Queries and Views of Programs Using a Relational Database System

Mark A. Linton

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

December 1983

Submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate Division of
the University of California, Berkeley.

Research supported by NSF grant MCS-8010686, a State of California MICRO grant, and Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

ABSTRACT

Large software systems are expensive to develop and maintain. A significant amount of programmer activity in understanding, changing, and debugging these systems is information management. To support these activities more directly, we have designed a programming environment, called OMEGA, that provides powerful mechanisms for accessing and displaying the information in a large software system.

OMEGA uses successful ideas from existing programming environments while trying to correct deficiencies. The major deficiency in these systems is that programmers can only view and manipulate a single logical representation of programs.

To support multiple representations, OMEGA uses a relational database system to manage all program information. Using a database system provides a powerful mechanism for efficient access to a variety of cross-sections of program information, as well as providing traditional database facilities such as concurrency control, data integrity, and crash recovery.

The user interface to OMEGA separates input specification from output display, relies on pointing rather than typing, and exploits interaction in semantic analysis to detect many errors as soon as they are made. By eliminating the traditional textual interface to programs, OMEGA also allows the unification of the different abstraction mechanisms present in traditional programming environments.

We have experimented with the ideas in OMEGA by designing a relational schema for software written in a particular programming language, and by implementing a program that transfers existing programs, stored as text, into a database managed by an "off-the-shelf" database system. A prototype visual interface to the program database has also been implemented.

The results of this thesis are new models of program representation and user interaction for software development systems. The model of program representation can be expressed in the relational data model, and software can therefore be manipulated easily and powerfully using relational calculus. Our experimental implementation demonstrates the feasibility of using a relational database system, and provides insights into potential problems and how they might be solved.

Acknowledgments

I am very grateful to Mike Powell, my advisor, for his support and guidance throughout this work. His tremendous creative and analytical abilities have taught me more than I can show with this dissertation; his willingness to listen, his patience to wait for me to understand his ideas, and his friendship has made working with him an enjoyable as well as educational experience.

I am also very grateful to Larry Rowe and Lucien LeCam, the other members of my committee, for their time and helpful comments on this dissertation; to Mike Stonebraker, for his help and advice on how to approach the database issues discussed in Chapter 3; and to everyone else at Berkeley, particularly the other members of the OSMOSIS group, for providing a stimulating and friendly environment in which to work.

I am lucky to have had for the support of my friends and family who, throughout this work, and throughout my life, have believed in me and my abilities. My special thanks to my parents who have given me their love and support no matter how far away I have been.

Lastly, and most importantly, I am thankful for the love and support of Susan Ellis, my wife, who has helped me both technically and spiritually throughout this work, and whose companionship and terrific personality makes my life a happy one.

Table of Contents

Chapter 1

The Software Beast

In the master's chambers
they gathered for the feast.
They stab it with their steely knives,
but they just can't kill the beast.

- from the song "Hotel California" by The Eagles

1.1 Introduction

Millions of lines of software are written, executed, and debugged every year. The cost of producing this software is a large percentage of the cost of using computers today, and this percentage is increasing as hardware costs drop. *That* software is expensive to develop and maintain is accepted, but *why* it is so costly and *how* to reduce these costs is still an issue.

One reason why software is so expensive is that the amount of information in programs today is far too large for one person to absorb and comprehend. Furthermore, collections of programs and pieces of programs are combined together into *software systems*, and the interdependencies between these various pieces are complex.

As software systems evolve, they continually grow larger in order to add functionality, improve reliability, and enhance performance. This growth increases both the amount of information in the system and the complexity of the interconnections of the pieces. Making a change to a software system requires an understanding of how the part being changed fits into the system.

With the use of computers growing, more and more programs are being built for both existing and new applications. Paradoxically, less and less *new* software is actually being created; most of the code written today replicates or nearly replicates the function of existing programs. To reduce the cost of software then, programmers need to be able to adapt existing software to new requirements and to be able to understand the interdependencies of the pieces of a software system.

A major part of understanding is simply seeing the information relevant to what one is trying to understand. In this thesis, we present the design of a programming system, called OMEGA, that provides mechanisms for seeing and manipulating software in a much more powerful and general way than current systems. Instead of the traditional linear or hierarchical view, we use recent ideas in database systems to provide multiple *relational* views of the information in a software system. The relational model provides very powerful operations for describing portions of a database of information. By using it we can give programmers the opportunity to view and change a wide variety of cross-sections of a software system.

In the remainder of this chapter, we present the philosophy behind the design of OMEGA, discuss the goals and non-goals of this thesis, and present ~~the~~ basic results. In Chapter 2, these ideas are put in perspective by looking at other programming systems.

Chapter 3 presents our model of program information and how it can be supported by an existing database system; in Chapter 4 we show how this information is communicated between programmer and machine. Chapter 5 describes how we have investigated the feasibility and practicality of our ideas. A complete implementation of OMEGA was impractical for the purposes of a dissertation. Consequently, we focused on the details of the database interface and the basic features of the user interface. In Chapter 6 we summarize our work, suggest areas for future research, and draw some conclusions from the principles we have developed.

1.2 OMEGA Philosophy

The approach to constructing a program in a traditional system, pictured in Figure 1.1, begins with a programmer having an idea of an algorithm to implement. An implementation for the algorithm (or part of the algorithm) often already exists, but usually the programmer has no way of finding it, or else what he or she can find does not function as desired for the particular application.

To create a program that can be executed, the programmer "encrypts" the programming constructs that represent the algorithm (procedures, statements, variables, etc.) into a particular programming language. The resulting "cryptotext" is transmitted over an unreliable "wire" (i.e., fingers) to the system, which uses a compiler to try to "decrypt" the code and determine the intended combination of programming constructs.

Primarily because of the textual medium through which programs are transmitted, programmers must manipulate software at a physical rather than logical level. The concepts of files, lines, and characters have nothing to do with software, yet many programming environments force users to communicate in those terms.

In OMEGA, the programmer communicates in software terms such as procedures, statements, and versions. Instead of being *entered* as independent entities, programs are *constructed out of existing software parts*. If parts are required that are slightly different from existing ones, the existing parts are modified or new variations of them are created. Maintenance also requires looking at and modifying existing parts of software. Therefore, in OMEGA *the activities of program construction and maintenance are identical.*

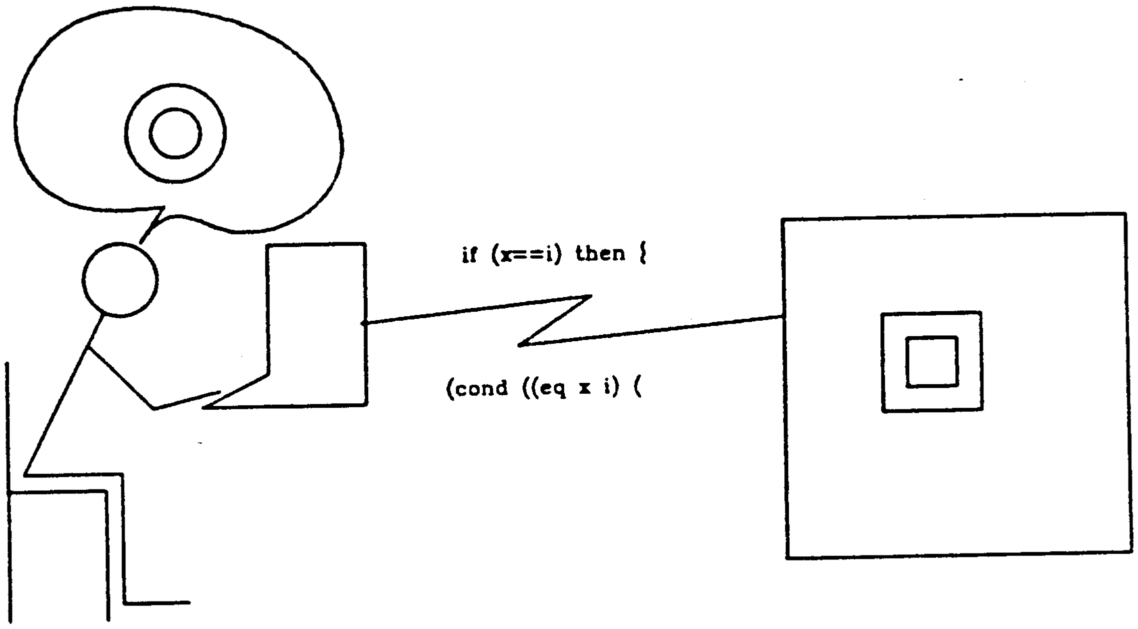


Fig. 1.1: Traditional programmer-machine communication.

For example, suppose one wished to create a compiler for the programming language Ada† [Ada 82]. Since the combined language features implemented by existing compilers cover at least 95% of the features of Ada, one theoretically should be able to build an Ada compiler quickly from parts of existing compilers. Furthermore, these parts have been debugged and tested, so they are likely to be more reliable than parts built from scratch.

Unfortunately, existing compilers are hard to understand, hard to interface to (even well-defined pieces such as code generators), and it is difficult to continue to share common parts over time. Consequently, the new compiler must be built at the same high price as the previous compiler and will require the substantial amount of maintenance that all large programs need. The new compiler is another victim of the software "beast" that strikes all large software projects today and consumes programmers, managers, and computing resources.

A single weapon, such as a new language or methodology, cannot stop the software beast. The design of OMEGA therefore combines ideas from programming languages and software methodologies with principles of database management and graphics-oriented user interfaces.

The most unusual aspects of OMEGA are its use and extension of current database ideas and its simple, pointing-based user interface. In focusing on these aspects we have left unspecified certain decisions, such as the particular language semantics built into the system.

1.3 Goals and Non-goals

The dream of being able to quickly build a reliable Ada compiler out of old compilers and to have fixes in the old compilers reflected in the new compiler exemplifies the fundamental goals of the OMEGA system. This thesis is only part of the work necessary to achieve those goals. We did not try to implement a complete system nor try to measure the effect this kind of system would have on programmer productivity.

We can only argue qualitatively that the activities of software construction and maintenance ought to be unified, and that a system that facilitates reuse of existing software will substantially improve programmer productivity. The purpose of this thesis is to solve some of the problems that lie within this framework through the design and experimental implementation of a programming system that provides powerful mechanisms for viewing and manipulating software.

We are not concerned here with the time and space costs of OMEGA. In the five years or so it may take to build a production OMEGA system, computing cycles and storage will become at least one order of magnitude cheaper than they are now. Also, it is usually easier to make a general solution fast and small than it is to make a fast and small solution general.

†Ada is a registered trademark of the Ada Joint Program Office, U.S. Government.

Nonetheless, it is important to make sure a design does not rely on any inherently expensive components. In the case of OMEGA the most important question about performance is whether the mechanism that is used to manipulate software (a relational database system) is too expensive. Another goal of this thesis, then, is to implement enough of OMEGA to discover performance problems and determine how they might be overcome.

1.4 Results

This work contributes a new model of program representation and user interaction for software development systems. The representation of program information to the user provides powerful operations on arbitrary cross-sections of programs. We show that this model can be described with relational calculus, and therefore that program information can be managed by a relational database system.

Our model of user interaction takes advantage of a relatively recent style of entering information into a computer – pointing. Pointing is a quick, precise way of identifying and rearranging objects that removes the need for parsing and name resolution. It therefore allows the facilities of various programming languages to be unified and extended. Programs can be displayed in the form desired by the user rather than in a form required by a compiler.

The model of program representation has been explored further by actually implementing an interface to the relational database system INGRES [Stonebraker, Wong, and Kreps 76]. A schema for a particular programming language, called Model [Morris 80], has been designed and a program written that translates existing Model program text and stores it in an INGRES database. Another program has been written that provides views and allows updates of the information in the database.

The implementation demonstrates the feasibility of using a relational database system and therefore the practicality of our theoretical model of program representation. It also provides an insight into potential performance problems and how they might be solved.

Chapter 2
Historical Perspective

I tip my hat to the new constitution,
take a bow for the new revolution,
smile and grin at the change all around,
pick up my guitar and play
just like yesterday,
then I get on my knees and pray
we don't get fooled again.

– from the song "Won't Get Fooled Again" by The Who.

2.1 Introduction

In designing OMEGA we have tried to use and generalize the positive aspects of existing programming systems while avoiding the negative ones. For example, Interlisp [Teitelman and Masinter 81] is an *integrated* environment; it has a single, consistent user interface and a uniform representation of programs. These are clearly good attributes for a programming system, therefore we designed OMEGA to be integrated. However, unlike Interlisp, which is centered around a list-oriented database, OMEGA is based on relations, thus providing more powerful and efficient operations.

Ideally, whether a feature is good or bad should be measured by the quantitative effect it has on programmer productivity. Unfortunately, not only is this effect hard to measure, it is difficult to measure productivity itself. The traditional measure of programmer productivity is the number of lines of code written. However, this number is misleading when building upon existing software rather than writing new software.

For example, a programmer who adds or changes 100 lines in an existing 10,000 line program will produce a working version much faster than someone who writes the 10,000 line program from scratch, but much slower than someone who writes a 100 line program from scratch. The larger program performs much more than the 100 line program, so the programmer who modifies the existing software should be considered more productive.

Counting the number of lines modified is not an accurate measure either. For example, changing the name of a variable changes every line in which the variable appears, but is not as difficult or useful as a more substantive change that involves the same number of lines.

Even if it were feasible to measure the effect of a particular system on programmer productivity, it is very difficult to construct an experiment that compares existing systems while factoring out their idiosyncrasies. We therefore turn to a qualitative analysis.

To analyze and compare programming systems, it is necessary to use a common model of the facilities that the systems provide. Systems as different as UNIX† [Kernighan and Mashey 81] and Smalltalk [Goldberg and Robson 83] cannot be compared by comparing respective editors, languages, or debuggers. A more general model is needed to take into account things such as the fact that Smalltalk provides an integrated environment whereas UNIX does not.

To provide the necessary generality and completeness, we model programming as a *communication* process. In the remainder of this chapter, we describe this model and then use it as a framework for discussing the programming systems that have influenced the design of OMEGA.

2.2 Communication Model

Programming is a communication activity between programmer and computer (as well as among programmers). As such, it has the three basic components:

- medium,
- protocol, and
- data.

The medium of communication is the physical connection, e.g., punched cards, printer, terminal, mouse, or voice. The protocol is the way data is transmitted through the medium to and from the machine. For the programmer, this includes both how individual commands are specified and the order in which commands are applied to perform a particular task. For the system, the protocol determines how commands and operands are interpreted, and where responses are sent. In a screen-oriented system, for example, the protocol determines where on the screen the response to a command is displayed.

The final component of communication is the data that is exchanged. There are two important aspects of the data: the kinds of objects that are communicated, and the operations that can be performed on the objects. The kinds of objects manipulated through a text editor, for example, are files, lines, words, and characters; typical operations allowed include insert, delete, and replace.

Programmers manipulate both objects associated with the definition of a program, called "program" or "static" data, and objects associated with its execution, called "runtime" or "dynamic" data. Runtime objects are represented

†UNIX is a registered trademark of Bell Laboratories.

and manipulated in terms of the underlying machine on which the program is executing.

Objects may have constraints placed on their values to ensure that they are meaningful. For example, the left-hand side of an assignment statement can be a complicated expression, but it must be an expression that will correspond to a storage location when the program is executing.

2.3 Individual Systems

Using our communication model of programming, we now turn to a discussion of the particular systems that have influenced OMEGA. For each system, we present an overview followed by an analysis of the features that we have tried to include or avoid in the design of OMEGA. This analysis is in terms of the communication model, thereby allowing us to focus on the fundamental elements of each system and avoid a myriad of specific details.

2.3.1 UNIX

The UNIX programming environment† is one of the most popular *tool-based* systems. A tool-based system provides a number of individual programs that each aid some part of the overall task. Ideally, each tool is small enough that it is relatively uncomplicated to build and use, while the combination of tools provides substantial power for developing and maintaining software.

The nucleus of a tool-based programming system consists of an editor for entering and modifying programs, a compiler for translating programs into an executable form, and a command interpreter for executing programs (including the editor and compiler). In addition, UNIX provides tools for building a configuration of a system, managing versions of modules in a system, interactive debugging, and indexing symbol definitions. Figure 2.1 shows the organization of the UNIX environment.

One of the advantages of the tool-based approach is that tools can easily be added to the system and users can build tools for their own particular needs. This ability is facilitated by the pipe and file redirection operations provided by the UNIX command interpreter that allow tools to be easily combined into new tools without modification. Thus, programmers can use existing programs instead of re-implementing them, one of our main goals for OMEGA.

Despite these desirable features, there are several problems with tool-based systems in general and UNIX in particular. First, if more than one tool is interactive (e.g., editor and command interpreter) then there is more than one user interface. Not only does this replication lead to varieties and inconsistencies that the user must remember, but each tool must have a command line scanner, parser, and execution processor. For example, the UNIX command interpreter

†We do not distinguish between the UNIX environment and the programmer's workbench (PWB/UNIX) [Ivie 77].

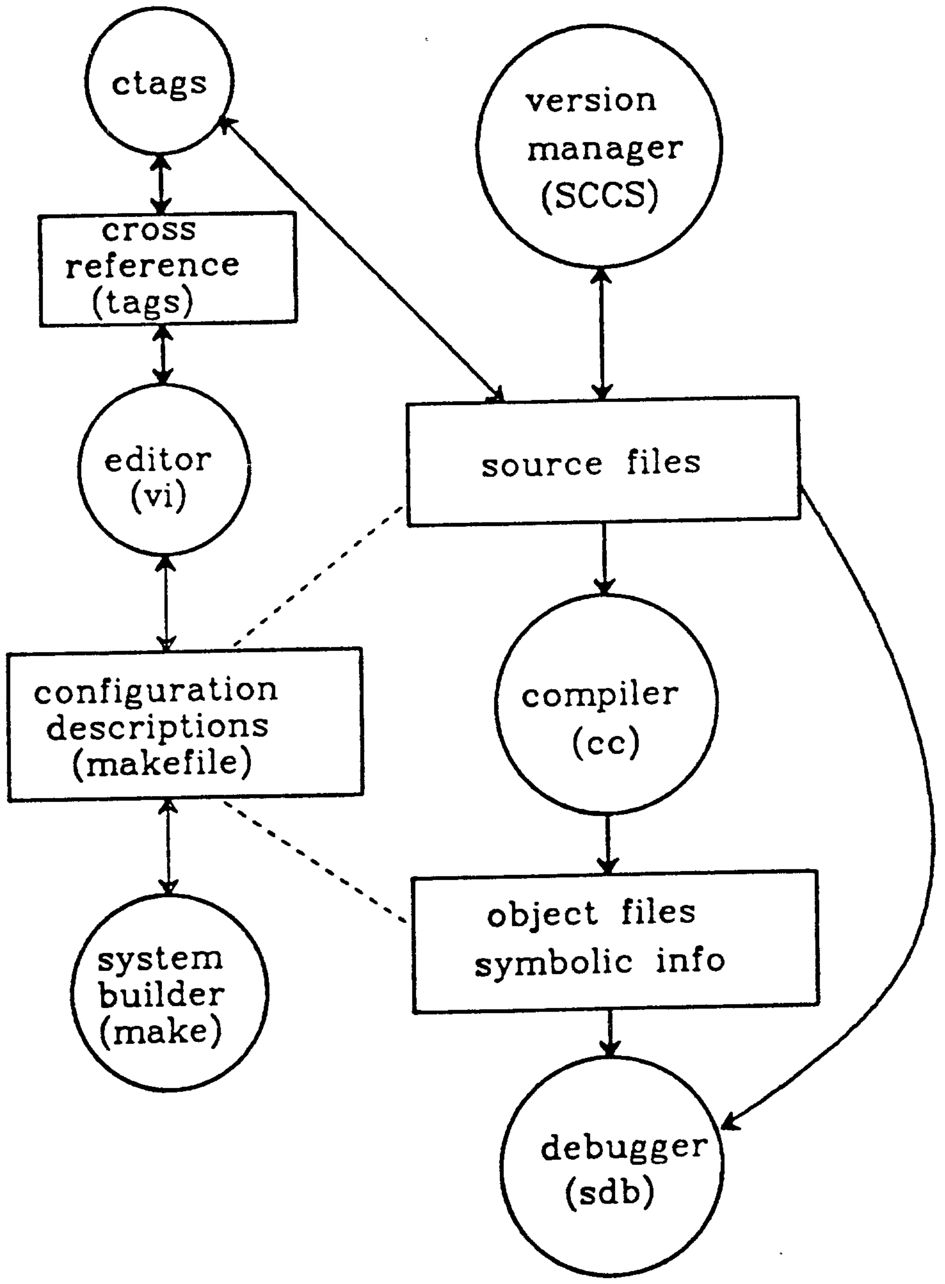


Fig. 2.1: Unix information and tools.

expands special characters in filenames to match files the user can access. The UNIX editor *vi* [Joy 79] duplicates some of the matching facilities in processing commands that involve filenames, but the facilities are not completely consistent with those provided by the command interpreter.

In terms of the communication model presented earlier in this chapter, the problem is that the user needs different protocols to perform the same operation in different contexts, and may not even be able to perform the operation in some contexts. The programming system must either implement these multiple, redundant protocols or not consistently provide access to all the operations.

The most important problem with UNIX is that the objects that a programmer enters and manipulates are at a physical level, namely text. There are several levels of text (files, lines, words, characters), but any correspondence between these objects and programming entities such as modules, statements, or variables is only by convention, and is not enforced by the system. Since only text is stored, semantic interpretation of the information (e.g., the binding of names and propagation of types for expressions) is expensive.

Historically, text was a natural and compact way to represent programs communicated through the medium of punched cards. As Figure 2.1 shows, UNIX has taken advantage of the availability of on-line storage by augmenting the program text with configuration files, symbol definition indices, and differential files for restoring previous versions. Except for information about previous versions, this data is already available from the program text. It is replicated to significantly improve the performance of the corresponding tool. For example, the configuration file contains dependency information that can be determined from the program text, but it is too expensive to have the system builder analyze all the program text.

The problem here is not that the information is replicated; this is a standard technique for improving performance. The problem is that usually the user has to maintain the redundant information by hand because it is too inefficient to have it continually recomputed.

In summary, UNIX is a popular tool-based environment with a simple, text-oriented representation of data and a powerful command interpreter. The environment is easy to extend and existing tools can be combined in flexible ways to perform new tasks. However, the tool-based approach leads to a proliferation of user interfaces (protocols). Representing programs as text provides a single, linear view of programs, which makes it difficult for tools to manipulate small portions of programs. Consequently, these portions are often replicated and therefore likely to become inconsistent.

2.3.2 Interlisp

In contrast to the tool-based approach of UNIX, Interlisp provides an integrated environment. All the pieces of the system share a common user interface and a common representation for programs and data. Facilities such as error recovery for misspelled words and the repeating of previous commands can be used when editing, debugging, or executing programs.

Like UNIX, the facilities of Interlisp can be extended through the modification of an existing piece of the system. Since Interlisp already contains many interconnected pieces, there is a package called Masterscope that helps programmers see the interconnections between modules.

Masterscope has an English-like interface for questions about the structure of a program. For example, to find the functions that use the variable *x* one might type

```
(who uses x)
```

Masterscope also works in conjunction with the Interlisp editor. For example, one can ask to edit functions that use the variable *x* or satisfy some predicate. More powerful editing can also be performed using the program information maintained by Masterscope. For example, it is possible in a single command to replace all occurrences of the variable named *x* with the variable *y*.

The predicates that can be specified to Masterscope are described by a set of templates that can be extended by the user. The result is an ad hoc language for a particular set of queries on program information.

The structure and semantics of objects in Interlisp is based on lists, and the operations provided include creating, extending, and searching these lists. Lists are a simple data structure that provide a way of grouping objects analogous to sets, but are inefficient to search. As with text in UNIX, to make any kind of analysis of the program it is necessary to either scan through all program information or have the desired subsets of information replicated. Interlisp also does not allow constraints on programs (e.g., the number and types of parameters to a function) to be enforced automatically prior to execution.

One nice feature of the Interlisp environment is that runtime data has the same representation as program data. This feature, together with the uniform user interface, makes Interlisp a much better environment for debugging than tool-based systems such as UNIX. Unfortunately, the price paid by Interlisp is that execution is interpretive and therefore an order of magnitude slower than it be if it were compiled. It is possible to compile Interlisp programs, but doing so disables the high-level debugging facilities.

OMEGA is designed to be like Interlisp in that it has a single protocol for manipulating a uniform representation of programs and data. However, the protocol in Interlisp is text-based, which requires parsing commands and resolving names. Processing text input opens up the possibility for syntactic and semantic errors, something we want to avoid.

The major weakness of Interlisp is its representation of programs as unconstrained lists of objects. List operations are not powerful enough to provide efficient access to the wide variety of program views that are useful in manipulating programs. For example, there is no general mechanism for quickly indexing on an element of a sublist within a larger list; this mechanism would be useful in processing requests such as "edit all functions that expect more than one argument".

2.3.3 Smalltalk

Smalltalk, like Interlisp, is an integrated environment. However, it is very different from both Interlisp and UNIX in its communication medium and protocol. Programmers communicate with UNIX or Interlisp through a conventional keyboard and CRT; with Smalltalk they use a large screen and a pointing device (e.g., a mouse).

This medium allows a protocol in Smalltalk based on pointing at objects and displaying output for different activities in separate "windows", independent rectangular areas on the screen. The use of a large screen in conjunction with windows and pointing makes attractive the use of menus for selecting commands and data. Previous menu-based systems such as UCSD Pascal [Softech 79] used only a small portion of the screen for the menu and allowed access to only one menu at a time. As a result, using menus was tedious and systems like UNIX have disdained them.

Pointing is an important mechanism because objects can be denoted without the user remembering and typing a string of characters, and without the system having to resolve the name using some form of symbol table. A mouse or similar pointing device makes pointing feasible since it allows the user to move the cursor quickly and accurately.

Smalltalk also provides a different structure and semantics for objects than previous systems. Objects are grouped hierarchically into classes; each class is itself an object and, except for the root class, belongs to another, higher-level class. Associated with each class is a set of operations that are defined on objects in that class.

Binding of operator names to operations is done entirely at runtime, meaning constraint checking cannot be done before execution begins. This dynamic binding also causes a severe execution performance penalty.

Smalltalk is a single-user environment; there are no aids for developing multi-user projects. There is nothing like Masterscope to help a programmer see the interconnections between pieces of programs. The intention is to use Smalltalk for quickly implementing and testing out ideas. Once the ideas work a complete implementation can be done in a more traditional environment.

We do not believe this dichotomy between a prototyping environment and a production environment is either necessary or advantageous. The fastest way to produce any implementation of ideas is to use and modify existing implementations. In OMEGA both prototype and production software is evolved from existing software. The difference is that a prototype is built quickly from pieces that might only approximate a desired function, whereas the production software has refined the pieces to provide a complete and more precise implementation.

The importance of Smalltalk is in its medium (large screens and mice) and associated protocols (pointing, windows, and menus). We have tried to use and improve upon these concepts in designing OMEGA. Chapter 4 discusses how the use of pointing is generalized in OMEGA.

2.3.4 Cornell Program Synthesizer

Interlisp and Smalltalk are both integrated environments for simple, unconstrained languages. The Cornell Program Synthesizer [Teitelbaum and Reps 82] was one of the first attempts to build an integrated environment for a language that has complex syntactic and semantic constructs. It features syntax-directed editing, incremental semantic analysis, and reversible execution. Like Smalltalk, the Synthesizer is a single-user environment with no facilities for handling large, multi-programmer systems.

To provide an integrated environment for the language PL/C [Conway and Constable 76], a simplified version of PL/I, the Synthesizer provides a uniform command language in which some of the commands create syntactic constructs. For example, the command "WH" causes a prototype **while** loop to be inserted where the cursor is located in the program.

In the Synthesizer, the visual representation of a syntactic construct is called a "template", and templates can contain placeholders for other objects. For example, the while loop template might look as follows:

> **while** *condition* **do**
>   *statement*

Here both the *condition* and *statement* are holes to be filled with a boolean expression and simple or compound statement respectively.

Unlike other syntactic constructs, expressions are entered as text and parsed. This distinction was done to avoid requiring a large number of commands to produce a comparatively small amount of code. Semantic analysis in the Synthesizer is done ad hoc.

An important idea behind the Synthesizer is that there is a separation between the way a program is entered and the way it is displayed. This separation is the key to avoiding the problem of integrating an environment for a language with a non-trivial syntax. In the Synthesizer this facility is not extensible; in OMEGA we allow the display templates and programming constructs to be defined by the user. This facility allows one, for example, to define a variety of loop constructs that are useful for solving common problems.

2.3.5 Gandalf / IPE

Although the Synthesizer provides an integrated environment for a more structured language than Smalltalk or Interlisp, it still interprets execution and does not support development of large, multi-programmer systems. The Gandalf project [Habermann, et al. 82] is an attempt to provide an integrated environment that executes compiled code and supports multi-programmer systems.

The part of Gandalf that interfaces directly to the programmer is called the Incremental Programming Environment [Medina-Mora and Feiler 81]. The IPE user interface is similar to the Synthesizer, except expressions are built the same way as other constructs, not as text as in the Synthesizer. This uniformity makes it easier for IPE to support a number of different languages since there is no parsing done at all. Semantic analysis, however, is still done ad hoc.

One of the traditional advantages of interpreting over compiling is that execution can begin almost immediately without waiting for code generation and linking. IPE performs code generation on one function in the background while the user is editing another. Since editing as an activity is rarely CPU bound, this technique allows the load of compilation to be distributed more evenly over time and helps avoid long delays for the user.

Like the Synthesizer, Gandalf is ***construction-based***; it is designed to support the creation of new software. Although there are aids to create large systems, the user interface and data model provide no mechanism for easily querying or viewing the interconnections within a program.

Both Gandalf and the Synthesizer provide a hierarchical model of programs. This model is natural for top-down development but unsatisfactory for program maintenance or improvement. For example, when adding a parameter to a procedure one would like to be able to view and modify each call to the procedure. This view is not easily attainable with only a hierarchical view of the program.

Gandalf also separates operations on objects within modules from operations on modules and objects that contain modules. For example, the configuration and version control facilities work with modules as the basic element whereas IPE can only manipulate the information in an individual module. The Cedar system [Deutsch and Taft 81] also makes this separation, but we feel that this is an artificial separation that hinders rather than helps programmers.

Modules are one form of grouping objects in a program. Other forms include procedures that use a given module, statements that reference a variable, and variables that may be modified by a particular module. What group of software objects a programmer wishes to see depends on what the programmer is doing. The hierarchical structure of modules is useful in top-down development, but when debugging, modifying, and extending software other kinds of grouping are necessary.

2.3.6 Programmer's Apprentice

The programmer's apprentice (PA) [Waters 82] is aimed at augmenting rather than replacing existing programming environments. The idea is to use a software knowledge base to make it easier for programmers to construct, understand and modify programs.

The philosophy of the PA is similar to that of OMEGA. Using the PA, a programmer manipulates *plans*, common programming constructs with associated semantics, to construct and modify programs without concern for the details of the code associated with a plan. The system can automatically generate code from plans, extract intended plans from existing code, and display plans graphically on a terminal.

Using a library of constructs such as plans during software development and maintenance is one of the goals of OMEGA; however, our approach is very different. Interaction with the PA is through typing pseudo-English commands, whereas OMEGA is menu and pointing based. Our approach eliminates the problem of misinterpreting the structure or semantics of commands and automatically presents the possible options to the user.

The library of plans is an ad hoc database that the PA accesses as it interprets commands; in OMEGA we use a general-purpose database system. By doing so, we do not have to implement efficient access methods or manage secondary storage, and the user can express a wide range of queries.

In time it may be possible to combine ideas from both the PA and OMEGA to provide both flexibility in accessing the database and intelligence in interpreting the knowledge that is in the database. However, in this thesis we are focusing on the problems that are involved in supporting flexible access to, rather than human-like interpretation of, programs.

2.4 Conclusions

There are many more systems we could discuss, but they are similar to one of the systems we have analyzed. Although the details of the systems we have presented are very different, the systems are actually quite similar in the way that programmers communicate with them. Except for Smalltalk, they all use a conventional terminal as the medium. Except for UNIX, each system is integrated, meaning a single protocol or command interface for the user. The Smalltalk protocol is also different in its use of pointing, menus, and windows.

The structure of the data that the programmer manipulates is either text (UNIX), lists (Interlisp), or hierarchical (Smalltalk, Synthesizer, and Gandalf). Data integrity is either checked periodically by a compiler (UNIX), incrementally by the system (Synthesizer and Gandalf), or mostly left to the user (Interlisp, Smalltalk). The incremental checking that systems perform is implemented ad hoc and therefore difficult to extend or change.

Smalltalk, the Synthesizer, and the IPE part of Gandalf are all construction-oriented; the operations they provide on data are aimed at creating and deleting objects in a small, contiguous piece of a program. They do not provide facilities for maintaining and evolving software that is viewed and modified by a number of programmers.

UNIX and Interlisp do provide some help, but because of their representation of programs, the operations they provide are inadequate and low level. In UNIX, for example, the basic data structure is a file of text. By convention, each module is stored in a separate file. Programmers must decide on the physical implementation of sharing part of a module, whether it be to periodically copy files or to share a single file that contains conditional compilation tags to identify non-shared portions.

The major problem of all these systems is that their representation of programs is not well-suited to the needs of a programmer working on a large software system. Text-oriented systems provide a physical rather than logical view, and tree-oriented systems provide a single logical view that does not match the view a programmer needs when maintaining and enhancing a software system.

In the next chapter we describe the representation that OMEGA presents, and show how this representation provides the functions of the systems presented here as well as facilities these systems cannot. Recognizing that the manipulation of program information is an instance of the general problem of data management, we use a general-purpose database system to provide a powerful set of operations on software and free OMEGA from storage management concerns.

Chapter 3

The Program Database

Prisoner: *What do you want?*
Number 2: *We want information.*
Prisoner: *You won't get it.*
Number 2: *By hook or by crook, we will.*

– from the television show *The Prisoner*

3.1 Introduction

Software constantly changes as new features are added, bugs are fixed, and new hardware technology is exploited. Programs are not self-contained; they use and interact with algorithms, data structures and subroutines from existing programs or libraries. Consequently, programmers need to understand existing software in order to use and modify it to meet new requirements as well as to create additional, compatible, software.

Most programs are too large to understand in complete detail; hence, programmers select different views to understand different aspects of them. Understanding the implementation of a procedure may involve looking at its statements; understanding how a variable is manipulated may involve looking at the statements that access the variable; understanding how a running program reaches a certain state may involve looking at statements in the order in which they are executed.

In the previous chapter we saw that current programming systems do not support these and other views of large collections of software. To provide the most powerful mechanism for describing views, OMEGA needs a general solution to the problem of supporting multiple logical views of a large amount of information.

General-purpose database systems provide this solution with facilities for defining views of a database and, through the processing of requests called queries, for retrieving views from the database. By using a database system to manage procedures, statements, variables, and the other information that makes up a program, OMEGA avoids constraining the ways in which a programmer can view software, and avoids duplicating the function of a database system.

Database systems provide many other useful facilities in addition to the ability to retrieve and define general views of data. They manage permanent storage, support efficient data access, provide concurrency control, attempt to recover from crashes, and try to ensure the integrity of the data. All of these problems arise in software development systems; using the work of database researchers allows us to address issues specific to programming environments.

In the remainder of this chapter we describe the representation of programs that OMEGA presents to the user, and how this representation can be translated into a schema for a traditional database system. This representation includes the integration of runtime information into the database, thereby providing support for debugging activities. We suggest two extensions to database systems to increase the power and improve the efficiency with which program information can be accessed. Finally, we show how OMEGA uses the database system to support traditional programming operations, including editing, symbol table management, and configuration building.

3.2 Semantic Data Model

Two popular program representations are text and trees. Text is expensive to extract program semantics from and, therefore, inefficient to use in processing most queries. Simply distinguishing comments from program statements requires scanning each character. Non-trivial queries, such as finding all the uses of the "+" operator in which both operands are integers, requires parsing and semantic analysis of the entire program.

Update operations on program objects (statements, expressions, and variables) must be translated by the programmer into operations on text objects (lines, words, and characters). This translation can be complicated; for example, changing the name of a variable requires string substitution every place the variable is used, which is not necessarily the same as every place the string appears.

The hierarchical view provided by tree-oriented systems is better than the linear view of text-oriented systems, but is still only a single view and therefore inadequate. Suppose, for example, a programmer wanted to port some software to a new machine. In doing this, the programmer might wish to look at all the constants defined throughout the software. However, it is likely that the constants would have been defined in the different modules where they logically belonged. A system that provides programmers only one organization of programs cannot satisfy the variety of activities that make up software development and maintenance.

The program database is so large that the complete structure of it is not of interest to programmers. At any given time, depending on the particular task, a programmer needs to see some cross-section of the database that contains the relevant objects. We call these cross-sections *program threads,* as they correspond to strands of connected objects in a large fabric of software.

Text and tree organizations are two kinds of program threads. Other examples include the following:

- statements that reference a variable
- procedures that use a module
- statements executed for certain input
- modules written before a certain date
- constants defined for a particular machine

The concept of a program thread captures many separate facilities from current programming environments, including structure-oriented editing, module dependency analysis, cross-reference listings, call graph generation, version history manipulation, and execution trace generation and analysis. In addition, there are many other possible program threads corresponding to particular information in which a programmer might be interested. For example, one might wish to see the uses of an I/O procedure where the parameter designating a file has a particular value.

The use of program threads also eliminates redundancies that arise from having separate facilities. For example, both module dependency analysis and cross-reference listing generation require preliminary analysis in text or tree systems. Frequently this analysis is replicated for each facility, and therefore is likely to become inconsistent.

Using OMEGA, programmers define, retrieve, and update objects and threads of objects. These operations provide a semantic model of software that supports many different views of programs, and therefore can support the various activities that are performed during the different stages of software development.

3.3 Picking a General-Purpose Database System

To use a general-purpose database system to implement program threads, it is necessary to map the program thread model onto a general-purpose data model.

The three data models whose implementation has been pursued most extensively are referred to as the *relational*, *network*, and *hierarchical* models†. The relational model is based on collections of objects, called relations, where each collection is made up of homogeneous objects, called tuples. A tuple is made up of fields that contain individual values. One or more of the fields form a *logical key*, whose value distinguishes the tuple from other tuples within the relation. A small number of powerful operators are defined on relations that allow general queries and views to be specified.

†For background and a more complete discussion of these data models, see [Ullman 80].

In the network model, data is represented by nodes of information and links between related nodes. The resulting graph has the advantage of more naturally representing program information; however, network systems do not provide the query processing and view definition capabilities of relational systems.

The hierarchical model is a special case of the network model; data is organized as a tree rather than a general graph. This restriction often makes it possible to access data more rapidly than in other models. Since program threads include non-hierarchical information, both the network and relational models are better suited for our needs than the hierarchical model.

Other data models, such as the *entity-relationship* model [Chen 76], have been proposed to add semantic extensions to the relational model. The entity-relationship model provides a semantic model that is very close to program threads, with program objects corresponding to entities and threads corresponding to relationships. However, work on this model has, until recently (e.g., [Cattell 83]), focused on semantic rather than implementation issues.

Our choice was to have OMEGA use a relational database system rather than build query and view capabilities on top of a network system, and therefore offer programmers flexibility in describing views of programs without having to provide our own query processor. This choice allowed early experimentation with program queries, while sacrificing performance and elegance. In the next section, we show how program information can be stored in the traditional relational model and suggest enhancements to help manage this kind of information.

3.4 Storing Program Information

To process queries on program semantics, it is necessary to have the information that a compiler builds during its parsing and semantic analysis phases. This information consists of some form of program graph and symbol table. Figure 3.1 shows part of the program graph for the following program fragment:

```
prevmax := max;
if a > b then
   max := a;
else
   max := b;
end if;
```

The tables in Figure 3.2 show how the information in the graph in Figure 3.1 could be stored in a relational database system. For the sake of clarity we have simplified this description by omitting some relations, such as those associated with type information, and using names rather than numbers for certain values. A complete schema for program information is described in Chapter 5.

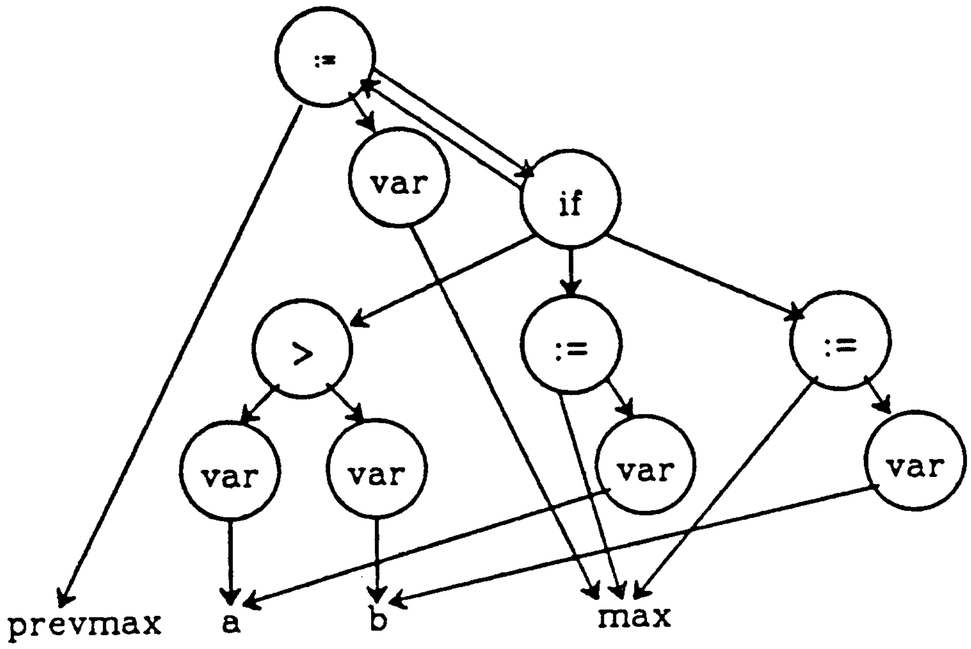


Fig. 3.1: Program graph for fragment.

statements relation

id	stmt-rel	stmt-id	next
6922	asgstmts	6923	6924
6924	ifthens	6925	0
6930	asgstmts	6931	0
6932	asgstmts	6933	0

exprlists relation

id	expr-rel	expr-id	next
6928	variables	6578	6929
6929	variables	6579	0

variables relation

id	name	type-rel	type-id
6578	482	typenames	6221
6579	484	typenames	6221
6580	838	typenames	6221
6581	6577	typenames	6221

asgstmts relation

id	lhs-rel	lhs-id	rhs-rel	rhs-id
6923	variables	6581	variables	6580
6931	variables	6580	variables	6578
6933	variables	6580	variables	6579

names relation

id	identifier
6577	prevmax
838	max
241	>
482	a
484	b

ifthens relation

id	condlist	else-rel	else-id
6925	6926	statements	6932

condlists relation

id	cond-rel	cond-id	then-rel	then-id	next
6926	fcalls	6927	statements	6930	0

fcalls relation

id	function	exprlist
6927	6231	6928

functions relation

id	name	class	parameters	type-rel	type-id	decls	stmtlist
6231	241	builtin	6628	typenames	6206	0	0

Fig. 3.2: Relations for information in 3.1.

Each tuple in the ***statements*** relation corresponds to a program statement. We associate a unique identification (UID) with each tuple, represented by a number, and use this number to refer to the tuple from other tuples. A UID is a logical key for a tuple in a particular relation, since it uniquely identifies the tuple within the relation and does not depend on the tuple's physical location.

Since some statements can contain an arbitrary number of other statements, this key is required to associate all of the contained statements with the containing statement. Statements may be nested in other statements to arbitrary depth. UIDs thus also provide a way to represent a hierarchy in a relational database.

Many program objects are like statements in that they may contain objects of their own kind. We call data structures to represent such objects ***recursive data structures.*** UIDs represent instances of recursive data structures from within other structures.

OMEGA allocates UIDs and can request a tuple from the database system using its relation and UID. Unfortunately, this interface does not allow the database system to retrieve the data efficiently nor does it allow OMEGA to perform the queries it needs.

Consider, for example, the relations introduced in Figure 3.2. If we wanted to print a tuple from the *ifthens* relation, we might use the following algorithm:

```
PrintString("if ");
PrintCondList(if-condition-list);
if if-else-part ≠ 0 then
   PrintString("else");
   NewLine; Indent(+4);
   PrintObject(if-else-rel, if-else-id);
   Indent(-4); NewLine;
end if;
PrintString("end if");
```

A straightforward implementation of this code generates separate queries for the *if-condition-list* and *if-else-part* fields of the *ifthens* tuple. Performing several independent queries is more expensive than a single, larger query because the database system can optimize operations for the larger query. There is also an inherent overhead for a query that involves reading the schema and maintaining information for concurrency and crash recovery. To display a procedure may involve traversing several hundred objects; trying to process the resulting several hundred queries quickly enough to avoid making the programmer wait could require an unnecessarily large amount of processing power.

An alternate approach is to add an attribute to each of the relations to indicate in which procedure they are located. Before processing any part of a procedure, one query could be used to retrieve all the tuples associated with the procedure into memory. The individual queries are then performed on this in-memory data. Although this provides more efficient access, the mechanism is outside the normal database system, and thus only a short-term solution.

This approach raises some problems that must be understood before suggesting extensions to the database system to replace it. Retrieving all the information at once for a large procedure is undesirable because the user must wait for the entire procedure to be retrieved before viewing any part of it. This wait would be particularly annoying if only a simple query needed to be done. For example, to display the statements that reference a particular variable, it is not appropriate to retrieve all the statements in all the procedures containing references to the variable.

When traversing information in the database, what we would like is for the database system to prefetch and cache tuples that are about to be referenced. Since the system is not aware of the semantics of the UIDs, it will be difficult for it to know which tuples are best cached in memory. The standard cache consistency problems must also be addressed.

A second issue is raised by the recursive nature of program structures. Consider a query that asks for all the statements that reference a particular variable. To discover whether or not the variable is in the statement, this query needs to examine the expressions in a statement and all subexpressions of those expressions, to whatever depth expressions are nested in the statement. There is no way in the relational model to express queries that involve a transitive closure. Therefore, such queries can only be made through separate queries for each stage of the closure.

3.4.1 Managing Recursive Data Structures

The issues of efficient access to, and transitive closure queries on, recursive data structures can be solved only by having the database system understand the recursive nature of the data. We propose to supplement the standard database value domains of integers, strings, etc., with a domain of *tuple references*. Values in this domain would provide information the database system could use to prefetch or retain tuples likely to be accessed. In addition, the transitive closure of a tuple reference can be defined and used in queries.

Tuple references are similar to foreign keys [Codd 70], and unique ids as proposed in [Codd 79]. We have extended these ideas, allowing tuple references to be manipulated through the query language; such usage may cause implicit join operations to be processed. The GEM database language [Zaniolo 83] has an equivalent facility. Unlike this and other work, whose motivation has been to provide a better semantic data model, our motivation has been to improve the performance of a series of small queries that are the result of traversing a portion of a graph. We now examine this proposal in more detail.

3.4.2 Tuple References

A tuple reference denotes a logical key for a tuple in some relation in the database. We use the notation "A = **ref**" to define the attribute A as a tuple reference. The only difference between tuple references and other fields of relations is that *their values are generated and interpreted by the database system.* All normal database operations apply to tuple references. Additional operations, described below, are also valid.

Although there are several possible implementations of tuple references, we assume that it is always possible to determine in which relation a referenced tuple is by saying *relation*(*r*), where *r* is the tuple reference. Thus, without loss of generality, we may think of a tuple reference as a pair (relation, tuple UID), even if the implementation is otherwise. The value of a tuple reference is generated automatically and is independent of the physical location of the tuple. One distinguished value that any tuple reference can have is a reference to no tuple, similar to the value *nil* in many programming languages.

Often an attribute always refers to a particular relation; in this case we use the notation "A = **ref** R", where R is the name of the relation. This notation implicitly defines an integrity constraint that restricts the attribute to refer to tuples in one particular relation. It also improves the readability of attribute definitions and allows the database system to perform optimizations such as minimizing the space needed to store a tuple reference.

Whereas the *ifthens* relation in Figure 3.2 would be defined in INGRES as

ifthens (id=integer, condlist=integer, else-rel=integer, else-id=integer),

it can be defined using tuple references as

ifthens (condlist = **ref** condlists, else = **ref**).

The value of a range variable in a query is the tuple reference for a tuple in the associated relation. For instance, the following example creates an *ifthens* tuple for an if-then statement (which requires a condition list, and an else statement):

```
range of c is condlists
range of e is statements
append to ifthens (condlist=c, else-rel="statements", else-id=e)
where {predicates to select the c and e we want}
```

The language we use for database operations here and in examples throughout this chapter is QUEL, the query language for INGRES, with extensions for tuple references.

In addition to normal database operations, it is possible to *dereference* a tuple reference by qualifying it with an attribute name. For example, the following query finds the **if** statements that have a condition that is simply a boolean variable:

```
range of i is ifthens
retrieve (i.all) where
   i.condlist.cond-rel = "variables"
```

If the specified attribute of a tuple reference is itself a tuple reference, it too may be dereferenced. It is therefore possible to qualify "i.condlist" as a normal range variable (in this case, of the *condlists* relation), and refer to its *cond-rel* attribute as "i.condlist.cond-rel".

A dereference is a simple notation for expressing an equi-join, with the result known to contain a single tuple. Using normal notation, the query in the example above is expressed as

```
range of i is ifthens
range of c is condlists
retrieve (i.all) where
   i.condlist = c and c.cond-rel = "variables".
```

This form is more complicated than the form using a dereference, and therefore more difficult for the database system to recognize as the retrieval of an individual tuple.

If the database system retrieves tuples only on demand, then the same performance problems arise dereferencing tuples that occur with the use of a sequence of simple queries. However, tuple references provide the information necessary for the database system to apply optimization and caching techniques to improve performance.

3.4.3 Multi-relation Tuple References

It is often advantageous to have an attribute that can refer to one of several relations. For example, a tuple in the *statements* relation contains a reference to a tuple in one of the individual statement relations, such as *asgstmts* or *ifthens*. Although storing references to different relations presents no problem to the database system, it is necessary to provide a means to determine the relation that contains a tuple designated by a tuple reference. This facility is provided by the *relation* operator. For example, to find all the **if** statements we would say

```
range of s is statements
retrieve (s.all) where
   relation(s.value) = "ifthens"
```

In the case where a tuple reference should refer to a subset of the relations in the database, an integrity constraint can be used to restrict the possible relations. For the *statements* example, this constraint could be expressed as

```
range of s is statements
define integrity on s is
   relation(s.value) = "asgstmts" or
   relation(s.value) = "ifthens" or
   relation(s.value) = "whilestmts" or
   relation(s.value) = "forstmts"
```

3.4.4 Transitive Closure Queries

Some properties of programs are transitive. For example, if a variable is used in an expression on the right-hand side of an assignment statement, then it is also used in the assignment statement. We define the relation

uses(user = **ref**, thing = **ref**)

and add the tuple (e, v) to it where e refers to an expression that contains a reference to a variable v. When an assignment statement is created with e as the right-hand side, we add the tuple (s, e), where s refers to the statement. In general, the *uses* relation contains tuples of the form (a, b) where a refers to an object that logically contains the object referred to by b.

To determine if the variable x is referenced in some statement y, it is necessary to ask if there exist tuples in *uses* (y, a_1), (a_1, a_2), ..., and (a_N, x) for some sequence of $a_1, ..., a_N$, $N \geq 0$. This question is simply a matter of determining if (y, x) is in the transitive closure of the relation *uses*.

We define "closure(R)" to be the relation that represents the transitive closure of a binary relation R. Given the *uses* relation and the closure operator, the statements that use the variable named "a" can then be found by saying

```
range of s is statements
range of v is variables
range of u is uses
range of uclosed is closure(uses)
retrieve (s.all) where
   u.user = s and u.thing = uclosed.user and
   relation(u.thing) ≠ "statements" and
   uclosed.thing = v and v.name = "a"
```

The first line of the predicate for this query specifies that a qualifying statement s must *directly* contain some object that is not a statement (i.e., an expression or variable), and that this object *recursively* contains a reference to a variable named "a". Requiring the statement to directly contain an expression or variable is necessary to avoid reporting enclosing statements, something implicitly desired in this kind of request.

3.5 Execution Information

Tuple references and a transitive closure operator offer substantial assistance in managing static program data. We now turn to the problem of managing the data that results from the execution of a program. Our semantic model, program threads, includes runtime objects such as the values of variables, activation of procedures, and the rest of the state of the executing program. This approach integrates debugging facilities naturally into the programming environment, since the same user and database interfaces used for program construction can be used during debugging.

Although we want to provide the appearance of uniformity between the source program and runtime data, we also wish to execute compiled code. We therefore use an interface between the database system and the executing program, called the *program monitor*, to provide the illusion that runtime data and program state are in the database. Figure 3.3 shows how the program monitor fits into the system.

The program monitor in effect provides relations such as

valueof(variableId, value),

that the user can access in the same way as relations for static program information. To the database system, the program monitor appears as a collection of relations that are physically separate from the rest of the database. In a distributed database system it is possible to perform a join operation across relations on different machines. Similarly, the value of a variable can be obtained by retrieving the *value* attribute from the tuple in the *valueof* relation with the desired variable's id.

Runtime data structures that are more complex than single-valued variables (e.g., tables, linked-lists, trees) may have more complex operations defined on them. For example, there might be an operation on a tree to find the node that has the maximum value for some field. One way to perform such an operation is to define a relation for the nodes of the tree, put a tuple in the relation for each node, and perform the corresponding query. In addition, if some update is made to this relation, it could be possible to reconstruct the runtime data structure from the relation. These kinds of manipulations require the use of program-dependent data formatting routines. It is natural for OMEGA to support such routines, since the database can store the formatting routines with the definition and implementation of the data structure.

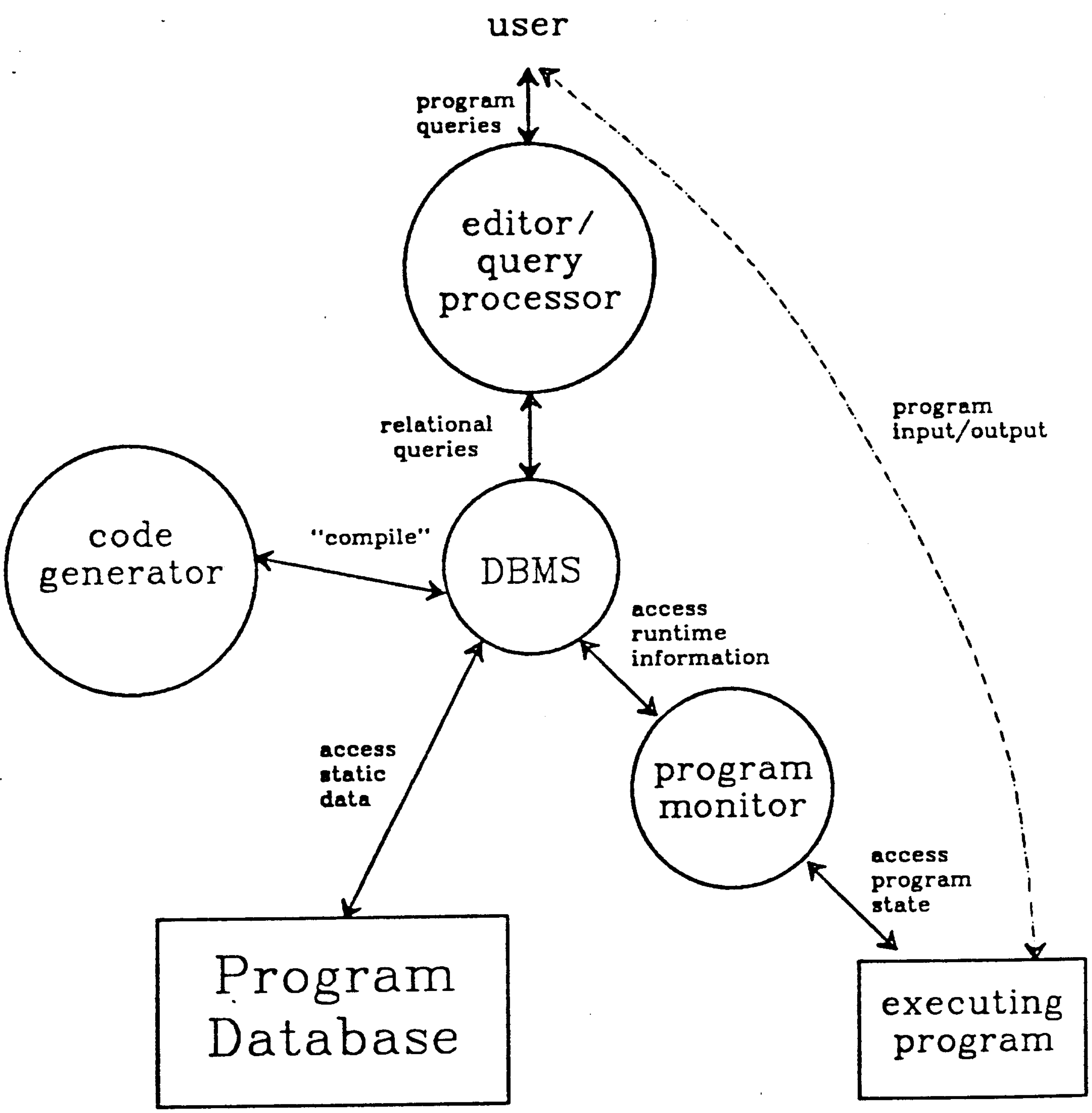


Fig. 3.3: Role of program monitor in OMEGA.

3.6 Monitoring Execution

When debugging a program, a user often wishes to monitor execution by having actions performed when specified conditions are satisfied. The most common actions are either to suspend execution or to print out the program location and values of certain variables. The concept of a *breakpoint*, as provided in most systems, is a condition that is satisfied when a particular instruction is reached and can be implemented directly on most machines.

We call a condition that interests the user an *event*, and allow the user to specify a set of actions that are to be performed whenever the event occurs. Unlike most debugging systems (e.g., [Johnson 77], [Katseff 79]), in which events are specified in a special debugging language, we allow events to be expressed as arbitrary relational qualifications. This form of expression naturally integrates debugging requests into the programming environment, while allowing the specification of a broader range of requests than most systems.

Some database systems provide a form of events and associated actions, called *triggers* [Eswaran 76]. A trigger specifies a set of commands to be performed when a particular database command is executed, subject to an additional boolean expression being satisfied. More generally, a trigger can be thought of as a predicate involving the information in the database and a set of database commands to execute when the value of the predicates changes from false to true. We could specify this construct using the following syntax:

```
when event
do
    actions
end
```

The *event* clause has the same form as the **where** clause of a QUEL retrieve statement, the *actions* are database commands.

Triggers are difficult to implement efficiently for general events. The detection of debugging events can also be complicated; for instance, consider the event that occurs when two variables defined in different procedures have the same value. The program monitor can help make detection of events easier.

In many existing debuggers, the only recognized events are the execution of particular statements. Thus, although one can imagine debugging events that are expensive to detect, many useful events can be trapped with simple breakpoints. To allow efficient implementation of more complex events, the database system must allow the program monitor to translate events into breakpoints and report when they occur. Depending on the hardware and operating system, the program monitor may use a variety of techniques for determining when changes in the program state (e.g., the value of a variable, the activation of a procedure, etc.) should trigger events.

The following examples show the power and generality of the OMEGA approach of expressing events as relational qualifications. First, suppose we wish to trap a call to procedure "buggy". To express events that refer to locations in the program, we assume the program monitor implements a relation

```
callstack(procedure, level)
```

that at any time contains a tuple for each procedure that is active. The *level* attribute is the runtime depth of procedure calls; it is highest for the most recently called procedure.

To notify the user when the procedure "buggy" is called we might say

```
range of p is callstack
when p.procedure.name = "buggy" and p.level = max(p.level)
do
   print p
end
```

Printing a tuple of the *callstack* relation might consist of displaying the name of the procedure, the values of its parameters, and the place from which it was called.

Suppose we are interested in seeing this information when "buggy" is called from "cause". We would place additional qualifications on the query as follows:

```
range of p is callstack
range of q is callstack
when
   p.procedure.name = "buggy" and p.level = max(p.level) and
   q.procedure.name = "cause" and q.level = p.level - 1
do
   print p
end
```

The last qualification indicates that "cause" called "buggy" directly. If we omitted the fourth constraint, the event would be triggered whenever "cause" is active and "buggy" is called. Other qualifications may test for particular parameter values, or other aspects of the program state.

Finally, suppose we wish to suspend execution of the program when "buggy" is called with parameter *n* equal to 0. To stop execution, we "update" a relation called *program-state*, which contains a single tuple of attributes of the executing program. For the purposes of this example, we are interested in the *status* attribute, which may have a value of RUNNING or SUSPENDED.

```
range of s is program-state
range of p is callstack
range of q is parameters
when
   p.procedure.name = "buggy" and p.level = max(p.level) and
   q.procedure.name = "buggy" and q.name = "n" and q.value = 0
do
   replace s (status = SUSPENDED)
end
```

Using triggers in conjunction with access to runtime information provides an extremely powerful mechanism for viewing the execution of a program. Conventional debuggers provide a limited set of events and conditions that may be brought to the attention of the user. Often this means the programmer has the poor choice of too little data or too much. Debugging in OMEGA offers a general way for the user to select those events and that information that is most useful. Moreover, output provided by the debugger can be immediately entered into the database. These facilities provide a powerful mechanism for obtaining and examining execution traces.

The program monitor provides access to data not stored directly in the database. In addition, the database system must allow the program monitor to indicate that a particular trigger condition is true. This ability is a natural extension of the definition of triggers to a distributed environment.

3.7 Displaying Information

Displaying program information on a terminal is a matter of translating the internal program representation into a human-readable, perhaps pictorial, form. Ideally, we would like to be able to define this pictorial representation as just another logical view of the database. However, there are problems with both the semantics and implementation of this approach.

Unlike traditional views, some portion of a pictorial view will be displayed on a terminal for a period of time during which the programmer wants to be the only one who can change the underlying information. The particular information to be displayed is determined by three independent factors:

- the thread of program information that is desired,
- the pictorial description of that information, and
- the amount of screen space available

What we want is to be able to define the information that is displayed on the terminal as "under surveillance" and locked. Since the pictorial view may be larger than the entire screen, we also need to be able to perform browsing operations such as scrolling.

The *portal* mechanism proposed in [Stonebraker and Rowe 82] provides the semantics we need. A portal defines some portion of a pictorial view that is to be displayed, and locks the underlying information in the database. Scrolling is provided by built-in operations to move a portal within a pictorial view.

The traditional implementation of views is to compute the tuples in the view each time the view is accessed. This computation is expensive for pictorial views of programs since they are accessed frequently but changed only occasionally. For example, scrolling first forward and then backward returns the display to its original state.

The caching of views could save substantial computation, particularly if the information were kept in main memory rather than on disk. This facility should be provided by the database system. Otherwise each application (in our case, OMEGA) undoubtedly will have to do its own caching and therefore have to worry about consistency issues that the database system is better equipped to handle.

3.8 Name Resolution

Whenever a user enters a name the system must try to determine to what object the user is referring. In a compiler, a *symbol table* provides the means for finding an object associated with a particular name in a given context. Context-dependent name resolution is an important aspect of a good program development system. People tend to build a "mental working set" of objects and make frequent references to them, using names that would be ambiguous if the context were ignored.

OMEGA provides the function of a symbol table by using the database system to manage context information and expressing the resolution of a name in terms of a single query. Using the approach of [Rowe 82], the relevant information is kept in three relations defined as follows:

symbols (object = **ref**, name = **string**, context = **ref** contexts)

visible (from = **ref** contexts, to = **ref** contexts)

contexts (priority = integer)

The first relation, *symbols*, associates an object with a particular name in a given context. The *visible* relation defines a structure between contexts so that names in the context referred to by the *from* attribute can be resolved in the context referred to by the *to* context. Since this property is reflexive, the *visible* relation always contains tuples of the form (*context-reference*, *context-reference*). The *contexts* relation associates an integer with each context that determines the precedence of contexts structured by the *visible* relation.

To understand these relations and the way they can be used, we consider the example of name resolution in a block-structured language. The basic unit of naming in such languages is called a *block*, within which names must be unique. The *scope* of a block is a collection of blocks that are searched in some order when resolving a name.

Suppose we have the following declarations in a Pascal program:

```
procedure A;
   var C : integer;
   procedure B;
      var C : integer;
   end;
end;
```

There is a block associated with each of the procedures A and B. The scope of A consists of only A; the scope of B is the set {B, A}.

For this example, the *symbols* relation defined above contains four tuples, one for each of the procedures and variables in the program. The scope rules of Pascal require the *visible* relation to contain three tuples: one to indicate that names in procedure B are visible in procedure B, one to indicate that names in procedure A are visible in procedure A, and one to indicate that names in procedure A are visible in procedure B.

The *contexts* relation associates the number 2 for the context associated with procedure B, and the number 1 for procedure A. This way, names defined in procedure B take precedence in that context over names defined in procedure A. In particular, in procedure B the name "C" refers to the C defined in B, not the C defined in A.

Given this information, we can find the symbol with name *x* in block *y* with a single query. This query is rather complicated to express in QUEL; to simplify things we separate it into a view definition and subsequent query on the view.

To define a view of all the symbols named *x* that are visible from *y*, we say

```
range of s is symbols
range of v is visible
range of c is contexts
define view x-symbols
   (object = s.object, name = s.name, context = s.context)
where
   s.name = "x" and s.context = v.to and v.from = y
```

Now to retrieve the desired symbol, we select the *x*-symbol associated with the highest priority context by saying

```
range of s is x-symbols
retrieve (s.all) where
    s.context = c and
    c.priority = max(c.priority where s.context = c)
```

3.9 Version and Configuration Management

Although software changes over time, it is not always the most up-to-date copy that is of interest. Organizations often must support older releases while developing new ones. A *version* is a snapshot of a program or part of a program at a particular moment of time. Because of, and despite, greater portability of software, it is often necessary to support different but largely identical pieces of software for different hardware or application environments.

A *configuration* is a specialization of a program or part of a program to meet a particular set of constraints. The difference between versions and configurations is that versions are ordered in time, with newer ones presumed to supercede older ones, whereas all configurations are equally important, and may coexist forever.

At the core of both version and configuration management are two requirements that differ from traditional database applications. The first is that there must be several valid and consistent instances of data in the database. The second is that it must be possible for multiple users to access and update these instances of data concurrently. This form of access is not necessarily the normal database sense of concurrent access; it is sometimes convenient to allow new instances of data to be created that will subsequently be coalesced into a single instance.

When a new version of a program is created, it would be inefficient to duplicate the database. Doing so would also make it more difficult to establish the relationship between the old and new versions. Software version control systems such as SCCS [Rochkind 75] use a differential file to compactly store program versions. The original version of the file is kept as are all updates necessary to transform the file to the latest (and all intermediate) versions. Hypothetical relations [Stonebraker and Keller 80] can be implemented using this technique and can be used to support multiple versions of information stored in a database.

One of the problems with systems like SCCS is that they require the user to explicitly state when new versions are created. Hypothetical relations do not solve this problem since there is no way to have old versions automatically removed. To save space and speed up queries involving past versions, the user must explicitly dispose of old versions. Coalescing of versions is also a manual process; the exact semantics of a change to an old version is a complex issue currently being studied.

Configuration management involves automatically building a program out of its various pieces according to a given set of parameters. For example, a common parameter is the target machine or system on which the program is going to be run. To minimize the time it takes to build an executable program, only the pieces that have changed or depend on pieces that have changed should be recompiled.

Tools such as *make* [Feldman 78] provide this service, but require the user to specify the program interdependencies. *Make* uses an auxiliary file that contains dependency information; this file must be continually updated by the user as the program changes.

Since *make* uses a text file as its basic unit of software and files usually contain several procedures, it also often recompiles more code than is necessary. By using a database, dependency information is not duplicated and the OMEGA build process can be done without any user assistance. Moreover, the information is directly retrievable at whatever granularity is desired. For example, to find all the procedures that depend on a procedure named "changed" we could say

```
range of p is procedures
range of s is statements
range of uclosed is closure(uses)
retrieve (p.all) where
   uclosed.user = p and uclosed.thing = s and
   relation(s.value) = "callstmt" and
   s.value.proc.name = "changed"
```

Configuration management also requires the ability to determine which program information belongs to which configurations. A common way to implement this feature in conventional programming systems is with conditional compilation facilities. Simple control statements are introduced to indicate which statements ought to be compiled for different configurations.

The database provides more complete control over which program elements relate to which configurations, since each object potentially could be associated with a set of configurations. A relational qualification can then be used to specify a particular configuration. For example, suppose we have the following relations:

```
configurations (name = string, created = time)

configof (object = ref, config = ref configurations)
```

We can then retrieve all the constants associated with the configuration called "VAX" by saying

```
range of c is constants
range of cf is configurations
range of cfof is configof
retrieve (c.all) where
   cf.name = "VAX" and cfof.config = cf and cfof.object = c
```

The most important idea that databases bring to version and configuration control is that ***a version or configuration is a view of the program.*** To get the most out of this notion, it is necessary that the difficult problems of view updates and consistency be solved. View updates are very difficult in general; however, database researchers are working on determining constraints under which updates to views can be processed while maintaining the consistency of the database [Dayal and Bernstein 82].

3.10 Conclusions

Storing program information in a general purpose database system provides a powerful mechanism for manipulating existing software. In designing OMEGA, we have chosen to take advantage of this power by using a relational database system to manage all program information.

To represent and manipulate program information, we have suggested the addition of a domain of tuple references and a transitive closure operator to the relational model. Both these ideas are similar to other proposed extensions, we have focused on them because they are critical for simple and efficient manipulation of programs.

These extensions do not represent a radical change in the relational model and are sufficiently general to be of use to a wide variety of applications. For example, computer-aided design (CAD) systems for integrated circuits must manage both hierarchical and relational data and could use a construct like tuple references.

Data management is a fundamental problem of computing. For general purpose database systems to be useful through a wide variety of applications, they must provide primitives for data modeling and access. In analyzing the database needs of a software management system, we have tried to identify those features that will provide the most leverage for manipulating complex data structures.

Database systems also provide user interfaces for defining and accessing information. As we can see from the examples given in this chapter, database languages can be as complicated and difficult to understand as programming languages, if not more so. Although we have shown how to manage program information, the software beast we described in Chapter 1 will continue to roam out of control unless this power can be harnessed. In the next chapter we describe the basic principles of the OMEGA user interface and show how they allow easy and simple access to the database without sacrificing power.

Chapter 4

The User Interface

I'm looking through you, where did you go?
I thought I knew you, what did I know?
You don't look different but you have changed.
I'm looking through you, you're not the same!

– from the song *I'm looking through you* by the Beatles

Seeing is Forgetting the Name of the Thing One Sees

– title of a book by Robert Irwin

4.1 Introduction

To simplify the construction and manipulation of software, programmers abstract recurring concepts into reusable parts. Current programming languages provide built-in parts, (e.g., statements, variables, data types, modules) and mechanisms for creating new constructs (e.g., by writing a procedure, declaring a variable, defining an abstract data type, or instantiating a module). These mechanisms allow programs to be modified easily, since a change to the definition of a part affects all its uses.

Due to the independent evolution of program structures and their different requirements for parsing in conventional programming systems, each has its own way (syntax and visual representation) for programmers to specify abstractions in terms of simpler elements. For example, in some languages, a program may define a new kind of integer that can be used just as easily (with overloaded operators), efficiently (with inline expansion of procedures), and cleanly (with implementation details hidden) as the native integer type. However, in most languages, it is not possible to define a new kind of **for** loop.

In Chapter 3 we showed how to represent static and dynamic program information in a relational database, thus providing powerful operations for viewing and manipulating software. In this chapter we describe the way the user interacts with OMEGA to create, view, and modify abstractions.

Using the concept of "what you see is what you get" that has been applied in many applications, we let users define visual representations of their programs' objects and structure. Thus, they can directly manipulate objects and immediately observe the results of those manipulations.

We call this approach ***visual abstraction***, since it provides a uniform abstraction mechanism based on pictorial and logical views of programs. This approach is in contrast to conventional software development, where a description of the desired computation is written in a language and then subsequently compiled.

The remainder of this chapter presents the details of the design of the OMEGA user interface. We first present our goals for the interaction between programmers and OMEGA, then describe the mechanics of this interaction, and finally discuss the detection and correction of semantic errors.

4.2 Goals

We want OMEGA to help programmers produce correct software, not just prevent them from producing incorrect software. Our approach is to have OMEGA use a visual medium and conversational protocol to provide a user interface that has the following characteristics:

- no input syntax
- multiple output formats
- interactive semantic analysis
- multi-threaded program organization

No input syntax means that the user is not required to cast the program in one particular form, as for a compiler. As in a menu-based system such as Smalltalk, OMEGA should provide suggestions and ask questions during program construction rather than forcing the user to remember and type long strings of symbols that must obey some rigid structure.

Support for multiple output formats means that the user may have program structures displayed in a variety of ways, depending on the aspect of the program that is of interest at the moment. Programming systems typically use a language as both the input specification and the displayed form of the program. As a result, compromises must be made between what can be parsed and what information should be displayed. In OMEGA, we want to use graphical output and icons to convey the most information in an easily assimilated way. At the minimum, output formats must support the multiple ways of building programs, so that the user can work without mentally switching between points of view.

Interactive semantic analysis means that a program is examined as it is being built. Just as oral communication is more effective than written communication because the speaker can adjust to the response of the listener, the system should provide feedback to the programmer as the program is built. Errors due to inconsistency or ambiguity should be resolved immediately. In addition, by displaying the structure of the program as it is being built, it may help the programmer see higher-level problems that the programming system cannot detect.

Multi-threaded program organization means that the programmer can manipulate the various threads accessible from the database within a single interface. Conventional programming systems provide only one view of a program. The programmer, however, may wish to see the program in different ways when it is being built, modified, or debugged. For example, a group of statements might be edited as a unit because they appear in the same procedure, because they all reference the same variable, or because they will be consecutively executed.

4.3 Mechanics of Interaction

The key to lifting the burden of syntax from a programming environment is to stop using text as the medium of program construction. We have already argued that text is a poor representation of program information for data manipulation reasons; we argue here that it is also inadequate as the sole interface between programmer and system.

Text hampers human understanding because it is not unique visually; "free format" languages allow tokens to be placed in many different positions. Text is also not a good representation for editing. Logically one wishes to operate on program structures (e.g., statements, variables, types, etc.); using a text editor one must manipulate some combination of lines, words and characters.

OMEGA resolves the different needs for program representation by allowing the program to be entered, displayed, edited, and analyzed in different formats. This flexibility is provided by

- separating the pictorial representation of an object from the object itself,
- pointing rather than typing to identify objects, and
- using multiple windows to allow pieces of programs to be constructed and viewed independently.

We now examine each of these ideas individually.

4.3.1 One Picture is Worth a Thousand Keywords

Most programming environments do not distinguish between an object and the pictorial representation of that object. In OMEGA, program structures are displayed consistently as *pictographs*. A pictograph is a view of an object displayed on the screen. Pictographs may be arbitrarily assigned to objects; different pictographs for the same object may be selected when different aspects of the object are to be emphasized.

A pictograph consists of letters or graphical images arranged in a two-dimensional area. Ideally, the display device would provide high-resolution and allow color, intensity, and non-character graphics to be used. The principles of pictographs also apply to lower resolution, black and white, or character-only displays, but existing 24 by 80 character CRTs probably hold too little information for the ideas presented here to be used on any significant scale.

A pictograph is the visual object that a programmer sees and manipulates. Shapes and spatial relationships help convey structural information. An important feature of a pictograph is that parts of it can be used to represent slots into which parameters are placed.

Figure 4.1 shows an example pictograph for a table search. The table search is a two-exit control structure since the desired element may or may not be in the table. The slots in the pictograph show places where parameters may be inserted for the table to be searched (*Table*), the key for the desired entry (*Key*), and the variable to point to the object desired (*Element*). Note that *Element* has a default value; use of the pictograph defines an object if no other one is substituted.

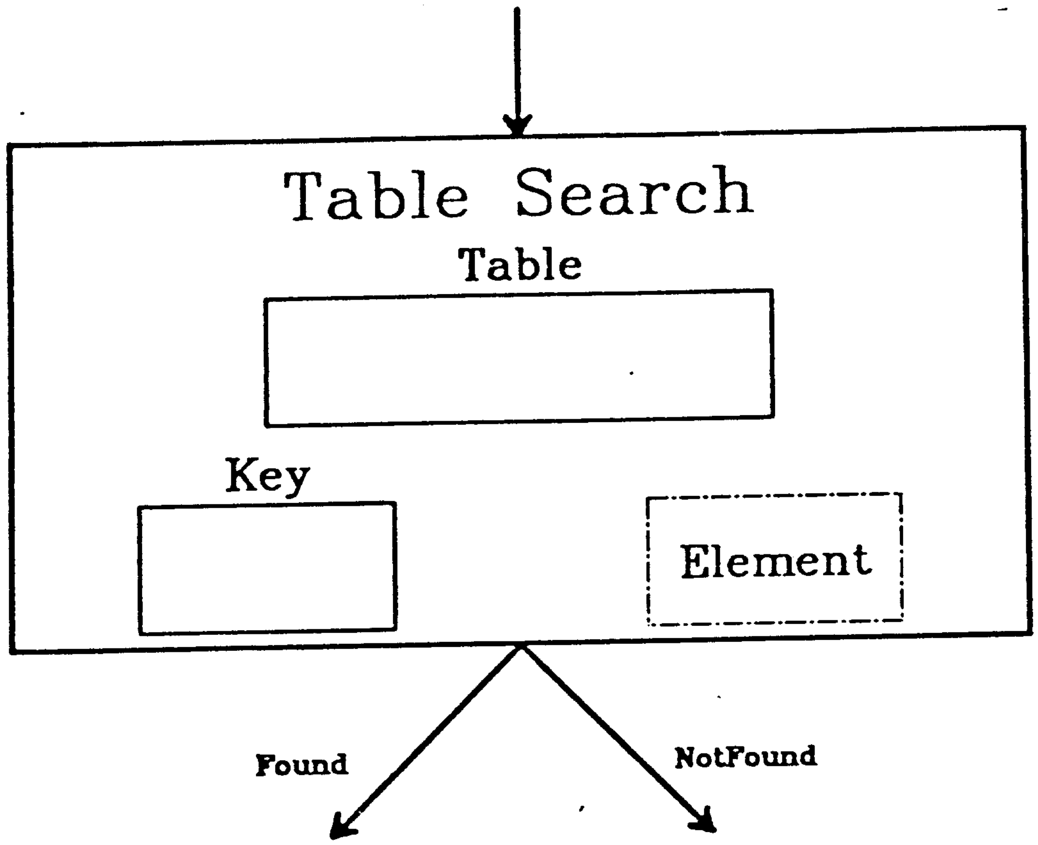


Fig. 4.1: Pictograph for table search abstraction.

Figure 4.2 shows another pictograph for the same control structure. This pictograph shows more details of the implementation and is in the traditional text form. This lower-level view of the control structure reveals aspects that are hidden by the higher-level view. In Figure 4.1, the parameters to the pictograph are represented by boxes; in Figure 4.2, by italicized words.

```
label NotFound, Found
var element : subscript of Table
if empty(Table) then goto NotFound
element := first(Table)
loop
   if element.key = Key then goto Found
   if element = last(Table) then goto NotFound
   element := next(Table,element)
endloop
```

Fig. 4.2: Implementation view of a table search.

An important collection of pictographs are those representing objects in the program. These pictographs may appear in the program structure, but also appear in a *glossary*. A glossary is simply a list of pictographs and their meanings. Figure 4.3 shows a glossary that might exist in a program using the table search of Figure 4.2.

Employees	array of EmployeeRecord, table of all employees
InputName	EmployeeName, name of employee just read
CurrentEmployee	EmployeeIndex, points to the record of the current employee

Figure 4.3. A sample glossary

4.3.2 It *Is* Polite To Point

Our alternative to entering text is to display relevant pictographs on the screen and have the user point at, pick up, and put down the corresponding objects using a pointing device (e.g., a mouse, light pen, finger, etc.). "Picking up" and "putting down" generally mean pointing at something and pressing a key or button.

The act of picking up an object and putting it down someplace may have a different effect depending on the objects and the parts of the pictograph selected. Picking up the *EmployeeRecord* pictograph in the glossary in Figure 4.3 and pressing the "what is this?" button would cause a description of the type *EmployeeRecord* to be displayed. Picking up the *Employees* pictograph and putting it down in the *Table* box of Figure 4.1 makes *Employees* the actual parameter of the *TableSearch* pictograph.

To use the *TableSearch* control structure in the program, we first pick up a copy of it by moving the mouse to the pictograph and pressing the pick-up button. We place it at the desired point in the statement list we are working on by moving the mouse just below the statement we wish it to follow and pressing the put-down button. This action causes the entry line of the pictograph to be connected to the previous statement. Figures 4.4(a) and 4.4(b) show how the list of statements appears before and after the insertion of the *TableSearch*.

The parameters of *TableSearch* are filled in by picking up the objects and putting them down in the boxes. The two possible exits are now sites for additional statements to be connected. Figure 4.4(c) shows the statements after the *TableSearch* has been filled.

4.3.3 What Is In A Name?

Identifiers in programs serve two functions: they provide a visual tag that the reader uses to associate together different instances of the same object, and they provide a mnemonic description of some properties of the object. In traditional systems, these two purposes run against each other. Shorter, more distinct identifiers are easier to resolve visually, yet longer identifiers that often may be similar are more descriptive. In OMEGA, these two functions are separated. Pictographs may be assigned to objects arbitrarily to improve the visual representation of the program; properties of the object are instantly accessible (and may be displayed on part of the screen as the glossary) from the database.

The ability to name by pointing adds significant power to the programming environment. For instance, it is not necessary for displayed pictographs to be unique. If it is necessary to disambiguate a name, the user simply points to the intended pictograph in the glossary (or somewhere else on the screen). Since the system always references objects and merely displays pictographs for the convenience of the user, the same pictograph may be used in different parts of the program without causing confusion about what object they refer to.

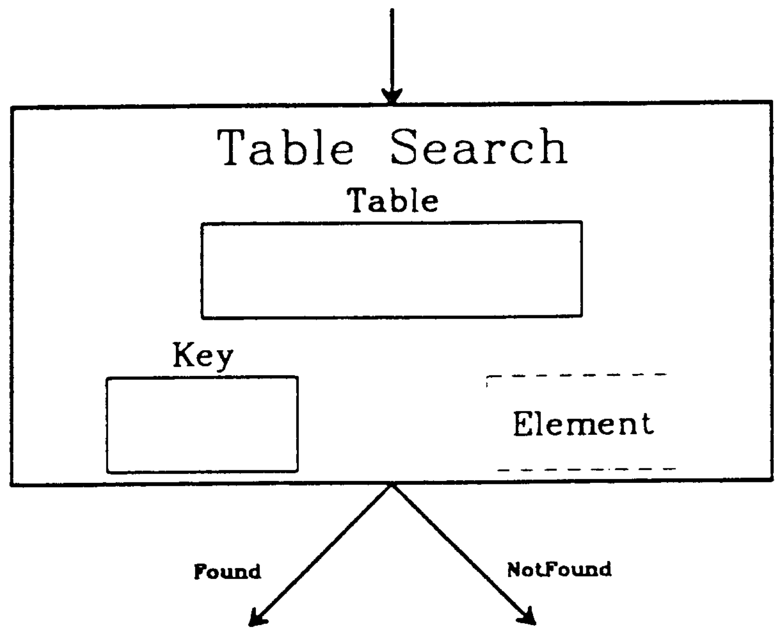
In conventional programming systems, the case often arises that the best name for an instance of a data structure is the name of the type of the data structure. This conflict must usually be resolved by adding a prefix or suffix to one or the other of the names. A similar problem occurs here; when pointing to a pictograph, it may be meaningful to pick up either the actual object or a new instance of the object. Such problems are easily avoided by allowing several pick-up keys. For example, after pointing to a variable, the user might choose to pick up the variable itself, the variable's type or value, or even a new variable of

...
Read identifier *id* from *input*
⟶

...

Fig. 4.4(a): Statements and point device before insertion.

...
Read identifier *id* from *input*



...

Fig. 4.4(b): After insertion.

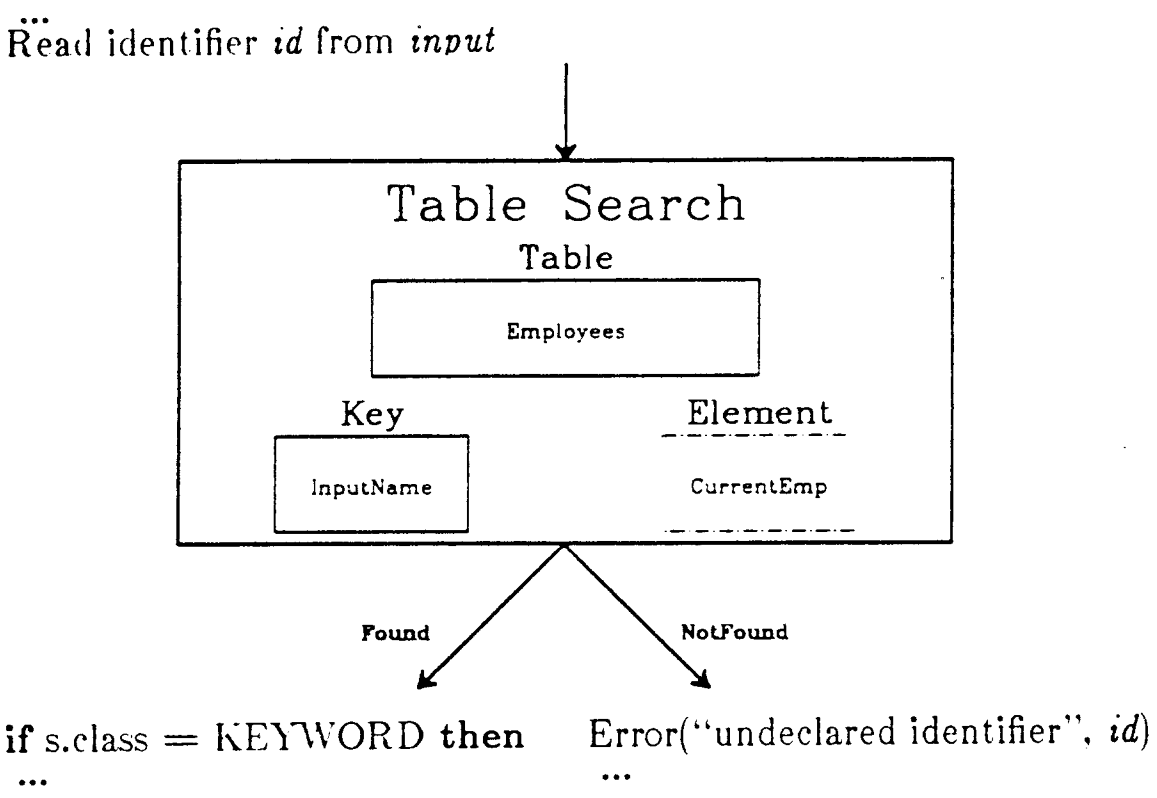

...
Read identifier *id* from *input*
Table Search
Table
Employees
Key
InputName
Element
CurrentEmp
Found
NotFound
if s.class = KEYWORD **then**
...
Error("undeclared identifier", *id*)
...

Fig. 4.4(c): After filling parameters.

the same type as that variable.

The use of multiple pick-up keys does not require a pointing device with many buttons, in fact the pointing device does not have to have any buttons at all. As the programmer's right hand controls the pointing device, the left hand stays on the keyboard and uses keys within reach as buttons. Unlike Smalltalk, in which menus pop-up in the middle of the active portion of the screen, or Cedar, in which the pointing device must be moved to a menu elsewhere on the screen, this approach allows commands that are used frequently (e.g., "zoom in", "add", "delete") to be entered with a single keystroke without repositioning the pointing device.

4.3.4 Rome Was Not Built In A Day

One of the advantages text-oriented interfaces have had in the past is the support of partially-formed programs. Since no examination of the program occurs until the user requests it, it is easy to leave loose ends to be fixed up later. Tree-oriented systems often have restrictions, for example, that nodes must be added top-down. Moreover, the transformations possible on text are limited only by the power of the text editor and the imagination of the user. Structure-oriented editors often make some transformations difficult; for example, it may not be possible to change one kind of a node to another without first deleting and then recreating the node's children.

There are some transformations that can be accomplished only with text-oriented systems. For example, moving delimiters to make what used to be a string or comment into program statements requires parsing. "Commenting out" code is a meaningful and straightforward transformation in OMEGA, however, and it is not necessary to resort to text tricks to accomplish it.

Programs are not represented linearly on the screen in OMEGA. It is possible to build several program fragments independently in different windows and connect them together by picking up and moving around pictographs. For instance, in the previous section, it would have been equally possible to assign the parameters to the *TableSearch* construct before inserting it into the program as a statement.

One freedom a pointing interface does not allow is that of referring to an object that is not yet defined. This restriction is not so bad since the parameters of an operation can be defined without defining its implementation. For example, one cannot create a call to procedure *f* before creating the procedure, but one can create *f* and refer to it before specifying its body. Eventually, the program will reach a state in which all necessary objects and attributes have been specified, and then be ready to run.

4.3.5 Screen Management

During an OMEGA session the screen is composed of a collection of windows that belong to one of the following classes:

- catalog
- glossary
- program
- response

A *catalog* window displays a subset of the available operations that are defined, including objects, control structures, operations, etc. A *program* window displays program fragments. These fragments can be either complete or partially assembled programs. A *glossary* window displays information about pictographs displayed in other windows on the screen. A *response* window displays output from some command or program, such as an error message. Figure 4.5 shows a sample screen.

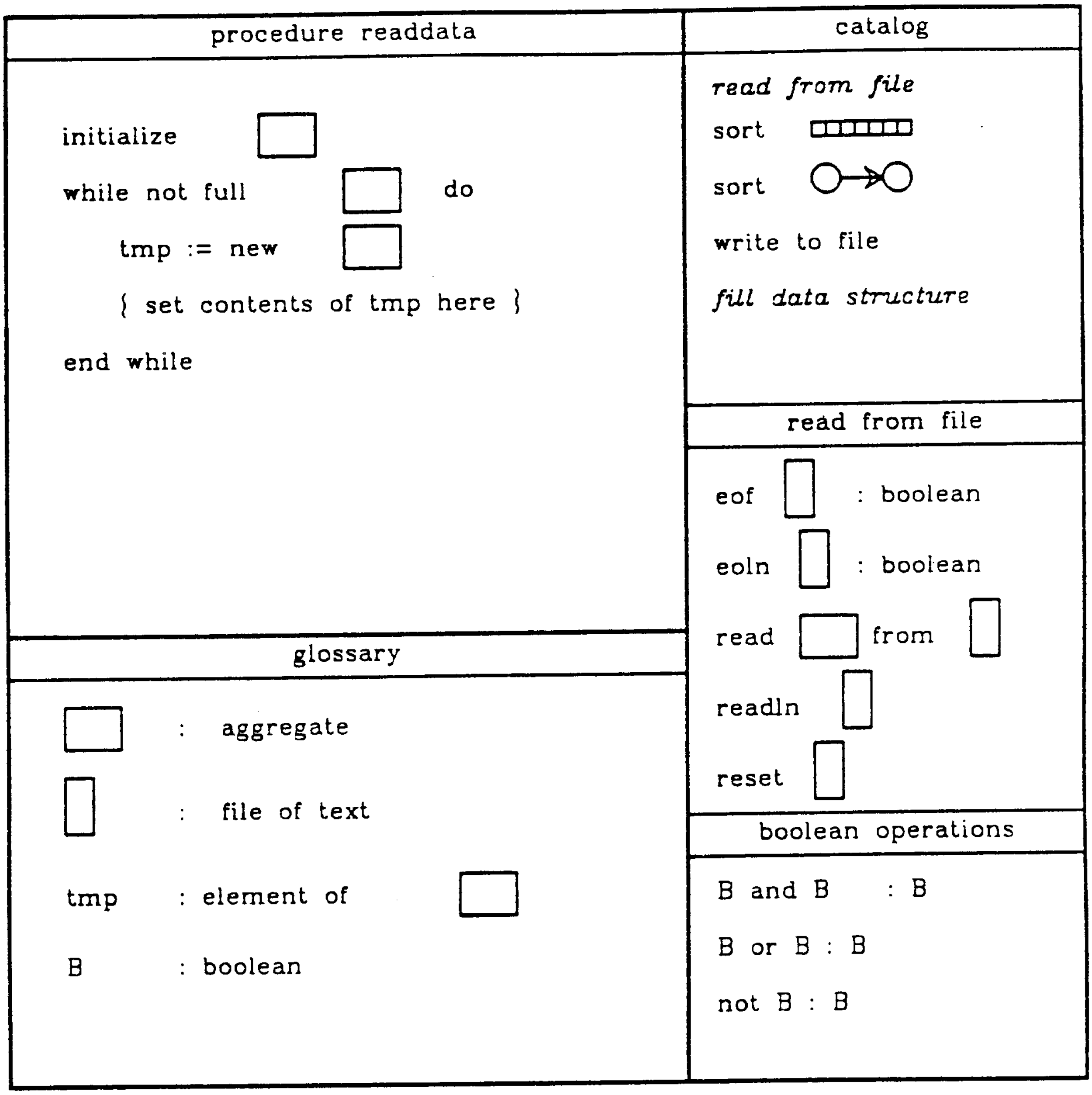
Catalog windows are the primary means of searching for information in the database. Standard queries allow users to locate previously defined operations, objects, and program fragments that they can use. Things in the catalog may be displayed in different ways. For example, the lower right window shows operations on booleans; the middle right window shows operations used to read from a file.

Glossary windows are created in conjunction with program windows. The glossary is the place where the two functions of traditional identifiers, tags and descriptions, are brought together. It displays the pictograph for objects and descriptions of what the objects are. Normally, the glossary associated with a program window contains entries for each object displayed in the program window. Of course, it is possible to have some of the well-known entries omitted.

The window in the upper left portion of the screen shows a program fragment under construction. As the programmer creates, fills, and moves around objects a number of windows need to be created, enlarged, and perhaps discarded. With this kind of interaction we cannot expect to use either the user-controlled window allocation style of Smalltalk or the traditional "virtual terminal" approach.

Both these strategies rely on a more static kind of window, one that is used at a particular size for a relatively large amount of time before being enlarged or discarded. For OMEGA, we prefer to have the size, placement, growth, and shrinkage of windows determined automatically. An implementation of this facility must take into account

- the importance of a window (a window may be important because it was recently touched or because it contains the time of day),



```
procedure readdata

initialize □
while not full □ do
    tmp := new □
    { set contents of tmp here }
end while
```

```
glossary

□     :  aggregate
▯     :  file of text
tmp   :  element of □
B     :  boolean
```

```
catalog

read from file
sort □□□□□□□
sort ○→○
write to file
fill data structure
```

```
read from file

eof ▯ : boolean
eoln ▯ : boolean
read □ from ▯
readln ▯
reset ▯
```

```
boolean operations

B and B : B
B or B : B
not B : B
```

Fig. 4.5: A sample OMEGA screen.

- the minimum size of a window (it is better to not display something than to display it too small to recognize), and
- the relationships between windows, so that related windows are placed together.

The last factor in the list above, inter-window relationships, is partially handled in Smalltalk through the *window pane* mechanism. This mechanism provides small windows that display different menus depending on the information displayed in a larger, primary window. This use of menus is particularly useful when traversing a hierarchy, since the menus show each level of objects. However, this facility cannot be applied in general in Smalltalk (one cannot define dependencies between windows), and the placement of related windows is done ad hoc.

4.4 Abstractions

Thus far we have relied on the reader's intuition for an understanding of what will happen when pictographs are put together. In this section, we describe more details of the abstractions that pictographs represent.

We use the term *abstraction* to refer to the general class of things that pictographs represent. An abstraction may be a program object such as a variable, type, control structure, or operation; it may be a program constructor such as a variable declarator, procedure template, or type former; or it may be a program manipulation command such as a query, configuration definition, or directive.

Abstractions are defined using other abstractions. OMEGA provides abstractions that are used to create program objects; which particular abstractions are available depends on the underlying programming semantics to be supported. An operation that places or instantiates an abstraction causes some semantic changes to the program database. For example, instantiating a variable abstraction causes entries to be made in the database to indicate that a new variable of the specified type has been created.

An abstraction has three parts: the pictograph that represents it, the parameters (and how they appear in the pictograph), and its semantics in terms of operations on the database. The pictograph determines what the user will see, and what the visual interaction is. The parameters specify what kinds of objects can be connected to the abstraction and how that is done using the pictograph.

Creating a new object that is an instance of an abstraction causes information to be added to the database. The operations performed are similar to those done during syntactic and semantic analysis of conventional programming languages.

4.4.1 Defining and Using Abstractions

Consider the following simple abstraction for creating variables.

Abstraction:	declare a variable
Pictograph:	var *name* : *type*
Parameters:	*name* is a pictograph
	type is a type object
Actions:	Create a new variable object
	Set the variable's pictograph to *name*
	Set the variable's type to *type*

The pictograph in the example is similar to declarations in conventional languages. Note that simply by changing the pictograph in the *declare a variable* abstraction to be "*type name;*", declarations could be displayed in a C-like format instead of a Pascal-like one.

Suppose we wish to define the exponentiation operator. The following abstraction would be used:

Abstraction:	declare a function
Pictograph:	function *name* (*parameters*) : *type*
	body
Parameters:	*name* is a pictograph
	parameters is a parameter list object
	type is a type object
	body is a statement list object
Database:	Create a function object
	Set its parameter list to *parameters*
	Set its return type to *type*
	Set its body to *body*
	Define its database semantics to insert a call to the function body

As one might expect, there are also abstractions for statements, parameter lists, and other program structures. If we wish to define the exponentiation abstraction, we would perform the following steps:

- Create a new function by pointing at the "declare a function" pictograph and pushing the "new" button. The pictograph for the definition of the new function will be displayed in a newly allocated window.

- Construct its parameter specifications using the "build a parameter list" abstraction. It would presumably contain a real parameter called *base* and an integer parameter called *exponent*.

- Connect the parameter list to the ***parameters*** part of the function definition.
- Pick up a reference to the data type "real" pictograph and place it on *type*.
- Construct the function body in the ***body*** slot by creating and connecting the necessary declarations and statements.
- Build a pictograph for exponentiation referencing the *base* and *exponent* pictographs and place it in the *name* slot.

Once the exponentiation function has been defined, we may install it in the catalog. This operation would be done using the "create catalog entry" abstraction, which might have parameters such as the pictograph for the function and a list of attributes on which to index the function. A subsequent reference to the function creates an instance of the function abstraction, which will cause the specified database operations to be performed when all of the parameters have been bound.

4.5 Semantic Error Detection

As the user manipulates abstractions, updates are made to the database. These updates do not necessarily change the resulting program immediately. Any change, such as defining a variable or creating a new statement, modifies the database. The program is altered only when the statement or variable is connected to the program. Moreover, the program will be changed only when a complete and consistent modification has been made.

Once the abstraction has been completed (i.e., all parameters are specified), the updates specified by the abstraction are attempted. This updating takes place as a transaction on the database system. Erroneous transactions do not complete and improper objects do not appear as part of the program. For example, a statement may refer to variable objects whose type has not yet been specified. The insertion of such a statement would not take effect until the type is defined. When the type gets defined, all references to the variable are checked to be sure they are consistent with the type. If they are, the statements are added to the program; otherwise, the statements, though in the database, do not yet affect the program.

Each time an object is connected to a parameter, a check is made to see if the object meets the parameter's specifications. If it does not, the object is not connected and an error message is generated. For example, connecting a variable object to the *type* parameter in "define a function" would result in an error. This approach is sort of a "square peg into a round hole" approach: the user cannot bind an object to a parameter if doing so would result in a type violation.

An application-level database transaction mechanism is used to manage partial updates to the program. Since the completion of one update may trigger the initiation of others, it is essential that multiple transactions be allowed at once. These transactions are built on top of the standard lower-level transaction mechanism provided by the database system, ensuring the reliable and consistent storage of the state of the programming environment even if that state describes a partial or incorrect program.

The semantic analysis necessary to determine if a parameter "fits" is equivalent to that done in a compiler after names have been resolved to objects. Although the user may give an object a name by putting an identifier in its pictograph, references to a pictograph lead directly to the associated abstraction. This interface eliminates the problem of resolving overloading for procedures since the user points at the actual procedure, not the name of a procedure.

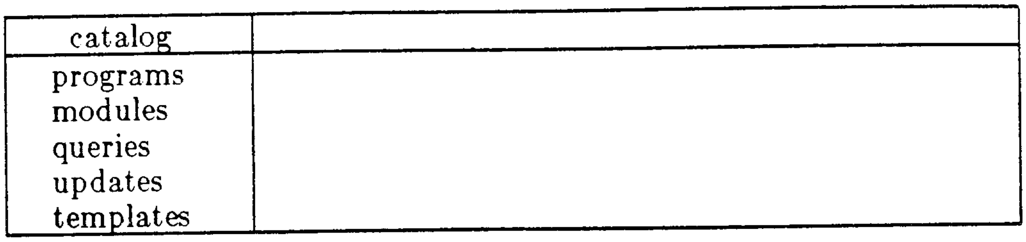
Because semantic error detection is done as the program is constructed, violations of constraints such as type incompatibilities are reported immediately, not after some period of time during which the user has forgotten the context in which the error occurred. Many sorts of errors (missing parameters, undefined variables) simply cannot occur due to the sequence of operations necessary to create the program.

Global changes that affect many parts of the program can be performed reliably because OMEGA can detect incomplete changes. If it is necessary to add a parameter to an operation, the system can find and request modification of each instance. Of course, it is not required that all instances be fixed immediately. Such temporary inconsistencies or incomplete objects form a task list of work to be performed by the user.

4.6 A Bigger Example

We have used isolated examples to illustrate the individual features of the OMEGA user interface. Now we show how OMEGA could be used in a longer, practical context. We consider the problem of constructing a module for managing a queue of processes waiting to run on a processor.

Figure 4.6(a) shows the screen upon entering OMEGA. The only information that is present is the catalog of the relations in the database. The first thing to do is to find existing modules that manage queues. To do this, we enter and execute a query to find all modules whose associated glossary information contain the string "queue". To enter this query, we first create a new query by moving the cursor to the "queries" entry in the catalog and pressing the "create a new one of these" key.

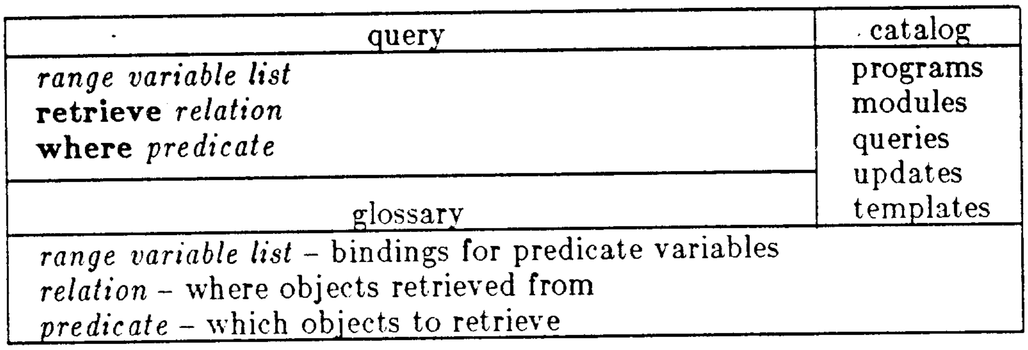


catalog

programs
modules
queries
updates
templates

Fig. 4.6(a): Initial screen.

The resulting screen is shown in Figure 4.6(b). A prototype query is displayed in a new window; it looks much like a query expressed in QUEL. The glossary window gives a short definition of each of the pieces of a query.



query

range variable list
retrieve *relation*
where *predicate*

catalog

programs
modules
queries
updates
templates

glossary

range variable list – bindings for predicate variables
relation – where objects retrieved from
predicate – which objects to retrieve

Fig. 4.6(b): After creating a new query.

To build the query we desire, we pick up the "modules" relation from the catalog window and place it on the "relation" slot in the window for the query, which specifies that we want to see qualifying tuples that are in the "modules" relation. To create the range variable list, we move to the slot and press the "create a new one of these" key. This action expands the slot into a prototype range variable list, which is a range statement followed by a range variable list.

The range statement prototype has slots for the name of the range variable and for the associated relation. To fill in the name slot, we press the "here comes a string" key and type the string "m". The relation slot is filled in by picking up the "modules" relation from the catalog window, moving to the slot, and pressing the "put down" key.

The predicate is filled in the same manner. The only difference is that when the "create a new one of these" key is pressed with the cursor on the "predicate" slot, a window is created that shows the different possibilities from which to chose from in constructing a predicate. The state of the screen after constructing the query in shown in Figure 4.6(c).

query	catalog
range of m **is** modules **retrieve** modules **where** m.glossaryInfo = "*queue*"	programs modules queries updates templates
glossary	
m – range variable modules – relation glossaryInfo – field of modules relation	

Fig. 4.6(c): After constructing query.

With the cursor on the query, we press the "execute" key and the results of the query are shown in a new window. The state of the screen after executing the query is shown in Figure 4.6(d). Two modules for managing queues already exist, one a queue of processes waiting for disk I/O and one a queue of jobs spooled to a printer. In addition to the type of queue they manage, these modules also differ in the semantics of their operations. For example, the printer queue manager may process jobs in the order they are added to the queue whereas the disk queue manager may take into account the current position of the disk head in choosing the next request to process.

query	catalog
range of m **is** modules **retrieve** modules **where** m.glossaryInfo = "*queue*"	programs modules queries updates templates
result of query	

module DiskQueue;
module PrinterQueue;	

glossary	
DiskQueue – manager of queue of processes waiting for I/O PrinterQueue – print queue manager m – range variable modules – relation glossaryInfo – field of modules relation	

Fig. 4.6(d): After query is executed.

Suppose we believe that the printer queue manager would be more appropriate as a starting point for the new queue manager we are constructing. We therefore move the cursor to "PrinterQueue" and press the "zoom in" key. The resulting screen is shown in Figure 4.6(e).

module PrinterQueue	catalog

procedure add(Process, PrinterQueue);
procedure remove(Process, PrinterQueue);
	programs
	modules
	queries
	updates
	templates

glossary	
PrinterQueue – pointer to record ... Process – record ...	

Fig. 4.6(e): After zooming in on "PrinterQueue".

To use the "PrinterQueue" module as the basis for the new module, we create a new configuration of the module called "RunQueue". Creating a configuration allows common portions of the module to be shared with the "PrinterQueue" module, meaning that improvements to one module, such as a bug fix or enhancement, can be easily passed on to the other module. Most systems do not provide this capability, but instead force the user to either copy the existing module or add code to the existing source that conditionally performs either the new or old functions.

Since no reference to configurations is currently on the screen, we must zoom in on the "templates" entry in the catalog. This operation will create a new window listing all the relations in the database, one of which is configurations.

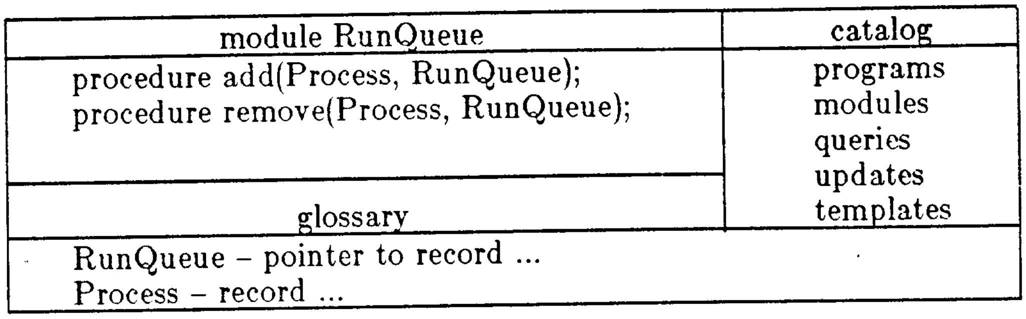
After creating a new configuration, we must define the type "RunQueue", change occurrences of the type "PrinterQueue" to the type "RunQueue", and add any other features not in the printer queue manager. To change type occurrences, we construct a global update in a similar manner to the way we first constructed a query. Figure 4.6(f) shows the screen after the update has been constructed.

update	catalog
range of v **is** variables **replace** v (type = RunQueue) **where** v.type = PrinterQueue	programs modules queries updates templates
glossary	
v – range variable variables – relation type – field of variables relation RunQueue – pointer to record ... PrinterQueue – pointer to record ...	

Fig. 4.6(f): After constructing update.

In addition to this update for the types of variables, it would also be necessary to perform similar updates for functions, parameters, and record fields. All of these updates could also have been defined as a single, higher-level operation that changes all references to a type to refer to a second type. In this case, the change would require simply selecting and executing this operation.

Figure 4.6(g) shows the "RunQueue" module after the updates have been performed.



module RunQueue

procedure add(Process, RunQueue);
procedure remove(Process, RunQueue);

catalog

programs
modules
queries
updates
templates

glossary

RunQueue - pointer to record ...
Process - record ...

Fig. 4.6(g): Newly constructed "RunQueue" module.

This construction shows an example of how the use of pointing, templates for abstractions, and interactions with database fit together. Since it is not taken from usage of a real interface, it is unnecessarily verbose. In practice, popular "short cuts" are likely to be used to specify frequently occurring combinations of commands. Nonetheless, this example shows the style of interaction that the OMEGA user interface provides.

4.7 Conclusions

Graphical input and output provide efficient and effective ways of expressing and representing the relationships between different program elements. Rather than expressing a program in terms of a language, programmers using OMEGA define and manipulate abstractions visually. Consequently, instead of having to parse lines of text and resolve identifiers to objects, causing conflicts that leads to multiple languages, OMEGA can provide a uniform model of abstraction and a simple structural interface.

Using a database, the same program may be manipulated according to several different viewpoints, including a traditional hierarchy. Recent developments in programming languages have favored modular structures, with restrictions on which objects and operations are available to which modules. OMEGA not only makes such constraints easy to describe and check, but allows auditing of usage in a natural way.

Languages such as Ada require the programmer to describe modules twice – once from the perspective of the implementor, and once from the perspective of a user. OMEGA allows these two perspectives to be defined as views on a single description, along with indications of what parts of the implementation should be visible to users. This facility can be generalized to allow different classes of users to have different levels of access to the implementation of a module.

Because the partially constructed program is stored in the program database, it is possible to immediately check for compile-time errors. Moreover, because the program is built rather than typed, a variety of common errors cannot be made.

The user interface described in this chapter together with the database organization described in chapter 3 forms the design of OMEGA. Although we have tried to fit the user interface and database together, we also have strived to focus each on its respective task so that implementation issues can be isolated to a particular area and solved using the general principles of the individual fields. In the next chapter, we discuss what we have done in the way of implementation to experiment with these ideas.

Chapter 5
Experimental Implementation

Dorothy: *Gee Toto, I don't think we're in Kansas anymore.*

. . .

Wizard: *Ignore that man behind the curtain!*

– from the movie *The Wizard of Oz*

5.1 Introduction

Often an implementation is more interesting than a design because it is a concrete representation of some ideas. People frequently confuse the two, not distinguishing between solutions to problems and the realization of these solutions. The best example of this kind of confusion is in programming languages, where a compiler is often used as an operational definition of the language.

It is important not to confuse the ideas presented in previous chapters with the implementation that we describe in this chapter. The number of ideas in the design of OMEGA make a complete implementation too much effort for this dissertation, not to mention the effort involved in preparing and polishing a system for use in a production environment.

Since we cannot build a production system, we certainly cannot evaluate the effect of OMEGA on programmer productivity. As we mentioned in Chapter 2, this would be a difficult task even with a working version of OMEGA, since just measuring programmer productivity is a hard problem.

The purpose of doing some of the implementation, then, was to learn more about our ideas. We particularly wanted to see if using a relational database system was feasible since the use of such a system is one of the more unusual ideas in the design. Some of the specific problems that had to be solved to do this include specifying a relational schema for a particular set of programming language constructs and interfacing to the database in terms of this schema.

We also wanted to experiment with our pointing-driven user interface and the window allocation ideas that are needed to support it. Problems in this part of the implementation include representing and displaying pictographs, allocating windows, and interfacing to the database system through the display so the user can ask queries on the program information.

In the remainder of this chapter we present an overview of the approach to implementing some of these ideas and then discuss the individual implementations in detail. We also briefly discuss the performance of our implementations and what it means about the design.

5.2 Approach

To experiment with our ideas, we needed an apparatus with which we could quickly build parts of OMEGA. Since our design is targeted for future hardware and database technologies, it was necessary to use components that do not provide the full power and speed we eventually expect to have available. In particular, we did not have access to a commercial database system or a bit-mapped graphics workstation.

The underlying environment that we could use to develop OMEGA was UNIX running on a VAX-11/780†. One of the main points of Chapter 3 was that OMEGA should use a general-purpose database system; we therefore chose the relational system INGRES because it was convenient (it runs on UNIX), familiar (we had used it before), and "supported" in the sense that the database research group at Berkeley was (and still is) using and doing research on INGRES. Talking to members of this group helped us both to understand how INGRES has evolved and what problems in database systems are currently being solved.

Once we had decided on the database system, we had to decide what to put in the database. The intent of OMEGA is to manage large software systems, we therefore wanted to be able to use OMEGA on a relatively large system. This desire meant we needed a way to quickly enter some existing software into the database, which in turn implied that we had to support the structure and semantics of some existing programming language even though we eventually wanted to provide the general form of abstraction discussed in Chapter 4.

The requirement of an existing body of software eliminated our first choice for a language to support, namely Ada. We also decided against the popular languages C and Pascal. The textual macro facilities available in C make it difficult to store the program in the database as the programmer really thinks of it.

We chose the programming language Model [Morris 80] over Pascal because it supports more recent programming concepts such as abstract data types and generic modules. We felt it was important to understand how to handle these facilities since they are present in Ada and other, newer languages. Also, the DEMOS operating system [Baskett, Howard, and Montague 77], which is being used as the basis for some operating systems research at Berkeley, is written in Model and provides an interesting testbed of evolving software.

†VAX is a trademark of Digital Equipment Corporation.

Given these tools, our implementation consists of *parse*, a program that takes Model text and enters it into the INGRES database, and *peruse*, a program that lets the user view, query, and modify the software that is stored in the database. Figure 5.1 shows how the various pieces of our implementation fit together.

5.3 Storing Program Information in an INGRES Database

The first thing we had to do was decide how to store the program information in the database. This decision involved designing the schema by defining the initial set of relations and views. Although we eventually wanted to be able to extend the kind of information we store in the database, for simplicity we assumed a static schema.

The schema has undergone several iterations, and is certainly not yet ideal. We used the following ideas to guide our design:

- Represent classes of objects by relations, individual objects by tuples.
- Use traditional programming language structures (variables, if-statements, etc.) as the object classes.
- Store a unique identification (UID), represented as an integer, in the first field of each tuple and use this number to refer to the tuple from other tuples. Use the number 0 to indicate a nil, or unassigned, reference.
- Use a (relation UID, tuple UID) pair to refer to an object whose representation can vary, e.g., an expression can be a function call, subscript operator, constant, etc.
- Represent information once; use views to represent different kinds of references to objects, e.g., variable usage is a view of the variables relation.

The resulting schema consists of 58 relations and 15 views for storing program information. A little less than half of the relations (26) are for traditional symbol table information, almost a third (19) for representing expressions, and the remainder split between representing statements (10), the string table (2), and a relation associating objects with the objects they contain (referred to as the *uses* relation in Chapter 3). The tables in Figures 5.2(a) and 5.2(b) list all the relations and their fields.

In addition to these relations, there are four auxiliary relations: *uniqueid*, *abstraction*, *bodyof*, and *viewof*. The *uniqueid* relation contains a single tuple with a single integer field that is the last UID to be assigned to some tuple. To allocate a UID this field is conceptually retrieved, incremented, and stored back in the database. In practice, it is too expensive to allocate UIDs one at a time like this, so a group of *n* UIDs are allocated at once by adding *n* to the field of the *uniqueid* tuple.

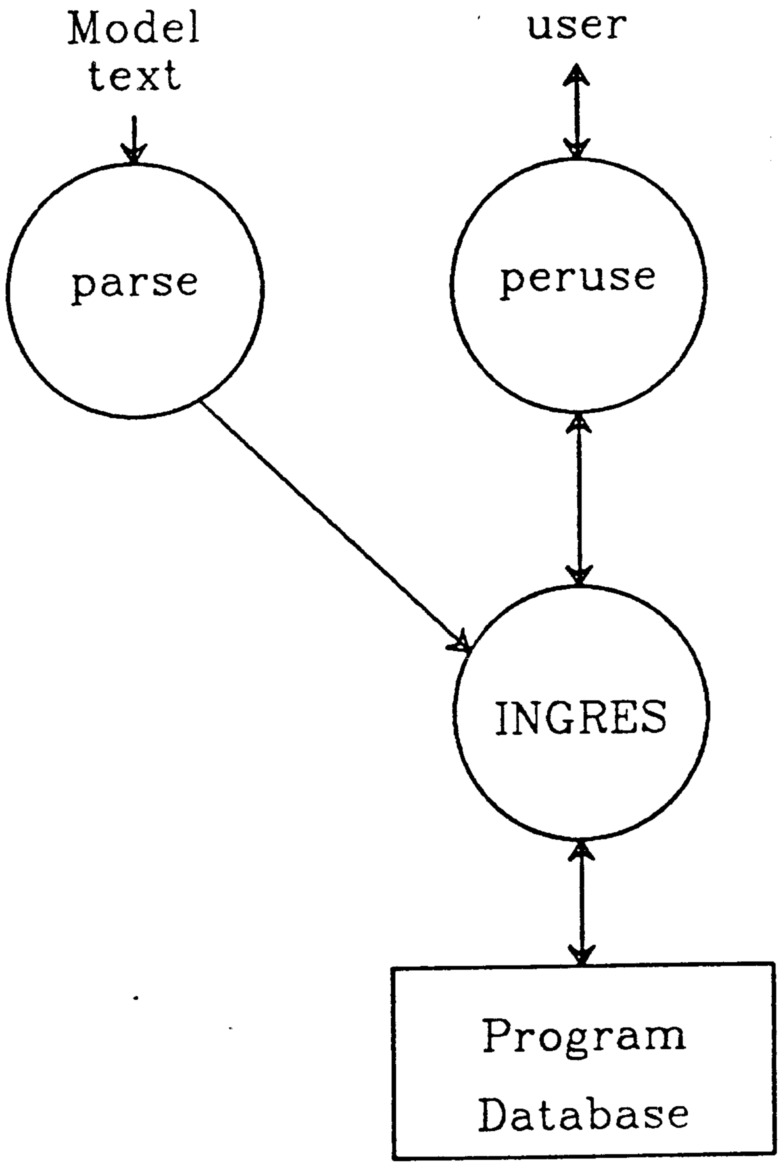


Fig. 5.1: Overview of implementation.

relation	fields (integers unless otherwise indicated)
programs	id, procedures
procedures	id, pictographs, proc-class, paramlists, decls, stmtlists
functions	id, pictographs, proc-class, paramlists, type_rel, type_id, decls, stmtlists, expr-rel, expr-id
proc_class	id, string = (public, inline, external, or builtin)
paramlists	id, parameters, paramlists
parameters	id, pictographs, param-class, type-rel, type-id
param_class	id, string = (readonly, varies, copied)
variables	id, pictographs, type-rel, type-id
constants	id, value-rel, value-id
decls	id, symbol-rel, symbol-id, decls
typerefs	id, pictographs, typelists
typenames	id, pictographs, type-rel, type-id
spaces	id, pictographs, typelists, reptype-rel, reptype-id, decls
space_param	id, pictographs
typelists	id, type-rel, type-id, typelists
ranges	id, type-rel, type-id, lower-rel, lower-id, upper-rel, upper-id
arrays	id, eltype-rel, eltype-id, typelists
dynarrays	id, eltype-rel, eltype-id
records	id, fieldlists
recurrecs	id, fieldlists
unions	id, fieldlists
fieldlists	id, fields, fieldlists
fields	id, pictographs, type-rel, type-id
enums	id, constlists
constlists	id, const-use, constlists
proctypes	id, proc-spec
pictographs	id, format = array[1..80] of char
pictof	object-rel, object-id, pictographs

Figure 5.2(a): Relations for symbol table information.

relation	fields (values are integers unless otherwise indicated)
stmtlists	id, stmt-rel, stmt-id, stmtlists
asgstmts	id, var-rel, var-id, expr-rel, expr-id
callstmts	id, proc-use, exprlists
ifthens	id, condlists, else-rel, else-id
condlists	id, cond-rel, cond-id, then-rel, then-id, condlists
casestmts	id, expr-rel, expr-id, pictographs, caselists, stmtlists
caselists	id, valuelists, stmtlists, caselists
valuelists	id, lower-rel, lower-id, upper-rel, upper-id, valuelists
loopstmts	id, before-rel, before-id, cond-rel, cond-id, stmtlists
forstmts	id, var-use, range-rel, range-id, incr-rel, incr-id, cond-rel, cond-id, stmtlists
fcalls	id, func-use, exprlists
exprlists	id, expr-rel, expr-id, exprlists
fieldrefs	id, record-rel, record-id, field-use
subscript	id, expr-rel, expr-id, exprlists
abstract	id, expr-rel, expr-id
concrete	id, expr-rel, expr-id
typerename	id, expr-rel, expr-id, type-rel, type-id
newexpr	id, type-rel, type-id, expr-rel, expr-id
lbnd	id, type-rel, type-id
ubnd	id, type-rel type-id
width	id, type-rel, type-id
strings	id, pictographs
intcons	id, value = integer
octcons	id, value = integer
charcons	id, value = integer
realcons	id, value = real
undefined	id
nilexpr	id
contains	outer-rel, outer-id, inner-rel, inner-id

Figure 5.2(b): Relations for statements and expressions.

The *abstraction* relation associates each relation name with a UID to allow references to relations to be stored as integers rather than character strings. The *bodyof* relation associates implementation views with definition views, and is used by *peruse* to implement the "zoom in" command. The *viewof* relation associates the predefined views with their underlying relations, and is used when directly accessing information from the database.

5.3.1 Predefined Views

The predefined views of the database represent definitions or uses of program objects. The use of views allows objects to be displayed differently depending on their context. For example, when displaying a variable as part of an expression, only the variable's pictograph is printed, but when displaying a declaration of a variable, its type is printed as well. Of course, it is possible in OMEGA to have the type displayed in expressions as well, but this is not the normal way people wish to see expressions displayed. Therefore, there is a view, called *var-use*, that is referred to by expression tuples. This view is defined by the following QUEL statements:

```
range of v is variables
define view var-use (id = v.id, pictographs = v.pictographs)
```

There are also views for uses of procedures, functions, parameters, constants, spaces, and type names.

The definitions of an object, such as a module, are also represented as a view of the corresponding implementation object. For example, there is a view of procedures called *proc-spec* defined as follows:

```
range of p is procedures
define view proc-use (
   id = p.id, pictographs = p.pictographs,
   proc-class = p.proc-class, paramlists = p.paramlists
)
```

Using views keeps the information on a procedure in a single place, but allows either the definitions or implementation of the procedure to be displayed.

5.3.2 Representation of Pictographs

The pictograph associated with each relation and view specifies how a tuple belonging to the relation or view should be printed. A pictograph is represented by a format string containing meta-characters to indicate where and how the fields of a tuple should be displayed. For example, the pictograph for the *var-use* view is

%2r

The "%" character indicates that the value of a field is to be displayed at the current output location, and the digit following the "%" (in this case a "2") indicates which field is to be displayed.

The character following the digit indicates how the field should be displayed. For the ***var-use*** example, this character is an "r" and means that the field is a reference to another tuple that should be retrieved and displayed according to the pictograph for its relation.

The "r" also indicates that the name of the relation to which the field refers is the same as the name of the field. For the example, the pictograph "%2r" for a *var-use* tuple therefore specifies that the second field is a reference to a tuple in the *pictographs* relation, because the name of the second field is "pictographs".

To indicate that a field to be displayed is a reference to some relation that is designated by an adjacent field in the tuple, the character "R" is used. For example, the *variables* relation, which is defined as

variables(id = integer, pictographs = i4, type-rel = i4, type-id = i4),

has as its pictograph the following:

%2r : %3R

The "3R" means the third field of the relation contains the UID for a relation, and the next (fourth) field contains the UID of a tuple in the relation.

Other characters to indicate how to display a field are "s", "d", "o", "c", and "f" for character string, decimal integer, octal integer, single character, and real number. In addition, the character "^" means to use the pictograph associated with the current tuple in the *pictof* relation, rather than the pictograph for the tuple's relation. This facility is used for displaying procedure and function calls, since calls to different procedures are displayed differently. It avoids the need for having a separate view defined for each individual procedure and function.

5.3.3 Semantic Constraints

In addition to the relations and views describing the structure of programs, we intended to specify constraints on the data to ensure that the program information was meaningful. For example, the *type-rel* and *type-id* fields of the *variables* relation together refer to a tuple that contains the type of the variable. The *type-rel* field contains the UID of some type relation (e.g., *typenames*, *arrays*, or *records*).

To ensure that the *type-rel* field actually refers to a type relation and not something else (e.g., *forstmts*), we would like to use database integrity constraints. If we had a relation called "types" that contained references to each relation that is legitimate as a type, then we would specify an integrity constraint as follows:

> **range of** t **is** types
> **range of** v **is** variables
> **define integrity on** v **is** v.type-rel = t.legit-type-rel

INGRES does not currently support this kind of integrity constraint. We therefore do not currently have any provisions within the database for ensuring the integrity of relation references, or enforcing any other constraints.

Parse is assumed to generate only correct program information into the database. We now turn to the problem of generating this information from Model text.

5.4 Parsing Model

The parsing necessary to compute the database tuples from text is the same as that which is performed during the first phase of a compiler. Ideally, we would have taken the first phase of the existing Model compiler and used it as the basis for *parse*. We spent some time trying this approach, since it would have guaranteed that our parser would recognize exactly the same language as the compiler, but we decided against pursuing it further for the following reasons:

1. The existing compiler was (and still is) quite slow. Since it is itself written in Model, we found it took an undesirably long time to test modifications.

2. Although the control structure of the compiler corresponded to the basic phases (parsing, name resolution, semantic checking, and code generation), the data structures contained a mix of information from all phases. They also did not match our schema very closely.

3. The compiler uses ad-hoc recursive descent parsing, making it difficult to build our own data structures.

Undoubtedly, (1) and (2) were the main reasons our attempt to use the existing compiler was not successful; (3) convinced us to start over using the parser generating program YACC [Johnson 78]. Using YACC, we were able to quickly construct a parser based on the grammar in the appendix of the Model reference manual.

5.4.1 Interfacing to INGRES

We initially tried writing tuples to the database while parsing, but this turned out to be undesirable since our schema was designed to contain references to *objects*, whereas information from parsing yields references to *names*. For example, the usage of a variable named "i" can be recognized during parsing, but which object it refers to cannot be determined because Model allows forward referencing.

The situation is more complicated for procedure and function calls, since Model allows (and OMEGA will also) the same name to be used for different procedures or functions so long as the types of the parameters differ. For example, "+" used on two integers is a different function from "+" when used on two real numbers.

As we showed in Chapter 4, our visual interface to the program database largely eliminates the process of resolving names to objects present in a traditional text interface. This level of interface, combined with our experience with INGRES performance, made it more desirable to have *parse* do semantic analysis than to perform analysis directly on the information in the database. We therefore changed the parsing phase to build data structures for the program information in memory, and then added a phase for analysis of these structures and a final phase to dump the information into the database.

The analysis phase performs both name resolution and some semantic checking, though we were not concerned with catching all semantic errors since we planned to enter only correct programs into the database. Semantic checking was helpful, however, in detecting bugs in *parse* as it occasionally reported errors for correct programs.

5.4.2 Name Resolution

Since in Model, any object can be referenced before it is defined, we construct special objects during parsing that are references to identifiers. From the syntactic context we know whether the reference is to a procedure, function, type, or else one of parameter, variable, and constant. Definitions of objects are also entered into a traditional symbol table.

Model allows blocks to be nested and has scope rules similar to Pascal. Since name resolution cannot be done during parsing, we cannot discard symbols at the end of the block in which they are defined. Resolving names after completing parsing also means that the symbol table lookup routine cannot assume that symbols are ordered by nesting depth.

To handle this forward referencing, each symbol in the symbol table is associated with a particular block and the block is used along with the identifier as a key for insertion and lookup. Since blocks are represented as pointers and identifiers are represented as pointers into a string table, we can use these two addresses as a key and therefore make the cost of hashing and comparing keys very cheap.

After parsing, we traverse all the objects in the program and attempt to resolve any of the references. The basic algorithm to resolve a reference could be written in Pascal as follows:

```
procedure resolve(name : Identifier; var sym : Symbol);
var b : Block;
   done : boolean;
   s : Symbol;
begin
   done := false;
   b := curblock;
   repeat
      s := lookup(name, b);
      if s <> nil then begin
         done := true;
         sym := s;
      end else begin
         b := outerblock(b);
         if b = nil then begin
            done := true;
            writeName(errorFile, name);
            writeln(errorFile, ' undefined');
         end
      end
   until done;
end
```

Parse also uses two variations of this algorithm, one for procedures or functions and one for types. Procedure and function names can be used more than once within the same scope if each definition has a different set of parameter types. The effect of this facility is that for each block it is necessary to iterate over all the procedures and functions in the block with the desired name and check to see if their parameters have matching types. It is not sufficient to stop when a match is found, since more than one procedure or function might satisfy the conditions and this should cause an error message to be printed.

Some procedures and functions (e.g., "=") are built-in but can be overriden by a user definition. If two functions match a call but one is built-in, then this is not an error and the call should be resolved to the user-defined function.

The other variation of the resolution algorithm is for types. In Model, abstract types (called *spaces*) can be parameterized by one or more types. Each distinct use of a space causes a new instance of the body of the space to be created with the actual types substituted for the formal type parameters. The approach to resolving a reference to a space is similar to that for a procedure or function call, except that there are neither conflicts nor builtin spaces, and if a reference is not resolved then a new instance of the space should be created.

5.4.3 Current Status

The implementation of *parse* is complete. Though it has not been rigorously tested or heavily used, it has successfully parsed, performed type analysis, and stored both the DEMOS kernel and the Model compiler (over 15,000 lines) into an INGRES database.

Parse is approximately 10,000 lines of C code, and took a total of about 4 months full-time effort to develop. We chose C to make interfacing to YACC easy; we would have preferred a language that provides for more semantic checking at compile time.

5.5 Viewing Programs from the Database

Although it has been necessary at times, looking at programs as tuples in INGRES is quite painful, being similar in many respects to looking at a hex memory dump. The purpose of implementing *peruse* was as much to see that the information was really in the database as to experiment with the user interface ideas presented in Chapter 4.

Since we did not have the graphics capabilities we wished and since we could not hope to implement the entire interface in a short time, we focused our implementation efforts on two areas: the interface between *peruse* and INGRES and the management of the screen area. These two areas are related by the convention that each program thread (i.e., view of the database) is displayed in a different window on the screen.

A query, then, causes the information to be retrieved from the database and displayed into a new window. In the four subsections that make up the remainder of this section we discuss how *peruse* interfaces to the database, how information is displayed on the screen, how commands are entered and processed, and what the current status of the implementation is.

5.5.1 Database Interface

The first version of *peruse* had static knowledge of the kinds of objects and corresponding names of relations in the database, as well as having explicit code for displaying objects. This version enabled us to see the information in the database displayed as normal program text. However, we quickly ran into the following three problems:

1. It was very expensive to display the body of a procedure. For a 10-line procedure it took over 20 seconds of CPU time (about 2 minutes elapsed time) on our VAX.

2. Every time the schema changed, a substantial portion of *peruse* had to be changed.

3. The general concept of a pictograph, as presented in Chapter 4, was missing from the implementation.

In the next version of *peruse* we therefore looked at improving performance and generalizing it, both by removing most schema dependencies and by using pictograph information stored in the database to drive the display algorithm.

5.5.1.1 Improving Performance

The problem with response time was due to retrieving each object with a separate query. For example, suppose the user wishes to see the body of a procedure. This object is represented by a single tuple from the implementation view of the *procedures* relation. When this tuple is displayed, all the different objects within the procedures (statements, variables, etc.) have to be retrieved.

The problem of processing a large number of small queries is a general one. Queries have an inherent amount of overhead due to the parsing, access strategy selection, and locking that is necessary. Our first attempt to solve this problem was to retrieve all the objects in a procedure at once rather than through individual queries. To do this we had to know in what procedure every object was defined. We kept this information first as a field in each object and later in the *contains* relation.

Using the *contains* relation worked well for very small programs, but was too expensive for larger programs, such as the DEMOS kernel. A single query involving a join of *contains* (over 20,000 tuples) with *stmtlists* (over 1,000 tuples) took over 13 seconds of CPU time.

Keeping the procedure where an object belonged in a field within the object was not as expensive, but still required a query for every relation to retrieve all the objects in a particular procedure. For example, even if a procedure did not have any variables defined in it, a query would be generated on the *variables* relation.

As mentioned in Chapter 3, retrieving all information for a given procedure is only helpful for one particular, albeit common, view. When some other view is desired, such as a collection of statements or declarations that cross procedure boundaries, it is not desirable to retrieve all the information associated with all the different procedures.

We therefore went back to the approach of retrieving a tuple at a time and tried to reduce the amount of time it took to do an individual retrieval. The most frequent queries were of the form

```
range of t is some-relation
retrieve (t.all) where t.id = some-id
```

for a given UID and relation. To minimize the searching necessary to perform this query, we advised INGRES to keep a hash table on all relations using their *id* field as the key.

Knowing the exact form of the query and the appropriate access strategy for it, we modified *peruse* to perform these queries using the INGRES access methods directly. This approach avoids the overhead associated with query processing, and in addition, since INGRES runs as a process separate from *peruse*, also avoids the overhead of exchanging messages via UNIX pipes.

This modification gave the effect of compiled queries, since what we did was "hand-compile" a particular class of queries. These queries still ran as separate transactions, meaning no pages were buffered across queries. To simulate transactions, or more precisely buffering across queries, we modified *peruse* to keep relations open rather than closing them at the end of each query. The table below shows the performance of peruse zooming in on the body of 5-line program using standard queries, hand-compiled queries, and hand-compiled queries with buffering.

Queries	# tuples	# pages read	CPU time (seconds)	Elapsed time (seconds)
standard	36	281	30.7	40
compiled	36	156	4.8	13
buffered	36	93	3.4	7

Compiling queries had a dramatic effect on performance, reducing CPU time by more than a factor of six, while buffering had a more modest effect. This effect might be more pronounced for larger programs. These results indicate that a production implementation of OMEGA requires a database system that can compile queries. Some buffering capability would also help performance.

5.5.1.2 Generalizing *Peruse*

Eliminating dependencies on the database schema in *peruse* code required a general mechanism for retrieving, displaying, and updating information based on input commands. We implemented this mechanism by using a dynamic schema rather than a static one, and by using the pictograph for an object to display the object.

Using a dynamic schema means keeping the information on how to interpret a particular command for a particular class of objects in the database rather than having it written into the *peruse* code. For example, *peruse* provides a command to "zoom in" on an object. Originally, the semantics of zooming were made explicit for particular program objects by having a different routine for each class of object (programs, procedures, etc.). The routine for program objects, for example, generated a query to retrieve and display the body of the main procedure in the program.

To generalize zooming in on a particular object we created the *bodyof* relation with two fields, one that names a definitions view and one that names the implementation view. This relation is then used to find the implementation view for the object's class. The pictograph for the implementation view indicates how to display the body of the object.

For example, *bodyof* contains the tuple

(*var-use*, *variables*).

If the user asks to zoom in on the usage of a variable, this tuple is used to retrieve the tuple in the *variables* relation with the same UID as the specified tuple in the *var-use* view. Using the pictograph for *variables*, the variable's name and type are then printed.

A consequence of accessing the schema dynamically is that most operations cause several accesses to the database and therefore are slowed down substantially. Since many of these accesses are to the *abstraction*, *relation*, *attribute*, and *pictographs* relations, we read a copy of these relations into memory when starting up and access this copy instead of the database.

5.5.2 Display Management

When a view of the database is requested, the information is retrieved and transformed into text using pictographs and stored into a *picture*. During this transformation, each object and its location within the picture is recorded in a *map*. Afterward, a rectangular portion of the screen, called a *window*, is allocated and as much of the picture as will fit is displayed in the window. The associated picture, map, and window are kept together in a data structure called a *scene*. Figure 5.3 shows an example of a *scene*.

Also associated with each scene is a cursor that refers to the current program object of interest in the associated view. This cursor is not a character cursor as in a text editor since the object can be represented by more than one character (or even more than one line) in the picture. The text associated with the current object is highlighted on the screen.

Windows are allocated at a static location and have a fixed size; the sophisticated allocation scheme discussed in Chapter 4 has not been implemented. The screen is divided in half horizontally and vertically to form four partitions that are used as windows. This partitioning does not include the top and bottom lines of the screen, which are used for status information and error messages respectively. When a new window is to be allocated, *peruse* uses a window that is either unallocated or least recently touched.

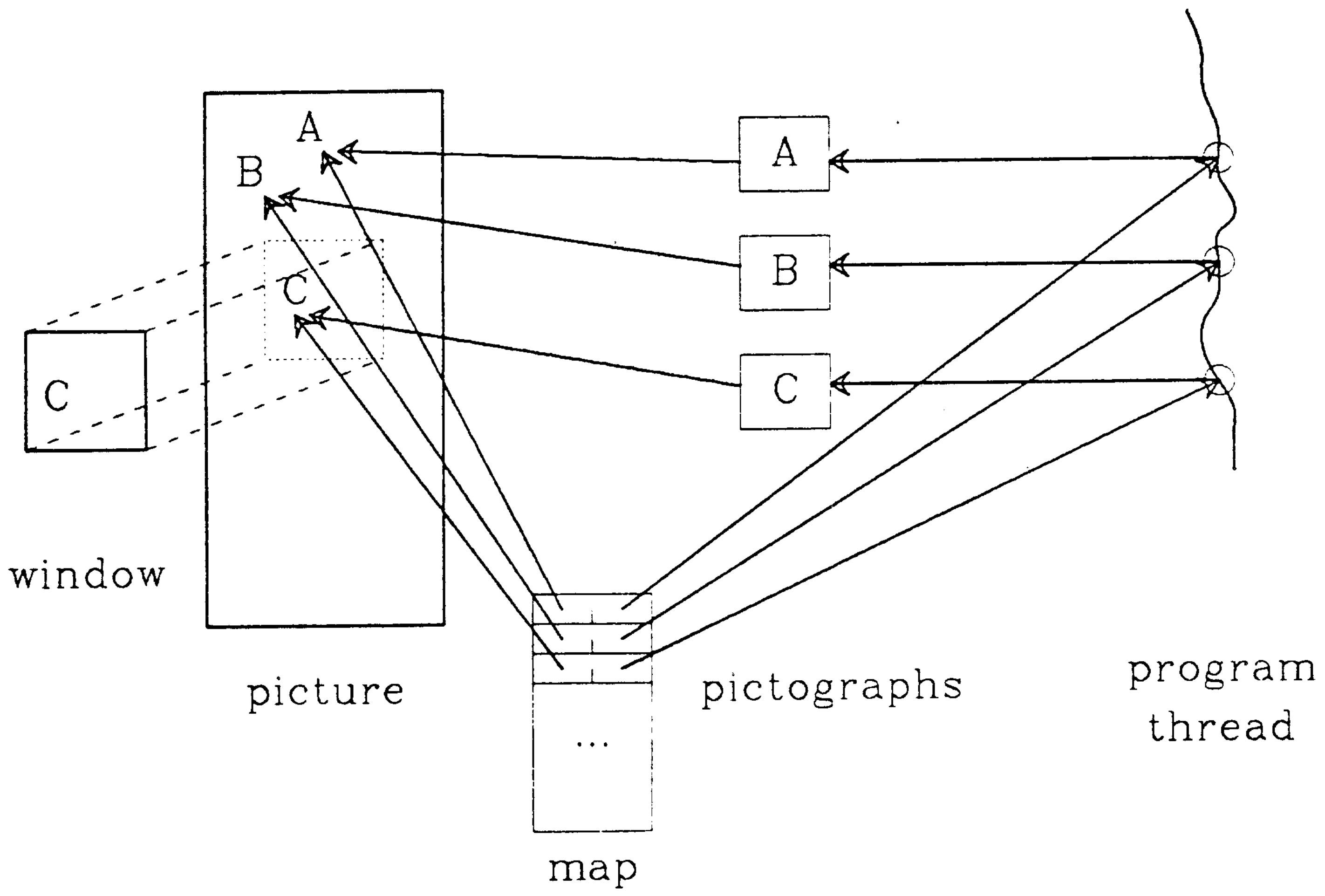


Fig. 5.3: The structure of a scene in peruse.

5.5.3 Input Commands

Commands are entered by a single keystroke, and specify an operation on the current object in the current scene. The table below shows the commands that are recognized by *peruse* (the notation "↑X" indicates the control key is held down while pressing the key "x").

key	command
↑F	scroll forward
↑B	scroll backward
↑R	rotate left
↑G	rotate right
↑D	redraw
s	select
S	pick up
w	move cursor forward
b	move cursor backward
e	zoom in
v	show slots
c	create
f	fill in

The "select" command requests that the scene's cursor be moved to the object nearest the input cursor. Ideally, the input cursor is controlled by a pointing device; however, it can also be controlled by cursor movement keys. "Pick up" is just like "select" except that the object is pushed on a stack for use with future commands.

An input cursor refers to a particular character location and therefore could be ambiguous, e.g., selecting the "a" in "a := b + c" could refer to either the variable named "a" or the entire assignment statement. To allow fine tuning of the cursor position, the "move cursor forward" and "move cursor backward" commands move the cursor according to the order in which the objects were traversed when they were displayed.

The "zoom in" command finds an object's relation UID in the *bodyof* relation and uses the UID of the associated relation to display the object. For example, since the tuple (*proc-spec*, *procedures*) is in the *bodyof* relation, pressing "zoom in" when the current object is a *proc-spec* causes a new window to be allocated and the body of the procedure to be displayed in it.

The "create" command creates a new object with the same relation as the current object. The "show slots" command forces unfilled fields of an object to be displayed as "<*relation-name*>"; they are not normally displayed. The "fill in" command can be used to set the value of an unfilled field to either a reference to an object that has been picked up, or a literal value (e.g., pictograph or integer) entered by the user.

5.5.4 Current Status

Peruse is continually being changed in an attempt to evolve it toward the interface described in Chapter 4. The first version used a static schema and could zoom in on program objects only. It also provided a command for finding all the uses of an object, causing a set of predetermined queries to be sent to INGRES and the results displayed. This facility demonstrated the power of using a database system, since providing it required very little additional code in *peruse*.

The second version of *peruse* used a dynamic schema and direct access to the database to dereference UIDs. It displayed objects using the pictographs stored in the database, but it did not allow objects to be created or modified. It did provide multiple windows, unlike the first version, but they were allocated in a static location and had a fixed size.

The third, and current, version completes the basic capabilities of the interface. Although it does not yet manage windows in the desired manner, it does have the ability to create objects and fill in slots. General queries and global relational updates are not currently supported, but only require query (or update) objects to be able to be defined and executed. Executing a query involves translating it into QUEL and sending it to INGRES, with the result displayed in a newly-allocated window.

5.6 Performance

Throughout this thesis we have focused on power and largely ignored performance. This focus has not been because we think performance is unimportant, but because experience has led us to prefer to tune a general, powerful solution to a problem than to extend the power of an already tuned system.

Since we have not had the time to do a careful analysis of the performance of any of the pieces of OMEGA, it is somewhat misleading to present any execution time measurements of *parse* or *peruse*. Nonetheless, our experience is undoubtedly of interest, so we present some simple measurements.

5.6.1 Execution Time

Figure 5.4 shows a table with execution times for the time it takes *parse* to process the DEMOS kernel and store the resulting information in the database, and the time it takes *peruse*, using compiled queries and buffering as described earlier, to zoom in on the body of the main procedure of DEMOS. For comparison purposes, the time *parse* takes to perform the parsing and type analysis phases, without storing into the database, is also shown.

Operation	CPU time	Elapsed time
	(seconds)	(mm:ss)
parsing DEMOS and storing in database	422.1	27:29
parsing DEMOS only	53.3	1:32
peruse main body	276.6	11:36

Figure 5.4: Time measurements for *parse* and *peruse*.

The time it takes to zoom in on the main body of DEMOS is an interesting benchmark, but does not reflect actual response time since the main body of DEMOS is fairly large (over 3000 tuples, the equivalent of nearly 1000 lines of text). Ideally, *peruse* should stop retrieving tuples when the output will no longer fit on the screen, and the picture data structure that is built from the output of a query should be filled on demand instead of all at once.

5.6.2 Storage Requirements

The table in Figure 5.5 shows the number of tuples and total size of the largest relations in the database. Although the total size is close to that of the corresponding text, this is somewhat misleading because it does not include space for indices (in this case hash tables) or comments. The elimination of comments was done for simplicity; there is no reason they could not also be stored.

relation	# tuples	width (bytes)	total size (bytes)
pictographs	1563	84	131292
typeof	4521	16	72336
exprlists	3507	16	56112
parameters	1720	20	34400
functions	712	40	28480
stmtlists	1567	16	25072
fieldrefs	1471	16	23536
paramlists	1720	12	20640
decls	1180	16	18880
fcalls	1473	12	17676
asgstmts	812	20	16240
pictof	1001	12	12012
procedures	399	24	9576
condlists	314	24	7536
intcons	700	8	5600
variables	338	16	5408
callstmts	448	12	5376
fields	335	16	5360
constants	292	16	4672
ifthens	290	16	4640
OTHERS	2115	–	38372
total	26515	–	536316
size of text	–	–	418792

Figure 5.5: Space usage in database for DEMOS kernel.

5.6.3 Analysis

The critical performance problem is response time in *peruse*, which is roughly an order of magnitude too slow. In general, the database system should be able to use main memory and semantic information, such as denoted by tuple references, to provide substantially better performance. Particular issues in the performance of current relational systems are discussed in more detail in [Chamberlain, et al. 81], [Stonebraker, et al. 83], and [Bitton, DeWitt, Turbyfill 83].

Improved data management algorithms will be a major factor in improving the performance of OMEGA, the other major factor will be running the system on faster hardware. Within five years it is likely that most programmers will use personal workstations that have the same or greater computing capacity than the VAX that we currently share with 10 to 20 other users.

The size of programs and information pertaining to programs is growing, meaning it will be necessary to continue to analyze system performance and look for techniques to improve it. We are convinced the use of a general-purpose database system to manage all program information will soon be practical in terms of performance when compared with conventional systems. The algorithmic complexity of database algorithms is sublinear whereas the text-oriented algorithms require linear time; it is only a matter of time before the increase in volume of information and reduction in database overhead makes the database approach faster.

It is true that it will always be possible to construct a special-purpose database system tuned to managing program information that is faster than the general-purpose system. Similarly, it is always possible to write assembly language by hand that executes faster than that generated by a compiler. However, just as for writing in a high-level language, it will be worth the slight loss in efficiency to use the more general, better supported, and more reliable system. General-purpose database systems are rapidly approaching this threshold of being cost-effective for use in managing program information.

5.7 Conclusions

We did not attempt a complete implementation of OMEGA; such an effort would have been premature. Instead we have experimented with the program database and user interface pieces of the system.

We have built *parse*, a program that takes Model source text and stores all the information in the program into a database managed by INGRES. *Parse* recognizes the full Model language and has been used to load a medium size program (DEMOS) into the database.

We have also built *peruse*, a program that displays information from the database onto the screen in a traditional text format. *Peruse* processes single keystroke input commands that allow the full power of the database to be accessed without using a command syntax or names to refer to objects.

Overall, we are pleased with the results of using INGRES. Although there are problems with performance, we have not had to worry at all about managing permanent storage or processing queries. The ability to define general views was particularly useful.

Our experience interfacing to INGRES has affected the ideas presented in Chapter 3. In particular, doing this implementation and examining the resulting problems helped produce the idea of tuple references.

We have not had as much experience in implementing the user interface as we would have liked, due both to time limitations and lack of hardware with the power we wish to use. Whereas with the database we had a powerful, albeit slow, system, we simply did not the have terminal capabilities with which we wanted to experiment.

We did implement enough of the user interface to realize that window allocation was an important issue. This realization was a surprise; we had expected to be able to use an existing window allocation scheme.

Chapter 6

Conclusions and Future Directions

And here I sit so patiently,
waiting to find out what price
you have to pay to get out of
going through all these things twice.

– from the song *Stuck inside of Mobile with the Memphis Blues Again*
by Bob Dylan

6.1 The Software Beast (reprise)

We began this thesis by describing our perception of why software is a "beast" that is so difficult to control. The major reason for the beast's strength is that in existing environments it is difficult for programmers to get a good view of software that has already been written, and therefore constantly re-implement algorithms.

To attack these problems we focused on system support for viewing and manipulating existing pieces of software. Now, after visiting other programming environments, traveling through the land of relational database systems, and synthesizing a user interface centered on the use of pointing, we conclude by summarizing the ideas of our work and describing problems for future research. At the end of the chapter we finish this dissertation with some general thoughts on our experience.

6.2 Summary of Work

The ideas in OMEGA have been aimed primarily at supporting flexible visual and logical manipulation of large software systems. To support manipulation of programs, we have designed an interface to a relational database system to provide a mechanism for querying and modifying software. We have built an experimental interface to the INGRES database system and used it to store and retrieve programs written in Model.

The use of a database system provides a more general and powerful mechanism for manipulating software than is available in current software development environments. If there is one point in this thesis that is more important than any other, it is that using a general-purpose database system to manage program objects offers significant advantages over text-oriented or special-purpose systems.

The medium of communication between programmer and system is equally important as the data that is transmitted. We have designed a medium similar to that provided by Smalltalk, generalizing the interface to consistently use pointing at objects as the means of conversing.

Pointing avoids traditional obstacles to software development such as syntax errors and mistyped identifiers. As a result, concepts that are normally offered in separate languages (e.g., pipes in UNIX and procedures in Pascal) can be integrated together without sacrificing the visual presentation.

Using the database concept of integrity constraints adds interactive semantic checking to OMEGA so that errors can be detected as soon as possible. This facility, combined with an extended notion of a database transaction, provides a systematic and flexible mechanism for ensuring that only meaningful (though not necessarily correct) programs objects are created.

We have implemented the basic elements of the user interface to allow browsing and querying of information in the database using pointing. Our implementation uncovered important problems in allocating and positioning windows, and we have begun to develop solutions to these problems. Due to time and hardware constraints, we have not been able to experiment with this interface and satisfactorily evaluate its usefulness.

6.3 Future Directions

Although this chapter completes this dissertation, there is much work to be done to further test and refine our ideas as well as solve new problems. Overall, we would like to continue trying out ideas that will take the implementation of OMEGA toward a complete and usable system, both for our own use in further experimentation and to confirm that our approach is practical. In addition, we would like to generalize our solutions so that they can be applied to other interactive computing environments.

There are many problems that need to be solved to achieve these goals. We will discuss the following areas in more detail:

- analyzing and improving the performance of the database interface,
- executing and debugging programs stored in the database
- experimenting with the user interface on a graphics terminal
- application to office information and VLSI design environments

6.3.1 Database Interface

Currently, the database system is certainly the bottleneck of OMEGA in terms of performance. In Chapter 3 we suggested tuple references as a way of providing the database system with the information necessary to enable it to cache and prefetch tuples in memory, and thereby reduce the cost of simple queries that retrieve individual tuples. Caching in general offers great gains, but conflicts with equally important crash recovery and concurrency control facilities.

General techniques for caching data in memory and processing queries on it will even be more important for larger databases. The database we created in our implementation corresponds to a relatively small piece of software (about 10,000 lines). If this approach is to be applicable to systems several orders of magnitude larger, we must be able to use logical information in the database to restrict the amount of data that has to be searched. For example, a programmer working on a single module will rarely access information in the database outside that module.

In Chapter 3 we also noted that for certain queries we need to be able to search the transitive closure of a binary relation. Although the semantics of this operation are well-defined, it is not immediately obvious what the correct implementation should be. It may be desirable to retain a previously computed closure, but the tradeoff in additional cost for updates needs to be examined more closely.

Finally, our experience is limited to a single data model, schema, and database system. It would be helpful in understanding the effects of database system facilities on the performance of OMEGA to be able to compare the same operations with different systems.

Although *peruse* has evolved to become schema independent, it is still dependent on INGRES. We need a single interface that can be adapted to different database systems.

6.3.2 Execution and Debugging

For OMEGA to be usable, it must be possible to run the programs that are stored in the database. Generating code is primarily a matter of traversing the information in the database, but there are interesting issues concerning what information should be stored back into the database (e.g., storage locations, a record of optimizations performed) and what additional information should be kept in the database incrementally to aid code generation (e.g., data flow information).

The question of what information a code generator should add to the database is related to the debugging facilities that are to be provided. In OMEGA, the program monitor will undoubtedly need to share information with the code generator. Further implementation of both creating and debugging executable programs is necessary to determine the best way to support these facilities.

In Chapter 4 we showed how to specify debugging events as relational qualifications on the information in (or virtually in) the database. This approach provides a powerful, high-level description of events, but complicates the translation of these conditions into low-level actions such as machine traps on references to particular instructions or data.

The optimal translation of an event, the one that degrades execution the least while correctly detecting the event, is dependent on the target machine, operating system, code generator, and language. We need a model and analysis of target hardware systems coupled with a characterization of debugging events that provides an algorithm for optimal translation.

6.3.3 Graphical Interface

Although we have designed and partially implemented the graphical interface described in Chapter 4, we have not been able to experiment with using it. We would like to actually use the interface to evaluate how effective it is in manipulating programs.

We suspect that relying on pointing may at times be verbose; that is, many objects may have to be selected for a relatively simple operation. To solve this problem we would need to provide some "shorthand" for this class of operations. To correctly "tune" the user interface in this manner requires more experimentation with its implementation.

Fundamentally, we want to be able to easily modify a piece of software for a slightly different use. We therefore also need to experiment with the user interface to find out how conveniently it allows these kinds of manipulations, and work to solve problems that may be encountered.

6.3.4 Application to Other Environments

Many of the problems in managing software are general problems that arise in other applications. We have tried to find general solutions to the specific problems of a programming system, and would like to see if these solutions can be applied to other environments.

The model of a graphical, pointing-oriented interface to a general-purpose database system is representative of many interactive computing systems. We would particularly like to experiment with our ideas for office information and VLSI design environments.

Forms [Tsichritzis 83] in office systems are similar in many ways to our abstractions and pictographs, and the problems of VLSI design data [Katz 82] are quite similar to our problems in managing program information. Both environments have been developed independently of software environments, it would be beneficial for both areas to look into exchanging and merging ideas.

6.4 Concluding Remarks

Just as we have promoted the use of existing software in the construction of new programs, throughout this thesis we have tried to use ideas from other programming environments and research on database systems to design and partially implement OMEGA.

To adapt ideas such as the relational data model and the Smalltalk medium, we first had to understand and generalize the problems we were trying to solve. An important part of understanding and generalizing these problems depended on separating the semantics of each problem from a potential implementation of its solution.

We hope by generalizing and building upon previous good ideas this thesis is a step toward a time when building upon good software is easy and straightforward, not weighed down by problems of information management and visual presentation.

Bibliography

[Ada 82]
Reference Manual for the Ada Programming Language, U. S. Department of Defense, July 1982.

[Arnold 80]
Arnold, K., "Screen Updating and Cursor Movement Optimization: A Library Package", Computer Science Division, University of California, Berkeley, 1980.

[Baskett, Howard, and Montague 77]
Baskett, F., Howard, J. H., and Montague, J. T., "Task Communication in DEMOS", *Proceedings of the Sixth Symposium on Operating Systems Principles*, November 1977.

[Bitton, DeWitt, and Turbyfill 83]
Bitton, D., DeWitt, D., and Turbyfill, C., "Benchmarking Database Systems: A Systematic Approach", to appear in *Proceedings of the Int. Conf. on Very Large Data Bases*, October 1983.

[Bonanni and Glasser 77]
Bonanni, L. E., and Glasser, A. L., "SCCS/PWB User's Manual", Bell Laboratories, 1977.

[Cattell 83]
Cattell, R., "Design and Implementation of a Relationship-Entity-Datum Data Model", Xerox PARC Tech. Report CSL-83-4, May 1983.

[Chamberlain, et al. 81]
Chamberlain, D. D., Astrahan, M. M., King, W. F., Lorie, R. A., Mehl, J. W., Price, T. G., Schkolnick, M., Selinger, P. Griffiths, Slutz, D. R., Wade, B. W., and Yost, R. A., "Support for Repetitive Transactions and Ad Hoc Queries in System R", ACM *Transactions on Database Systems*, Vol. 6, No. 1, March 1981.

[Chen 76]
Chen, P. P., "The Entity-Relationship Model – Toward an Unified View of Data", ACM *Transactions on Database Systems*, Vol. 1, No. 1, March 1976.

[Codd 70]
Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, Vol. 13, No. 6, June 1970.

[Codd 79]
Codd, E. F., "Extending the Database Relational Model to Capture More Meaning", ACM *Transactions on Database Systems*, Vol. 4, No. 4, December 1979.

[Conway and Constable 76]
Conway, R., and Constable, R., "PL/CS - A disciplined subset of PL/I", Tech. Report No. 76-293, Dept. of Computer Science, Cornell University, 1976.

[Dayal and Bernstein 82]
Dayal, U., and Bernstein, P., "On the Correct Translation of Update Operations on Relational Views", *ACM Transactions on Database Systems*, Vol. 7, No. 3, September 1982.

[Deutsch and Taft 80]
Deutsch, P., and Taft, E., eds., "Requirements for an Experimental Programming Environment", Xerox Corporation, Palo Alto Research Center, June 1980.

[Eswaran 76]
Eswaran, K., "Specifications, Implementations, and Interactions of a Trigger Subsystem in a Integrated Database System", *IBM Research*, RJ 1820, San Jose, Ca., August 1976.

[Feldman 78]
Feldman, S. I., "Make - A Program for Maintaining Computer Programs", Bell Laboratories, Murray Hill, New Jersey, 1978.

[Goldberg and Robson 83]
Goldberg, A., and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

[Green, et al. 81]
Green, C., Phillips, J., Westfold, S., Pressburger, T., Kedzierski B., Mont-Reynaud, B., Tappel, S., "Research on Knowledge-Based Programming and Algorithm Design - 1981", Memo KES.U.81.2, Kestrel Institute, Palo Alto, California, 1981.

[Habermann, et al. 82]
Habermann, A. N., Ellison, E., Medina-Mora, R., Feiler, P., Notkin, D., Kaiser, G. E., Garlan, D. B., and Popovich, S., "The Second Compendium of Gandalf Documentation", CMU Department of Computer Science, May 24, 1982.

[Ivie 77]
Ivie, E. L., "The Programmer's Workbench - A Machine for Software Development", *Communications of the ACM*, Vol. 20, No. 10, October 1977.

[Johnson 78]
Johnson, S., "Yacc: Yet Another Compiler-Compiler", Bell Laboratories, Murray Hill, NJ, 1978.

[Joy 79]
Joy, William N., "An Introduction to Display Editing with Vi", University of California at Berkeley, Berkeley, CA, 1979.

[Katz 82]
Katz, R., "A Database Approach for Managing VLSI Design Data", *Proceedings of the 19th Design Automation Conference*, June 1982.

[Kernighan and Mashey 81]
Kernighan, B., and Mashey, J., "The Unix Programming Environment", *Computer*, Vol. 14, No. 4, April 1981.

[Kernighan and Ritchie 78]
Kernighan, B., and Ritchie, D., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Medina-Mora and Feiler 81]
Medina-Mora, R., and Feiler, P., "An Incremental Programming Environment", *IEEE Transactions of Software Engineering*, Vol. SE-7, No. 5, September 1981.

[Morris 80]
Morris, J. B., *A Manual for the Model Programming Language*, February 1980.

[Powell and Linton 83a]
Powell, M., and Linton, M., "A Database Model of Debugging", accepted for publication in *Journal of Systems and Software*. Preliminary draft in *Proceedings of the ACM SIGSOFT-SIGPLAN Symposium on High-Level Debugging*, March 1983.

[Powell and Linton 83b]
Powell, M., and Linton, M., "Database Support for Programming Environments", *Proceedings of the Database Week Special Session on Databases for Engineering Applications*, May 1983.

[Powell and Linton 83c]
Powell, M., and Linton, M., "Visual Abstraction in an Interactive Programming Environment", *Proceedings of SIGPLAN 83: Symposium on Programming Language Issues in Software Systems*, June 1983.

[Rochkind 75]
Rochkind, M. J., "The Source Code Control System", *IEEE Transactions on Software Engineering*, Vol. SE-1, December 1975.

[Rowe 82]
Rowe, L., *private communication.*

[Rowe, et al. 81]
Rowe, L., Cortopassi, J., Doucette, D., and Shoens, K., *RIGEL Language Specification*, Comp. Sci. Div., Dept. of EECS, University of California at Berkeley, June 1981.

[Schmidt 82]
Schmidt, E., "Controlling Large Software Development in a Distributed Environment", Xerox PARC Tech. Report CSL-82-7, December 1982.

[Softech 79]
UCSD Pascal User Manual, Softech Microsystems, Inc., San Diego, California, 1979.

[Stonebraker 82]
Stonebraker, M., "Application of Artificial Intelligence Techniques to Database Systems", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/31, May 1982.

[Stonebraker, et al. 82]
Stonebraker, M., Stettner, H., Kalash, J., Guttman, A., and Lynn, N., "Document Processing in a Relational Data Base System" Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/31, May 1982.

[Stonebraker, et al. 83]
Stonebraker, M., Woodfill, J., Ranstrom, J., Murphy, M., Meyer, M., and Allman, E., "Performance Enhancements to a Relational Database System", *ACM Transactions on Database Systems*, Vol. 8, No. 2, June 1983.

[Stonebraker, Johnson, and Rosenberg 81]
Stonebraker, M., Johnson, R., and Rosenberg, S., "Extending INGRES with a Rules System", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 81/93, December 1981.

[Stonebraker and Kalash 82]
Stonebraker, M., and Kalash, J., "TIMBER: A Sophisticated Relation Browser", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/17, January 1982.

[Stonebraker and Keller 80]
Stonebraker, M., and Keller, K., "Embedding Hypothetical Data Bases and Expert Knowledge in a Data Manager", *Proceedings of the 1980 ACM-SIGMOD Conference on Management of Data*, Santa Monica, Ca., May 1980.

[Stonebraker and Rowe 82]
Stonebraker, M., and Rowe, L., "Database Portals: A New Application Program Interface", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/80, November 1982.

[Stonebraker, Wong, and Kreps 76]
Stonebraker, M., Wong, E., and Kreps, P., "The Design and Implementation of INGRES", *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[Teitelbaum and Reps 81]
Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-directed Programming Environment", *Communications of the ACM*, Vol. 24, No. 9, September 1981.

[Teitelman and Masinter 81]
Teitelman, W., and Masinter, L., "The Interlisp Programming Environment", *Computer*, Vol. 14, No. 4, April 1981.

[Tsichritzis 82]
Tsichritzis, D., "Form Management", *Communications of the ACM*, Vol. 25, No. 7, July 1982.

[Ullman 80]
Ullman, J., *Principles of Database Systems*, Computer Science Press, 1980.

[Waters 82]
Waters, R. C., "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 1, January 1982.

[Woodfill, et al. 79]
Woodfill, J., Whyte, N., Ubell, M., Siegal, P., Ries, D., Meyer, M., Hawthorn, P., Epstein, B., Berman, R., and Allman, E., "INGRES 6.2 Reference Manual", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 79/43, May 1979.

[Wulf, London, and Shaw 76]
Wulf, W., London, R., and Shaw, M., "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976.

[Zaniolo 83]
Zaniolo, C., "The Database Language GEM", *Proceedings of the 1983 ACM-SIGMOD International Conference on Management of Data*, San Jose, California, May 1983.