

A VISUAL OBJECT-ORIENTED ENVIRONMENT
FOR LISP

By
LEONG Hong Va

A THESIS

Submitted to
The Chinese University of Hong Kong
in partial fulfillment of the requirements
for the degree of

MASTER OF PHILOSOPHY

Department of Computer Science

May 1989

303883

thesis

QA

76.73

L23L46



ABSTRACT

A VISUAL OBJECT-ORIENTED ENVIRONMENT FOR LISP

By LEONG Hong Va

A multiparadigm environment allows the programmer to choose a notation, which is the most natural for the problem at hand, among a number of alternatives. This thesis is concerned with the design of the multiparadigm programming environment VOCOL, in which an attempt to integrate the visual, functional, concurrent and object-oriented programming paradigms within a LISP framework is made. With a layered approach, the VOCOL system is segmented into the visual LISP layer, the object-oriented layer and the application layer. VCLISP, a visual concurrent LISP environment, is conceived both as an enhanced stand-alone LISP environment and as the visual LISP layer of VOCOL, on which the object-oriented layer is built. On the object-oriented layer, the standard features of objects, classes, inheritance and message passing are defined. In addition, the notions of abstract data types, procedures and processes are embraced by objects. Relational links, function modularization objects and extra message passing mechanisms are devised to extend the power and flexibility of the system. Programming tools provided on the application layer further alleviate the programmers' burden and increase their productivity. Besides the design, a prototype of VOCOL has been implemented on the VAX workstation, with a sliced evaluation technique. A variety of sample programs have been developed on both the visual LISP layer and the object-oriented layer with the aim of shedding light on the capability and illustrating the usefulness of VOCOL.

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude and appreciation to Dr K.W. Ng, my supervisor and to Dr A. Choi, my previous supervisor. Their guidance and encouragement during the past two years have been invaluable to my research project. Their sound suggestions and critiques are indispensable for the successful preparation of this thesis. In addition, they gave many valuable comments and insight on my project and proofread this thesis. Thanks are also attributed to Dr D.G. Bobrow who made a request to the ENVOS Corporation to grant me the LOOPS Manual free of charge for my research.

TABLE OF CONTENTS

ABSTRACT	i
ACKNOWLEDGEMENTS	ii
TABLE OF CONTENTS	iii
TABLE OF ACRONYMS	vi
CHAPTER 1 INTRODUCTION	
1.1 PROGRAMMING ENVIRONMENTS	1
1.2 THE IDEAL PROGRAMMING ENVIRONMENT	2
1.3 RESEARCHES TOWARDS THE IDEAL	4
1.3.1 New Programming Style	4
1.3.2 Multiparadigm Programming Environment	5
1.3.3 Automatic Programming	6
1.4 PURPOSE OF THIS RESEARCH	7
CHAPTER 2 MULTIPARADIGM PROGRAMMING ENVIRONMENTS	
2.1 VARIOUS PROGRAMMING PARADIGMS	9
2.1.1 Conventional	9
2.1.2 Functional	10
2.1.3 Logic	12
2.1.4 Object-oriented	13
2.1.5 Visual	14
2.1.6 Concurrent	15
2.2 DESIGNING A MULTIPARADIGM PROGRAMMING ENVIRONMENT	16
2.2.1 Methodologies	17
2.2.2 An Evaluation	18
2.3 EXISTING MULTIPARADIGM SYSTEMS	19
2.3.1 Enhancing Conventional with Object-oriented Programming: C++	20
2.3.2 LISP-based Logic Programming: LogLisp and Qlog	20
2.3.3 Integrating Functional and Object-oriented Programming: LOOPS, CommonLoops and Flavors	21
2.3.4 Object-oriented Logic Programming	22
2.3.5 Visual Based Systems	23

CHAPTER 3	DESIGNING THE VOCOL SYSTEM	
3.1	MAKING A DECISION	25
3.1.1	Choosing Component Paradigms	25
3.1.2	Design Methodology	27
3.1.3	The System VOCOL	28
3.2	SYSTEM ARCHITECTURE	28
3.3	IMPLEMENTATION ISSUES	30
3.4	A PROTOTYPE IMPLEMENTATION	32
CHAPTER 4	VISUAL LISP LAYER	
4.1	VCLISP SPECIFICATION	33
4.2	THE VCLISP ENVIRONMENT	34
4.2.1	Visual Aspects	36
4.2.2	Non-visual Aspects	45
4.2.3	Concurrent Aspects	46
4.2.4	Name Space	50
4.3	AN IMPLEMENTATION ON COMMONLISP	52
4.3.1	An Anatomy of LISP	52
4.3.2	Sliced Evaluation of A Form	53
4.3.3	Process Scheduling	55
4.3.4	Icons and Interfaces	56
4.3.5	Concluding the Prototype	58
CHAPTER 5	OBJECT-ORIENTED LAYER	
5.1	AN OVERVIEW	59
5.2	ELEMENTS OF THE OBJECT-ORIENTED LAYER	61
5.2.1	Object	61
5.2.2	Relation and Inheritance	64
5.2.3	Message Passing	70
5.2.4	Concurrent Process	78
5.2.5	Function Modularization Object	80
5.3	OBJECT-ORIENTATION ON VCLISP	82
CHAPTER 6	APPLICATION LAYER	
6.1	PROGRAMMING TOOLS	86
6.2	PROLOG INTERPRETER	86
6.3	EXPERT SYSTEM SHELL	91

CHAPTER 7	PROGRAMMING IN VOCOL	
7.1	OVERVIEW	97
7.2	PROGRAMMING WITH VISUAL LISP FEATURES	97
7.3	CONCURRENT PROGRAMMING WITH VCLISP	98
7.3.1	Dining Philosopher	98
7.3.2	Producer and Consumer	102
7.4	OBJECT-ORIENTED PROGRAMMING IN VOCOL	106
7.4.1	Binary Tree	107
7.4.2	Distributed Sorting	111
7.4.3	Project Assignment	114
7.4.4	Dining Philosopher Revisited	117
CHAPTER 8	EVALUATION AND RELATED WORKS	
8.1	RELATED WORKS	122
8.1.1	LOOPS	122
8.1.2	ConcurrentSmalltalk	124
8.2	COMPARISON	126
8.2.1	Evolution from LISP and Smalltalk	126
8.2.2	Comparison to LOOPS and ConcurrentSmalltalk	127
8.3	EVALUATION	129
8.4	FUTURE DEVELOPMENT	130
CHAPTER 9	CONCLUSION	133
APPENDICES		
A	The BNF of Some Meta-forms	136
B	An Example of the Evaluation Cycle	138
C	Some Iconic System Functions	140
PUBLICATION		141
REFERENCES		142

TABLE OF ACRONYMS

BBF	blackboard framework
BFJ	breadth first join
BFO	breadth first only
BFS	breadth first search
BMP	best matched pair
CE	class editor
DWM	double wildcard matching
FMO	function modularization object
GNS	global name space
IE	icon editor
KS	knowledge source
LNS	local name space
MA	message agent
MC	message centre
SCM	system configuration manager
SGE	system graph editor
SWM	single wildcard matching
VFE	visual function editor
WYSIWYG	what you see is what you get

CHAPTER ONE

INTRODUCTION

1.1 PROGRAMMING ENVIRONMENTS

A computer (hardware) is designed to work for human beings. Computers are dumb and must be told explicitly what to do (by means of software). The art of instructing a computer to do the required work is known as programming. Software can be grouped functionally into software systems. A software development cycle is undergone before a piece of software is produced. The collection of hardware and software tools used to develop software systems is termed an environment [Dart87].

A programming environment is an environment that supports the coding phase of the software development cycle, providing such facilities as editor, compiler, debugger and well-defined user interface. It makes the programming-in-the-small tasks relatively easy and aims at increasing the productivity of the programmer. In a broader sense, it is an environment under which programs are developed, either from scratch or by means of automated tools and library modules. It supports the various phases from design to development, and from documentation to maintenance. When pushed to the extremity, a programming environment

evolves into a software development environment, which supports all the activities in the software development cycle. Tools for programming-in-the-large activities such as configuration management, version control and project management are integrated and are at the programmer's own disposal.

Software systems are vital to the operation of computer systems. They are produced under programming environments. With a little thought, we can infer that programming environments are vital to computer systems. With the additional fact that software is costly and unreliable [Shooman83], the software crisis continues to threaten the computer society. A good programming environment that relieves the burden of program development and increases productivity is therefore crucial.

1.2 THE IDEAL PROGRAMMING ENVIRONMENT

The importance of programming environments leads us to pursue the ideal. What is an ideal programming environment then? What do we expect to gain from it? How can we develop such an environment? Now, let us take a glance at a few widely accepted criteria for an ideal programming environment and our expectations. The last question is answered in the next section. The ultimate goal of researches on programming environments and other related fields is towards an intelligent computer system which can generate programs for

us. We just need to describe what is to be accomplished. The intelligent computer system is expected to totally take up the role currently played by programmers and even system analysts.

We want to put the computer to work with minimum effort and minimum skill, without the need for esoteric training if it is avoidable [Martin85]. Programmers should be able to utilize the computer with minimum time for application development and human errors must be minimized. The underlying programming languages must be simple in both concept and syntax. They should be very high level and non-procedural. A structured editor is often appropriate. Programs must be maintainable, portable, extensible, powerful and reusable. Incremental program development and immediate computation [Reps87] should be supported so that *what you see is what you get* (WYSIWYG). Adequate support to error detection (good compiler and debugger), error repair (good structured editor) and error recovery should be provided, in addition to an experimental system with checkpoints. High quality user interface emphasizing user friendliness is of the utmost importance.

Different people have different expectations on an ideal programming environment. Expert programmers are usually anxious for the efficiency of the program and on the availability of debugging and documentation tools. They are ready to sacrifice some "fancy" features for efficiency and response time. Novice programmers are more concerned with

the friendliness of the user interface, adequacy of online guidance and ease of casual programming. They are willing to yield run time efficiency to a better user-computer interaction. Intermediate programmers would probably expect both but to a lesser degree. They are less concerned with program efficiency and fancy features. Nevertheless, the classification is not a clear cut one and the programmers' expectations may vary.

1.3 RESEARCHES TOWARDS THE IDEAL

Although it is relatively easy to point out what characteristics an ideal programming environment should have, such an environment is an unattainable goal. What we can do is to get as close as possible and this is where researchers are making progress. A variety of approaches are under research, including exploration of new programming styles, new multiparadigm programming environment designs and new automatic programming techniques.

1.3.1 New Programming Style

Since the construction of the first computer, von Neumann programming style dominates contemporary programming languages. As a substitute, it is desirable to develop new styles of programming in which a program is defined in some other forms or by some other means, in the hope of overcoming the weaknesses of existing styles. For example, logic

programming was a new style in the sixties. Programming is achieved by exhibiting axioms and constraints. It is claimed to be non-procedural. The design of dataflow languages and machines marked the beginning of yet another programming style in relieving the inherent *von Neumann bottleneck*¹ and exploiting a maximal degree of parallelism. In visual languages, icons are drawn and connected together to perform some computations, with the idea of letting the picture be the program. In the course of searching for new programming styles, many new programming paradigms, such as the object-oriented paradigm, were conceived.

1.3.2 Multiparadigm Programming Environment

A paradigm is a model on which programs are constructed, according to certain syntactic rules governed by that paradigm. Each paradigm has its own characteristics, merits and demerits. A multiparadigm language is one that encompasses features of several programming paradigms. The design of a multiparadigm language aims at blending together the relative advantages of the component paradigms and alleviating their relative disadvantages whilst keeping their interactions low. A multiparadigm programming environment is built by integrating a multiparadigm language into an environment that supports relatively easy program development.

¹ A change of state in a von Neumann computer *always* involves the movement of one or more data items between the CPU and the memory (or register), the bottleneck [Backus78].

A programming paradigm is analogous to a tool in a toolbox. Each tool is specialized for its application. We would not use a screwdriver to pry up nails. Similarly, we would like to implement a concurrent program with a concurrent language like Ada rather than the sequential language Pascal and to do simulation with a simulation language like GPSS rather than Fortran. Prolog is excellent in implementing expert systems but weak in abstract data types and user interfaces. Baroque idioms must be adopted to get around the problem when the tool we are using does not suit our need. Therefore paradigms (tools) must be integrated within a powerful environment (toolbox) to provide various program development tools so that incremental program development is possible with little cost [Bobrow85]. In this way, we are able to utilize the right tool at the right time and this is one of the driving force behind multiparadigm programming environments.

1.3.3 Automatic Programming

Automatic programming means having the computer help write its own programs [Heidorn77]. In a broad sense, an automatic programming system is one that carries out part of the programming activity currently performed by a human programmer [Hammer79]. Under this definition, almost every effort made in computer science is attributed to automatic

programming². As a contemporary yardstick, automatic programming aims at the generation of the program from something superior to programming languages, either in the form of specification or by inference.

An automatic programming system has four characteristics: a specification method, a target language, a problem domain, and an approach to operation [Barr82]. Given a specification of a program in the problem domain, certain operations are performed to generate the required program written in the target language. Specification languages, programming by example (program induction) and program-building expert systems are all automatic programming systems with different characteristics.

1.4 PURPOSE OF THIS RESEARCH

An investigation on the ideal programming environment and various research efforts on programming environment reveals that all the above-stated research directions are equally important. However, programmers are usually reluctant to abandon their current programming style and the introduction of new programming styles will certainly be met with great resistance at first. As for automatic programming, formal specification is precise and guarantees correct code

² In the early days of programming in assembly languages, the Fortran compiler was claimed to be an automatic programming tool because it relieved the programmer from going into the details of expression evaluation, multi-dimensional array indexing, etc. Similarly, Prolog and other 4GLs were means to automate programming in the seventies.

generation but hard to learn. Natural language specification is easy to write but hard to translate and to resolve the inherent ambiguity. The code generated is highly probable to be incorrect. Programming by example is generally related to casual programming. Automatic programming is effective but not suitable for programmers of diversified aptitudes.

The accommodation of programmers of different talents and capabilities is one of the objectives of this research. Multiparadigm programming environment is probably a promise which has the ability to satisfy all the three levels of programmers: novice, intermediate and experienced. Therefore, the main purpose of this research is to design a multiparadigm programming environment, a survey of which is presented in chapter 2, with well-mixed component paradigms combined in suitable proportions and in a useful way. Good user-computer interfaces can assist both intermediate and novice programmers, particularly the latter. Structural and pictorial inputs and responses are helpful. This can be realized through the use of pointing devices and high resolution graphics terminals. We are looking forward to giving the novices the impression that programming is no more a nightmare but a pleasure.

CHAPTER TWO

MULTIPARADIGM PROGRAMMING ENVIRONMENTS

2.1 VARIOUS PROGRAMMING PARADIGMS

There are many programming paradigms in use. Some of them have a very long history but some are still in their very infancy. Before examining contemporary multiparadigm programming environments, a few common programming paradigms are presented.

2.1.1 Conventional

Conventional languages are often called procedural languages. This programming paradigm originated from the natural architecture of von Neumann computers. Perhaps it got the name *conventional* from the fact that all the very first programming languages were similar in nature and derived from the von Neumann architecture. All of them emphasize on the control flow of the program.

In a conventional programming language, the assignment statements play the predominant role. All other language constructs are built around assignment statements to make the desired computation possible. Conventional languages use variables to imitate the computer's storage cells and assignment statements to imitate the fetching and storing of

data, and arithmetic and logical operations on them. Program execution is reflected by the change of states of the system. Most of the languages used today, from assembly language to Fortran, from COBOL to Algol, and from Pascal to C, are conventional. They form the mainstream of contemporary programming languages.

Conventional languages are both fat and flabby in a global sense with the von Neumann bottleneck [Backus78]. The world of assignments, mixed with many other statements, is chaotic. The programmer must do much housekeeping work in a computation and is deprived of the ability to concentrate on the actual solution. To compute the product of two matrices, he must use three nested loops and to manage the change of indices. This can be alleviated. In the hardware aspect, SIMD computers transfer and manipulate a vector of data simultaneously and in the software aspect, languages like APL¹ operate on collections of data item and support functional forms.

2.1.2 Functional

Functional programming is based on mathematical theories on functions. The strong relationship between functional programming and mathematics facilitates program description and proof. The programming language can often be described in terms of itself, without the need of a meta language. A functional program is composed of a set of function

¹ APL is sometimes not considered as a conventional language.

definitions and a sequence of function applications. All functions map objects into objects. New functions can be built by a fixed set of combining forms called functional forms such as composition and reduction. Functional languages are normally history insensitive and computational results are temporary. Values computed from functions are passed about without being actually changed.

The first functional language in use was LISP [McCarthy60], in which recursive functions are defined in terms of propositional expressions, predicates and conditional expressions. The programming system is built upon the lambda calculus. In LISP, those objects manipulated by functions are called atoms. Composite functions are returned as functional arguments. Strictly speaking, pure LISP is functional but not ordinary LISP because the latter allows side effects which is not an element of functional programming.

Besides LISP, Backus [1978] also defined functional programming systems called FP and FFP. They are similar to LISP but the lambda calculus was abandoned. Functional forms are used extensively. Storage cells and a simple naming convention are provided in FFP, the super set of FP, to overcome the drawback of history insensitivity. Cells are used to store newly defined functions and naming convention helps to locate them.

2.1.3 Logic

In logic programming, the programmer defines relationships among values or objects. A program is a set of axioms and rules. A computation is a deduction of the consequences of the program [Davis85]. To solve a problem, a description of the problem or the solution is given, together with the constraints and conditions, in the form of logic statements. The system will try to find out the solution by unification - the deduction mechanism. Logic programs are elegant with control and logic separated [MacLennan83].

Prolog [Clocksin81] is recognized as the dominant logic programming language. Most people even have the impression that logic programming is synonymous with programming in Prolog. Prolog shares the same merits with logic programming languages but it has the implementation drawback of the sequential execution of subgoals in a rule and the sequential unification of the alternatives of each subgoal. An intrinsic depth-first search is implied and the order of subgoals in a rule *does have* influence on the execution of a Prolog program. This property violates the idea of logic programming. Also Prolog is logically incomplete. Furthermore, the cut in Prolog is believed to behave like the *goto* statement, which is considered harmful [Dijkstra68].

2.1.4 Object-oriented

The object-oriented paradigm sprang up since the design of Smalltalk [Goldberg83], the pioneer of object-oriented programming languages. The concepts are simple and the notations are natural. An object program can be considered a simulation of the activities of entities in the universe of interest. In an object-oriented programming environment, the system is composed of a collection of *objects* with *attributes*. Similar objects are grouped together into *classes*, each of which encapsulates the common properties of these objects. A class can be thought of as a stereotype. Specialized properties are defined in the objects themselves, known as *instances*. There can be classes of classes, called *metaclasses*, and so on. The propagation mechanism of properties from parent classes to child classes or instances is termed *inheritance*. A class can inherit from one (*single inheritance*) or more parents (*multiple inheritance*). An alternative to inheritance is delegation, adopted by the actor model [Lieberman86,87].

Objects can communicate with one another by means of *message passing*. An object responds to a message by the invocation of a *method*, causing a change to its internal state and/or messages to be sent to other objects. The set of messages recognizable is termed the *protocol* (external interface). An important concept is *polymorphism*: different

objects behave in their own ways upon the receipt of the same message even if they belong to the same class because each object is autonomous and has its own internal state.

The programmer defines the object classes, attributes, protocols and inheritance properties in an *object program*. Computation takes the form of message exchanges among objects and state changes within objects. The state of a program is the collection of the states of objects in the system.

Smalltalk is a pure object-oriented programming environment, built on the Smalltalk virtual machine [Goldberg83]. In the contour of Smalltalk, everything is considered to be an object and treated in a unified manner. Each object must be an instance of a class. Classes are instances of metaclasses. Single inheritance is adopted. Class variables are shared among instances of the same class whereas instance variables are private to the instances. The protocol of an object is defined by its methods. Instances attributes and methods are inheritable but can be overridden whereas class variables are global names and can never be redefined. Message passing is the sole mechanism of program execution.

2.1.5 Visual

The relatively new visual paradigm is causing greater and greater impact on programming environments. As graphics processors and dedicated hardware are designed and

manufactured with low costs, visual programming is brought from aspiration to reality. Many programming environments are built with visual interfaces and interactions. A simple implementation of the visual environment involves the use of icons as an alternative to system objects like files, folders and programs as in the Macintosh. User can select and apply operations on them. These graphical user interfaces have been widely accepted.

In more advanced implementations, pictures and icons are used as program constructs. The programmer just draws the program. The advantage is the direct visualization of the program constructed. Programs can be animated to give the programmer a feeling of how the program works, aiding him in understanding and debugging the program.

2.1.6 Concurrent

Concurrent programming originates in the sixties as an approach to the design of operating systems. As the cost of processors has been decreasing in the past ten years, more and more multi-processor systems and distributed systems are being built. Concurrent programming begins to invade every kind of application, including database management systems, parallel algorithms and computations, and real time systems. Concurrent programs are advantageous in that their exploitation of parallelism during execution results in a great reduction in execution time in a multi-processor environment.

A concurrent program is composed of more than one sequential programs executing in parallel, possibly as parallel processes. The basic issues in a concurrent programming system are parallelism, communication and synchronization [Andrews83]. Parallelism is concerned with how to express concurrent computations and what notation is to be adopted. Communication is concerned with the means of information exchange between parallel computations. Synchronization is concerned with the coordination between them.

Concurrent programming, as with the visual paradigm, can be incorporated into any programming paradigms. Most concurrent programming languages are extensions to existing conventional languages or are themselves conventional, such as concurrent Pascal and Ada. Some popular programming languages belonging to other programming paradigms are also enhanced with concurrent constructs, like PARLOG (concurrent Prolog) [Clark84] and ConcurrentSmalltalk [Yasuhiko87].

2.2 DESIGNING A MULTIPARADIGM PROGRAMMING ENVIRONMENT

Several factors must be considered when designing a multiparadigm environment: the initial cost of learning, the costs of debugging, maintenance and running [Bobrow85], and design objectives and philosophies. To design a multiparadigm language, there are at least five different

methodologies [Hailpern86]. After designing the underlying language, it remains to devise suitable tools like the editor, compiler, debugger and file manager. These tools, together with the new language, are integrated together into a programming environment through carefully designed user-computer interface.

2.2.1 Methodologies

Some multiparadigm systems are developed by combination of existing languages. The idea is to develop a new system that supports all the component languages individually, using the same syntax and semantics. They can be loosely integrated or fully integrated.

Another method is to select one of the languages as the basis and embed the other languages in it. This may be done via special function calls. For example, Qlog and LogLisp are Prolog systems in Lisp built in this way.

The third method, together with the second method, are among the most popular. New language constructs are added to a selected base language. Concurrent Pascal is built from Pascal by defining concurrent constructs. Object-oriented programming is partially supported in the implementation of C++ by defining class and inheritance.

The fourth method is the redefinition of a language in the context of new theoretical insights and goals. Undesirable features of the language can be corrected.

Meanwhile, a new paradigm may be added through redefinition. This is equivalent to introducing new language constructs in parallel with modifying existing ones.

The last method is to learn from past experience and to design a new programming language out of nothing, trying to incorporate useful features from other languages. There are numerous of them. Nial [Jenkins86] supports conventional paradigm and functional paradigm directly and many others indirectly. Lucid [Faustini86], a real time dataflow language, is also developed from scratch.

2.2.2 An Evaluation

The first method in the last section is good for experienced programmers because they are already familiar with those constituent languages. Programs already written in one of these languages are portable. However, in a primitive multiparadigm environment such as an operating system containing LISP and Pascal, the interface between different languages is difficult. In a completely integrated system, definitions must be given to data structures and language constructs of one language in the context of another. The interactions, especially the unintended interactions not covered by the definitions, among these languages are great and complex.

The second method and the third one are very similar. They require fewer system development efforts. The new system

is a superset of the base language and all the facilities in the original programming environment are available. Compared with the first approach, these methods suffer from a degraded efficiency and flexibility.

The fourth method shares similar advantages and disadvantages with the above methods. Better embedded new language constructs in a more consistent environment is resulted. However, the redefinition of the semantics of certain language constructs may render the new language incompatible with the base language. This is hazardous to the portability and maintainability of software.

Adopting the last method is a task of tremendous complexity. The major advantage lies in the design of a consistent and elegant language and environment. Relative merits in different paradigms can be explored but the initial cost of learning is extremely high. The environment must be an excellent one before it can attract a reasonable community of users.

2.3 EXISTING MULTIPARADIGM SYSTEMS

Researches on multiparadigm programming environments are actively being carried out to exploit their merits towards the ideal. By conducting a survey on some existing multiparadigm systems, we can gain insight into the situation and into the current trends of research.

2.3.1 Enhancing Conventional with Object-oriented Programming: C++

Some conventional languages are flourished with object-oriented capabilities. A typical example is the C++ language. It is a superset of C and enforces strong typing, including typing to function parameters. New data types which are supposed to be as efficient as the built-in types are readily defined, with appropriate manipulation functions. These data types are similar to packages in Ada. To support object-oriented programming, the class concept is introduced. Classes are organized in a strictly hierarchical manner. C++ combines the advantages of the flexibility and efficiency of C and the capability of hierarchical representation of data types with inheritance in an object-oriented language. It is found useful in a diversity of systems programming.

2.3.2 LISP-based Logic Programming: LogLisp and Qlog

Both functional and logic programming are clean, elegant and powerful. It is possible to simulate logic programming with functional programming and vice versa but such a simulation is both inefficient and often burdens the programmer with extra load. A combination of these two programming paradigms is suggested and implemented in several systems. The design of LogLisp [Robinson82] and Qlog [Komorowski82] aims at the well-developed LISP programming

environment as a basis for logic programming. They hope to provide the programmer with two different paradigms at his own disposal.

In LogLisp, Prolog clauses are written as S-expressions and a query is a list with a leading keyword. Assertions and queries are executed as if they were LISP special forms. Expressions can be simplified according to LISP semantics, leading to the utilization of almost the full power of LISP in a logic programming environment. He can also devise a query whose solution is returned as a LISP data structure, used for LISP computation. The interface between logic programming and the LISP programming environment is implicit.

The standard syntax of Prolog clauses is preserved in Qlog. Prolog clauses defined are transformed into callable functions. Backtracking and variable instantiation are performed explicitly by manipulating stacks. The interface between logic programming and LISP in Qlog is explicit. LISP functions must be called from a Prolog program to initiate a LISP computation and vice versa.

2.3.3 Integrating Functional and Object-oriented Programming: LOOPS, CommonLoops and Flavors

The popular object-oriented paradigm is often combined with the functional paradigm, mainly the language LISP, such as LOOPS [Bobrow82], CommonLoops [Bobrow86] and Flavors [Moon86], to name a few. LOOPS is a LISP based multiparadigm

environment implemented in the interactive InterLisp-D environment. It supports LISP style programming and provides extensive visual effects, with object-oriented paradigm incorporated. Active values (annotated values) lead to access-oriented programming. Rule-oriented programming is supported as extension modules. On the success of LOOPS, one of the most popular multiparadigm system, CommonLoops is developed on CommonLisp, the most prevalent LISP dialect. This new system is built based on the experience of LOOPS but is a departure from it. It is compatible with the functional programming style in LISP. Programs can become incrementally object-oriented. Flavors is a CommonLisp package encouraging object-oriented programming. All these systems benefits from the specialization of functions and messages.

2.3.4 Object-oriented Logic Programming

SPOOL [Fukunaga86] is the result of a marriage between object-oriented and logic programming. The data structures are organized as objects with multiple inheritance. Methods are Prolog clauses. On the receipt of a message, the body of the method whose head is unifiable with the message is invoked in a Prolog-like manner. Objects with reasoning capabilities can be built. Anonymous and partially filled message is allowed which is much more powerful than conventional object-oriented languages like Smalltalk.

Orient84/K [Ishikawa86] is a knowledge representation language built on top of Smalltalk-80. An object is called a

knowledge object and comprises three parts: monitor, behaviour and knowledge base. Monitor controls incoming messages and exercises control over the other two. Behaviour is similar to Smalltalk-80 in syntax and semantics and knowledge base is organized as Prolog clauses. Multiple inheritance is allowed, with concurrent programming constructs added, like synchronization and mutual exclusion.

There are still many other systems such as Concurrent Prolog [Zaniolo84; Kahn86] which supports message passing and encourages object-oriented programming.

2.3.5 Visual Based Systems

Active researches are being made on the visual paradigm and combination of other paradigms with it. Visual based systems are particularly helpful to novices and offer advantages to other programmers.

The Pict/D [Glinert84] system, an interactive graphical programming environment, is the implementation of conventional paradigm upon visual paradigm. A program in Pict/D is constructed by drawing and editing the control flow of the program with the aid of a joystick. Testing and debugging are carried out by program animation.

Graphical FP [Pagan87] is a graphical based FP [Backus78] system. Functions and functionals are represented as boxes and names and parameters as text. Animation is achieved by highlighting the boxes. PiP [Raeder84] is also a functional

language, in which icons are used extensively, including the definition of functions and composition of functionals. Textual names only serve as secondary identifiers. PiP is designed to support programming by example, which can greatly benefit the novices.

ThinkPad [Rubin85] is a constraint-based programming by example system. Data structures are drawn with their constraints specified. Functions are then defined as a sequence of constrained transformations from input data structure to output data structure. These data structures and transformations are mapped into a series of Prolog clauses for execution.

CHAPTER THREE

DESIGNING THE VOCOL SYSTEM

3.1 MAKING A DECISION

A multiparadigm programming environment is to be designed to achieve the objective of this research: to accommodate programmers of all levels of competence. The first step in the design of a multiparadigm programming environment is to choose component paradigms with immense potential. The selected paradigms are then integrated into an environment with a suitable methodology, bearing in mind the objectives of the system.

3.1.1 Choosing Component Paradigms

Recall that our system is to be attractive to the novices. The most user-friendly paradigm is certainly the visual paradigm. It is one of the most promising field of research because it is still young and possesses affluent potential. There are remarkable benefits in visual programming. Pictures are superior than text [Raeder85] in such ways as a faster transfer rate, higher dimension of expressions and direct references. Visual representation can narrow the gap between the programmer and the programming system and it is nice to let the picture be the program. It facilitates the implementation of the powerful WYSIWYG principle and program animation which can serve as an

educational tool. The morale of novice programmers, who would like to deal with a system of icons rather than a system full of text, can be boosted by the visual paradigm.

Object-oriented programming is based on object entities and is quite natural to our daily life. It is proven to be powerful and flexible and encourages data encapsulation, classification and abstraction. Association of procedural information with object class improves maintainability and may reduce storage¹. Complex data structures may be defined as objects. An object program can manipulate these objects via encapsulated operations, facilitating the modularization of large and complex programs. Incremental addition of objects are easy, without having to modify existing objects because object methods are distributed and hidden.

LISP has been a very good experimental programming language and environment. It is a powerful language with a unified syntax and dynamic behaviour and is well recognized as an exploratory language. Interactive LISP programming environments are widely available and there are numerous works on LISP machines. Therefore it is very suitable to choose LISP's functional paradigm as a component. Furthermore, the functional style and the object-oriented

¹ For example, a vector is usually stored as an array but a unit vector can be associated with a procedure that returns the constant 1 when given any legal index. No array is needed to store the vector elements of such a vector.

style are orthogonal² and complement each other. This combination is particularly powerful when enhanced with visual programming capabilities.

The attempt of embodying the conventional programming paradigm is relinquished because the potential of conventional languages has almost been exhausted. Their advantage as a close connection with the architecture of von Neumann machines is losing its significance as new architectures and paradigms become mature.

3.1.2 Design Methodology

The methodology of defining new language constructs on a base language is adopted (section 2.2.1). In this way, the new system is able to furnish all the facilities provided in the base language environment and is a super set of it. Programmers (intermediate and expert) already familiar with the base language can move onto the new system without much difficulty. After they have become accustomed to it, they can incrementally modify their programs and programming style to take full advantages of new language constructs and features. Even though efficiency may be degraded, especially in a prototype system, better software and hardware support will evolve, should the system be proved to be successful.

² The procedural or functional programming style of LISP is orthogonal to the object-oriented style. In procedural programming, class (data structure) information is associated with general purpose procedures. In object-oriented programming, procedural information is associated with the properties of classes.

3.1.3 The System VOCOL

After a careful choice of the paradigms to be combined and the design methodologies, a multiparadigm programming environment is conceived. It is a LISP-based one with object-oriented and visual programming augmented. To increase the capability of the system, multiprogramming is supported by introducing concurrent programming concepts with the notion of LISP processes. This system is given the name *VOCOL*, the acronym for Visual Object-oriented COncurrent Lisp, encompassing the four designated paradigms.

3.2 SYSTEM ARCHITECTURE

VOCOL is a complex system with many features. To bridge the gap between the system and the implementation machine, the system is segmented into fragments of manageable size. It is designed with a layered approach in the light of better organization, understanding and modularity. The system is composed of three layers of abstraction. The bottom layer is a visual LISP layer. The intermediate layer is an object-oriented layer built upon the bottom layer, with the aid of the latter. The top layer is an application layer of programming tools. The configuration is reflected in figure 3.1.

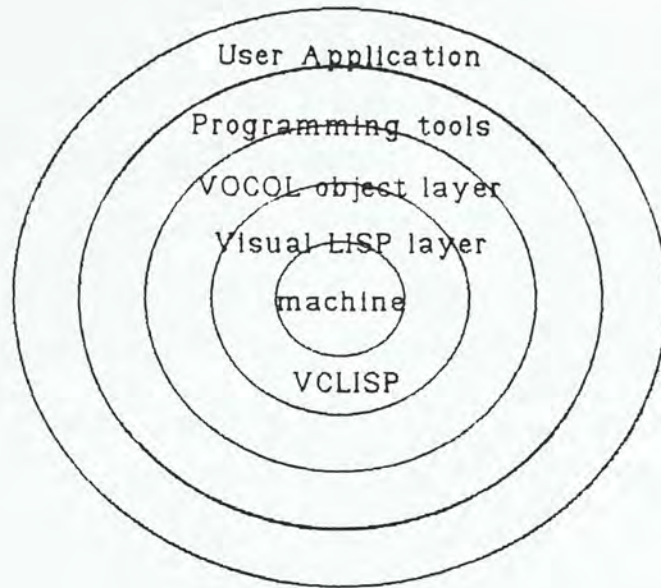


Figure 3.1 Architecture of VOCOL

The bottom layer serves as a bridge between the hardware or the operating system and the higher layers of the system. It is possible to port the system to other machines by suitable modification of this layer. The layer can be viewed as a LISP programming environment which most LISP programmers are familiar with. In fact, the visual LISP layer is intended to be an extension to a LISP programming environment. To take advantage of the visual paradigm, visual representations for LISP functions are supported. They reveal the internal structure of the underlying functions. To enhance the functionality of the layer, concurrent programming constructs are defined. This layer is designed to be a stand-alone programming environment and denominated *VCLISP*, an acronym for Visual Concurrent LISP. The programming language for *VCLISP* is a LISP dialect, named *VCLisp*.

The intermediate layer, the heart of the VOCOL system, is built upon VCLISP. On this layer, the programmer finds the object-oriented programming style. Objects are organized as a network. They denote concurrent processes with protocols through object methods defined in terms of VCLisp functions. Inter-object communication is achieved by means of message passing. This layer eventuates in a combination of the Smalltalk style with concurrent programming in LISP.

The uppermost layer is an application layer, which serves a dual purpose. Application tools are implemented in terms of VOCOL features to experiment with the power and expressiveness of the system. It also provides a better programming environment by equipping the programmer with program generation and application tools.

The programmer can develop programs on any of these three levels. Programs can be developed in the LISP environment through interaction with the user interface process. Programming in an object-oriented style can be achieved by interacting with the various editors and the user interface. Program generation tools can also be used to generate programs.

3.3 IMPLEMENTATION ISSUES

The layered approach has been adopted in the design of the VOCOL system. The upper layers need few modifications.

if any, when ported to a new machine. The programming environment supports concurrent programming and therefore a multiprogramming environment must be implemented on the target machine. Direct implementation on existing operating systems is possible but context switching between LISP processes must be handled carefully. A multiprocessor machine is more suitable to exploit the inherent parallelism in concurrent object programs.

The ideal machine to implement VOCOL is a multiprocessor LISP machine which can interpret concurrent LISP programs and execute compiled ones and with extensive graphic capabilities. Then only a graphic shell around the operating system suffices to support operations of the visual LISP layer. Visual programming features must be disabled when a graphic terminal is not available, without affecting the functionality of the system, though it results in textual representations of object programs and inter-object and intra-object interactions. The availability of this dual mode can also increase the flexibility of the system. Expert programmers may choose to program in text with only minor graphic features because it is normally faster to program in text if one is familiar with the system and sometimes they prefer command line inputs rather than selection from menus.

3.4 A PROTOTYPE IMPLEMENTATION

The design of VOCOL has been implemented in a prototype. This prototype, also named VOCOL, is developed on VAX workstation under the VMS operating system with CommonLisp [Guy84]. VOCOL is a LISP based system and it is conveniently implemented in LISP [Winston84]. This approach enables the fully utilization of available LISP built-in functions and leads to a drastic reduction of much of the memory management workload. Rapid prototyping is therefore possible.

In the VOCOL prototype, the visual LISP layer is implemented in VAX Lisp. VOCOL supports concurrent programming but there is no context switching facilities in VAX Lisp. Therefore the multiprogramming environment is simulated and VCLISP processes are scheduled by the VCLISP interpreter. Visual programming features are available as VAX Lisp functions which are regarded as if they were VCLISP system calls.

Most objects and procedures making up the object-oriented layer are programmed in VCLisp. This can increase the portability of the system and demonstrate the usefulness of the VCLISP programming environment. This prototype implementation will be presented in greater details in the following chapters, along with the detailed design of the system.

CHAPTER FOUR

VISUAL LISP LAYER

4.1 VCLISP SPECIFICATION

The visual LISP layer is the lowest conceptual layer of VOCOL, supporting visual programming in LISP. As a system requirement, a concurrent dialect of LISP is to be implemented as the basis on which upper layers of VOCOL are built. VOCOL unifies the notions of objects and processes and supports concurrently executing objects. This consideration gives rise to the design of VCLISP. VCLISP is a stand-alone LISP environment and supports concurrency and parallelism which are very important assets to the implementation of the object-oriented layer. The latter dictates a high degree of concurrency.

VCLisp is the programming language recognized and processed by VCLISP. For the sake of compatibility and familiarity, features (textual syntax and semantics) of VCLisp follow closely those of CommonLisp but with icon management, concurrent programming and process control primitives enriched. All VCLISP processes are programmable in the language VCLisp.

VCLISP is able to perform a few process scheduling tasks. It executes concurrently a number of VCLISP processes in a round-robin manner, copes with icons and visual function

definitions, supports basic process control operations, communication and synchronization primitives, and provides a globally shared space accessible to all VCLISP processes. Each process is able to spawn coprocesses to introduce a multiple number of threads. The interpreter is extensible by the inclusion of system routines written in other programming languages. A high priority console process exists to enable the programmer or operator to monitor the system and tune the performance.

4.2 THE VCLISP ENVIRONMENT

Different classes of programmers may have different views on the organization of VCLISP. The views of the configuration of the VCLISP environment by visual LISP programmers, who stick to single user LISP programming, and by VCLisp programmers, who are concerned with concurrent programming, are depicted in figures 4.1 and 4.2.

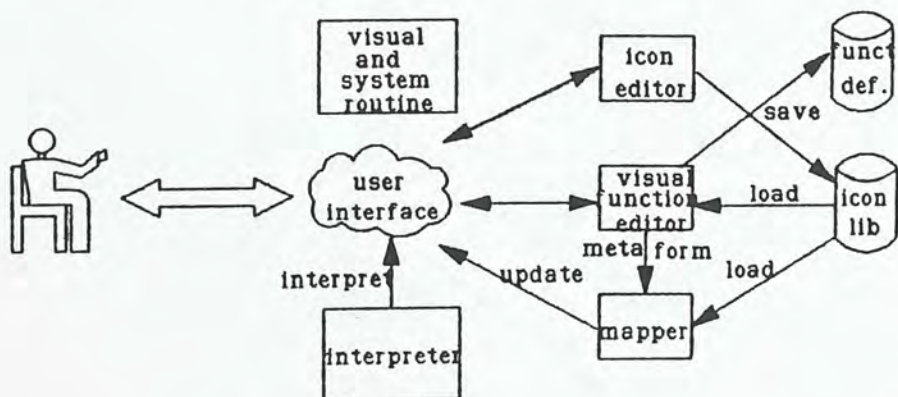


Figure 4.1 VCLISP in the View of Visual LISP Programmers

Programmers who are familiar with the LISP environment would like to have their programming environment enhanced with visual programming capabilities. In this way, they can develop applications in a more convenient way and such an environment is reflected as a rudimentary view of the VCLISP system as shown in figure 4.1. The programmer interacts with the system through a special process called the *user interface* process, which is a top level read-eval-print loop, accepting and processing requests from the user. It evaluates forms and S-expressions. In addition, it can invoke the icon editor and visual function editor to modify icons and VCLisp function definitions stored in files.

A user-defined visual function is stored as a meta-form which is converted into the internal representation by the mapper before being executed in the context of the user interface process. The programmer regards the user interface process as a visual LISP programming environment. Concurrent programming features are totally ignored.

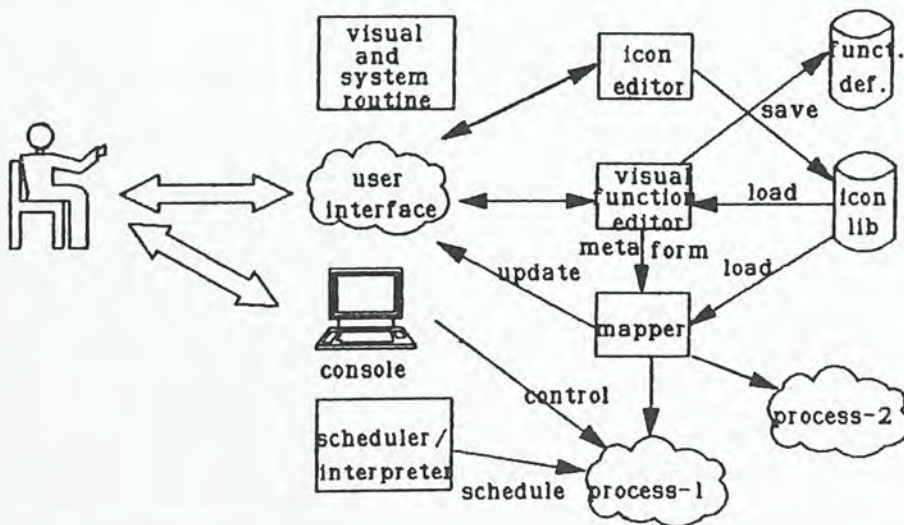


Figure 4.2 VCLISP in the View of VCLisp Programmers

There are still some programmers who are accustomed to program in a multiprogramming environment. The incorporation of concurrent programming features in VCLISP suggests the possibility of devising processes which execute concurrently. The configuration of VCLISP in their point of view is shown in figure 4.2. The *system console* process provides a means for the programmer or operator to control and monitor system activities. It also enables the programmer to preempt all the processes in case of a deadlock, and to take appropriate corrective measures. Meanwhile, the programmer interacts with the *user interface* process which can create new processes, define their code segments and activate them. Processes are created dynamically, and their interactions are governed by VCLISP concurrency semantics.

In the subsections that follow, various facets of the VCLISP environment are described and examined.

4.2.1 Visual Aspects

VCLISP is a visual environment for LISP in which VCLisp programs are built and executed. Icons and VCLisp functions can be defined and modified with various editors, to be expounded subsequently.

Icons

The basic element of a visual program is the icon. An icon in VCLISP and VOCOL is made up of an optional bitmap image, an optional drawing procedure, a logical operation, an

optional icon name and an ID (system index). The ID is used to store the icon in file (e.g. ICON18.BIT and ICON18.CTL) and to identify the icon. The appearance of an icon is the result of applying the logical operation on the pattern represented by the bitmap (a precise representation) and the image generated by the drawing procedure (a concise representation). The programmer has the freedom to choose between the two, with time/space trade-off. Icons in the VOCOL system are managed by the icon manager and modified by the icon editor.

Icon Editor

The *icon editor* (IE) is used to design new icons and modify old icons. Within the icon editor, the icon is edited with a pointing device. The bitmap is easily changed by the mouse. The drawing procedure is a sequence of calls to graphic primitives, such as *polyline* and *circle*. This sequence of graphic operations will cause an image to be drawn. The final appearance is determined by the logical operation, the bitmap and the drawing procedure. Several display modes are supported to disclose component appearances (bitmap, drawing procedure) or the complete icon appearance for ready visualization. A missing bitmap or a missing drawing procedure is considered to generate a blank image on which the logical operation is performed. A typical IE in operation is shown in figure 4.3.

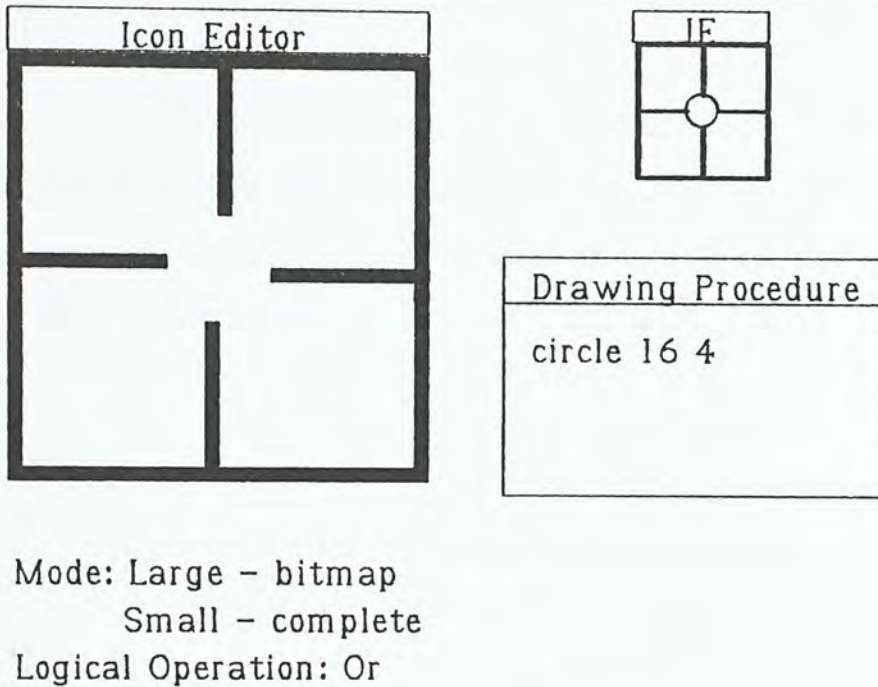


Figure 4.3 Icon Editor in Action

Visual LISP Functions (VCLisp functions)

A VCLisp function can be identified through an icon. Iconic forms have been designed for some common system primitives. They are organized in a hierarchical manner, and can then be identified and invoked from their appearances. The structure (lambda body) of a function is represented as nested blocks similar to that of a Nassi-Shneiderman diagram [Raeder85], particularly control constructs like *repeat-until*, *while-do*, *loop* and *if-then-else*, some of which are syntactic sugars¹ for constructing more complex programs.

¹ A syntactic sugar is a syntactic construct whose existence does not add any new power to a language. It solely simplifies the syntax of some language constructs or serves as a convenient vehicle to express some computations. Syntactic sugars are exemplified by the multi-dimensional array in Pascal and the *let* statement in LISP [MacLennan83].

Some of them are shown in figure 4.4. The programmer is also able to customize his programming environment by designing new iconic representations for functions. Organizing a function in a block-like manner allows a direct visualization of its logical structure and how statement blocks are nested. Owing to resolution constraint, deep levels are revealed only on zooming.

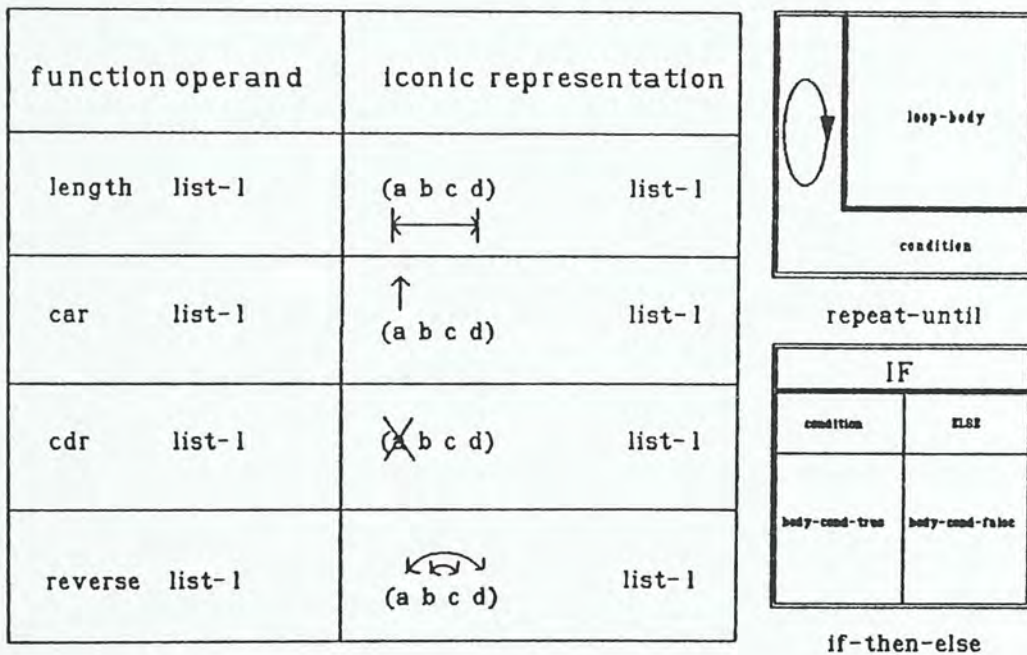


Figure 4.4 Iconic Functions and Constructs

A visual function is made up of four sections: *identification section*, *parameter section*, *local variable section* and *function body*.

a. Identification section. This section contains the icon that identifies the function. This is the atom whose lambda definition part is defined with the function body.

b. Parameter section. This section lists in iconic representation the parameters to be passed to the function. Prompts to be displayed when the function is applied can be defined. Prompted arguments combine keyword arguments and optional arguments in LISP. At every function application in the course of defining a function, the prompt is displayed along with the default value. The programmer should fill in the actual parameter before proceeding. This ensures a better consistency in the function defined.

c. Local variable section. This section lists in detail all the local variables to be used within that particular function. A local bindings of those variables will be created after evaluating the actual parameters. This is implemented with a *let* closure.

d. Function body. This part of the function definition contains the lambda body of the VCLisp function. It is a sequence of S-expressions represented in the form of nested blocks.

A function application is represented by a block with double boundary. The evaluation of forms and S-expressions, as well as application of functions follow the convention of LISP. The value returned from a function is the value of the last evaluated form, unless interfered by special forms or macros like *progl*. The natural infix notation for

expressions and assignment statements are maintained. Infix expressions are differentiated from ordinary prefix LISP S-expressions by enclosing in ovals.

To retain access to the familiar LISP programming environment, textual LISP programs are acceptable and the system is able to interpret these programs in addition to visual LISP programs. Quite often, visual LISP functions are used interchangeably with textual LISP functions. A mechanism must be adopted to translate between the two. The mapper serves this purpose. Visual representations are difficult to process. They are converted into meta-forms before storage and into internal forms before execution.

Visual Function Editor

Another basic editor provided by VCLISP is the *visual function editor* (VFE). VFE is a template-based graphic editor. The programmer interacts with the VFE to define the four sections of a VCLisp function via template filling with prompts by system through the use of pointing devices and menus. User-defined icons can be picked from the icon library as an alternative to names. The course of interaction is a major improvement over that of the graphical FP system [Pagan87].

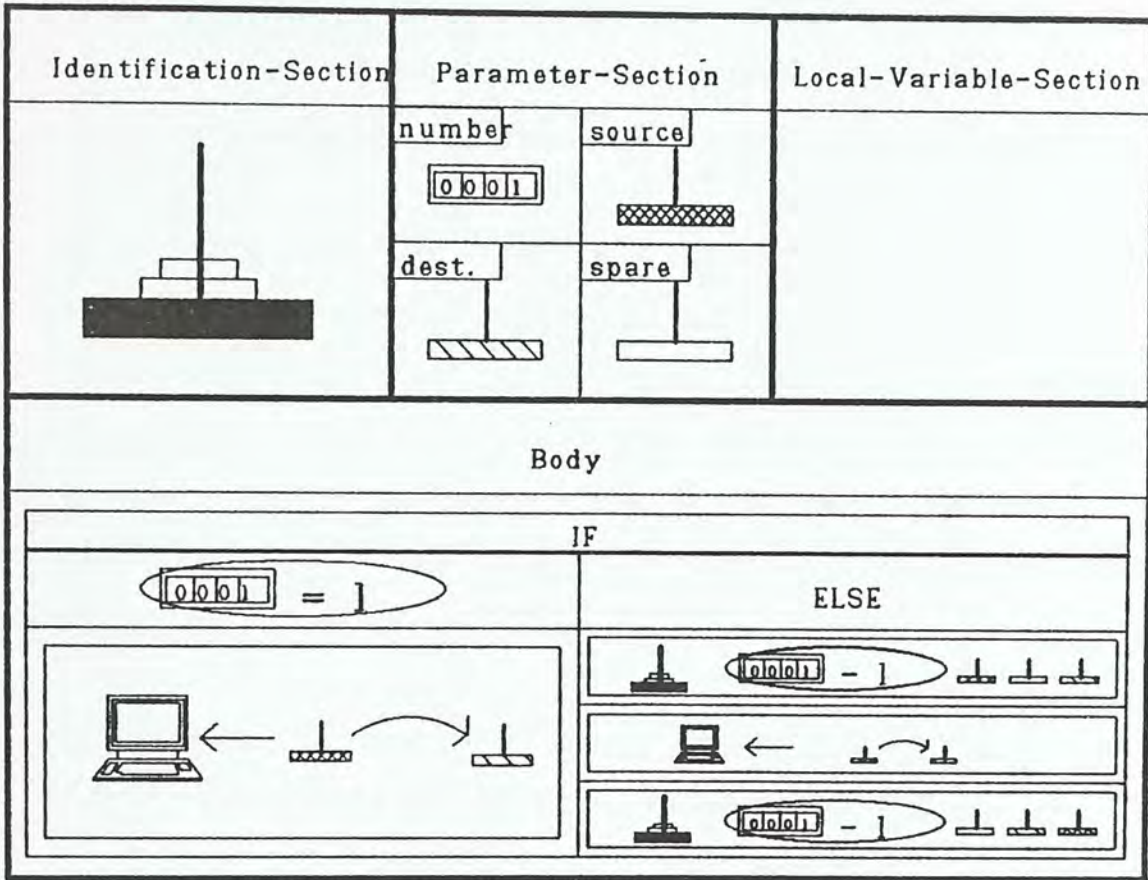


Figure 4.5 Tower of Hanoi

Consider the definition of the recursive function *tower of Hanoi*, shown in figure 4.5. In this figure, the logical structure of the function is clearly depicted. Icons almost completely substitute the role of names, except section titles and notations to certain control constructs. The solution is broken up into two distinct cases. The first case is that there is only one disc, which corresponds to the recursion basis. The operation is to output the movement of the disc from the source pole to the destination pole to terminal. The second case is the opposite. When there are more discs to be moved, we will first move all discs except the bottom one from the source pole to the spare pole, then

move the bottom one to the destination, followed by moving the rest from the spare to the destination. These operations are shown structurally in the right part of the if-block.

The output from the VFE is a function definition in the meta-form which can be stored directly on files. In a meta-form, all the icons in a visual function are replaced by their unique indices in VCLISP. It is a textual representation but the structure is almost identical to its visual counterpart. For example, the meta-form for Tower of Hanoi is

```
(visual-funct (id (name tower-of-hanoi 15))
  (para (((name number 18) (name number 18) nil)
        ((name source 20) (name source 20) nil)
        ((name dest 21) (name dest 21) nil)
        ((name spare 25) (name spare 25) nil)))
  (local nil)
  (body ((if (infix ((name ? 18) = 1))
            (then-part ((name ? 2) (name ? 20)
                       (quote (name ? 5))
                       (name ? 21)))
                     (else-part ....
                     ))
        ))),
```

where 15, 18, 2, etc. are the system indices for the icons of tower-of-hanoi, counter, print, and so on. VCLISP uses these indices to locate the iconic appearances for them. The

second tuple (name number 18) in the second line is the prompt to be displayed. The third value nil is the default value. In this example, it is not specified by the programmer, hence the value nil. The special form *infix* is used to denote an infix expression. Note here whenever the special form *name* is encountered, it is filtered by the mapper and handled immediately. A sample BNF of meta-forms is shown in appendix A.

Mapper

The iconic representation of a VCLisp program is converted into an internal form before being processed by the interpreter. This task is carried out by the mapper, which maintains a table for each VCLisp process (section 4.2.3), showing the *unique identity* (UID) for each icon, each name or each pair. UID is unique within the whole VCLISP environment. The same icon will have different UID in two different processes because they refer to different storage locations. Similarly, the UID of the *global name space* (section 4.2.4) is also distinct from those of other processes. The mapper replaces all the names and icon identities in the meta-form by internal symbols determined by their UIDs. The mapper is also responsible to convert the internal symbols back into icons and names during I/O operations. Syntactic sugars in the meta-forms are preprocessed into ordinary LISP S-expressions.

4.2.2 Non-visual Aspects

After having been processed by the mapper, all VCLisp statements and function definitions are reduced to linear textual S-expressions. Each S-expression can be an atom, a system call, a special form or a function application. An atom leads to the evaluation of its value. A system call is a call to the VCLISP system routines, such as a call to the graphics package, a process control operation or a communication primitive. A special form is implemented by the VCLisp interpreter in a special way. It may involve the establishment of bindings (*let* and *let**), the evaluation of a block of code (*block*), or the generation of a closure of the environment (*function*). A function application involves the evaluation of arguments and application of the function body. The function is either a system function or a user-defined one. System functions are intercepted and executed directly, as if they were system calls. Others are executed by the VCLisp interpreter, according to the function body.

In addition to visual functions defined by the VFE, an experienced programmer may write his program in the form of conventional LISP S-expressions, which are also mapped into internal representations before executed for the reason of compatibility. For example, the textual equivalent of the function tower of Hanoi is


```

(defun tower (number source destination spare)
  (cond ((= number 1)
        (print `(move ,source to ,destination)))
        (t (tower (- number 1) source spare destination)
            (print `(move ,source to ,destination))
            (tower (- number 1) spare destination source))))

```

This function is executed in the same way as the one shown in figure 4.5 and the meta-form presented shortly after the figure.

4.2.3 Concurrent Aspects

The basic notion in concurrent programming is the process. Parallelism, communication and synchronization are the key factors to a concurrent program [Andrews83]. In VCLISP, two levels of parallelism are defined: external concurrency and internal concurrency. External (inter-process) concurrency is the parallelism exploited within the system by parallel processes. This is the most natural way of expressing parallelism in a multiprogramming environment. The scheduler dispatches the processes to be executed by the CPU, creating the illusion that they are executed concurrently. Internal (intraprocess) concurrency is the parallelism exploited within a single process. This is expressed with the notion of *coprocesses*. A coprocess is a child process spawned or forked by a process or another coprocess (the parent) and granted full access to the parent's name space. The coprocess created differs in the

process body from the parent, contrasting with the case of a fork statement. More explicitly, the *lexical environments*² of a process and of all its coprocesses coincide. All of them share the same pool of lexical data but they have separated *dynamic environments*. All the coprocesses execute in parallel with the parent process. Therefore, coprocesses can be viewed as multiple threads within a single process, hence the term internal concurrency.

Primitive process control operations such as *create*, *activate*, *suspend*, *resume*, *delay* and *kill* are defined. They are provided to support external concurrency. The function *create* accepts a sequence of S-expressions and creates a new process whose body is made up of those S-expressions in return. The new process must be assigned a name and will not start its execution unless it is *activated*. The semantics of the suspension and resumption of a process are equivalent to those defined in other operating systems. The *delay* operation is used to suspend a process for a certain time interval. When the time expires, the process is resumed (put back to the ready queue only). A process can be terminated prematurely by *killing* it, perhaps due to its malfunctioning or gets deadlocked.

Besides the above-mentioned primitive process control operations, primitives to control internal concurrency are defined. The primitives include *spawn**, *enter-cs* and

² The lexical environment is the set of bindings of symbols in a LISP process. Dynamic environment is the set of local bindings created during a function execution. See section 4.2.4.

exit-cs. *Spawn** is the primitive to create a coprocess. It accepts a sequence of VCLisp S-expressions and creates a coprocess in much the same way as the creation of a new process. *Enter-cs* and *exit-cs* are associated with critical sections and data integrity. A critical section [Deitel84] is a segment of code not interruptible by external messages or user interrupts. It ensures the integrity of a piece of shared data within a process and among processes. When a critical section is entered, all user interrupts are disabled and all coprocesses are suspended. Only the coprocess executing in the critical section is eligible to be dispatched. Restoration is made after the critical section is exited. The critical section defined in this way is a critical region within the process. However, the process is still subject to timer interrupt and scheduling.

Along with means to specify parallel computations, VCLISP must provide measures for interprocess communication and synchronization. The primitive mechanisms for message passing enable processes to communicate and synchronize with one another. These primitives include *send* and *receive*. *Send* is a nonblocking operation but *receive* is a blocking one. The communication channel is point to point. A process can send a message to another process either by direct naming or indirect naming. Direct naming involves the explicit specification of the receiver process. Indirect naming involves a third entity called the *mailbox*, which can be created and destroyed dynamically. To ensure integrity, only empty mailboxes can be destroyed. The message sent is put

into the mailbox for the receiver to collect. When a message is sent, the sender continues to execute statements after the send operation (nonblocking) regardless what status the receiver process is in. However, when a process attempts to read a message by the *receive* operation and no message is available, the process blocks itself and waits for the message (blocking). This strategy corresponds to the unbounded buffer approach to message passing.

Message passing is an effective means of interprocess communication and synchronization and furnishes a direct support for high level message passing mechanisms (remote procedure call, plug and socket matching) defined in the object-oriented layer of VOCOL. Since VCLISP has a global name space (section 4.2.4) accessible to all VCLisp processes, it is better to define another type of synchronization primitives based on shared resources. The best candidate is the *semaphores* because they are simple and efficient to implement and are elegant in many simple applications. In VCLISP, the traditional operations *P* and *V* are defined on integer semaphores (semaphores with values $n \geq 0$). Mutual exclusion is much simplified with semaphores, compared with solutions using message passing, despite the fact that message passing is able to simulate semaphores and semaphore is not able to solve all synchronization problems [Peterson83].

4.2.4 Name Space

The set of accessible names and their designated storage locations is termed a name space. The name space accessible to a VCLisp process is composed of two parts: *local name space* (LNS) and *global name space* (GNS). The LNS is local to the dedicated process and the GNS is shared by all the processes in the system.

VCLISP is a LISP programming environment, whose name space is made up of cons cells denoting atoms with properties and bindings. Similarly, the LNS of a VCLISP process is the set of names locally accessible to that particular process. It is made up of the *lexical environment* and the *dynamic environment*. The lexical environment is the set of global bindings of atoms and names found in an ordinary LISP process. The dynamic environment is the set of local bindings such as lambda bindings and closures created by special forms like *let* and macros like *do* during the execution of the process. Stated in another way, the LNS is equivalent to the entire name space known to an *isolated* LISP environment which is executed as a process in an operating system.

The GNS is a lexical environment whose bindings are accessible to all VCLISP processes. This space is a pool of globally shared symbols, which features shared resources in a multiprogramming environment or the shared memory in a tightly coupled multiprocessor architecture. This name space

is used for interprocess communication and synchronization, and resources sharing. The names of mailboxes, semaphores and their associated information can be catalogued in the GNS to be referenced. Furthermore, shared data structures can be stored and global functions and macros can also be defined in the GNS.

The evaluation of the binding of an atom normally takes place in the LNS unless specified otherwise. An undefined value will be returned if the atom is unbound. The special form *global* can redirect the evaluation into the context of the GNS. Values can be retrieved from and stored to global names by means of the *global* qualification. Another way to achieve the same effect is through the use of special functions like *gsymbol-value*, *gsetf*, etc. This act is to protect the GNS from being accidentally changed by misspelling within a process. However, a special measure is adopted towards functions and macros. Global functions and macros can always be invoked directly within the process itself, unless they are shadowed by local functions and macros. This means that the lambda definition of a function is searched in the LNS, followed by the GNS, before signalling an error. Global function and macro definitions are reentrant codes for the purpose of sharing. The evaluation of global functions takes place in the LNS of the caller, unless *global* is used. Global functions and macros are defined with special functions *gdefun* and *gdefmacro*.

4.3 AN IMPLEMENTATION ON COMMONLISP

A prototype of VCLISP is programmed in CommonLisp shelled around the VAX/VMS operating system. The system console and other system components such as the visual function editor are written in VCLisp, in CommonLisp and in other languages supported by VMS. Routines written in other programming languages are interfaced to VCLISP as system and library routines.

The implementation of VCLISP is concerned with process scheduling in the interpretation cycle, visual features and the interfaces and interactions with system routines. However, before portraying the implementation, let us take a glance at how LISP works. Then the technique of continuation [Allen78] is introduced as a means to perform explicit LISP process scheduling. Finally, the system interfaces and user interactions are presented.

4.3.1 An Anatomy of LISP

LISP data structures and programs are S-expressions represented by linked lists of cons cells [Allen78]. The basic element is the symbolic atom which has properties (p-lists) and values. As a functional language based on lambda calculus, lambda expressions and function applications play a central role. LISP functions are applied in an environment which can be modified by pseudo-functions (functions with side effects) [Siklossy76].

Many LISP interpreters and compilers are implemented in terms of LISP as LISP itself is both the meta language and the object language. LISP interpreters are often implemented as a read-eval-print loop. The definition of the evaluator *eval* can be found in books as [Winston84; Waite73; MacLennan83]. A typical execution cycle of a LISP function application is composed of four phases: fetching the lambda definition, evaluating the arguments, establishing the lambda bindings and executing the function body in the newly established environment.

4.3.2 Sliced Evaluation of A Form

The VCLisp interpreter is able to schedule and execute a number of VCLisp processes concurrently. With the aid of timer interrupts, the interpreter is able to switch its context of execution and the proper context switching between processes is of the utmost importance. The lack of environment context switching information in most LISP interpreters, including VAX CommonLisp, decreases the explicit representation and manipulation of this information. This is achieved by implementing the interpreter in LISP and splitting the evaluation cycle into slices consuming a finite (preferably short) time, hence the term *sliced evaluation*. The term *slice* refers to the smallest indivisible phase in the course of an S-expression evaluation. Context switching is only allowed between slices. During the evaluation cycle, the split slices are threaded together by the technique of continuation [Allen78].

In the read-eval-print cycle of a LISP interpreter, a form is read, evaluated by the evaluator *eval* and then printed to the output stream. The steps involved in the evaluation of the form can be simple or complex. In the simplest case, the form is just a constant or an atom, and the constant or the value of the atom is simply returned. In case that the form is special, it will be handled by the system in a special way. The arguments may or may not be evaluated and evaluation may occur before or in the course of the computation. The form may also be a system call or a built-in function application. Then the arguments are evaluated and passed to the corresponding routine. It is possible for the system call to be sliced. For example, in a *P* operation (semaphore operation), the calling process may be suspended and later resumed. The *mapcar* function must also be sliced because of its long execution time.

Quite often, the form is a user-defined function application. The four phases in a typical function application are to be carried out and are split into slices. There is a continuation field associated with each slice, indicating what the next action is. At the end of each slice, the continuation field is updated. A phase may be simple, such as fetching the lambda definition of a function. It is indivisible and constitutes a slice. A phase may also be complex, such as evaluating the arguments presented to a function application. The evaluation of arguments can be

split into a sequence of evaluations. Each of them involves the evaluation of a form and is itself an evaluation cycle. A further split is then necessary.

A stack is maintained to hold the intermediate configurations during evaluation. After executing a slice, the current context has been saved on the stack and the interpreter can continue to execute the process by taking the action indicated by the continuation field. If the time slice for the process is used up, the interpreter just turns its attention to and services another process.

The evaluator is defined as a function *vc-eval-loop*. It dispatches the current task to the appropriate executor function according to the continuation field. The nature of the task dictates a direct execution or a split into a sequence of smaller tasks. Control is then yielded back to *vc-eval-loop* and the cycle is repeated indefinitely. The complete evaluation of an example is shown in appendix B.

4.3.3 Process Scheduling

A VCLisp process is represented by a structure containing such slots as process descriptor, process body, lexical and dynamic environment, argument list, partial result and continuation field. The context of execution is reflected by the dynamic environment, argument list, partial result and continuation field.

The process scheduler maintains a process list and a dispatch list. The process list contains all the processes in the system whereas the dispatch list contains the list of processes ready to be dispatched. Each process has its own process descriptor recording its status, resources consumed, remaining time slice and the like. When the evaluator is free, the first process in the dispatch list is dispatched. The evaluator function is then called with the process as argument. This process is executed until its time slice is used up or an exception occurs. It is then returned into a suitable position in the list, and the next process is dispatched.

When a process control function such as *create* is executed, the status of the appropriate process, and possibly the dispatch list, are modified. The process being interpreted may be suspended as a result, like the execution of a *suspend* operation. Communication and synchronization primitives can also affect the process status and scheduling. For example, a *P* operation on a semaphore may cause the calling process to be suspended and queued at the semaphore. All these process related operations are handled by the interpreter and the scheduler.

4.3.4 Icons and Interfaces

In the prototype, icons are created by the IE. Constrained by terminal resolution and implementation complexity, icons are assumed to be composed of a matrix of

32 by 32 dots. The actual appearance is the same as the one shown in figure 4.3. The bitmap for the icon is stored in one file and the control information like the name, and logical operation are stored together with the drawing procedure in another file. The system of icon is handled by an icon manager with which the programmer can perform operations (figure 4.6). However, the deletion of an icon is dangerous because the dangling pointer problem poses a severe threat. Consistency checking can be performed but this is slow and tedious, as all functions and object classes must be examined in detail, as well as the contents of symbols in the environment, which is even more disastrous.

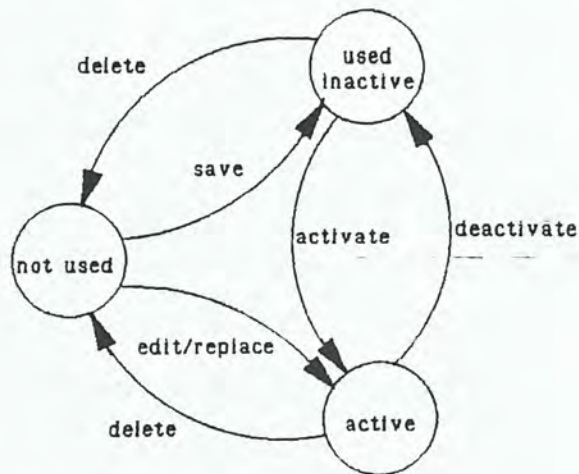


Figure 4.6 Operations on Icons

A symbol can be identified by name or by icon. Therefore the mapper must hold both attributes. The wildcard notation ? is used to match the unknown part. Entries in the mapper tables are updated when the remaining part of the pair is known. As a special arrangement, local variables and

parameters are treated differently. Since these symbols are local symbols, their effects to the mapper tables should not be permanent and their associated entries are deleted after translation.

The interaction between VCLisp and the programmer is made via the user interface process and the system console. Keyboard inputs are interrupt driven. Processes are selected by means of window and mouse selection. Mouse inputs and menus are supported in certain system applications such as the icon editor.

4.3.5 Concluding the Prototype

After implementing the prototype, a few applications are developed. Some of them are presented in chapter 7. As a simple benchmark, it is found that the prototype runs at an order of magnitude slower than an ordinary LISP environment because of the need of multi-threading and the simulation of LISP data structures and operations in terms of the language LISP itself. The management of icons and mouse interrupts also burden the VMS system with extra workload, further reducing the run time efficiency. Nevertheless, with dedicated multi-processor machines and well-developed VCLisp interpreters and compilers, it is highly possible that the prototype runs at an order of magnitude faster than on an ordinary LISP machine!

CHAPTER FIVE

OBJECT-ORIENTED LAYER

5.1 AN OVERVIEW

The object-oriented layer is built upon VCLISP. Central to this layer are classes and autonomous instance objects connected together as a network. Object methods are defined in the form of LISP functions, with the same flexibility of LISP functions. Attributes and methods are selectively inherited from parent classes. A network topology is allowed so that a class can have more than one parent classes, resulting in multiple inheritance. User-defined relational links connecting different classes govern the rule of attribute and method inheritance. Interobject communication is carried out by means of message passing. Two message passing mechanisms, explicit and implicit, are provided. Furthermore, the notion of concurrent processes is unified with the notion of objects. A special type of object, denominated *function modularization object*, serves as the interface to the VCLISP environment.

The network of objects is shown in figure 5.1.

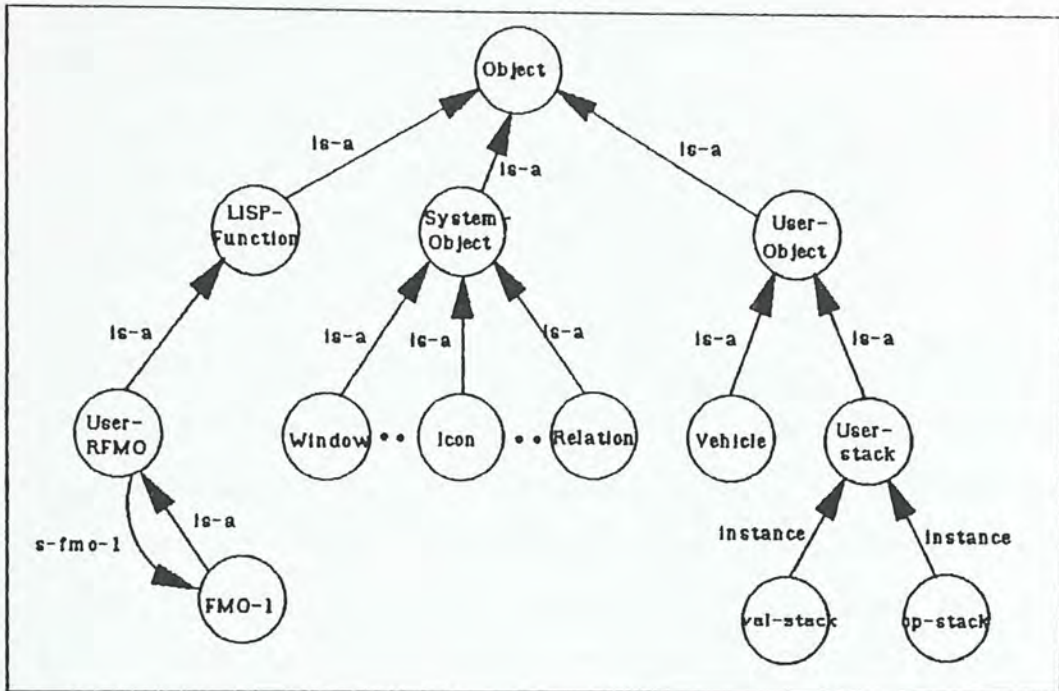


Figure 5.1 The System of Objects

The inheritance graph is made up of three subgraphs, each of which is responsible to its own system functionality. The subgraph rooted at *LISP-function* contains all the intrinsic (system defined) functions of LISP. The child *user root function modularization object* contains all user-defined global LISP functions accessible to all objects. *System-object* is the parent class of all system objects provided by the object-oriented layer, such as windows, relations, and icons. These objects are vital to the proper functioning of the object-oriented layer. The network of objects rooted at *user-object* is created entirely by the programmer. This collection of objects constitutes the object program specified by the programmer.

5.2 ELEMENTS OF THE OBJECT-ORIENTED LAYER

In this section, we will take a closer look at the various features of the object-oriented layer.

5.2.1 Object

The definition of an object instance is contained in its class, which serves as a mould for creating instances. A class is composed of three sections: *identification section*, *attribute section* and *function definition section*. As an example, the definition of a class called *node* is shown in figure 5.2. It is also part of the application on binary tree manipulation to be discussed in chapter 7, where a full narration on the definition of *node* is presented.

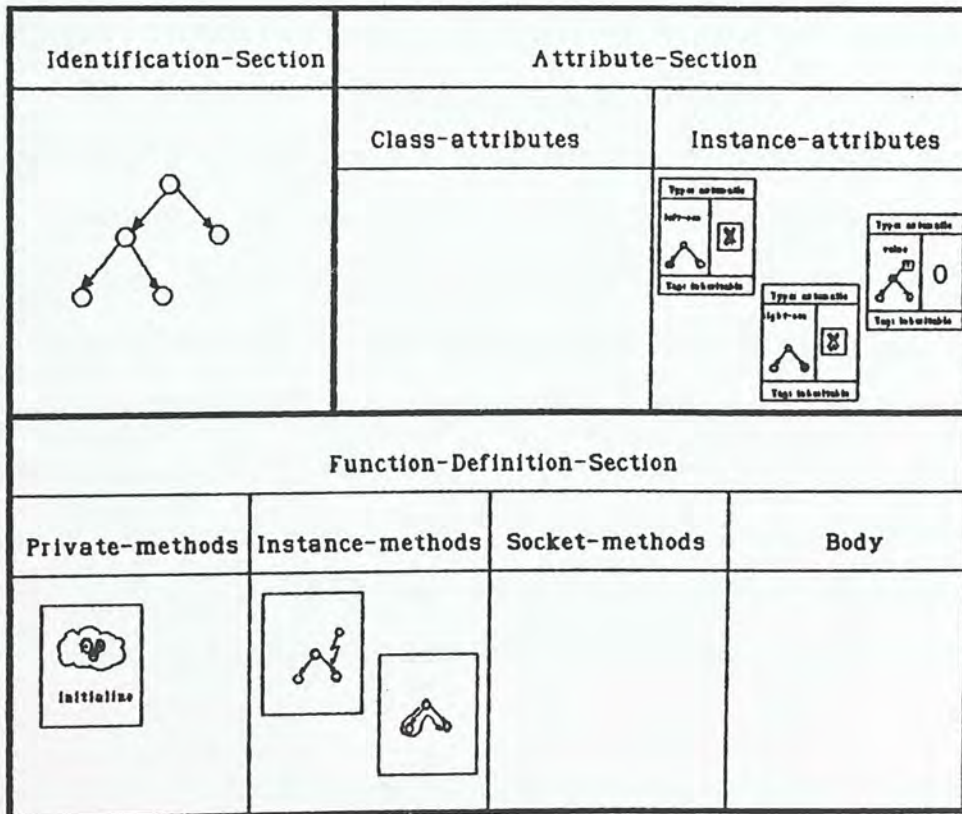


Figure 5.2 The Class Node

The identification section of a class contains the icon which identifies that particular class. There are two kinds of attributes, *class attributes* and *instance attributes*, to be defined in the attribute section. An attribute is a slot in which a value, an object name or an *object pointer* can be stored. An object pointer is the means of identifying and accessing another object, though an object may be referred to by its name. The function definition section is composed of four subsections: *private methods*, *instance methods*, *socket methods* and *body*.

A class attribute is an attribute shared among all instances belonging to that class. It is accessible to all of them and serves the purpose of a global variable of limited scope (not accessible to instances of non descending classes). An instance attribute is private to an instance. Every instance belonging to a class has its own copy of the attribute, which is only accessible to outsiders via the *protocol* of the object. An instance attribute can be either *manual* or *automatic*. Accessor functions¹ for an automatic attribute are defined automatically by VOCOL, analogous to the automatic definition of access procedures for a LISP structure. In this way, the programmer is relieved of the burden of having to define many accessor functions as the protocol. For manual attributes, accessor functions must be

¹ Accessor functions come from the concept of access procedures [Winston84] in data abstraction. Constructors create new data objects; selectors retrieve stored information in these objects; mutators modify their stored information. The three of them comprise the set of access procedures. In VOCOL, accessor functions are composed of selectors and mutators. No constructor is needed for object attributes which are defined explicitly in the definition of the object class.

defined explicitly. Thus the programmer is able to control the amount of privileges granted to other objects. A manual attribute without any accessor function becomes a private attribute. An instance attribute can also be either *inheritable* or *noninheritable*, stating whether the instance is inheritable to child classes.

The various methods in an object are VCLisp functions capable of performing some computations. The arguments to a method can be LISP objects or object pointers and the method body is the same as the function body of a VCLisp function. According to the method access mode, methods can be divided into private methods, instance methods and socket methods, all of which are under the function definition section. The private methods of an object are those used internally only. They are never accessible to other objects. The instance methods and socket methods constitute part of the external protocol of the object, the remaining portion being automatic accessor functions. Instance methods are invoked by *explicit messages* and socket methods by *implicit messages*. They will be discussed in section 5.2.3.

Finally, the last subsection of the function definition section is the body which is a sequence of VCLisp S-expressions. The body of an object is the code section of a process encapsulated by the object. Viewed in another way, an object with a non-empty body is a process integrated with protocols for communication with other objects. Objects in VOCOL are active.

A class is defined by means of the *class editor* (CE), which is a structured editor similar in structures and functions to the VFE. The programmer can fill in the templates with appropriate attributes. Methods can be defined with the VFE but in the context of the CE. Extra message passing primitives and object concepts are then available.

5.2.2 Relation and Inheritance

In an object-oriented environment, there exist relations among various objects. In VOCOL, all relations are *binary*. They describe the relationship between two objects only. An instance object must belong to one and only one class, through the *instance* relation. This classification is natural and trivial. The relationship between two classes is often of a hierarchical sense, such as the *subclass* relation in Smalltalk. In VOCOL, the equivalent relation *is-a* is defined as a built-in relation. In addition, the programmer can define new relations for connecting classes together, drawn as directed edges. Thus the representation of classes and their relations becomes a network topology or a digraph and this representation is called the system graph (figure 5.1).

The relationship between classes has an impact on the inheritance mechanism - what properties the child class can assume from its parent. In VOCOL, a class can have more than one parent class. In the terms of object-oriented

programming and in the context of inheritance, this is referred to as the *multiple inheritance* phenomenon. The child class is eligible to assume the existence of attributes from its parent classes according to certain rules, to be governed by the relations between the classes. These rules are used to resolve conflicts among inheritable attributes from different parent classes.

In the system graph, each user-defined relational link between two classes is itself an instance of a descendant class of the class *relation*. Therefore, a relational link inherits all the attributes of *relation*. As an instance, it can have an iconic appearance or a name. The attributes in a relational link are used to control the behaviour of inheritance. The attribute *priority* determines the order of method searching. A sequence of nodes which depicts the order will be generated according to the breadth first algorithms.

The Breadth First Algorithms

Simple breadth first search is adopted based on the value of the priority. The sub-branch with higher priority is explored before the others. No total ordering on all nodes at the same level is attempted. For example, in figure 5.3, the search order will be:

<level 0> A

<level 1> B,D,C

<level 2> F,G,E,I,H (the parents of B are searched in preference to parents of D and C)

<level 3> J.

The rationale is that the nearer ancestors should be given greater weights as they are related more closely with the object concerned. The first applicable method encountered is considered the candidate to be applied. Priority values range from 0 to 255 and the built-in relation *is-a* has a priority of 128, a median value. This rule is called the *breadth first only* (BFO) algorithm.

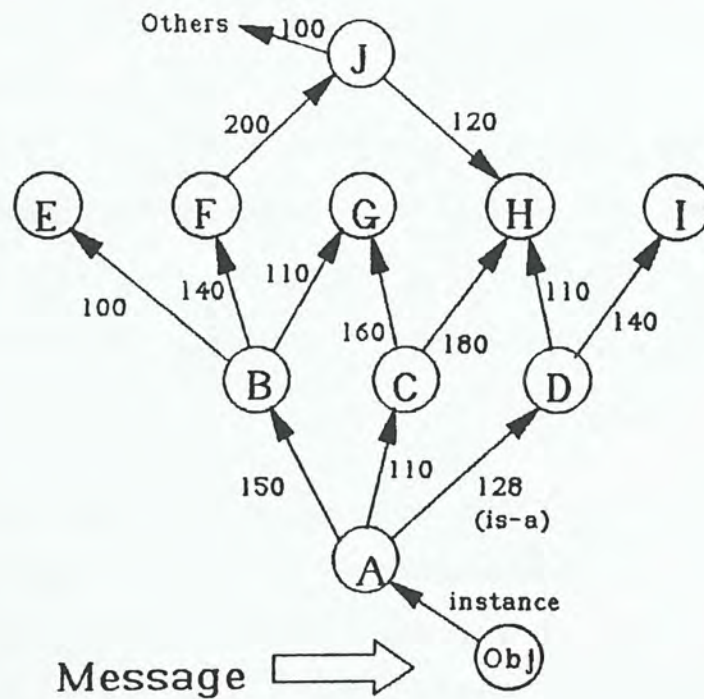


Figure 5.3 A Network of Object Classes with Priorities

The BFO algorithm is a counterpart of the *breadth first join* (BFJ) algorithm, which assumes an acyclic digraph.

The different branches to be searched reunite gradually and they must be synchronized with one another at those joints, such as the nodes G and H in figure 5.3. In the figure, the search order will be:

A,B,D,C,F,E,I,G,J,H, assuming that J has only sons of low priorities.

Here, the joint H is synchronized and visited after its children J and D.

In fact, both algorithms BFO and BFJ can be merged into the algorithm *breadth first search* (BFS):

1. put the node where the search is originated in a list,
2. pick up the first unmarked node in the list,
3. mark the node,
4. arrange all the children of the nodes according to priority in descending order,
5. append the list of children to the tail of the list,
6. repeat steps (2) through (5) until all nodes in the list are marked.

After applying the algorithm BFS, a list of nodes is obtained at step (6). The sequence for the BFO algorithm is the list with all duplicate nodes removed, keeping only their first occurrence. The sequence for the BFJ algorithm is obtained in much the same way, except that only the last occurrence of duplicated items is retained. The list of nodes generated by BFS on figure 5.3 is

(A,B,D,C,F,G,E,I,H,H,G,J,H,ancestors of J). Notice how this list reduces to the BFO and BFJ sequences stated above.

The BFJ algorithm works better and in a more consistent way when the system network is an acyclic digraph, because synchronization occurs at joints and the inheritance pattern is more natural and structured. On the other hand, in graphs where there are cycles, it is more natural to utilize the BFO algorithm. In VOCOL, the BFJ algorithm is adopted by default, unless specified otherwise.

In consistent with method searching, the set of attributes inheritable to a class is determined by the *add-list* and *remove-list* attributes, ordered by *priority* in a breadth first manner (with the BFJ or BFO algorithm). The set of attributes inherited to a class is the set of inheritable attributes in the parent plus attributes dictated by the *add-list* less attributes designated by the *remove-list*. The order of the operations is determined by the attribute *order*. By default, a child class can inherit all attributes marked inheritable from its parent classes when their relationship is *is-a*. An example of the definition of some relations and their usages are shown in figures 5.4 and 5.5.

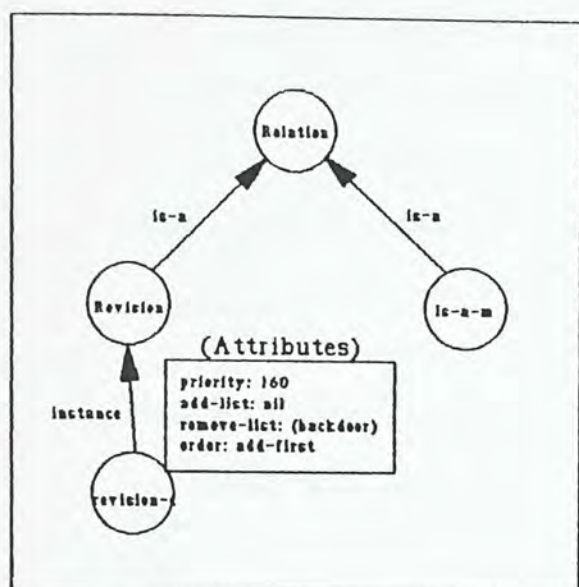


Figure 5.4 Definition of Relations

Two child classes of *Relation* have been defined: *revision* and *is-a-m*. An instance of *revision* can be used to relate two classes in which one is the revision of the other. For instance, Civics-82 (an imaginary brand of car) is a revision of Civics-80. We then use the relation *revision-c*, an instance of *revision*, to relate them, with Civics-80 as the parent. The relation has priority 160, empty

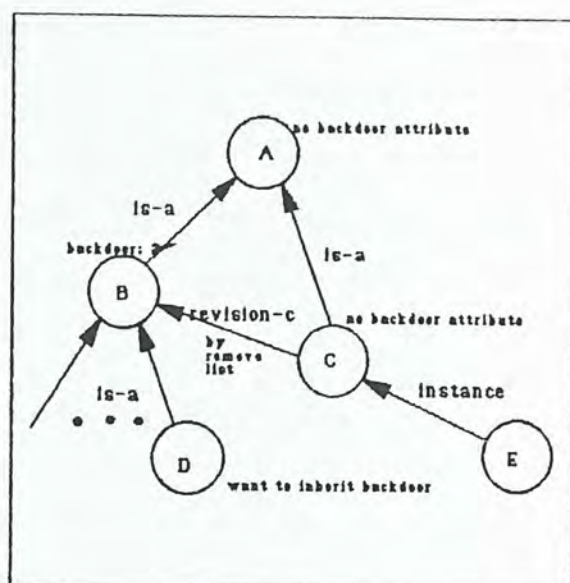


Figure 5.5 A Usage of Relations

Class A (say, small-car) does not have a *backdoor* attribute but B (say, Civics-80) does. Class D (say, those Civics-80's sold in Hong Kong) and other child classes of B wish to inherit this attribute. Class C (say, Civics-82) is a revision of B and has no backdoor attribute. A value of zero is not desirable. Furthermore, we want to guarantee that the attribute is not inherited

add-list and *remove-list* automatically. Thus B with attribute *backdoor*. removes the attribute from Inheritable attributes to the inheritable set to C child class are those of the through the use of *remove-list* parent class with *backdoor* in *revision-c*. removed and Civics-82 will have no *backdoor* attribute even if Civics-80 has.

The system graph is maintained by means of the *system graph editor* (SGE), with which the programmer can change the connections between various classes. He can also change the relation and the associated attributes between two classes. A change in the system graph often causes a change in the inheritance lattice. The recomputation of inheritable attributes is therefore necessitated.

5.2.3 Message Passing

The execution of an object program is reflected by interobject communication, which is achieved by means of message passing, as in interprocess communication. There is a variety of message passing mechanisms, formed by the combination of various message passing behaviours: synchronization (wait/no wait), threading, reply, and naming. A message can be *synchronous* or *asynchronous* [Andrews83]. The sender of a message waits for the completion of a synchronous message before proceeding. The sender of an asynchronous message can proceed after sending a message,

regardless of the status of the receiver. A multiple loci of execution may or may not be created through the sending of a message. The receiver may or may not be terminated at the completion of the message. A message can optionally return a value (reply). The sender and receiver can be named directly or indirectly. For example, the remote procedure call mechanism in Smalltalk is synchronous, single thread all the way, replying, direct naming; the rendezvous in Ada is synchronous, no division of threads, nonreplying, indirect naming; the future in ABCL/1 [Yonezawa87] is asynchronous, single then multiple and then single thread, replying, direct naming.

In VOCOL, two message passing mechanisms are defined: explicit and implicit. *Explicit message passing* is an one-way consent, two-way communication whereas *implicit message passing* is a two-way consent, one-way communication. In explicit message passing, the sender can send a message to a receiver. A result is generally returned. In implicit message passing, both the sender and the receiver must agree (mutual agreement) on a type which identifies the communication channel through which a message is sent. No result is required to be returned.

Explicit Message Passing

Explicit message passing mechanism bears much similarity to the message passing mechanism in Smalltalk. An object is actually making a remote procedure call when it sends an

explicit message to another object. To send an explicit message, an object makes the function call

`(send-message receiver-object method argument-list)`.

When an object is sent an explicit message, it checks its protocol for the existence of the method designated by the message. If the method is found, it is invoked. Otherwise, such a method is sought among the parent class chains ordered by priority with the breadth first algorithms until one is found. An error will be signalled if none of the inheritable methods of that particular object is applicable. The invocation of the designated method is catered for by a new thread whose life span is the same as the computation covered by the method. The sender then suspends itself waiting for the reply of a synchronous message. In the asynchronous case, no reply is expected. The value returned by a method is either an object pointer or a VCLisp entity, which can be used in the assignments of attributes and variables.

The receiver of a message can be a *pseudo-object*. A pseudo-object is the name of a special variable which denotes a context-dependent object. For example, the pseudo-object *self* when referenced by object A denotes object A but it denotes object B when referenced by object B. In VOCOL, the pseudo-objects *self*, *super* and *sender* are recognized. The pseudo-objects *self* and *super* are very similar to the corresponding *pseudo-variables self* and *super* in Smalltalk. *Self* refers to the object itself whereas *super* refers to the parent class (*superclass*) of the class where the applicable

method is found. The mechanism for *super* is to cross the boundary of the applicable method to a more general one. *Sender* can be referenced by the receiver of a message in case that the receiver acts as an agent on behalf of the sender of the message. With *sender*, the receiver is able to identify who the message sender is and take the appropriate actions for it.

A message is normally sent to an object instance but a class can also be a receiver. A class can recognize the special message *make-instance* by creating a new instance of it and sending the instance the message *initialize*. The list of arguments from *make-instance* will be handed over to the message *initialize*. When a class is sent a message of other type, the net result is equivalent to a sequence of actions: creating a new temporary instance; invoking the method designated by the message; killing the instance; and passing back the returned value. This arrangement is to limit the effect caused by the message. All changes are local within the temporary instance, except for a change in the class attributes, if any. Furthermore, it is in consistent with the responses of FMOs (section 5.2.5).

So far we have only discussed sole message receivers: instance and class. In fact, a message can be broadcast to a collection of objects. A broadcast of message is effected by
(**broadcast** *rec-objects-or-classes method argument-list*).

The first parameter of *broadcast* can be a list of objects who will be sent the same message concurrently. When a class receives a broadcast message, the message is redirected to all instances of the class. A wildcard receiver allows all instances in the system to be sent the same message. It must be pointed out that *broadcast* is asynchronous and there is not any reply.

Implicit Message Passing

Implicit message passing involves the concept of *plug* and *socket*. Let us take a glimpse at electrical outlets (sockets) and plugs in our daily life. There are many types of plugs and sockets. Some are big and some are small. Some have three pins and some have two pins. Some have rectangular pins and some have circular ones. They differ in shapes and only those with the same shape can be used together. The same concept has been adopted and formulated in VOCOL, in which object instances are capable of creating and destroying plugs and sockets dynamically.

A plug is composed of a type and zero or more slots. Each slot corresponds to an attribute. The plug is identified by the type, either in the form of a name (symbolic name) or a shape (an iconic appearance of the plug). In addition to the slots, it has also a list of arguments known as the *message packet*. A socket is also made up of a type and zero or more slots.

When we say that a plug matches a socket, we mean the shape and the contents of all slots in the plug are *matchable* with those of the socket. The use of a wildcard value is allowed for the matching purpose. In case of multiple possible matches, one of them is selected with the *best match criteria* to be discussed later. When a plug matches a socket, a communication channel is established. The corresponding socket method is invoked with the content of the message packet as method parameters. The term implicit message passing refers to the course of message packet transfer and method invocation. After the matching procedure, both the matched plug and the matched socket are removed to disable further implicit message concerned with them. No reply is expected and the mechanism is asynchronous. A new thread is created for the execution of the socket method.

In parallel to the broadcast mechanism in explicit message passing, a multiple number of plugs and sockets can be created at one time, causing a multiple number of implicit messages to be sent. When a *multiple-plug* is created, the owners of all matchable sockets will be sent the same implicit message with the message packet. Similarly, when a *multiple-socket* is created, each matchable plug will cause an implicit message to be sent to the destined object, resulting in a very high degree of parallelism.

Plugs and sockets are created and destroyed dynamically. A socket method will never be invoked if the instance does

not create a socket for it even though it has the associated socket method defined. Conversely, a socket is invalid for an instance if no associated socket method exists. In either case, an exception is raised.

The Best Match Criteria

The matching between a plug and a socket is determined by the best match criteria. A plug and a socket with different names are never matchable. Similarly, a plug and a socket with different number of slots are never matchable. In addition, a plug and a socket are not matchable if any pair of their attributes does not match. All the other plug and socket pairs are candidates for matching. The *best matched pair* (BMP) is trivial when there exists a candidate with an exact match of slot values without wildcard attributes between the plug and socket. The one with more attributes is preferred in case of conflicts. If such a candidate does not exist, the *best match algorithm* is applied to determine the best one.

Before presenting the best match algorithm, let us define a few terms. A *single wildcard matching* (SWM) between a plug and a socket is a matching of the content of a slot of the plug and the corresponding slot of the socket when one of them is a wildcard and the other is a known attribute. A *double wildcard matching* (DWM) is a matching between the two slots, both of which contain wildcards. The degree of mismatch is measured by three quantities: total number of

DWMs, total number of SWMs and total number of attributes. The pairs with the most number of DWMs are the worst. Pairs with the same number of DWMs can be further differentiated by checking the number of SWMs. Those with more SWMs are worse. When both DWMs and SWMs are the same, the pairs with the least number of attributes are worse. In the criteria, DWM has priority over SWM because a DWM implies a possible match in the second order² but a SWM implies a match in the first order only. A better matched pair should be a more constrained one.

Concluding the criteria, we have the best match algorithm formulated. First of all, we transform the three quantities into a 3-tuple of the form $(-DWM, -SWM, \text{attributes})$. Define the relations *greater* and *equal* on the tuples.

$$\begin{aligned} (a,b,c) > (d,e,f) & \text{ if } a > d \\ & \text{ or if } a = d \text{ and } b > e \\ & \text{ or if } a = d, b = e \text{ and } c > f \end{aligned}$$

$$(a,b,c) = (d,e,f) \text{ if } a = d, b = e \text{ and } c = f$$

The highest rank tuple on a set of n 3-tuples $\{t_1, t_2, t_3, \dots, t_n\}$ is t_r iff

$$\forall t_i, i \in \{1, 2, \dots, n\}, t_r \geq t_i.$$

² The second order match is similar to the unification of two uninstantiated variables in Prolog, in which the value is still indeterminate, whereas the first order match is the unification of an uninstantiated variable with an atom or structure, in which exact substitution is found.

A list of n 3-tuples $(t_1, t_2, t_3, \dots, t_n)$ is ordered iff

$$\forall i, j \in \{1, 2, \dots, n\}, j > i, t_i \geq t_j.$$

The ordered sequence for the tuples reveals the candidates for the BMP. The tuples with the highest ranks are the best. In case of multiple BMPs with the same highest rank, one of them is selected arbitrarily. The trivial case is also embedded by this formula. With zero DWM and zero SWM, those candidates must rank first because the first two elements can never be positive.

Furthermore, no explicit sorting is needed because only the highest ranking tuples are required. A list may be used to store the partial result for the best ones at hand. As a result, this algorithm is of $O(n)$ and is quite efficient (most unification algorithms, such as [Rich83; Wise86] are $O(n^2)$). In this way, a simple pattern matching capability is supported. Although this feature is not as powerful as the unification algorithm in Prolog, it is easily implemented without much sacrifice in complexity and efficiency.

5.2.4 Concurrent Process

In VOCOL, each object instance can behave in the way of a process. This is accomplished by defining an object with a non-empty body. When an instance of a class is created and activated, the sequence of code embraced in the body is executed as a concurrent process. System calls to process management operations in VCLISP are available with similar

semantics. The process denoted by the body is suspended when *sleep* is encountered. *Sleep* is a combination of *suspend* and *resume*. When a process goes to sleep, it suspends itself. It is awoken by a message. In that case, the message is serviced and the process is placed back in the ready queue. When the body code of an object instance finishes execution, the instance is not removed from the system but sleeps indefinitely instead. Conceptually, it remains dormant until a message comes in. It then services the message and sleeps again. An object instance is only removed by explicitly *kill*ing the instance.

We may view an object instance to be composed of data part, protocol part, and process part. The process part obeys semantics of VCLISP processes. At the end of the process body, we can regard the process denoted by the process part as having terminated. However, since objects are non-volatile entities, the data part and the protocol part still remain, giving the illusion that the object "sleeps indefinitely".

With the concept of concurrent process and process body, objects unify data structures, procedures and processes. At one extreme, all objects in the environment have empty bodies and a purely object-oriented environment is resulted. At the other extreme, object bodies contain extensive amount of codes, ending with the operation *kill*. In this manner, the system becomes a typical multi-programming environment.

Quite often, the system contains an intermix of both of them, in various states of execution, corresponding to a typical VOCOL environment.

5.2.5 Function Modularization Object

Function modularization objects (FMOs) are descendant classes of the system class *LISP-function*. A FMO is a class without any instance, instance method, socket method or body. It is a module containing a number of encapsulated functions. We can imagine a FMO to be a package and the bundle of VCLisp functions as instance methods though it is meaningless to use the term when instances do not exist. A child class of *LISP-function* is the *user root function modularization object* (*user-RFMO*) which is the ancestor class of all other FMOs. All these FMOs and *LISP-function* constitute the interface to the visual LISP layer. VCLisp functions contained in a FMO can be invoked by sending it a message and the programmer just regards the FMO as a class receiver. At the completion of the function specified by the message, the "temporary" instance is destroyed. VCLisp functions in FMOs can be defined to be internal or external. Only external functions are visible to outsiders. Internal functions can never be imported by other FMOs.

All VCLisp intrinsic functions (system functions) are defined and bundled in *LISP-function*. This FMO resembles the structure of the package *lisp* whereas the package *user* is contained in *user-RFMO*. By default, VCLisp functions defined

in the object-oriented layer by the programmer are added directly to the *user-RFMO*. All user-defined VCLisp functions in the VCLISP layer are encapsulated in *user-RFMO* as well. In addition to message passing, a direct invocation of VCLisp functions is allowed, and the syntax is the same as any other LISP function application. The function call can be prefixed by a class name of a FMO, of a similar syntax to the nomenclature of a packaged symbol. In the absence of a FMO prefix, these function calls are directed to the *user-RFMO*. The search for the applicable method follows the rules of inheritance described in section 5.2.2. Since *LISP-function* is a parent class of *user-RFMO* with a median priority 128, the programmer may let *user-RFMO* inherit from *LISP-function* such that if the function is not user-defined, try whether it is a system function. This search can be continued if there exist other FMOs until the whole subgraph rooted at *LISP-function* has been attempted. In this way, FMOs provide a means of access to the VCLISP environment with a functional hierarchy.

There are slight differences between the two mechanisms of accessing and using LISP functions. The direct application of LISP function will be performed in the context of the caller. This means that the function definition is executed in the caller's environment, with full access to its name space. However, sending a message to a FMO will cause a "temporary instance" to be created and the method is executed

in the context of the "instance" on behalf of the message sender. The effects of the method execution are localized to the receiver FMO.

Packages in LISP can be used to organize large LISP programs into modules of manageable sizes with fewer interactions. However, name conflicts and indiscriminate uses of *import* and *export* tend to render a LISP program complex. Organizing VCLisp functions into FMOs serves to limit such effects. When an FMO is related by an *is-a* relation to another, the effect is similar to the "using" of the parent class package, functions in the child class have priority over parent classes. The unrestricted uses of *import* and *export* can be relieved.

5.3 OBJECT-ORIENTATION ON VCLISP

VCLISP is a stand-alone programming environment supporting the visual, functional and concurrent paradigms. The object-oriented paradigm delineated in VOCOL is developed on VCLISP. The architecture of the object-oriented layer built upon VCLISP can be shown in figure 5.6,

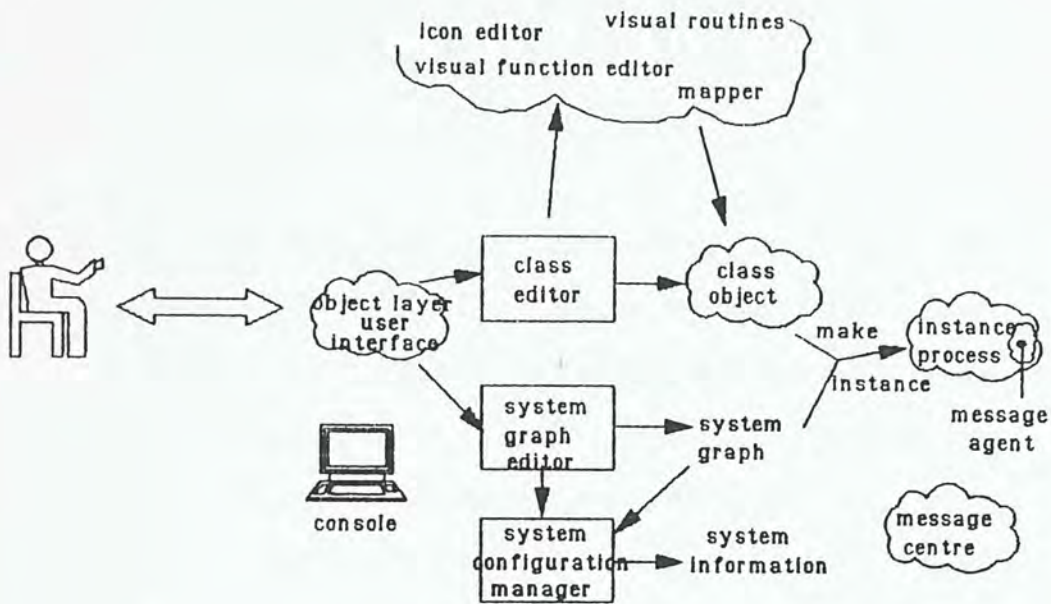


Figure 5.6 The Object-oriented Layer

This layer has a direct relationship with the VCLISP layer and there are interactions between them. The class editor (CE) must communicate and exchange information with the visual function editor (VFE), the mapper, and so on for the proper functioning. The object-oriented layer user interface process accepts and processes user requests. It invokes the CE to manipulate a class definition and the system graph editor (SGE) to maintain the system graph topology which governs the inheritance among object classes. As a new class definition is created, a process for the class will be created automatically to cater for the creation and management of its instances. When the system graph is modified by the SGE, the *system configuration manager* (SCM) keeps track of the new topology and makes appropriate changes to system information, possibly resulting in a recomputation

of the inheritance lattice. These components keep in close contact with the icon editor (IE), VFE and mapper to assure proper system functionality.

In VOCOL, the access of instance attributes and class attributes occurs in context. This means that the programmer can just use the attribute name in an assignment statement and it is the duty of the CE to replace the access with the appropriate attribute access functions like *set-inst-attr* and *get-class-attr*. This serves to provide the programmer with a simpler view with names and as a result, the name spaces for objects and LISP symbols are not distinct.

The special process named *message centre* (MC) monitors and controls the traffics of object level message passing. Every message sent in the object-oriented layer is redirected to the MC. Message related activities such as *make-plug* and *make-socket* are also directed to the MC. The MC takes the appropriate actions according to the message type. With the aid of the SCM, the proper receiver of the message is searched and the message is then sent to the receiver object(s). Reply destination is also included if the message dictates a reply. Operations concerned with plugs and sockets are examined by checking whether there is any matchable pair. The owner of the socket of the best match pair is then sent the implicit message, after sending messages to the MAs of the owners to remove their plug and socket.

Each instance is established as a process augmented with a *message agent* (MA) coprocess. The MA accepts incoming messages, queues them up and creates coprocesses to execute the designated methods. A reply is also handled by the MA during the creation of the coprocess, by appending a send operation at the end. Serialization is enforced by mutual exclusion. This ensures that a method excited by an incoming message can be serviced only after the completion of the previous one.

The class process also has a message agent to deal with the creation of new object instances, new processes with the associated code segment and MA coprocess. The MA for the class process is able to determine the effect of receiving the message sent to a class receiver and the appropriate action taken, as described in section 5.2.3 above.

The system console provides the familiar functionalities of system monitoring and performance tuning. The operator is able to take the appropriate measures when the execution of the system gets into stuck, such as a deadlock in the processes denoted by the object instances.

CHAPTER SIX

APPLICATION LAYER

6.1 PROGRAMMING TOOLS

To complete the design of VOCOL and to enhance its capabilities, tools which can assist programmers to develop programs are provided in an extra layer built upon the object-oriented environment. They are to be equipped with a variety of such tools. In addition, this layer serves the purpose of demonstrating the flexibility and usefulness of the environment.

In this chapter, two possible applications are considered and discussed. The first one is a Prolog interpreter and the second one is an expert system shell.

6.2 PROLOG INTERPRETER

Prolog is an elegant logic programming language. It would be an advantage to the programmer if he is able to write logic programs. The implementation of a Prolog interpreter is quite straightforward, with a unifier and the appropriate unification algorithm. It would be more advantageous if the inherent parallelism in a Prolog program can be exploited. Since the VCLISP and the VOCOL environments allow a high degree of parallelism, we would

like to explore the ways of mapping parts of the inherent parallelism in a Prolog program into the notions of objects and processes in VOCOL. This is the purpose of this section.

The semantics of ordinary (sequential) Prolog rely heavily on the sequential execution of clauses and subgoals. The execution order of the alternatives of a goal in a Prolog program is a determinant to the correct behaviour of the program. The control primitive *cut* also assumes a sequential execution. Slight modifications must be made to the syntax and semantics of Prolog to allow for the possible parallel execution of a Prolog program. Notations to express parallel computation must be defined and mechanisms to enforce the necessary sequential computation and to replace the *cut* primitive must be provided [Wise86; Gregory87]. As a result, different parallel Prolog languages are defined and systems for their execution are implemented.

There are many types of parallelism in a Prolog program. Some of them are static and some are dynamic. Static parallelism is easier to exploit by analyzing the static structure of the program. Dynamic parallelism depends on the run time instantiations and the detection algorithm creates a substantial overhead. The simplest form of parallelism is the OR-parallelism among clauses with the same predicate head and arity. All the alternatives of such a head can be explored in parallel. The set of solution is the union of solutions yielded by these alternatives. Independent and all-solution AND-parallelism [Gregory87] can also be

exploited within the subgoals of a clause. Subgoals exhibiting these AND-parallelism can be unified simultaneously. The sets of bindings obtained from the subgoals can then be joined together to produce the set of bindings for the underlying clause, or obtained through pipelined producer and consumer processes (the generate and test strategy). Examples of the forms of parallelism are shown in figure 6.1.

<p>p(X) :- q(X). p(X) :- r(X). p(X) :- s(X).</p>	<p>p(X) :- q(X), r(X).</p>	<p>p(X) :- q(X), r(Y), s(X,Y).</p>
	<p>all-solution AND-parallelism</p>	<p>independent AND-parallelism (q and r)</p>
<p>OR-parallelism</p>		

Figure 6.1 Some Forms of Parallelism in a Prolog Program

The most natural scheme to represent the parallel execution of a Prolog program is the AND/OR process tree, an example of which is shown in figure 6.2 with query p(X). In a AND/OR process tree, only the leaf processes can be active. The internal nodes are suspended processes waiting for the reply from their children. They can be marked AND or OR. The exploration of the OR-parallelism results in the creation of an OR-process with each alternative made into a child process. The subgoals within a clause leads to the creation of an AND-process with the head as the parent and the subgoal processes as children. The granularity in this scheme is, however, very fine.

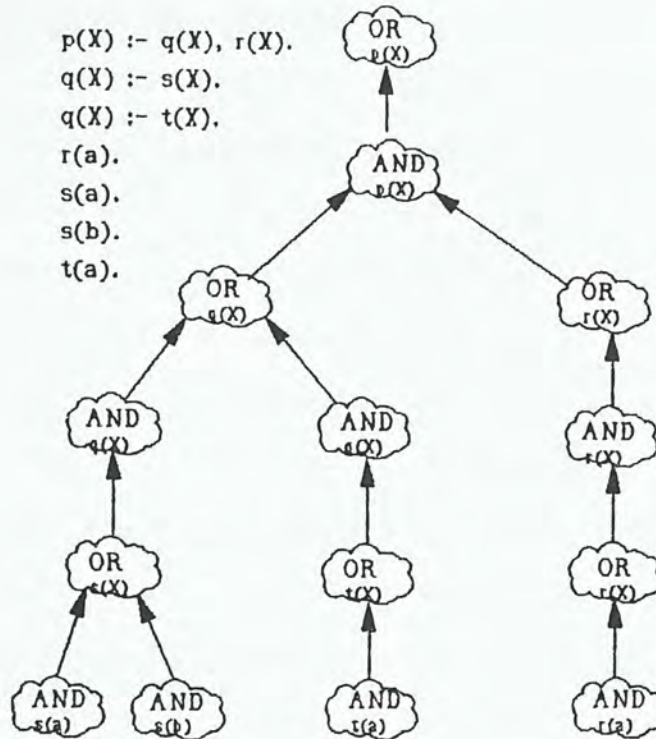


Figure 6.2 AND/OR Process Tree

A possible scheme to implement a parallel Prolog interpreter on VOCOL is to define a class for each clauses with the same predicate head, corresponding to the static structure of the Prolog program. For each dynamic invocation of a goal or subgoal, an instance of its class is created. Then messages are sent to *self* to initiate a method (coprocess) for each alternative of the goal. This corresponds to the OR-parallelism among alternatives. For each subgoal, if the resources are still available, a message is sent to the class of the subgoal to make a new instance of it. After receiving the reply of binding, the binding is passed to the next subgoal. The coprocess in charge must report a failure or the current set of bindings to its parent at the end of the clause. When all alternatives have been

tried, the coprocess is terminated and the result is reported back to the parent through the lexical environment. In this way, an OR-process corresponds to an instance of the class of the goal predicate. The AND-processes are replaced by coprocesses within the process represented by the object instance. An alternative scheme is to let each alternative be an instance and hence associated with a process, in much the same way as a *dframe* in EPILOG [Wise86]. Back-unification then takes place to hand back the list of bindings. A third scheme is to use coprocesses throughout, to effect the AND/OR process approach. Coprocesses have the advantage of a shared lexical environment so that it is not necessary to pass the result back and forth.

It must be pointed out that the granularity greatly affects the amount of parallelism available and run time efficiency. Too fine a granule will burden the system with excessive communication and process management overheads. Too coarse a granule will wipe out the inherent parallelism. An optimal grain size has yet to be determined. Algorithms to detect independent AND-parallelism and other forms of parallelism can be defined in the root class *Prolog* for code reusing, utilizing the inheritance capability of an object-oriented system. Extra control mechanisms are needed to enable the joining of partial bindings from the independent subgoals and to effect the pipelining of partial bindings. To prevent the system from being cluttered with object processes, explicit control strategies such as thread management should be adopted.

6.3 EXPERT SYSTEM SHELL

Expert systems find their applications in a wide variety of fields, such as medical diagnosis, risk analysis, and circuit design. Among these applications, one is automatic programming. Expert systems to build programs have been designed and implemented. With an expert system shell as a tool, it is ready to explore such a possibility.

There are many ways to design an expert system. An alternative to the classical model exemplified by MYCIN is the *blackboard model* [Engelmore88]. A *blackboard framework* (shell) is a specification of the components of a blackboard model or its implementation. In a blackboard model, there is a global storage called the *blackboard* and some autonomous *knowledge sources*. Each knowledge source (KS) is itself a knowledge base with the associated inference engine. Knowledge sources cooperate together to work towards the solution by putting their findings on the blackboard, which is accessible to all the knowledge sources. Partial solutions are produced in the form of islands, which tend to merge together into larger islands, in a similar manner to the solution of jigsaw puzzles. Blackboard systems are particularly suitable to problems with continuous input data stream and to evolutionary systems.

Expert systems can be built on expert system shells and blackboard systems are built on blackboard frameworks. A

blackboard framework is made up of three components: the knowledge sources, the blackboard data, and the control. A typical framework is depicted in figure 6.3.

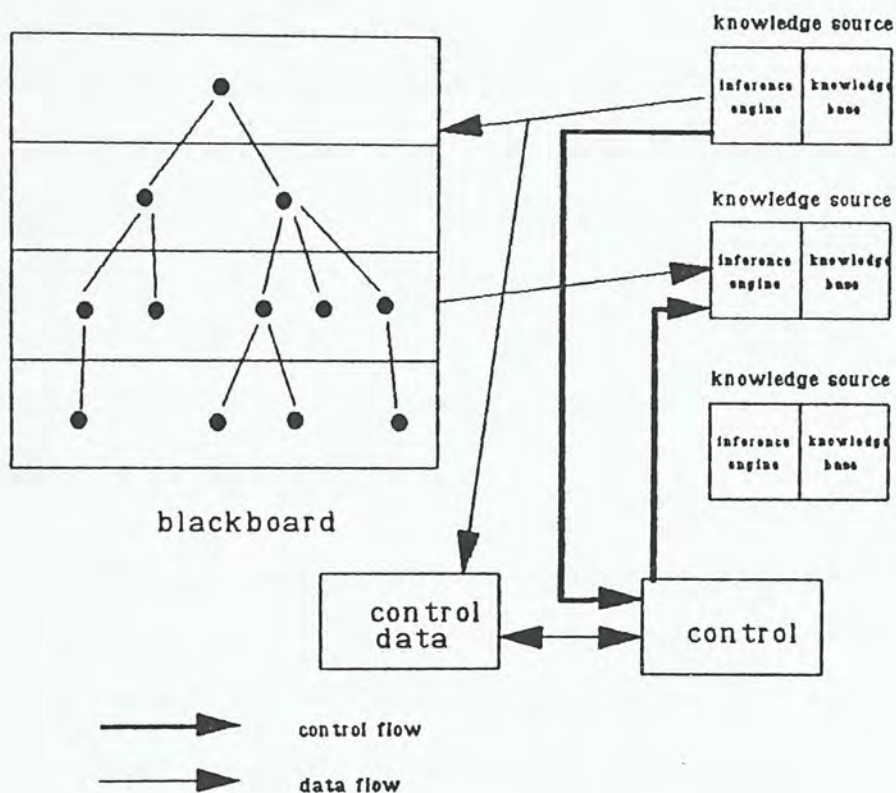


Figure 6.3 A Blackboard Framework

A KS reads the blackboard and updates it *opportunistically* if the knowledge is applicable and the KS has obtained the permission from the control module. The blackboard can be partitioned into panels or into layers to hold hierarchical or structured knowledge pieces. At any given time instant, several KSs may be possible to contribute to the solution. The control module then arbitrates among the KSs involved. Control focus can be KS-centred,

blackboard-centred, or mixed and strategies can vary, according to control data, which is part of the blackboard generally not readable to KSs.

In a blackboard model, the inference engine is distributed among the KSs, leading to greater flexibility. A great deal of parallelism is inherent in the autonomous KSs. The control module, which governs the behaviours of the resulting system, is programmable with complex strategies. This dynamic control adds to another dimension of flexibility. A hybrid knowledge base may be used and a mixed strategy may be adopted.

The blackboard model can be used to generalize simple inference mechanisms. The forward chaining OPS5 can be viewed as a rudimentary blackboard system. Each KS is an OPS5 rule. The control module is the conflict resolution module and the blackboard corresponds to the working memory. The control focus is blackboard-centred. The firing of a production rule involves the matching of the preconditions. This procedure can be carried out by the autonomous KSs monitoring the blackboard. The step of placing all matched rules in the conflict set is equivalent to the notification to the control module raised by the involved KS. The control module runs the algorithms of conflict resolution (LEX or MEA) and activates a KS, corresponding to the firing of a rule.

In a similar manner, the backward chaining Prolog is also representable as a blackboard system. The control module carefully selects the next KS (rule) to react. Subgoals are placed on the blackboard. They can be ordered, as in sequential Prolog. They may also be unordered. If the control module allows more than one KS to react, subgoals may be solved simultaneously, leading to a parallel execution of a Prolog program. In addition to simple forward chaining and backward chaining, the blackboard model is able to operate in a mixed mode, for example, to perform two forward steps, followed by three backward steps and then four forward steps. It is the control module that control this flexible behaviour. Furthermore, in this way, a blackboard framework can be used to build an expert system, by suitably devising the control strategies.

The blackboard model corresponds quite closely to an object-oriented environment, especially an environment in which objects are active, as in VOCOL. The exploitation of parallelism in VOCOL on a multiprocessor machine, say, also benefits the performance of the framework but it is then important to have a control mechanism which can maximize the degree of parallelism, for example, to reduce the memory contention problem of the blackboard. A typical blackboard framework on VOCOL is shown in figure 6.4.

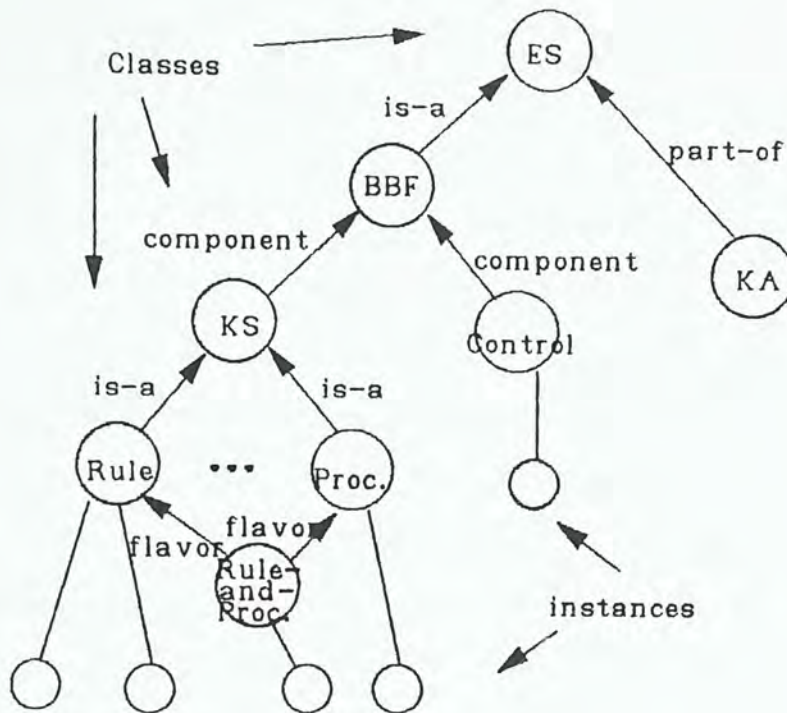


Figure 6.4 Blackboard Framework on VOCOL

Each KS is encapsulated by the notion of a VOCOL object. It is made up of a continuously executing process body, a few protocols and attributes. The body keeps track on changes in the blackboard, watching for the possible matching of KS preconditions. The KS then reveals itself to the control module, an instance of *control*. The control module then sends the appropriate selection message to the KS which is granted the operation privilege, after analyzing the blackboard and control data.

For rule-based knowledge, the message method to a forward chaining mode would be to produce new information based on the blackboard contents and then update it whereas that to a backward chaining mode would be to spawn subgoals onto the

board. For procedural knowledge, the message would probably initiate a computation, updating the board as a result. The inference engine is built-in in the knowledge object, as instance methods. This allows the sharing of codes through inheritance. Furthermore, the rule and procedure classes can be used as flavors to create a new class *rule-and-proc* which is a hybrid representation scheme for the rule-based and procedural knowledge. More hybrid classes can be built in this way. By suitably designing and constructing the internal inference engine, an instance of a class can represent a mini-expert system.

Blackboard data can be stored in a class attribute of *blackboard framework* (BBF), or in the global name space (GNS) provided by VCLISP. Control data can be stored in the class *control* (accessed by KS instances via protocol, or by defining a lateral link from KS to *control*), in the BBF, or in the GNS. Finally, the knowledge acquisition module serves to manipulate the knowledge. This module can be very complex and may have learning capabilities.

As an alternative to being implemented as an object with an active process body (an active KS), a KS can be made passive by defining the body code in a method. The control module can now exercise a greater degree of control over the KSs. It may broadcast a message to the KSs, when being triggered by the completion of the previous KS operation.

CHAPTER SEVEN

PROGRAMMING IN VOCOL

7.1 OVERVIEW

VOCOL is a layered programming environment supporting a variety of programming styles. In this chapter, sample application programs emphasizing on different aspects of VOCOL will be presented and discussed. As pointed out in chapter 4, the first layer of VOCOL is the VCLISP environment, on which programmers in different programming domains hold different viewpoints. Sections 7.2 and 7.3 are dedicated to VCLISP programming with visual LISP features and concurrent features respectively. Section 7.4 is reserved to programmers adopting an object-oriented style delineated in chapter 5.

7.2 PROGRAMMING WITH VISUAL LISP FEATURES

The simplest view of the VOCOL system is to focus on the sequential nature of the VCLISP environment. This simple view is equivalent to an ordinary LISP programming environment, enriched with a graphical user interface and the visual representation of LISP programs. It is not surprising that the sample applications are simple. The example to be exhibited is the Tower of Hanoi. This application had been detailed in section 4.2.1 and figure 4.5 and it is not

repeated here. There is not much peculiarity in sequential VCLisp programs compared with ordinary LISP programs, except for the visual effects.

7.3 CONCURRENT PROGRAMMING WITH VCLISP

The additional feature of concurrent processes in the VCLISP environment arms the programmer with the capabilities of developing concurrent programs. The first example to be discussed is the dining philosopher, which will be given a further treatment in section 7.4.4, where it is programmed in an object-oriented style. The second example is the producer and consumer synchronization problem. Several alternatives to the solution are possible. For the sake of convenience, only the solution to dining philosopher is shown in the visual form, and the rest are presented in their meta-forms (appendix A). Meta-forms bear the advantage of being more compact, without the need of zooming in the smaller parts as in visual programs, albeit the highly nested parentheses.

7.3.1 Dining Philosopher

Dining philosopher is a classical concurrent programming and synchronization problem. It is widely used to test the flexibility and expressive power of concurrent programming languages and constructs. VCLISP provides synchronization primitives like message passing and semaphore. The solution exhibited in this section is the semaphore solution. The

program is shown in figures 7.1, 7.2 and 7.3.

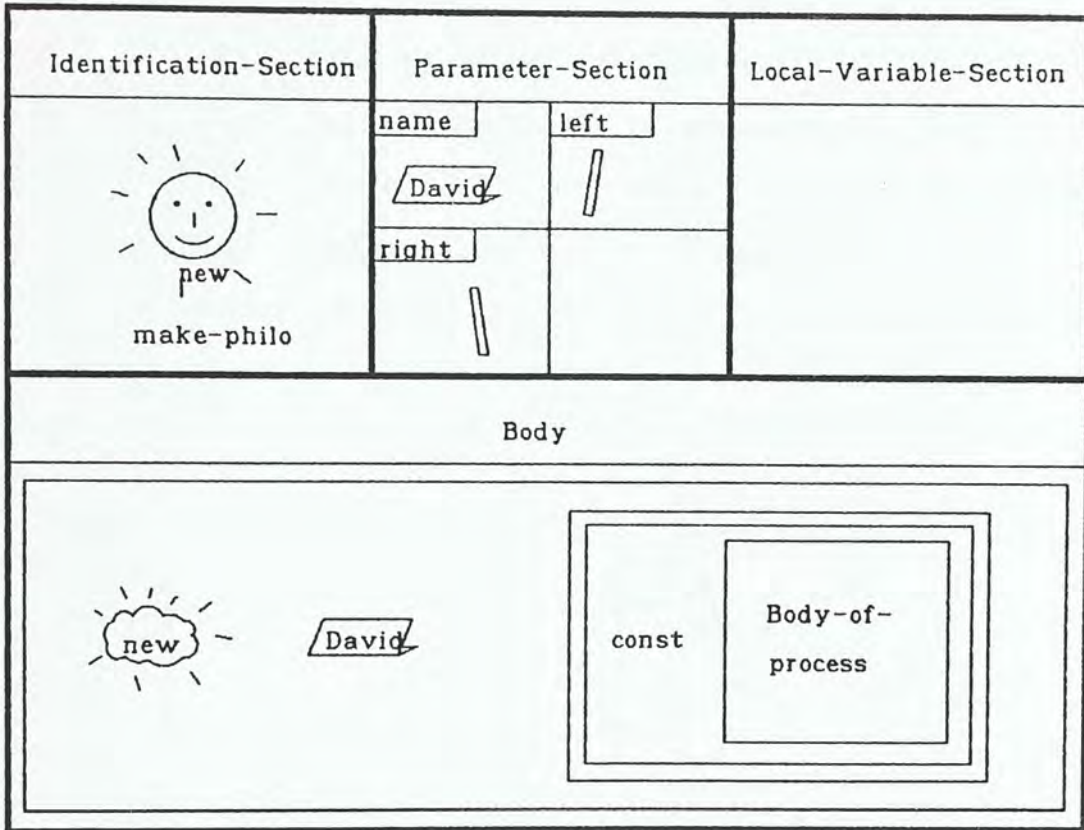


Figure 7.1 Visual Function Make-philo

The function *make-philo* has three parameters, the name of the philosopher, the left stick and the right stick. The name is an atom, a string or an icon. The sticks are binary semaphores, created by the operations on the user interface process, defined in figure 7.3. The purpose of *make-philo* is simply to create a new process with the code body revealed in figure 7.2 with process name defined by the input parameter *name*. The body of the process is a loop reflecting the life cycle of the philosophers: think; enter the room; pick up the left stick; pick up the right stick; eat; put down the left

stick; put down the right stick; and leave the room. Here the sticks and the room are all semaphores. The system visual functions used in this example and in the binary tree in section 7.4.1 are shown in appendix C. Visual syntax conforms to that described in chapter 4. For example, a double boundary refers to a function application. The special form *const* is a nickname for the special form *quote*.

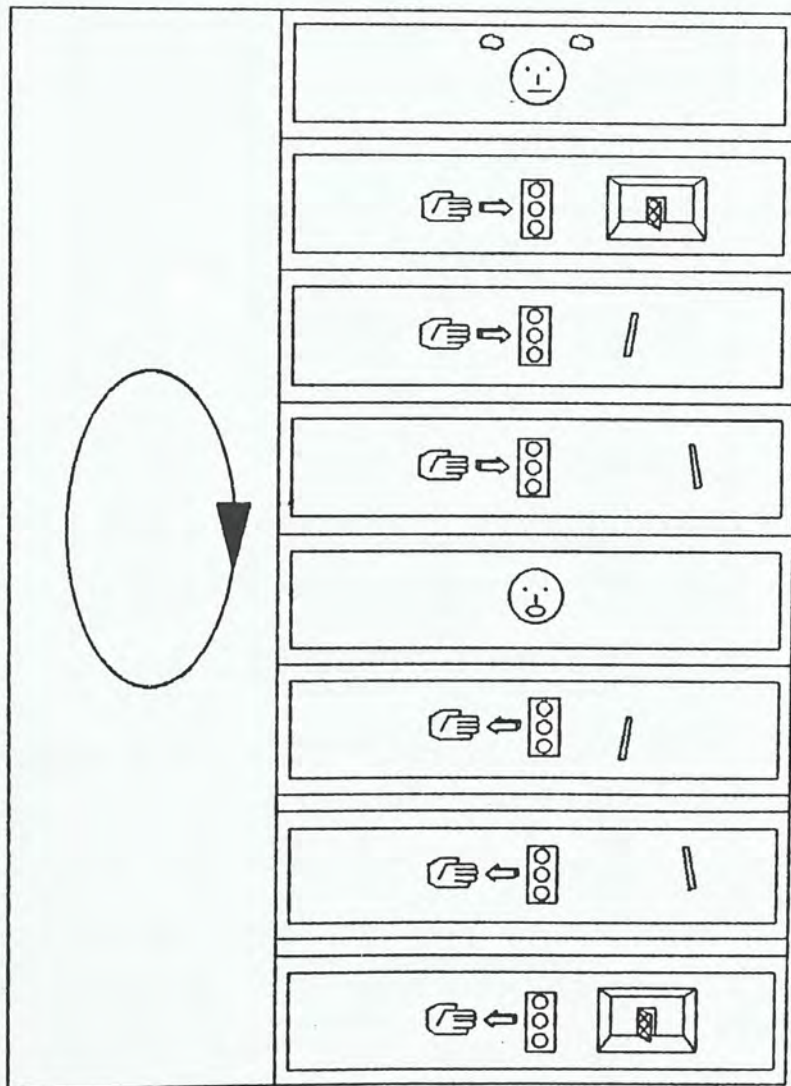


Figure 7.2 Zoomed Body of Make-philo

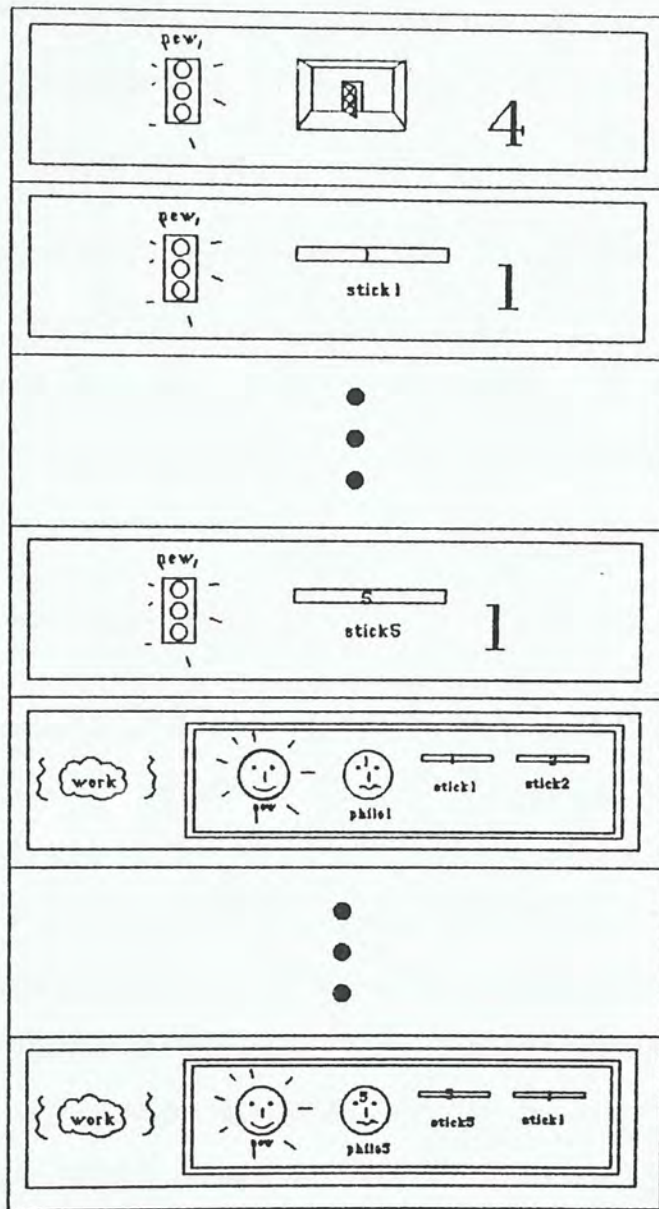


Figure 7.3 Operations At User Interface

Having defined the function *make-phil0*, the program must then be initiated by creating and activating the processes. This is carried out by the sequence of operations in figure 7.3. First of all, the six semaphores (a room and five sticks) for mutual exclusion must be defined. Then the five philosopher processes are created and activated using the

function *make-philos* with the appropriate arguments (the name and the stick semaphores). The names of the philosophers are iconic.

This particular solution of dining philosopher is deadlock-free. Deadlock is prevented by allowing at most four philosophers to eat simultaneously. However, starvation is possible and to avoid it, a more sophisticated algorithm must be adopted.

7.3.2 Producer and Consumer

There are several solutions to the producer and consumer problem. They are presented in the figures below. The first one is the traditional approach, in which a bounded buffer (a shared data structure) and semaphores are used. Here the global name space (GNS) is used. The second one is by means of primitive message passing. The VCLISP system call *send* is an unblocking one and automatic (unbounded) buffering is achieved. It is unnatural to bound the buffer size here and this is the most direct solution. The third approach is to utilize the concept of coprocess and critical section (critical region). Within a process, the segments of coprocess code in the critical section are strictly serialized. Under different programming situations, each approach has its own advantage and simplicity. It is up to the programmer to choose the one that is best for his application at hand.


```

Perform at the user interface the following:
=> (setf (global *max*) 10)
=> (setf (global wptr) 0)
=> (setf (global rptr) 0)
=> (setf (global buf) (make-array (global *max*)))
=> (vcsys-create-semaphore "mutex") ; default = 1
=> (vcsys-create-semaphore "empty" *max*)
=> (vcsys-create-semaphore "full" 0)
=> (vcsys-activate
    (vcsys-create "producer"
      '((loop (setf item (produce)) ; item is LMS of Producer
              (vcsys-p "empty")
              (vcsys-p "mutex")
              (setf (aref (global buf) (global wptr)) item)
              (setf (global wptr) (mod (+ 1 (global wptr)) (global *max*)))
              (vcsys-v "mutex")
              (vcsys-v "full")
              )))
    (vcsys-activate
      (vcsys-create "consumer"
        '((loop (vcsys-p "full")
                (vcsys-p "mutex")
                (setf item (aref (global buf) (global rptr)))
                (setf (global rptr) (mod (+ 1 (global rptr)) (global *max*)))
                (vcsys-v "mutex")
                (vcsys-v "empty")
                (consume item)
                )))
      )))

```

Figure 7.4 Producer Consumer 1

In the first algorithm, the first few steps are to define the bounded buffer as an array in the GNS, reset the read/write pointers and create the three operating semaphores. The next step is to define the producer process and to activate it. The final step is to define and activate the consumer process. The producer enters a loop: produce an item; store the item in a local variable; seize an empty slot in the buffer; obtain a mutual exclusive right to write; place the item in the buffer; release the mutual exclusion;

and generate a full slot. Similarly the consumer enters a loop: seize a full slot; obtain the mutual exclusion; read item from the buffer; store in a local variable; release the mutual exclusion; generate an empty slot; and consume the item.

```

Perform at the user interface the following:
=> (vcsys-make-mailbox "p-c-mail")
=> (vcsys-activate
    (vcsys-create "producer"
      '((loop (vcsys-send "p-c-mail" (produce))))
    )))
=> (vcsys-activate
    (vcsys-create "consumer"
      '((loop (consume (vcsys-receive "p-c-mail"))))
    )))

```

Figure 7.5a Producer Consumer 2a

In algorithm 2a, a mailbox is used for direct communication between the producer and the consumer. After creating a mailbox, the producer and consumer processes are created and activated. The former enters a loop of producing an item and sending it to the mailbox. The latter enters a loop of receiving an item from the mailbox and consuming it. The synchronization is completely handled by the message passing and the mailbox mechanisms.

```

Perform at the user interface the following:
=> (vcsys-activate
    (vcsys-create "producer"
      '((loop (vcsys-send (vcsys-process-id "consumer") (produce))))
    )))
=> (vcsys-activate
    (vcsys-create "consumer"
      '((loop (consume (vcsys-receive (vcsys-process-id "producer"))))
    )))

```


Figure 7.5b Producer Consumer 2b

In algorithm 2b, the communication between the producer and the consumer is even more direct: no mailbox is required. The producer process enters a loop of producing an item and sending it to the consumer, whilst the latter enters a loop of receiving an item from the producer and consuming it.

```

Perform at the user interface the following:
=> (setf (global *max*) 10) ; assume that *max* is used system wide
=> (vcsys-activate
    (vcsys-create "proc"
      '((setf buf (make-array (global *max*))) ; variables are in LNS
        (setf wptr 1)
        (setf rptr 0)
        (vcsys-activate
          (spawn* "producer"
            '((loop (setf pitem (produce))
                  (do () ; can use semaphore if busy waiting not desired
                    (if (= wptr rptr))
                    (critical-section
                     (setf (aref buf wptr) pitem)
                     (setf wptr (mod (+ 1 wptr) (global *max*))))
                  )))
          (vcsys-activate
            (spawn* "consumer"
              '((loop (do ()
                    (if (= wptr (mod (+ 1 rptr) (global *max*))))
                    (critical-section
                     (setf rptr (mod (+ 1 rptr) (global *max*)))
                     (setf citem (aref buf rptr)))
                    (consume citem)
                  )))
            (loop))
          ))
    ))

```

Figure 7.6 Producer Consumer 3

In algorithm 3, coprocesses and critical sections are used. A main process is created and activated. It then

executes its body: make a bounded buffer; reset the read/write pointers; and create and activate the producer and consumer coprocesses. The producer enters a loop: produce an item; loop until buffer is not full; and enter the critical section to put the item in the buffer. Similarly, the consumer enters a loop: loop until buffer is not empty; enter the critical section to get an item; and consume the item. Note that the mutual exclusion semaphore is not required because the critical section enforces the mutual exclusion on the buffer. To increase efficiency, busy waiting for non buffer full and non buffer empty can be replaced by other synchronization means, such as semaphores. Defining *max* in the GNS only permits the value to be used throughout the whole system. It makes no difference by defining it in the LNS of the process.

7.4 OBJECT-ORIENTED PROGRAMMING IN VOCOL

Besides concurrent programming in VCLisp, a major feature of VOCOL is object-orientation. In this section, a few programs are designed with the object-oriented approach. The first one is the binary tree, which is the simplest example, followed by more difficult ones. For compactness, only the first one is delineated in the visual representation.

7.4.1 Binary Tree

A binary tree is to be defined for storing and sorting a set of numbers. Functions are defined for insertion, inorder traversal and initialization. Each node of the tree will be an instance of the class *node*, each has two sons and a value. Figure 7.7 contains the definition of *node*, a replicate of figure 5.2 for the sake of convenience. The definition of these functions as object methods are depicted in figures 7.8, 7.9 and 7.10.

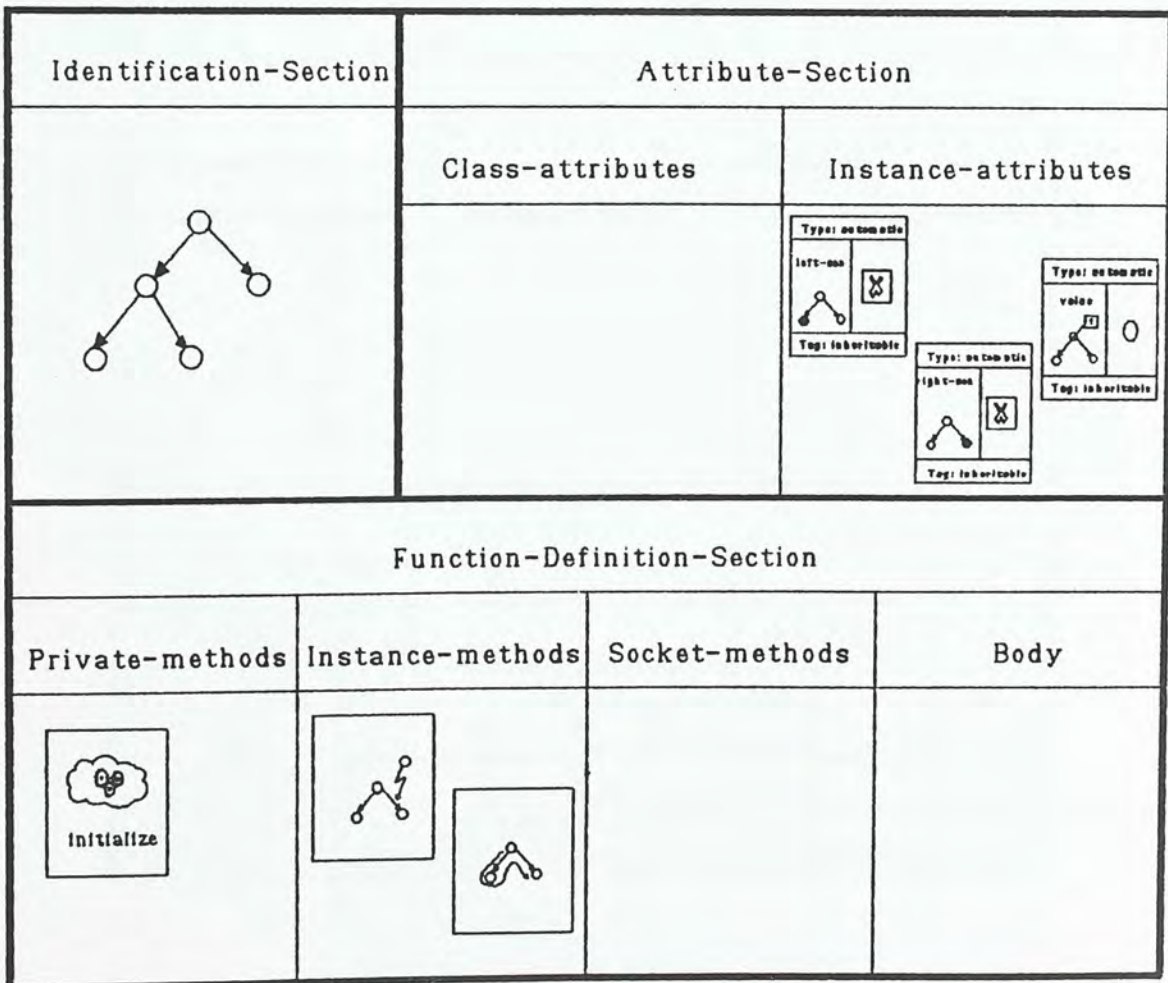


Figure 7.7 Class Node

The class *node* has three instance attributes: *left-son*, *right-son* and *value*. These attributes correspond to the fields of a node in a tree implemented in a conventional language. Each of them has an iconic representation. All attributes are automatic and inheritable. The first two have an initial value *nil* and the third *zero*. Automatic attributes are defined because the fields often need to be retrieved or modified. They are inheritable for the benefits of child classes of *node* so that they needed not to be duplicated. A private method *initialize* is defined to give the key an initial value. Two instance methods *insert* and *inorder* are defined for insertion and traversal.

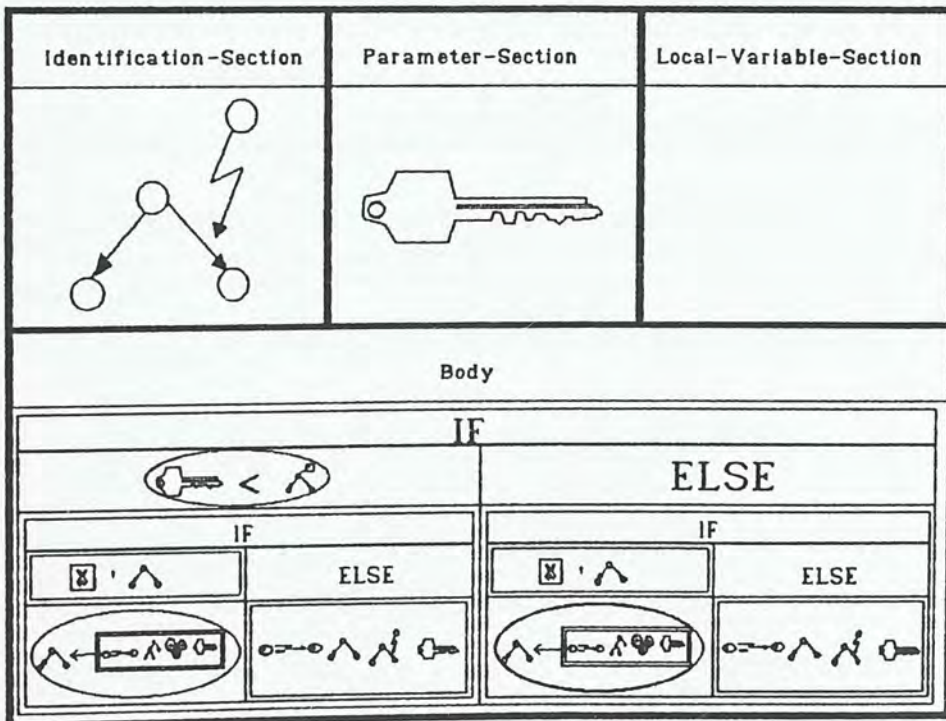


Figure 7.8 Function Insert

Insert takes an argument *key*. If *key* is less than *value*, it tries to insert *key* into the left subtree. If left-son is null, it creates a new instance with *value* set to *key* and connect it to *left-son*. If not, it inserts to the left subtree by sending it the message *insert*. If *key* is not less than *value*, the right subtree is treated in a symmetric way.

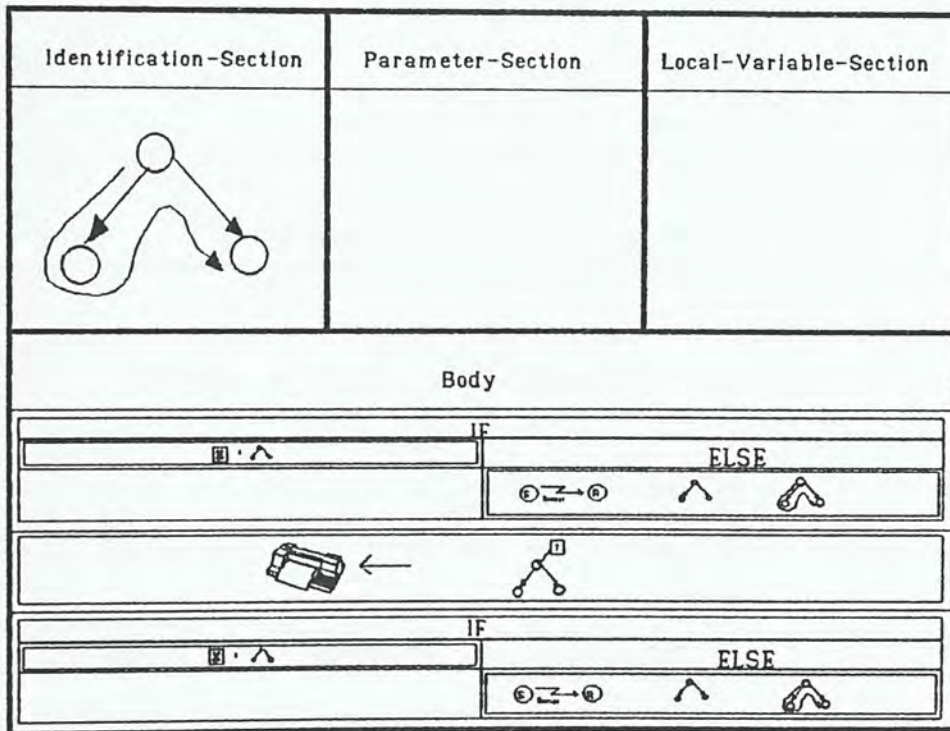


Figure 7.9 Function Inorder

Inorder takes no argument. If *left-son* is not null, perform inorder traversal on left subtree by sending it the message *inorder*. After visiting the left subtree, *value* is printed (this can be replaced by sending a message to an output method to increase flexibility). Lastly, the right subtree is traversed by sending it the message *inorder*.

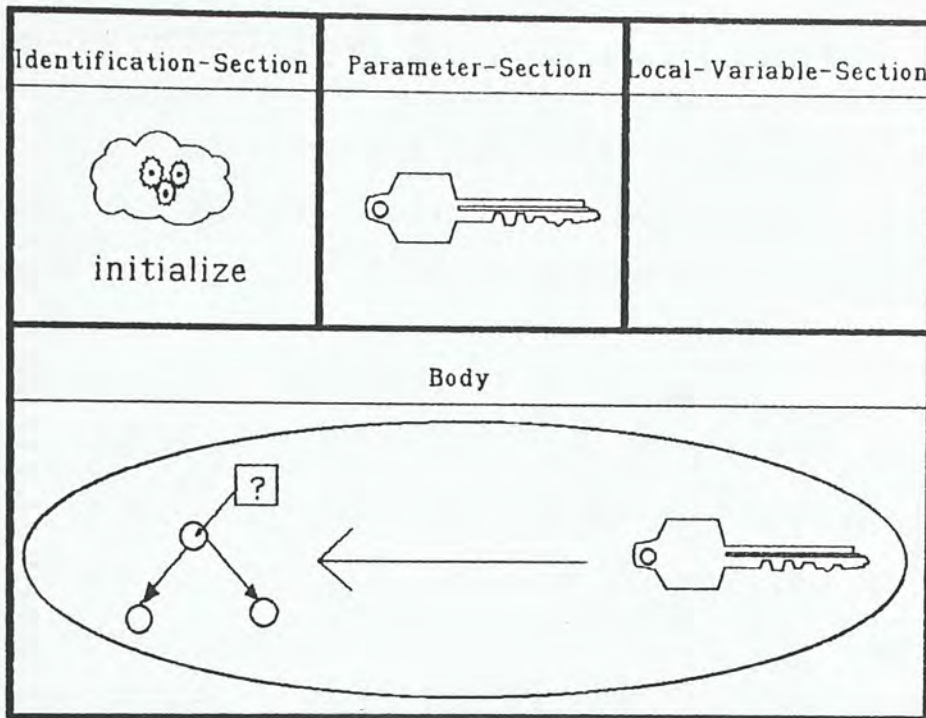


Figure 7.10 Function Initialize

The purpose of *initialize* is to assign an initial value to the node when a new instance of node is created.

This program is almost identical to how it would be written in Smalltalk since both make use of the notion of an object instance for each node. Also, when examined closely, the whole program exhibits strong similarity to one written in Pascal, the difference being the concepts on data structure. Here we are asking the object instances to perform such missions as insertion and traversal instead of directly operating on the data structures as in Pascal.

7.4.2 Distributed Sorting

There are many methods to sort a sequence of numbers. The binary tree in the previous section is one of them. Now, let us have a look on the procedure of distributed sorting.

The problem of distributed sorting is concerned with n processes. Each of them stores a record. The n records are to be sorted according to their keys in ascending order. It is natural to model these process records as objects. Each record is encapsulated by an object. In the sample program, each record broadcasts its contents to all the other records. A record just counts the number of records with a key less than itself. At the end of message broadcast, each of them is able to know its ranking. It then monitors the blackboard. At a call for records with the same rank as its own, it shrieks and reports itself. At the end, the report will contain the sorted list of records. The program is listed below.

```
(class (id (name controller ?))
  (attr (class-atir ())
        (inst-attr (((name num-rec ?) nil manual noninheritable)))
  (funct-def
    (priv-method)
    (inst-method)
    (sock-method)
    (body (self-inst-attr (name num-rec ?)
                        (send-message (name data-node ?) 'instance-count))
          (broadcast (name data-node ?) (name initiate ?) (name num-rec ?))
    )))
```

Figure 7.11 Class Controller


```

(class (id (name data-node ?))
  (attr (class-attr (((name index ?) 1)))
    (inst-attr (((name key ?) nil manual noninheritable)
                ((name value ?) nil manual noninheritable)
                ; attributes key and value comprise the record
                ((name ctr ?) 0 manual noninheritable)
                ((name pos ?) 1 manual noninheritable)
            ))
  (funct-def
    (priv-method)
    (inst-method
      (visual-funct (id (name initiate ?))
        (para (((name num-rec ?) (name num-rec ?) 0)))
        (local ())
        (body (self-inst-attr (name ctr ?) (name num-rec ?))
          ))
      (visual-funct (id (name count ?))
        (para (((name arg ?) (name arg ?) 0)))
        (local ())
        (body (if (< (name arg ?) (name key ?))
          (incf (name pos ?)))
          (decf (name ctr ?))
          ))
      )
    (sock-method)
    (body ; wait until being initialized with message count
      ; this may be replaced by the operation sleep
      (loop (exitif (infix ((name ctr ?) (> 0))))
        (broadcast (name data-node ?) (name count ?) (name key ?))
        ; wait until ctr = 0 (all messages received)
        ; and index >= pos (my turn to report)
        (loop (exitif (infix (((name count ?) = 0) and
          ((name index ?) >= (name pos ?))))))
      (report key value)
      (incf (name index ?))
    )))

```

Figure 7.12 Class Data-node

Instances of the class *data-node* are processes containing the records (key and value). The attribute *ctr* is used to keep track of the number of messages to be received, which equals the number of instances of the class. The attribute *pos* is used to count the number of records in front of it (to

determine its rank). The method *initiate* is used to set the counter for the number of messages to be received. The method *count* monitors the rank of the object itself. If a message is received, it checks whether the incoming key is less than the object's key. If so, it increments the rank by 1. When all messages have been received, the object instance waits until its turn to report. The class attribute *index* is used as the blackboard. When the index becomes the rank of an instance, the instance reports itself and increments the index. The purpose of *controller* is to find out the total number of instances of the class *data-node* and the value is used to initialize the message-count of the instances. A subtle point is that the operations *incf* and *decf* must be atomic for the program to function correctly. Another way is to guard the operations on *index* with semaphores rather than busy waiting. Efficiency can be increased if busy waiting on the condition can be avoided with the use of primitives like *await*.

An alternative way to handle reporting is to wait until the counter becomes zero. The record then makes a plug with *pos* as the attribute and with the record content as the message packet. The controller makes multiple-socket with argument *index* (defined in the controller, not in the class *data-node*) until index is greater than the number of records - the terminating condition. The socket method is to print out the received record (message packet) and increment the index by one. Finally remove all the redundant sockets made. In this method, the matching of position with index is

implicit. For some indices, there can be more than one matched plugs and for some indices, there may be no matched plug. This occurs when some of the keys are duplicated.

Although the idea of sorting is very simple, the amount of message passing is still a bottleneck. Totally, $O(n^2)$ messages are sent and the solution is not often practical unless the traffic problem, a research effort in distributed computing systems, is solved. Nevertheless, this serves as an example to write simple object programs, despite its possible inefficiency.

7.4.3 Project Assignment

In the university, every senior student is required to carry out a project on a topic. Totally, there are n students and m projects ($m \geq n$). Each student arranges the projects in a preference list. He is assigned his first priority project if no one chooses the same project. In case of conflicts, the first-come-first-served policy is adopted. Furthermore, if a student makes his decision late, he may not get his first priority project even if the project has been assigned to another student with a lower preference. At last, if he runs out of his preference list, it is assumed that he gives up to carry out a project. A solution to the project assignment problem is shown in figures 7.13 and 7.14.


```

(class (id student)
  (attr (class-attr ())
        (inst-attr ((preference nil manual noninheritable)
                     (delta-time 100 manual noninheritable)
                     (success nil manual noninheritable))))
  (funct-def
    (priv-method
      (visual-funct (id initialize)
        (para ((arg arg nil)))
        (local ())
        (body (self preference arg)
              )))
    (inst-method)
    (sock-method)
    (body (loop (self success (send-message (car preference) 'try))
                (exitif (or success (null preference)))
                (self preference (cdr preference))
                (delay delta-time)) ; delay to allow other processes to go ahead
          (if success (print (car preference))
                    (print "Not taking project")))
    )))

```

Figure 7.13 Project Assignment: Class Student

```

(class (id project)
  (attr (class-attr ())
        (inst-attr ((selected nil manual noninheritable))))
  (funct-def
    (priv-method)
    (inst-method
      (visual-funct (id try)
        (para ())
        (local ())
        (body (if selected
                (return nil)
                (self selected t))
              (return t)
            )))
    (sock-method)
    (body)
  ))

```

Figure 7.14 Project Assignment: Class Project

The students and the projects are considered objects. A student sends messages to his preferred projects, one by one. If the project is not assigned, the student gets the project which then updates its status. The delay in each student is used to allow slower processes to catch up because some students may get a project as his third preference before a student with the same project as his first priority sends the message. This solution to the problem is in fact a simulation of the actual situation.

The only protocol in a *project* is the message to *try* from a *student*. If the project is not selected, it marks itself as selected and reply OK. Otherwise, it rejects the student's request. A student has the private method *initialize* which is used to initialize his preference list when he is created. The body of a student is to enter a loop for his preference list until he gets a project or runs out of the list. He sends a message to the first project in the list and removes this first project when being rejected. Finally, he announces the project he gets.

A better solution to this assignment problem is to have an arbiter to count the number of messages and announce the commence of the second round selection, third round selection, and so on. Each student must sleep after sending a message. He is awakened when the next round comes again. A student who succeeds or surrenders must report to the arbiter who can then update the counter for messages. With an arbiter, each project is able to know the number of

students competing for it in each round. In case of conflict, the project can decide who is better and abandon the FCFS policy. This decision may also be made by the supervisor (another object class) who is notified by his project about the conflict.

7.4.4 Dining Philosopher Revisited

Besides being implemented as concurrent processes, it is possible to implement the solution to the dining philosopher problem as an object program. The program that defines the philosophers as objects in VOCOL is shown in figures 7.15 and 7.16.

```
(class (id philosopher)
  (attr (class-attr ())
        (inst-attr ((philo-id nil manual noninheritable))))
  (funct-def
    (priv-method
      (visual-funct (id initialize)
        (para ((id id nil)))
        (local ())
        (body (setf philo-id id)
              )))
    (inst-method
      (visual-funct (id dummy-to-wake)
        (para ())
        (local ())
        (body
          )))
    (sock-method)
    (body (loop
          (think)
          (case philo-id
            (1 (make-plug ('config 'free 'free '? '?'?) (1)))
            (2 (make-plug ('config '? 'free 'free '? '?'?) (2)))
            (3 (make-plug ('config '? '?' 'free 'free '?'?) (3)))
            (4 (make-plug ('config '? '?' '?' 'free 'free) (4)))
            (5 (make-plug ('config 'free '?' '?' '?' 'free) (5))))
          (sleep)
```



```

        (eat)
        (send-message 'table-0 'return-stick philo-id)
    )
)))

```

Figure 7.15 Class Philosopher

```

(class (id table)
  (attr (class-attr ())
    (inst-attr ((config-var nil manual noninheritable)
      (c1 free manual noninheritable)
      (c2 free manual noninheritable)
      (c3 free manual noninheritable)
      (c4 free manual noninheritable)
      (c5 free manual noninheritable)))
    (funct-def
      (priv-method
        (visual-funct (id initialize)
          (para ())
          (local ())
          (body (setf config-var (make-socket ('config c1 c2 c3 c4 c5)))
            )))
        (inst-method
          (visual-funct (id return-stick)
            (para ((philo-id philo-id nil)))
            (local ())
            (body (remove-socket config-var)
              (case philo-id
                (1 (setf c1 'free)
                  (setf c2 'free))
                (2 (setf c2 'free)
                  (setf c3 'free))
                (3 (setf c3 'free)
                  (setf c4 'free))
                (4 (setf c4 'free)
                  (setf c5 'free))
                (5 (setf c5 'free)
                  (setf c1 'free)))
              (setf config-var (make-socket ('config c1 c2 c3 c4 c5)))
            )))
        (sock-method
          (visual-funct (id config)
            (para ((philo-id philo-id nil)))
            (local ())
            (body (case philo-id
              (1 (setf c1 'used)
                (setf c2 'used))
              (2 (setf c2 'used)
                (setf c3 'used))
            )))
        )))

```



```

                (setf c3 'used))
            (3 (setf c3 'used)
              (setf c4 'used))
            (4 (setf c4 'used)
              (setf c5 'used))
            (5 (setf c5 'used)
              (setf c1 'used)))
        (send-message sender 'dummy-to-wake)
        (setf config-var (make-socket ('config c1 c2 c3 c4 c5)))
    )))
(body
)))

```

Figure 7.16 Class Table

Here each philosopher is an instance of the class *Philosopher*. He enters the *think-and-eat* cycle indefinitely. Five instances of *Philosopher* are created at the very beginning. The class *Table* has five attributes containing the status (free or in use) of the five chopsticks which are initialized to *free*. A single instance *table* is created. On initialization, it makes a socket *config* containing the sticks also. When a philosopher, say Philo-1, wants to eat, he makes a plug, expressing his desire to acquire the sticks and sleeps to wait for their availability. If the two sticks required by Philo-1 are free, the plug created by Philo-1 can be matched with the socket *config*. Thus an implicit message is sent and the socket method is invoked to update the configuration and awaken the sleeping philosopher to eat by sending him a dummy explicit message. When Philo-1 finishes eating, he sends the *table* a message to return his sticks. The *table* then updates the configuration, removes the outdated socket and creates a new one.

It is worth to point out that *table* should be serialized to avoid errors caused by data inconsistency, the typical problem in a multiprogramming environment. The solution would be to guard the socket method *config* and the instance method *return-stick* within critical sections.

This is a deadlock-free solution to the problem because both the sticks are matched and acquired simultaneously. However, it suffers from the problem of starvation. To solve this, we can define new instance attributes *priority* and *event-count* for each philosopher. Let the table keeps a count on the total number of turns of eating (the event count). Each philosopher keeps the event count of his last eating turn. When a philosopher wants to eat and the difference between his event count and the table's event count exceeds a certain limit, say, 10, he sets his priority to high. The default value is the wildcard. All the plugs and sockets for *config* will have the new slot priority. The table always makes a socket with a priority filled with the high value. In this way, those hungry philosophers with high priority can eat within 5 turns because their plug and socket pairs are better matched (one SWM less). After his dinner, a philosopher resets his priority to the wildcard.

Implicit messages play an important role in the formulation of dining philosopher. By suitably incorporating the power of simple pattern matching, the implementation becomes more elegant and easier to understand, as compared with a Smalltalk implementation in which only explicit

messages are adopted. The procedures of asking for free sticks, decision making and sticks acquisition are made in the single step of plug creation. The philosopher does not need to ask the table for the current configuration. Hence, many message passing overheads are eliminated. Although the semaphore solution in the last section seems more elegant, it is more difficult to have a thorough control over the object behaviours.

CHAPTER EIGHT

EVALUATION AND RELATED WORKS

8.1 RELATED WORKS

There are numerous researches on multiparadigm programming environments. Still quite a considerable amount of the research efforts are devoted to object-oriented programming. The VOCOL system is related to object-oriented programming in LISP and to concurrent programming. Two closely related systems, the Xerox LOOPS and ConcurrentSmalltalk, are therefore studied.

8.1.1 LOOPS

LOOPS [Bobrow82] is an object-oriented programming environment developed on the Interlisp-D environment. It is a commercial product of the Xerox Corporation and the ENVOS Corporation.

LOOPS integrates the object-oriented paradigm smoothly within LISP. In addition, it supports access-oriented programming and rule-oriented programming. Implemented in an interactive LISP programming environment and running on workstations, LOOPS provides certain graphical user interfaces.

Objects in LOOPS are classified into classes, which are instances of metaclasses. Attributes can contain values as well as property lists. An attribute slot is a generalized LISP variable. The only relationship between two classes is *parent-son (is-a)*. Multiple inheritance is adopted but cycle in the graph topology is forbidden. This means that the classes are organized as an acyclic digraph. The root is the class *tofu* (top of the universe). Abstract classes are defined but they are never instantiated. Remote procedure call is adopted as the message passing mechanism. A result is generally returned and the sender of message must wait. Parallelism is achieved by sending special messages to start new processes to execute methods.

The name space of LOOPS is distinct from that of LISP. The special qualifier *\$* is used to convert a LISP atom into a LOOPS object. Operations on object attributes must be explicitly specified, otherwise they will be performed on LISP atoms.

The procedure-oriented paradigm of LOOPS is concerned with the separation of data and instructions, as in conventional programming languages. This is encompassed by the functional style of LISP. The access-oriented paradigm is supported by *active values*. An active value is an instance of the class *ActiveValue*. It can be installed within an attribute or a property. Reading from or writing to the attribute or property will cause the appropriate message sent to the associated active value object, invoking

the designated method. In this way, actions of active values are triggered through the access of the variables and hence the term access-oriented. Active values are especially useful to system monitoring and debugging. The active value can be the housekeeper to a resource. The variable under watch by the debugger can be installed with active value methods to dump out its content or to take appropriate action. The rule-oriented paradigm is supported in LOOPS as an extension, contained in the users' modules.

LOOPS provides extensive functions and methods for built-in classes. Perhaps the system attempts to equip the programmer with every function he might wish to use. In addition, various browsers reflect the various views of the system and the structure of objects, classes and their hierarchy. Dynamic system behaviour is easily depicted pictorially with the aid of windows and gauges provided by the system.

8.1.2 ConcurrentSmalltalk

ConcurrentSmalltalk [Yokote87] is a programming environment incorporating concurrent programming facilities in Smalltalk-80, which only supports limited concurrency. In Smalltalk-80, a process is created by sending the message *fork* to a block. Both the sender and the forked process then share the same set of variables (same context). This is similar to the coprocesses in VCLISP. Although a multiple number of processes can exist, the system assumes no time

slicing. The programmer has to implement a high priority background process to explicitly schedule the processes. ConcurrentSmalltalk goes a step ahead in providing notions to express parallel computations and concurrent programs in a more natural way. It unifies objects and processes into the single notion of *concurrent objects*.

ConcurrentSmalltalk is both source-level and snapshot-level compatible with Smalltalk-80, to which concurrent constructs are added and semantics are defined. Two message passing mechanisms are supported. One is the familiar remote procedure call (synchronous method call), the other being asynchronous method call. The asynchronous method call acts like a future¹, if the returned value is needed later, through the use of the system-defined object *CBox*. A *CBox* object is created for each asynchronous method call. The message is then retransmitted by the *CBox* object to the destined receiver and it waits on behalf of the sender. The sender receives the *CBox* object after sending the message. This is a way to implement asynchronous message passing in terms of synchronous message passing. The retrieval of values from *CBox* objects gives rise to the necessity of defining new synchronization primitives on the *CBox* objects, such as *receiveAnd* and *receiveOrAll*. A new context is created when a new message comes in before the completion of an executing method, in a LIFO manner. Finally, the notion

¹ A future is a piece of computation initiated by a process. The result is not required immediately and hence the initiator needs not to wait for the completion of the computation. Later the process can use the result stored in a dedicated location and will wait if the computation has not yet finished. This is a means of exploiting an extra amount of parallelism.

of an atomic object is introduced for mutual exclusion and serialization. Messages arriving at an atomic object are executed strictly serially.

8.2 COMPARISON

After a thorough discussion of the various facets of the VOCOL system, it is now time to compare the system in the light of its parents - LISP and Smalltalk and its relatives - LOOPS and ConcurrentSmalltalk and to conduct an evaluation.

8.2.1 Evolution from LISP and Smalltalk

In this section, the difference between VOCOL and the parents LISP and Smalltalk and the evolution of VOCOL features from the two are highlighted. These features are grouped under tables 8.1, 8.2 and 8.3.

General Features	LISP	VOCOL	Smalltalk
paradigm	functional	visual; concurrent; functional; object-oriented	object-oriented
support for object-orientation	extension by Flavors system	object-oriented layer	primitive paradigm
mechanism of program execution	function call	message passing and function call	message passing
visual effects	no	extensive	yes but not extensive
concurrency	no	yes	limited

Table 8.1 Comparison on the General Features of LISP, VOCOL and Smalltalk

Abstract Data Type	LISP defstruct	LISP property list/atom	Object in VOCOL ²	Object in Smalltalk
direct access to attribute	no	yes	owner only	owner only
automatic accessor function	yes	no	yes	no
user-defined accessor function	no	no	yes	yes
private attributes	no	no	yes	yes
flexibility	medium	high	medium	medium

Table 8.2 Comparison on Abstract Data Type

Reusability	LISP (CommonLisp)	VOCOL	Smalltalk
mechanism	package	multiple inheritance	single inheritance
amount shared	selected subset by mutual consent	filtered set inherited	completely inherited
access to internal symbol / attributes	indiscriminate access	via protocol, if allowed	via protocol, if allowed
modularity	unstructured mixture of <i>use-package</i> , <i>import</i> and <i>export</i>	well-defined semantic network	well-defined tree structure
flexibility	high	medium high	medium

Table 8.3 Comparison on Resource Sharing and Code Reusing

8.2.2 Comparison to LOOPS and ConcurrentSmalltalk

In this section, features of the two systems LOOPS and ConcurrentSmalltalk are compared with those of VOCOL. Most

² Since VOCOL supports the full LISP environment in the visual LISP layer by the VCLISP environment, structures and property lists are supported in VOCOL and at the programmers' own discretion. The comparisons made here only serve for the notion of objects in VOCOL.

features in the tables in the previous section are shared by LOOPS and ConcurrentSmalltalk, as they are advancement to LISP and Smalltalk. They are shown in table 8.4.

Features	LOOPS	VOCOL	ConcurrentSmalltalk
paradigm	functional; object-oriented; access-oriented; rule-oriented	visual; functional; concurrent; object-oriented	object-oriented; concurrent
mechanism of computation	message passing; function call	message passing; function call	message passing
message passing mechanisms	remote procedure call	remote procedure call; asynchronous message; implicit message	remote procedure call; asynchronous message with future
inheritance	multiple: acyclic digraph	multiple: semantic network	single: hierarchical
search for applicable method	depth first search with join	breadth first only; breadth first join	along the superclass chain
metaprogramming	meta classes	user-defined relations	meta classes
parallelism	explicitly create a process for a method	default to multithreading of methods; mixed internal and external parallelism	by means of asynchronous message and CBox objects
name space	separate for LISP and LOOPS	name in context; not distinct between VCLISP and VOCOL	distinct global names and shared variable names
interface to supporting environment	interface to LISP via LISP functions and symbols	interface to VCLISP via FMOs	compatible with Smalltalk-80; no interface needed

Table 8.4 Comparison on Features of VOCOL, LOOPS and ConcurrentSmalltalk

8.3 EVALUATION

Concluding from the programs developed in the last chapter, VOCOL shows its usefulness in various aspects. As a LISP environment, it arms the programmer with a graphical user interface and enables better representations of LISP programs. As a concurrent LISP environment, the familiar notions of processes are at the programmer's own discretion, though they are programmed in the LISP language. It makes virtually no significant difference between conventional concurrent programming and concurrent programming in VCLisp, as exemplified by the sample programs developed in the last chapter.

The effectiveness of the VOCOL environment is even more strongly supported by the variety of sample applications, as well as the ability of defining higher level programming tools. Implicit message passing finds its applicability in the pattern matching of dining philosopher. Message broadcasting is useful in distributed sorting. A combination of object programming and message broadcasting solves the problem of project allocation in a human like manner. Object-orientation is closely related to real world simulation. Some of these problems require the computation power of the LISP environment. This power is granted through the FMO interface. Although there must exist some problems that have only awkward solutions in VOCOL, this is inevitable because VOCOL is not able to provide the programmer with all

kinds of tools that he can ever think of. Yet the system proves to be powerful and often produces elegant solutions to a wide class of problems.

8.4 FUTURE DEVELOPMENT

The prototype of VOCOL demonstrates its capabilities through the sample programs and applications. Nevertheless, there are still capabilities which need to be improved and to be explored.

Concerned with the visual aspect, icons are of the utmost importance. A better visual programming interface can be provided by conducting a thorough survey on the psychology of widely accepted icons and icons which easily identify themselves to outsiders, as well as new syntactic sugars which are more readily understandable.

In VOCOL, it remains to generalize the notions of classes and metaclasses in various aspects such as how to specify concisely and precisely the exact operations on the set of inheritable methods and attributes before passing them to the child classes and to have a more thorough control on the behaviours of the classes. In VOCOL, only binary relations are defined. Is it possible to define a set of consistent semantics for tertiary and, in general, n-ary relations in the context of inheritance?

As for message passing, should we extend the capabilities of plugs and sockets so that their contents can be varied after their creation (dynamic plugs and sockets); or should we introduce the concept of logical variables so as to attain the power of unification? The terms dynamic plug and socket mean that with the attribute of a plug or a socket *connected* with a variable or an attribute, the former is able to change whenever there is a change in the latter, resulting in an aliasing of the storage. The attributes of plugs and sockets in VOCOL are static - obtained from the copy rule.

The prototype implementation of VCLISP interpreter runs at one or two orders of magnitude slower than its counterpart of an ordinary LISP interpreter. The prototype for VOCOL is even slower, especially in the search of appropriate methods and location of class and instance attributes. When the determination of inheritable instances and methods is performed statically, recomputations are then necessary if there is a slight change to the system graph topology. Algorithms which can minimize the recomputation required are to be developed. On the other hand, if their determination is dynamic, algorithms must be devised for their efficient searching. Dynamic plugs and sockets and those with logical variables even pose a more severe threat to efficiency. Efficient execution and control strategies must therefore be designed before the system can become a practical one. For example, LOOPS attempts to improve its efficiency in garbage collection and method and instance lookup by careful management of reference counts, object copying and caching.

Another way to improve the efficiency would be to explore parallelism in the hardware frontier. The VCLISP system is best implemented on a multiprocessor machine for the maximum degree of parallelism. Algorithms and control strategies must be developed to deal with the allocation of resources, such as memory contention for the global name space, efficient message passing mechanisms and management, and allocation of processes on the processor nodes.

CHAPTER NINE

CONCLUSION

The ideal programming environment is yearned for by almost all programmers, who are often burdened by the need to instruct very concisely, precisely, and prudently the computer exactly what to do, and even more annoyingly, how. Research efforts are constantly being made to approach this unattainable goal. The design of an experimental multiparadigm programming environment, named VOCOL, attempting to draw fascination from the pool of programmers of different faculties is one of the numerous efforts.

In this thesis, not only the design of the VOCOL system is described, a prototype implementation and its performance, are also discussed and illustrated. The system VOCOL successfully blends together the visual, object-oriented, and concurrent programming paradigms with the basic ingredient: functional programming in a LISP environment. With a layered architecture, VOCOL is broken up into modules of manageable sizes.

As part of the whole design, a stand-alone LISP programming environment - the VCLISP environment is conceived and realized. VCLISP defines the semantics and operations on concurrent LISP processes, having visual representations and being compiled in the language VCLisp. To examine the validity and usefulness of the environment, a prototype is

developed and the VCLisp interpreter written. Sample programs are written to benchmark and evaluate VCLISP. A few of them are presented in the thesis. Experiments show that VCLISP is easy to program, as ordinary LISP is, and that it is more powerful and user-friendly.

On top of VCLISP, the object-oriented paradigm is built. Entities in the object-oriented layer, either active or passive, are expressed in terms of VCLisp statements and system calls. A high degree of parallelism is exploited with the unification of objects and processes. By suitably modifying VCLISP, VOCOL is portable to a new machine. In much the same way, sample applications are developed as object programs, running on the object-oriented layer. Some of these object programs illustrate their elegance and some their flexibility over conventional programs.

Finally, the application layer on the object-oriented layer furnishes the programmer with various programming tools with which his productivity can further be raised and his programming burden assuaged, in addition to the toolbox analogy of a multiparadigm environment designated by VOCOL.

The integration of visual programming, concurrent programming, and object-oriented programming into LISP, as in VOCOL, has proved to be a powerful combination in a multiparadigm programming environment. The author wishes that the design of VOCOL could lead to some contributions to

researches in programming languages and environments and is looking forward to the advent of such a day: *Programming is no more a nightmare but a pleasure!*

APPENDICES

A. The BNF of Some Meta-forms

1. Visual Function

Note that the BNF can generate meta-forms for all valid visual functions but it can also generate some invalid ones. For example, it is not allowed both the textual name and icon-id of a name to be wildcards.

```
<visual-funct> ::= ( visual-funct <id> <parameter> <local-vars> <body> )
<id>           ::= ( id <name> )
<parameter>   ::= ( para ( <arg>* ) )
<arg>         ::= ( <name> <name> <val> )
               semantics: the first name is the name of parameter; the
               second is the prompt; the last value is the default value
<local-vars>  ::= ( local ( <loc-var>* ) )
<loc-var>     ::= ( <name> <val> )
<body>        ::= ( body ( <s-exp>* ) )
<name>        ::= ( name <text-name> <icon-id> ) ; <text-name>
<text-name>   ::= <symbol> ; <wildcard>
<icon-name>   ::= <number> ; <wildcard>
<val>         ::= <constant>
<wildcard>    ::= ?
```

2. Class Definition

```
<class-def>   ::= ( class <id> <attributes> <funct-def> )
<attributes>  ::= ( attr <class-attr> <inst-attr> )
<class-attr>  ::= ( class-attr ( <c-att>* ) )
<inst-attr>   ::= ( inst-attr ( <i-att>* ) )
<c-att>       ::= ( <name> <val> )
<i-att>       ::= ( <name> <val> <type> <tag> )
<type>        ::= automatic ; manual
<tag>         ::= inheritable ; noninheritable
<funct-def>   ::= ( funct-def <priv-method> <inst-method> <sock-method>
                  <body> )
<priv-method> ::= ( priv-method <visual-funct>* )
<inst-method> ::= ( inst-method <visual-funct>* )
<sock-method> ::= ( sock-method <soc-meth-def>* )
<soc-meth-def> ::= ( <socket> <visual-funct> )
<socket>      ::= ( <name> <socket-attr>* )
<socket-attr> ::= <name> ; <val> ; <wildcard>
```


3. Others

Depending on the context of the visual function, the scope of <s-exp> varies. If it is under the context of a class definition, high level message passing mechanisms are allowed. The following rules are a few of them.

```

<s-exp> ::= <function-application> |
         <infix-exp> |
         <send-msg> |
         <make-plug> | <make-socket> | ...

<infix-exp> ::= ( infix ( <infix-ele> ) )
<infix-ele> ::= <expr> | <l-value> <-> <expr>
<expr> ::= <term> (<add-op> <term>)*
<term> ::= <factor> (<mul-op> <factor>)*
<factor> ::= [ <expr> ] | <s-exp>
<send-msg> ::= ( send-message <receiver> <method> <argument>* ) |
               ( send-async <receiver> <method> <argument>* ) |
               ( broadcast <rec-class> <method> <argument>* )

<receiver> ::= <class> | <instance>
<rec-class> ::= ( <class>* ) | <class> | ?
<make-plug> ::= ( make-plug <plug> <msg-packet> ) |
               ( make-multiple-plug <plug> <msg-packet> )
<make-socket> ::= ( make-socket <socket> ) |
                 ( make-multiple-socket <socket> )

<method> ::= <name>
<argument> ::= <name> | <val>
<plug> ::= ( <shape> <attr>* )
<socket> ::= ( <shape> <attr>* )
<shape> ::= <name>
<attr> ::= <name> | <val> | <wildcard>
<msg-packet> ::= ( <argument>* )

```


B. An Example of the Evaluation Cycle

Suppose the symbol `A` is bound the value `6` and `DOUBLE` the lambda definition `(LAMBDA (X) (* X 2))`. Let us trace the evaluation of `(PRINC (DOUBLE A))`. This sliced interpretation takes quite a long cycle (32 steps) to evaluate such a simple S-expression. Note that the execution can be suspended between any step and then resumed later.

1. Push `(PRINC (DOUBLE A))` in FORM stack and push `EVAL` in STATE stack (the continuation field) and start evaluation.
2. Form `(PRINC (DOUBLE A))` is a function call. Set STATE to function evaluation `EVAL-FUNC`.
3. Get lambda definition for `PRINC`. It is a system function and set STATE to evaluate argument `EVAL-ARG`.
4. Argument list `((DOUBLE A))` is not null. Therefore evaluate the first argument. Push the argument `(DOUBLE A)` in FORM stack and `EVAL` in STATE stack.
5. Evaluation of `(DOUBLE A)`. Form is a function call. Set STATE to `EVAL-FUNC`.
6. Get lambda definition for `DOUBLE`. It is a user-defined function and set STATE to `EVAL-ARG`.
7. Argument list `(A)` is not null. Push the first argument `A` in FORM and `EVAL` in STATE.
8. Form is an atom `A`. Its value is `6`. Put it in `RESULT` and set STATE to `FINISH` so that backpatch can be performed.
9. Backpatch occurs and `6` is appended to the `RESULT` list of `EVAL-ARG`. Set STATE to `EVAL-ARG`.
10. Argument list is null. Finish argument evaluation. `RESULT` contains the evaluated argument list `(6)`. Set STATE to bind arguments `BIND-ARG`.
11. Pair up the parameters with arguments to form the lambda binding. Thus `X` is bound to `6`. Set STATE to apply function body `APPLY-BODY`.
12. Function body may be a sequence of statements. Push the first statement in FORM stack and push `EVAL` in STATE stack. Retain the rest of the body.
13. Form is `(* X 2)`. Set STATE to `EVAL-FUNC`.
14. It is system function. Set STATE to `EVAL-ARG`.
15. Argument list `(X 2)` not null. Push `X` in FORM and `EVAL` in STATE.
16. Form is an atom `X`. Its value is `6`. Set STATE to `FINISH`.
17. Backpatch and append `6` to `RESULT` list. Set STATE to `EVAL-ARG`.
18. Argument list `(2)` is not null. Push `2` in FORM and `EVAL` in STATE.
19. Form is a constant `2`. Value is `2`. Set STATE to `FINISH`.
20. Backpatch and append `2` to `RESULT` list. Set STATE to `EVAL-ARG`.
21. Argument list is null. Finish argument evaluation. Arguments are `(6 2)`. Set STATE to `BIND-ARG`.

22. * is a system function. Pairing is not necessary. Set STATE to APPLY-SYSTEM.
23. Apply * on the argument list (6 2) to yield 12 (result of step 13). Set STATE to FINISH.
24. Backpatch 12 in RESULT stack. Previous action is EVAL (step 12). Set STATE to FINISH.
25. Backpatch again. Propagate 12 back. Previous action is APPLY-BODY (step 11). Set STATE to APPLY-BODY.
26. Function body is exhausted (DOUBLE has only one statement in body). Set STATE to FINISH.
27. Backpatch 12 in RESULT stack. Previous action is EVAL (step 4). Set STATE to FINISH.
28. Backpatch again and propagate 12. Append 12 in RESULT list. Previous action is EVAL-ARG (step 3). Set STATE to EVAL-ARG.
29. Argument list is null. Finish argument evaluation. Argument is (12). Set STATE to BIND-ARG.
30. PRINC is a system function. Set STATE to APPLY-SYSTEM.
31. Argument 12 is printed on the screen. Set STATE to FINISH.
32. Backpatch 12. Previous action is EVAL (step 1). Set STATE to FINISH.
33. Everything is empty and the whole evaluation cycle is completed.

C. Some Iconic System Functions

A few commonly used system functions and their iconic representations are shown in this table. Some of them have been used in the definition of the tower of Hanoi in chapter 4 and in the sample applications in chapter 7.

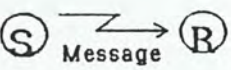


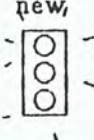
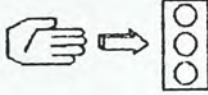
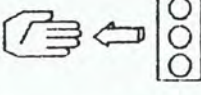



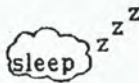


	send-message		print
	make-instance		vcsys-create-semaphore
	vcsys-p		vcsys-v
	vcsys-create		vcsys-activate
	delay		sleep
	true		nil

Figure C.1 Some Iconic System Functions

PUBLICATION

A Visual Object-oriented Environment for LISP,
in the Proceedings of International Computer Science
Conference 1988.

Hong Kong, December 1988, pp. 705-711.

REFERENCES

- Allen, J. (1978).
Anatomy of LISP, McGraw-Hill, New York, 1978.
- Andrews, G.R. and Schneider, F.B. (1983).
"Concepts and Notations for Concurrent Programming," in *Computing Surveys*, volume 15, number 1, March, 1983, pp. 3-43.
- Backus, J.W. (1978).
"Can Programming Be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs," in *Communications of ACM*, August 1978, pp. 613-641.
- Barr, A. and Feigenbaum, E.A. (1982).
The Handbook of Artificial Intelligence, vol. 2, Pitman, London, 1982.
- Bobrow, D.G. and Stefik, M. (1982).
The LOOPS Manual, Palo Alto Research Center Xerox, PARC, KB-VLSI-81-13, 1982.
- Bobrow, D.G. (1985).
"If Prolog is the Answer, What is the Question ? Or What it Takes to Support AI Programming Paradigm," in *IEEE Transactions on System Engineering*, November 1985, pp. 1401-1408.
- Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F. (1986).
"CommonLoops: Merging Lisp and Object-Oriented Programming," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, September 1986, pp. 17-29.
- Brown, G.P. (1985).
"Program Visualization : Graphical Support for Software Development," in *IEEE Computer*, Aug. 1985, pp. 27-35.
- Chang, S.K. (1987).
"Visual Languages: A Tutorial and Survey," in *IEEE Software*, January 1987, pp. 29-39.
- Clarisse, O. and Chang, S.K. (1986).
"VICON : A Visual Icon Manager," in *Visual Languages*, Chang, S.K., Ichikawa, T. and Ligomenides, P.A., eds., Plenum Press, 1986, pp. 151-190.
- Clark, K. and Gregory, S. (1984).
PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Department of Computing, Imperial College, London, 1984.
- Clocksinn, W.F. and Mellish, C.S. (1981).
Programming in Prolog, Springer-Verlag, Berlin, Heidelberg, 1981.
- Cox, B.J. (1984).
"Message/Object Programming : An Evolutionary Change in Programming Technology," in *IEEE Software*, Jan. 1984, pp. 50-61.
- Cunningham, W. and Beck, K. (1986).
"A Diagram for Object-Oriented Programs," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Sept. 1986, pp. 361-367.
- Dart, S.A., Ellison, R.J., Feiler, P.H. and Habermann, A.N. (1987).
"Software Development Environments," in *IEEE Computer*, November 1987, pp. 18-28.

- Davis, R.E. (1985).
 "Logic Programming and Prolog : A Tutorial," in *IEEE Software*, September 1985, pp. 53-62.
- Deitel, H.M. (1984).
An Introduction to Operating Systems, Reading, Mass., Addison Wesley, 1984.
- Dijkstra, E.W. (1968).
 "GOTO Statement Considered Harmful," in *Communications of ACM*, March 1968, pp. 147-148.
- Englemore, R. and Morgan, T. (1988).
Blackboard Systems, Addison-Wesley, Reading, Massachusetts, 1988.
- Faustini, A.A. and Lewis, E.B. (1986).
 "Toward a Real-Time Dataflow Language," in *IEEE Software*, Jan. 1986, pp. 29-35.
- Fukunaga, K. and Hirose, S.I. (1986),
 "An Experience with a Prolog-based Object-Oriented Language," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Sept. 1986, pp. 224-231.
- Glinert, E.P. and Tanimoto, S.L. (1984),
 "Pict : An Interactive Graphical Programming Environment," in *IEEE Computer*, Nov. 1984, pp. 7-25.
- Goldberg, A., Robson D. and Ingalls, D. (1983).
Smalltalk-80: The language and its implementation, Addison-Wesley, Reading, Massachusetts, 1983.
- Grafton, R.B. (1985).
 "Guest Editor's Introduction : Visual Programming," in *IEEE Computer*, Aug. 1985, pp. 6-9.
- Guy, L. (1984).
CommonLisp the Language, Digital Press, 1984.
- Hailpern, B. (1986).
 "Guest Editor's Introduction: Multiparadigm Languages and Environments," in *IEEE Software*, volume 3, January 1986, pp. 6-9.
- Hammer, M. and Ruth, G. (1979).
 "Automating the Software Development Process," in *Research Directions in Software Technology*, edited by Wegner, P., Cambridge, Mass., MIT Press, pp. 767-792.
- Heidorn, G.E. (1977).
 "The End of the User Programmer?" in *The Software Revolution*, Infotech State of the Art Conference, Copenhagen, Denmark, 1977.
- Hirahawa, M., Monden, N., Yoshimoto, I., Tanaka, M. and Ichikawa, T. (1986),
 "Hi-Visual : A Language Supporting Visual Interaction in Programming," in *Visual Languages*, Chang, S.K., Ichikawa, T. and Ligomenides, P.A., eds., Plenum Press, 1986, pp. 233-259.
- Ishikawa, Y. and Tokoro, M. (1986),
 "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K : Its Features and Implementation," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Sept. 1986, pp. 232-241.
- Jacob, R.J.K. (1985).
 "A State Transition Diagram Language for Visual Programming," in *IEEE Computer*, Aug. 1985, pp. 51-59.
- Jenkins, M.A. and Glasgow, J.I. (1986).
 "Programming Styles in Nial," in *IEEE Software*, Jan. 1986, pp. 46-55.

- Kahn, K., Tribble, E.D., Miller, M.S. and Bobrow, D.G. (1986),
 "Objects in Concurrent Logic Programming Languages," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Sept. 1986, pp. 242-257.
- Kmorowski, H.J. (1982),
 "Qlog - The programming environment," in *Logic Programming*, Clark, K.L. and Tarnlund, S.A., eds., Academic Press, 1982, pp. 315-322.
- Lang, K.J. and Pearlmutter, B.A. (1986),
 "Oaklisp: an Object-Oriented Scheme with First Class Types," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Sept. 1986, pp. 30-37.
- Lieberman, H. (1986).
 "Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-oriented Systems," in *Proceedings of Third Workshop on Object Oriented Languages*, Bigre+Globule #48, Paris, Jan. 1986.
- Lieberman, H. (1987).
 "Concurrent Object-oriented Programming in Act1," in *Object-oriented Concurrent Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press, 1987, pp. 9-36.
- London, R.L. and Duisberg, R.A. (1985),
 "Animating Programs Using Smalltalk," in *IEEE Computer*, Aug. 1985, pp. 61-71.
- MacLennan, B.J., (1983).
Principles of Programming Languages: Design, Evaluation and Implementation, Holt, Rinehart and Winston, NY, first edition, 1983.
- Martin, J. (1985).
Fourth-Generation Languages, Volume 1, Prentice Hall, N.J., U.S.A, 1985.
- McCarthy, J. (1960).
 "Recursive Functions of Symbolic Expressions and Their Computation by Machine," in *Communications of ACM* 3(4), April, 1960, pp. 184-195.
- Moon, D.A. (1986).
 "Object-Oriented Programming with Flavors," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, September 1986, pp. 1-8.
- Moriconi, M. and Hare, D.F. (1986),
 "The PegaSys System : Pictures as Formal Documentation of Large Programs," in *ACM Transactions on Programming Languages*, Oct. 1986, pp. 524-546.
- Nassi and Schneiderman, B. (1973).
 "Flowchart Techniques for Structured Programming," in *SIGPLAN Notices*, Vol. 8(8), August 1973.
- Nguyen, V. and Hallpern, B. (1986).
 "A Generalized Object Model," in *SIGPLAN Notices*, Oct. 1986, pp. 78-87.
- Pagan, F.G. (1987).
 "A Graphical FP Language," in *SIGPLAN Notices*, March 1987, pp. 21-39.
- Peterson, J.L. and Silberschatz, A. (1983).
Operating System Concepts, Addison-Wesley, Reading Mass., 1983.
- Raeder, G. (1984).
Programming in Pictures, PhD dissertation, University of Southern California Los Angeles, Calif., Nov. 1984.

- Raeder, G. (1985).
 "A Survey of Current Graphical Programming Techniques," in *IEEE Computer*, August 1985, pp. 11-25.
- Reiss, S.P. (1986),
 "An Object-Oriented Framework for Graphical Programming," in *SIGPLAN Notices*, Oct. 1986, pp. 49-57.
- Reps, T. (1987).
 "Language Processing in Program Editors," in *IEEE Computer*, November 1987, pp. 29-40.
- Rich, E. (1983).
Artificial Intelligence, McGraw Hill, New York, 1983.
- Robinson, J.A. and Sibert, E.E. (1982),
 "LogLisp: Motivation, Design and Implementation," in *Logic Programming*, Clark, K.L. and Tarnlund, S.A., eds., Academic Press, 1982, pp. 299-313.
- Rubin, R.V., Colin, E.J. and Reiss, S.P. (1985),
 "ThinkPad: A Graphical System for Programming by Demonstration," in *IEEE Software*, Mar. 1985, pp. 73-79.
- Shoeman, M.L. (1983).
Software Engineering: Design, Reliability and Management, McGraw Hill, Singapore, 1983.
- Siklossy, L. (1976).
Let's Talk LISP, Prentice-Hall, Inc., Englewood cliffs, N.J., 1976.
- Snyder, A. (1986),
 "CommonObjects: An Overview," in *SIGPLAN Notices*, Oct. 1986, pp. 19-28.
- Stefik, M.J., Bobrow, D.G. and Kahn, K.M. (1986),
 "Integrating Access-Oriented Programming into a Multiparadigm Environment," in *IEEE Software*, Jan. 1986, pp. 10-18.
- Strom, R. (1986).
 "A Comparison of the Object-Oriented and Process Paradigms," in *SIGPLAN Notices*, Oct. 1986, pp. 88-97.
- Synder, A. (1986).
 "Encapsulation and Inheritance in Object-oriented Programming Languages," in *Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Sept. 1986, pp. 38-45.
- Tanimoto, S.L. and Runyan, M.S. (1986),
 "PLAY: An Iconic Programming System for Children," in *Visual Languages*, Chang, S.K., Ichikawa, T. and Ligomenides, P.A., eds., Plenum Press, 1986, pp. 191-205.
- Tichy, W.F. (1987),
 "What Can Software Engineers Learn from Artificial Intelligence ?" in *IEEE Computer*, Nov. 1987, pp. 43-54.
- Waite, W.M. (1973).
Implementing Software for Non-Numeric Applications, Prentice Hall, Englewood Cliffs, N.J., 1973.
- Wegner, P. and Smolka, S.A. (1983).
 "Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives," in *IEEE Transactions on Software Engineering*, vol.9, no.4, July, 1983, pp. 446-462.
- Winston, P.H. and Horn, B.K.P. (1984).
LISP, Addison-Wesley, Reading, Massachusetts, second edition, 1984.

Wise, M.J. (1986).

Prolog Multiprocessors, Prentice Hall, Englewood Cliffs, N.J., 1986.

Yokote, Y. and Tokoro, M., (1987).

"Concurrent Programming in ConcurrentSmalltalk," in *Object-oriented Concurrent Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press, 1987, pp. 129-158.

Yonezawa, A. and Shibayama, E., Takada, T. and Honda, Y., (1987).

"Modelling and Programming in an Object-oriented Concurrent Language ABCL/1," in *Object-oriented Concurrent Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press, 1987, pp. 55-90.

Zaniolo, C. (1984), "Object-Oriented Programming in Prolog," in *Proceedings, International Symposium on Logic Programming*, Atlantic City, IEEE, 1984, pp. 265-270.

CUHK Libraries



000303883