappearance of the method's name from the PopUpMenu.

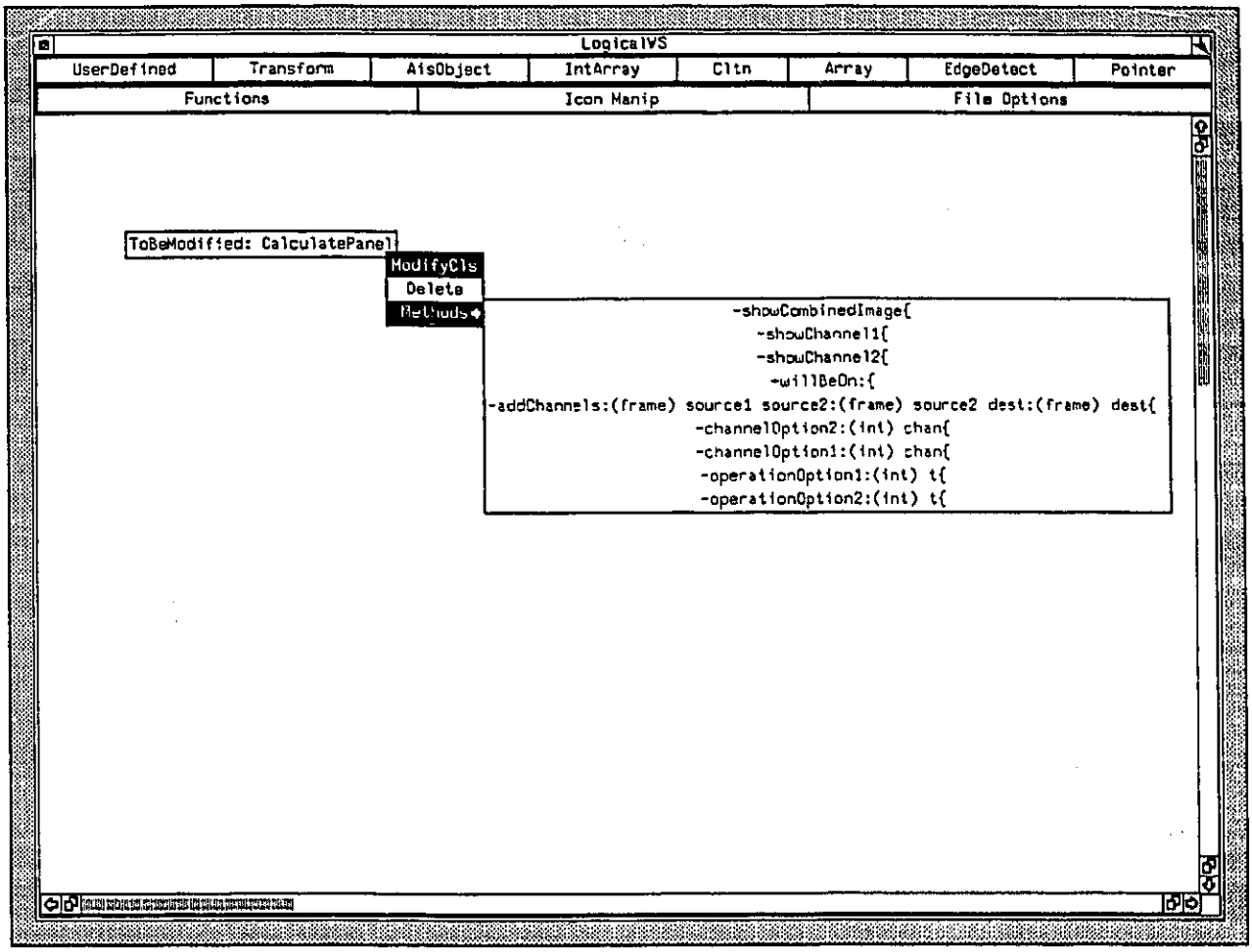

Figure 5.13: The CalculatePanel Class after the Method Deletion

The user creates a new method by retrieving the graphical structure of the old `-calChoice:` method and modifying it. The graphical structure is stored in a graphic-file and is not deleted when the source code corresponding to this method is deleted from the file CalculatePanel.m.

The graphical structure is modified according to the user's needs and the source code corresponding to this graphical structure is inserted in the file CalculatePanel.m by selecting the options: "StartCreateMethod", "Use Icons" and "StopCreateMethod" from the "Control Option" sub-menu. The name of the method need not remain the same.

The ACG is able to modify the structure of any object-oriented applications if the

graphic-files corresponding to its classes exist.

The ACG is able also to modify its own structure. The classes that are created with the ACG have the same properties and behavior as the classes in the initial AIS hierarchy. In order to access one of the classes implemented with the ACG the user has to use the "New Object" option. To allow the user to access a class by simply selecting its name from the menu-bar the ACG has to modify its own source files, namely the file LogicalWS.m. The names of the classes implemented in the current session are inserted in the LogicalWS.m file as menu options.

At the and of a session with the ACG source files modified are compiled and link-edited and a new version of the ACG executable is created.

5.6 Conclusions

In this chapter the ACG was used to implement by means of graphic manipulation one of the classes that comprise a complex image processing application called AREX (used for visualizing multi-sensor data).

The ACG presented an alternative method of implementing an object oriented application by generating the source code for an object according to the graphical structure created by the user on the screen.

The file generated and modified with the ACG was then cross-compiled on a Sun-SPARC workstation and link-edited with the other files that comprise the AREX and with the libraries containing the image vision functions. The executable was downloaded to the SIMD parallel computer and executed.

By allowing the user to concentrate on image processing issues instead of programming issues and by allowing code reuse the ACG represents a more efficient alternative for implementing image processing applications for the AIS-3500 parallel machine.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

The visual programming environment implemented in this research, called ACG, is a tool used to build object-oriented applications for image processing on a parallel machine.

Researchers in visual programming languages agree that research should avoid general-purpose programs for professional programmers by concentrating on specialized applications [26].

The system can be characterized as a visual programming system [5] in which the language itself is visually represented but the objects do not have a visual representation [6]. The system combines graphic and textual forms creating a more efficient way of implementing an application. The ACG itself is an object-oriented application, a complex interface for a textual language (Objective-C). The computation paradigm used is the object-oriented programming model. Complex structures are constructed from simple objects and relations.

The ACG assists the programmer to create source code for an object-oriented application for visualizing multi-sensor data. The source code is generated according to the graphical structure created by the user on the screen by means of graphic manipulation. The application is executed on a SIMD parallel machine designed for vision applications.

The ACG is not a system intended for the novice programmer and implies a profound

knowledge of the image processing techniques and a working knowledge of Objective-C.

The ACG provides a good visualization of both program flow and data structures, but the system interface is not sufficient for the programmer to understand the full semantics of his program; it is necessary to understand also the underlying implementation language.

The application for visualizing multi-sensor data is developed on a Sun platform and executed on a SIMD machine. The image processing function cannot preexist in an executable format on the parallel machine, because the amount of memory is very limited and also because the parallel machine does not have an advanced programming environment. The solution is to compile the source code for the application on a different platform and to minimize this way the memory requirements.

The ACG assists the user in modifying any function or class and in creating new functions and new classes whenever necessary. The user has the possibility to "save" the graphical structures built in the work-space and to "retrieve" them easily. The graphical representation of the source code allows a more efficient understanding of the application and increases the programmer's productivity.

The possibility to "collapse" large graphical structures into small icons and to use them as standard icons allows the user to build complex programs in a limited work-space. The initial standard object's hierarchy is enhanced by user's actions. The classes implemented by the user with the ACG can be further used in the same manner as the standard classes.

6.2 Future Work

6.2.1 Reverse Engineering

Most of the visual environments completed to date have been experimental prototypes which support program composition by means of some single (graphical) representation. This is undesirable, because no one such representation has yet been accepted as superior over all others for all tasks.

The solution is a production quality environment which support simultaneous program composition/editing in any of both textual and graphical representations, with correspon-

dence among all views of the program being automatically maintained by the system.

The best programming environment would be user extensible, to allow easy incorporation of experimental program representation at will, complete with appropriate syntax and semantics. It would allow programmers to work on each part of a program using the most natural means. Furthermore, it would smooth the migration from textual programming to graphical program composition (for experts), or from graphical to textual (for novices), or between alternative graphical representations (for all users), by allowing each user to program using his/her personal interface of choice while assimilating the other.

A direction for future research is related to reverse engineering. The ACG should take source code as input and generate the corresponding graphical representation for it. In this case any modification in the source code is directly reflected by the graphical structure in the work-space and the consistency between the two representation is automatically preserved.

In the present implementation, the user saves the graphic representations of an application in the so-called graphic-files. This feature can be eliminated if the ACG is able to "read" the source file of an application and to generate automatically on the screen the graphical structure corresponding to it. This will improve user's efficiency by eliminating the need to store the graphic-files in the file system.

6.2.2 Encapsulation

One of the problems faced by the visual programming systems is the difficulty to represent large programs on the screen. The graphical representation of a program takes far more space than the textual representation. One solution for this problem is the encapsulation of a number of interconnected icons in a new graphical structure that can be further used.

There are a number of issues that have to be solved in order to make this solution useful. An icon can be linked to another icon through three types of connections: data, control, or client/server connections. In the present stage of implementation ACG allows a complex graphical structure to be collapsed in a box (called sub-box). A sub-box can be eventually connected to another icon or sub-box with a control-connection. The control-connection is used to link the last icon in the first sub-box with the first icon in the second sub-box.

The user should be able to indicate a number of "connection" points to be represented as

arrows attached to the sub-box in which the graphical structure is collapsed. The connection points are selected by the user from the graphical structure before collapsing it.

This will allow the user to connect standard icons and/or sub-boxes using all three types of connections. The user should be able to expand a sub-box in an additional window to see the detailed implementation of that particular graphical structure. Also a graphical structure containing both standard icons and sub-boxes should be collapsible. This form of encapsulation makes the generation of large application simpler and also improves the code reusability.

6.2.3 Artificial Intelligence

It is generally accepted that the power of a programming language corresponds to the conciseness of the expressions which are required from the user to specify a set of computer actions. Power is sometimes achieved in languages by the use of overloaded operators - multiple meanings for the same symbols.

In order for the system to determine which of the possible meanings the user intends, the system must carry out a problem solving process. In addition, the system may automatically handle routine specifications of details in programs, further reducing the volume of information that the programmer must supply directly.

6.2.4 Visual Programming of User Interfaces

An application for image processing implemented for the AIS-3500 parallel computer consists of two conceptually different parts: the first part is the graphic user interface and the second part is the image processing application itself.

The user interacts with the image processing application running on the parallel computer through a graphic interface by pointing and selecting (the SIMD machine doesn't have a keyboard).

The implementation method for both the graphic interface and the image processing application is the same in the present implementation, although the structures to be implemented are very different.

In creating the user interface the most suitable solution is to give the user the possibility to select from a number of visual entities (buttons, menu-bars, sliders, etc.), to compose the graphic interface on the screen and to associate to each entity a specific function.

Chapter 7

Bibliography

[1] Clarisse O. and Chang S., *VICON: A Visual Icon Manager*, in Visual Languages, New York: Plenum Press, 1986, pp. 151-190

[2] Chang S., *Principles of Visual Languages*, in Principles of Visual Programming Systems, Prentice Hall, 1990, pp 3-59

[3] Cuniff N., Taylor R.P. and Black J.B., *Does Programming Language Affect the Type of Conceptual Bugs in beginners' Programs? A Comparison of FPL and Pascal*, Proceedings SIGCHI '86: Human Factors in Computing Systems, April 13-17 1986, pp. 175-182.

[4] Darrell R., *Characterizing Visual Languages*, in Proceedings IEEE Workshop on Visual Languages 1991. pp 176-182

[5] Dudley T. and Mahling D., *Report on E-mail Panel: Is Visual Programming a New Programming Paradigm*, in Proceedings IEEE Workshop on Visual Languages 1991, pp 82-88

[6] Ellis T.O., Heafner J.F. and Sibley W.L., *The Grail Project: An Experiment in Man Machine Communication*, in RAND Report RM-5999-ARPA, 1969

[7] Glinert E., *Visual Programming Environments - Applications and Issues*, in IEEE Computer Society Press Tutorial, Los Alamitos, California, 1990

[8] Glinert E., *Nontextual Programming Environments*, in Principles of Visual Programming Systems, Prentice Hall, 1990, pp 144-230

[9] Glinert E. and Tanimoto S.L., *PICT: An interactive graphical programming environment*, in IEEE Computer 17(11), 1985, pp 7-25

[10] Glinert E., *Out of Flatland: Towards 3-D Visual Programming*, in Proceedings of FJCC '87-1987 Fall Joint Computer Conference, IEEE Computer Society, Dallas Texas, October 28-29, 1987, pp 292-299

[11] Glinert E., Blattner M. and Frerking C., *Visual Tools and languages: Directions for the '90s*, in Proceedings IEEE Workshop on Visual Languages 1991, pp 89-95

[12] Glinert E. and McIntyre D., *The User's View of SunPict, an Extensible Visual Environment for Intermediate-Scale Proccdural Programming*, in Proceedings of The Fourth Israel Conference on Computer Systems and Software Engineering, 1989, pp 49-58

[13] Hirakawa M., Iwata S., Yoshimoto I., Tanaka M. and Ishikawa T.,*HI-VISUAL Iconic Programming*, in 1987 Workshop on Visual Languages, August 19-21, 1987. Linkoping, Sweden. IEEE Computer Society, pp 40-54

[14] Kodosky J., MacCrisken J. and Rymar G.,*Visual Programming Using Structured Data Flow*, in Proceedings IEEE Workshop on Visual Languages 1991, pp 34-39

[15] Kopache M. and Glinert E., *C2: A Mixed Textual/Graphical Environment for C*, in Proceedings IEEE Workshop on Visual Languages 1988, pp 231-238

[16] Korfhage R. and Korfhage M., *Criteria for Iconic Languages*, in Visual Languages, Plenum Press 1986, pp 207-232

[17] Lakin F., *Spatial Parsing for Visual Languages*, in Visual Languages, Plenum Press 1986, pp 35-86

[18] Lau-Kee D., Billyard A., et. al., *VPL: An Active, Declarative Visual Programming System*, in Proceedings IEEE Workshop on Visual Languages 1991, pp 40-46

[19] Luo D. and Yao S.B., *Form operation by example - A language for office information processing*, in Proceedings of SIGMOD Conference, June 1981, pp 213-223

[20] Myers B.A., *Visual programming, Programming by Example, and Program Visualization; A Taxonomy*, in Proceedings SIGCHI '86: Human Factors in Computing Systems, Boston, MA, April 13-17, 1986, pp 59-66

[21] Myers B.A., *The State of the Art in Visual Programming and Program Visualization*, in Proceedings SIGCHI '86: Human Factors in Computing Systems, Boston, MA, April 13-17, 1986, pp 12-21

[22] Najork M. and Kaplan S., *The CUBE Language*, in Proceedings IEEE Workshop on Visual Languages 1991, pp 218-224

[23] Penz F., *Visual Programming in the ObjectWorld*, in Journal of Visual Languages and Computing, 1991, pp. 17-41

[24] Pietrzykowski T., Matwin S., *PROGRAPH: A Preliminary Report.* University of Ottawa Technical Report TR-84-07. April 1984, 91 pages

[25] Pong M.C. and Ng N., *PIGS - A system for programming with interactive graphical support*, in Software Practice Experience 13(9), 1983, pp 847-855

[26] Rogers G., *The GRClass Visual Programming System*, in Proceedings IEEE Workshop on Visual Languages 1990, pp 48-53

[27] Rogers G., *Visual Programming with Objects and Relations*, in IEEE Workshop for Visual Languages, 1988, pp 29-36

[28] Selker T. and Koved L., *Elements of Visual Language*, in Proceedings IEEE Workshop on Visual Languages 1988, pp 38-44

[29] Shneiderman B., *Direct Manipulation: A Step Beyond Programming Languages*, in IEEE Computer 16(8) August 1983, pp 57-69

[30] Shu N.C., *Visual Programming Languages - A Perspective and a Dimensional Analysis*, in Visual Languages, Plenum Press 1986, pp 11 - 34

[31] Shu N.C., *FORMAL: A forms-oriented and visual directed application system*, in IEEE Computer 18(8), 1985, pp 38-49

[32] Smith D.C., Pygmalion: *A Computer Program to Model and Simulate Creative Thought*, Basel, Stuttgart: Birkhauser, 1977, 187 pages

[33] Tanimoto S., *VIVA: A Visual Language for Image Processing*, in Journal of Visual Languages and Computing, 1990, pp 127-139

[34] Taylor T. and Burton R., *An Icon-Based Graphical Editor*, in Computer Graphics World 9(10), Oct. 1986, pp 77-82

[35] Yao S.B., Hevner A.R., Shi Z. and Luo D., *FORMANAGER: An office forms management system*, in ACM Trans. Office Inf. Syst. 2(3), 1984, pp 235-262

[36] Williams C. and Rasure J., *A Visual Language for Image Processing*, in Proceedings IEEE Workshop on Visual Languages 1990, pp 86-91

[37] * * *, Object Layer Reference Manual, Applied Intelligent Systems, Inc., 1989.

[38] * * *, C Layer Reference Manual, Applied Intelligent Systems, Inc., 1989.

Appendix A

The ObjectBaseFile Text File

```
#AisObject
$
+new
%id
-intPerform:
?=char*
%int
-intPerform:with:
?=char*
?<int
%int
@
#Screen
$
+new
%id
-activate
%
-add:
?>id
```

```
%
-deactivate
%
-menu:
?>id
%
-name:
?<char*
%
-when:send:message:
?<int
?>id
?=char*
%
-while:send:message:
?<int
?>id
?=char*
%
@
#Monitor
$Screen
-readConfigFile
%
-display:
?<frame
%
-initImage
%
-green:
```

```
?<frame
%
-display
%
-greyLevel:
?<frame
%
@
#ScreenRect
$Screen
+on:
?>id
%id
+willBeOn:
?>id
%id
-client
%id
-client:
?>id
%id
@
#ParameterPanel
$ControlPanel
@
#ControlPanel
$ScreenRect
@
#Menu
$ScreenRect
```

```
@
#MenuBar
$ScreenRect
@
#PanelButton
$ScreenRect
@
#Notifier
$
+initialize
%id
-activate
%
@
#Choice
$ScreenRect
-readMethod:writeMethod:
?=char*
?=char*
%
@
#PointList
$
@
```

Appendix B

The FunctionBaseFile Text File

```
#acquire()
$int
<camera_lenght_side_sqber(int)
*frame
>dest_frame(frame)
@
#display()
$frame
<frame_to_display(frame)
@
#frame_get()
*frame
>ptr_to_frame(frame*)
@
#addkx()
$frame
<s(frame)
$int
<syshift(int)
$frame
```

```
<k(frame)
*frame
>d(frame)
$int
<dxshift(int)
*frame
>dc(frame)
$int
<dcxshift(int)
@
#and()
$frame
<s1(frame)
$int
<s1yshift(int)
$frame
<s2(frame)
$int
<s2yshift(int)
*frame
>d(frame)
$int
<dxshift(int)
@
#d_all()
$frame
<s(frame)
$int
<syshift(int)
*frame
```

```
>d(frame)
$int
<dxshift(int)
@
#eq()
$frame
<s1_8(frame)
$int
<s1yshift(int)
$frame
<s2_8(frame)
$int
<s2yshift(int)
*frame
>d(frame)
$int
<dxshift(int)
@
#gtk()
$frame
<s(frame)
$int
<syshift(int)
$int
<k(int)
*frame
>d(frame)
$int
<dxshift(int)
@
```

```
#ltk()
$frame
<s_8(frame)
$int
<syshift(int)
$int
<k(int)
*frame
>d(frame)
$int
<dxshift(int)
@
#not()
$frame
<s(frame)
$int
<syshift(int)
*frame
>d(frame)
$int
<dxshift(int)
@
#or()
$frame
<s1(frame)
$int
<s1yshift(int)
$frame
<s2(frame)
$int
```

```
<s2yshift(int)
*frame
>d(frame)
$int
<dxshift(int)
@
#subkx()
$frame
<s(frame)
$int
<syshift(int)
$frame
<k(frame)
*frame
>d(frame)
$int
<dxshift(int)
*frame
>dc(frame)
$int
<dcxshift(int)
@
#tfr8()
$frame
<s_8(frame)
$int
<syshift(int)
*frame
>d_8(frame)
$int
```

```
<dxshift(int)
@
#xor()
$frame
<s1(frame)
$int
<s1yshift(int)
$frame
<s2(frame)
$int
<s2yshift(int)
*frame
>d(frame)
$int
<dxshift(int)
@
#exit()
$int
<status(int)
@
#frame_free()
$frame
<fp(frame*)
@
#frame_release()
$frame
<fp(frame*)
@
```