Implementing Relational Views of Programs

Mark A. Linton

Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, California 94305-2192

ABSTRACT

Configurations, versions, call graphs, and slices are all examples of *views*, or cross-sections, of programs. To provide a powerful mechanism for defining, retrieving, and updating these views, the OMEGA programming system uses a relational database system to manage all program information.

We have built a prototype implementation of the OMEGA – database system interface. This implementation includes the design of a relational schema for a Pascal-like language, a program for taking software stored as text and translating it into the database representation, and a simple, pointing-oriented user interface. Initial performance measurements indicate that response is too slow in our current environment, but that advances in database software technology and hardware should make response fast enough in the near future.

1. Introduction

As software systems evolve, they continually grow larger as a result of added functionality, improved reliability, and enhanced performance. The 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. A major part of understanding is simply seeing the information relevant to what one is trying to understand.

The OMEGA programming system [Linton 83] provides mechanisms for seeing and manipulating software in a much more powerful and general way than current systems. Instead of a linear view, such as presented by UNIX† [Kernighan and Mashey 81], or a hierarchical view, such as presented by Gandalf [Habermann, et al. 82], OMEGA provides multiple *relational* views of the information in a program.

The relational model provides very powerful operations for describing portions of a database of information. OMEGA gives programmers the opportunity to view and change a wide variety of cross-sections of a software system.

We have begun building a prototype implementation of OMEGA using the relational database system INGRES [Stonebraker, Wong, and Kreps 76]. So far, we have implemented a program to extract and store the information in traditional program text into an INGRES database, and a simple pointing-oriented user interface for browsing programs in the database.

In the remainder of this paper, we describe the general views of software that OMEGA provides the user (including traditional views), describe how our prototype implementation interfaces to INGRES to support these views, briefly describe the user interface, and present preliminary measurements of the performance of this prototype.

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.

†UNIX is a registered trademark of Bell Laboratories.

2. Views of Programs

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. For example, to find all uses of the "+" operator where both operands represent real numbers requires parsing and semantic analysis of the entire program.

In a text-oriented system programmers must translate update operations on program objects such as statements, expressions, and variables into operations on text objects such as lines, words, and characters. This translation can be complicated; for example, changing the name of a variable requires string substitution every place that 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. For example, when porting some software to a new machine, a programmer might wish to look at all the constants defined throughout the software. However, the constant definitions are likely to be spread throughout the program hierarchy, leaving no convenient way to view them together. In general, a system that provides programmers only one organization of programs cannot satisfy the variety of activities that make up software development and maintenance.

Programmers are not interested in the complete structure of a large software system; at any given time they need to see some cross-section that contains the objects relevant to a particular task. The data management term for such a cross-section is a *view*.

Text and tree organizations are two common views of a software database. Other examples of views include the following:

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

Views of programs capture many separate facilities in current programming environments, including structure-oriented editing, module dependency analysis, cross-reference listings, call graph generation, version history manipulation, and execution trace analysis. In addition, there are many other possible program views corresponding to particular information in which a programmer might be interested. For example, one might wish to see a view containing the uses of an I/O procedure involving a particular file.

To provide a powerful, general mechanism for describing views, OMEGA uses a general-purpose relational database system to manage the procedures, statements, variables, and the other information that makes up a program. By using this approach, we do not constrain the ways in which a programmer can view software, and avoid duplicating the functions 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. In addition, database researchers are also looking at the problem of managing physically distributed data, which is rapidly becoming an important problem in managing the development and distribution of software.

We chose to use the relational data model [Codd 71] because it provides powerful facilities for defining and manipulating general views of data, and because we had a relational database system readily available. A more complete and general description of the use of a relational database system by a programming environment is described in [Powell and Linton 83a], including integration of runtime information. We focus here on our prototype implementation.

3. Overview of Prototype Implementation

To experiment with our ideas, we needed an apparatus with which we could quickly build parts of OMEGA. The underlying environment that we had available was UNIX running on a VAX†-11/780. We chose INGRES as our database system because it was both convenient (it runs on UNIX), and familiar (we had used it before).

The intent of OMEGA is to manage large software systems; we therefore wanted to be able to use OMEGA on a relatively large system.

†VAX is a registered trademark of Digital Equipment Corporation.

This desire meant that we needed a way to quickly enter some existing software into the database.

The requirement of an existing body of software eliminated our first choice for a language to support, namely Ada [Ada 82]. We also decided against the popular languages C and Pascal. The textual macro facilities available in C makes 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 language concepts such as data abstraction and generic types. 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.

Given these tools, our implementation consists of *parse*, a program that takes Model source and enters it into an INGRES database, and *peruse*, a program for viewing the software stored in the database. Figure 1 shows how the various pieces of our implementation fit together.

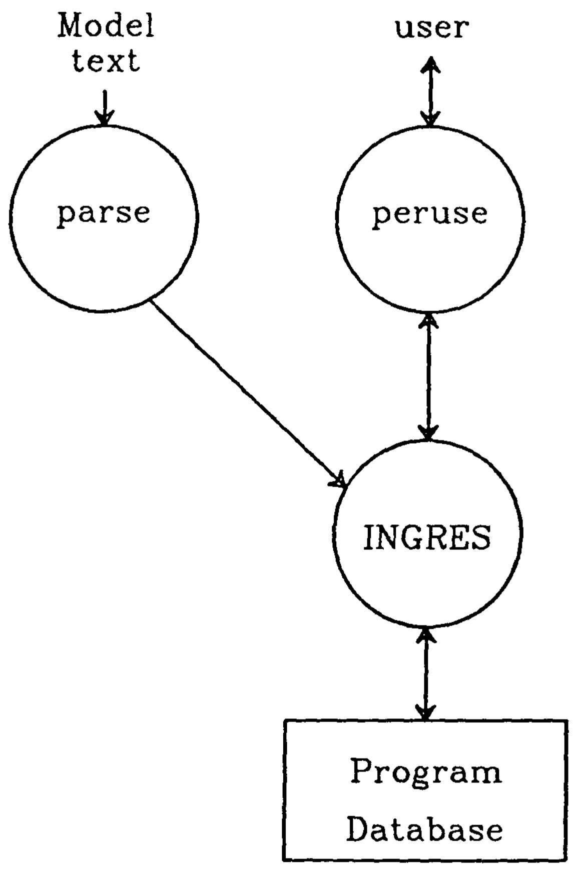

Fig. 1: Implementation overview.

4. Interfacing to INGRES

To use INGRES, we first had to decide how to organize program information into a relational schema. We used the following ideas to guide our design:

- Represent classes of program objects (e.g., procedures, variables, statements) by relations, and individual objects by tuples.
- 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 zero 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 that can be a function call, subscript operator, or constant.
- Represent information once; use views to represent different kinds of references to objects, e.g. variables 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 that they contain. Tables 1(a) and 1(b) list the various 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.

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 1(a): Relations for symbol table information.

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 (described in the next section). The *viewof* relation associates views defined on the database with their underlying relations, and was only necessary for our experiments with directly accessing information from the database.

The views that are predefined 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 name 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. Using the INGRES query language QUEL, this view is defined by the following statements:

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

There are also views for uses of procedures, functions, parameters, constants, and types.

The definition of an object, such as a module, is 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-spec (
    uid = p.uid, name = p.name,
    proc-class = p.proc-class,
    paramlists = p.paramlists
)
```

Using views keeps the information in a single

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 1(b): Relations for statements and expressions.

place, while allowing either the definitions or implementation of the object to be displayed.

5. User Interface

The purpose of implementing *peruse* was as much to see that the information was really in the database as to experiment with the OMEGA user interface described in [Powell and Linton 83b]. Our implementation focused on three areas: how to display objects in the database, the management of the screen, and the command interface.

5.1 Pictographs

A *pictograph* is the view of an object that is displayed on the screen. Each relation or view has a pictograph associated with it that specifies how a tuple belonging to the relation or view should be printed.

By design, a pictograph consists of graphical and textual images arranged in a two-dimensional area, where parts of it represent slots where objects can be placed to designate values for attributes of other objects. Since we did not have a terminal with graphical capabilities, we represented pictographs as format strings containing meta-characters to indicate where and how 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. 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 respectively.

The "r" in the *var-use* example also indicates that the name of the relation to which the field refers is the same as the name of the field. 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 of the *var-use* view is "pictographs".

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 2 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.

5.3 Input Commands

Commands are entered by a single keystroke, and specify an operation on the current object in the current scene. Table 2 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
s	select
S	pick up
w	move cursor forward
b	move cursor backward
e	zoom in
v	show slots
c	create
f	fill in
↑F	scroll forward
↑B	scroll backward
↑R	rotate left
↑G	rotate right
↑D	redraw screen

Table 2: *Peruse* commands.

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

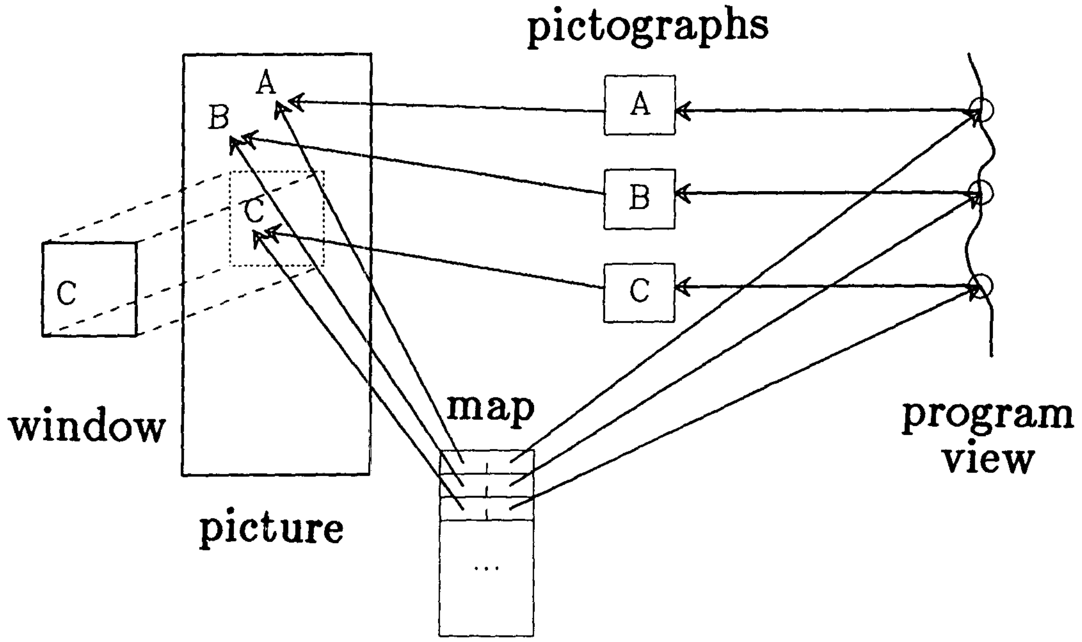

Fig. 2: Structure of a scene.

The 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 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., integer or string) entered by the user.

5.4 Current Status

Peruse provides some of the basic capabilities of the OMEGA user interface, but is not yet complete. It provides fixed size windows in a fixed location, rather than dynamic allocation and reshaping of windows without users having to specify a size or location.

It can create and zoom in on objects, but does not yet support general queries and global relational updates. To implement these operations requires that query objects be definable and executable, where executing a query involves translating it into QUEL and sending it to INGRES.

6. Performance Measurements

Using a relational database system offers substantial power, but an equally important aspect of a software development system is the speed with which it can respond to programmer requests. In addition to execution time, we wanted to make sure that using a database did not require an unreasonably large amount of disk storage.

In this section we present time and space performance measurements for a simple use of *peruse*. These measurements in no way represent a complete analysis; to do so would require a complete implementation of OMEGA and a collection of large software databases on which to gather measurements. Nonetheless, our initial experience has been helpful in determining problem areas.

6.1 Response Time

Our initial implementation of *peruse* was very slow in displaying the body of a procedure. The poor response time was due to each object being retrieved 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. The most frequent queries sent by OMEGA occur while traversing references, and are of the form

> **range of** t **is** *some-relation*
> **retrieve** (t.all) **where** t.uid = *some-uid*

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 UID 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 and thereby avoid the overhead associated with query processing. Since INGRES runs as a process separate from *peruse*, this also avoided 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.

Tables 3(a) and 3(b) show the performance of *peruse* retrieving and displaying the body of

Configuration	# 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

Table 3(a): Response time for viewing body of a 5 line program.

Configuration	# tuples	# pages read	CPU time (seconds)	Elapsed time (seconds)
standard	3746	33847	7477.5	12094
compiled	3746	14996	564.1	957
buffered	3746	6085	300.0	446

Table 3(b): Response time for viewing body of DEMOS.

both a simple, 5 line program, and the main program body of the DEMOS kernel (about 1,000 lines). We measured configurations of *peruse* using standard queries, hand-compiled queries, and hand-compiled queries with buffering.

The CPU time above includes both user and system time as measured under UNIX. All measurements were made on a lightly loaded VAX-11/750.

The time it takes to retrieve the main program body of DEMOS is a good benchmark, but does not reflect actual response time because the main body of DEMOS is so large. Ideally, *peruse* would stop retrieving tuples when the results will no longer fit on the screen, then continue retrieving and filling its data structure on demand. Currently, it is filled of all at once before displaying any results.

Compiling queries has a dramatic effect on performance, reducing CPU time by more than a factor of six. Buffering has a more modest effect, and as might be expected is more pronounced for the larger benchmark. These results indicate that a production implementation of OMEGA requires a database system that can compile queries, and that some buffering capability would also help performance.

6.2 Storage Requirements

Table 4 shows the number of tuples and total size of the largest relations in the database.

The total size is close to that of the corresponding text, but this is somewhat misleading because the database storage 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. Even with conservatively high estimates for the space needed for comments and indices, storing programs in a database does not appear to require substantially more storage than the corresponding program text.

relation	# tuples	width (bytes)	total size (bytes)
strings	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
nameof	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

Table 4: Space usage in database for DEMOS kernel.

7. Conclusions

To experiment with the implementation of support for the relational views that we designed into OMEGA, we have built *parse*, a program that takes source text and stores all the information in the program into a database managed by INGRES, and *peruse*, a program that displays information from the database onto the screen.

Although there are clearly performance problems using INGRES, the functionality it provides has allowed us to not worry about managing permanent storage or processing queries. The ability to define general views has been particularly useful.

Our initial measurements of performance show that compiled queries and buffering improve performance significantly. In general, the database system should be able to use main memory and more semantic information about the data to provide substantially better performance than is currently available.

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 a general-purpose system. Similarly, it is always possible to hand-code assembly language that executes faster than the code generated by a compiler. However, just as it is cost-effective to write in a high-level language and use a compiler, it will be worth using the general-purpose, better supported, and more reliable system. Database systems are approaching this threshold of being cost-effective for use in managing program information.

8. Acknowledgments

The ideas in OMEGA are the result of many discussions with Mike Powell; both he and Larry Rowe provided helpful comments on portions of this paper. The referees' comments were also helpful in improving the paper's focus and presentation.

9. References

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

[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.

[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.

[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.

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

[Linton 83]
Linton, M., "Queries and Views of Programs Using a Relational Database System", Report No. UCB/CSD 83/164, Computer Science Division, University of California, Berkeley, California, December 1983.

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

[Powell and Linton 83a]
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 83b]
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.

[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.