Proceedings

IEEE Third Workshop on
Program Comprehension

wpc '94

Proceedings

IEEE Third Workshop on
Program Comprehension

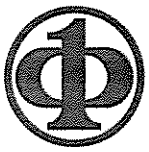
November 14 - 15, 1994

Washington, D.C.

Sponsored by

IEEE Computer Society

Technical Council on Software Engineering



IEEE Computer Society Press

Los Alamitos, California

Washington • Brussels • Tokyo

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society Press Order Number 5647-02
Library of Congress Number 93-81245
IEEE Catalog Number 94TH06767
ISBN 0-8186-5645-X (paper)
ISBN 0-8186-5646-8 (microfiche)
ISBN 0-8186-5647-6 (case)

Additional copies may be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264
Tel: +1-714-821-8380
Fax: +1-714-821-4641
Email: cs.books@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: +1-908-981-1393
Fax: +1-908-981-9667

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM
Tel: +32-2-770-2198
Fax: +32-2-770-8505

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama
Minato-ku, Tokyo 107
JAPAN
Tel: +81-3-3408-3118
Fax: +81-3-3408-3553

Editorial production by Regina Spencer Sipple
Cover design by Joseph Daigle, Schenk/Daigle Studios
Printed in the United States of America by Braun-Brumfield, Inc.

The Institute of Electrical and Electronics Engineers, Inc.

Table of Contents

Message from the General Chair

Welcome to the Third Workshop on Program Comprehension!

This is the third workshop in a series that began in 1992 in Orlando, Florida. The second workshop was held in 1993 in Capri, Italy, and the success of that conference led to the planning and organization of this third workshop in Washington, D.C., in 1994. These meetings have attracted leading researchers and practitioners in the the field of program comprehension from around the world and we are pleased to recognize the international appeal and focus of this particular workshop.

The field of program comprehension is often called by the synonym, "software understanding." It deals with the often neglected human side of software engineering, and it is there that the organizers believe many software engineering problems are rooted — and where they have to be resolved. Program comprehension has a direct impact on a programmer's productivity and capability and therefore, presents a significant and necessary area of research. Kristen Nygaard put it in a very succinct way: "To program is to understand!"

I would like to thank all of the volunteers who have made the Third Workshop on Program Comprehension a reality. I would also like to thank the professional staff members for their concerted efforts.

I wish all of the participants a pleasant stay in the Washington, D.C. area and I hope that they glean many new and stimulating ideas from the 1994 Workshop on Program Comprehension.

Vaclav Rajlich
Wayne State University

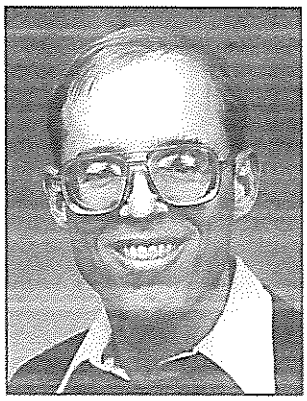
Message from the Program Co-Chairs

Welcome to the Third Workshop on Program Comprehension. As the role of program comprehension has come to be recognized as a substantial part of software engineering, so too has this workshop evolved. Not only is it fully refereed, but there has also been a conscious attempt to broaden the scope of the workshop — both in terms of the topics addressed and the participants involved.

The workshop includes a keynote address, seven paper sessions, and an open discussion of the future of program comprehension. The sessions range from experience reports to exploratory trends, from architecture to databases, from object orientation to parallelization, and from the role of people in the current process to automated analysis.

This workshop would not be possible without the help of many people. We wish to thank the authors for their efforts and timeliness, the program committee and the other reviewers, and the IEEE Computer Society — particularly Regina Spencer Sipple — for her assistance in putting the proceedings together. Last but not least, we wish to thank Vaclav, without whom there would be no Workshop on Program Comprehension at all.

Once again, welcome to WPC'94. We hope you gain new insights, ideas, and contacts from this truly international workshop.



Spencer Rugaber
Georgia Institute of Technology

Aniello Cimitile
University of Naples

WPC '94 Committees

General Chair

Vaclav Rajlich
Wayne State University, USA

Program Co-Chairs

Aniello Cimitile
University of Naples, Italy

Spencer Rugaber
Georgia Institute of Technology, USA

Program Committee

Aniello Cimitile, *University of Naples, Italy (Co-Chair)*
Spencer Rugaber, *Georgia Institute of Technology, USA (Co-Chair)*
Paolo Benedusi, *CRIAI, Italy*
Ted Biggerstaff, *Microsoft Research, USA*
William Chu, *Chia University of Taiwan, ROC*
Ugo De Carlini, *University of Naples, Italy*
Prem Devanbu, *AT&T Bell Laboratories, USA*
Philippe Facon, *IIE-CNAM, France*
John Foster, *British Telecom, UK*
Lewis Johnson, *Information Sciences Institute, USA*
Paul Layzell, *University of Bari, Italy*
Panos E. Livadas, *University of Florida, USA*
Ettore Merlo, *Ecole Polytechnique of Montréal, Canada*
Glenn Racine, *Army Research Laboratory, USA*
Steve Reiss, *Brown University, USA*
Harry Sneed, *SES, Germany*
Larry Van Sickle, *Great Hill Technology Group, USA*
Giuseppe Visaggio, *University of Bari, Italy*
Anneliese von Mayrhauser, *Colorado State University, USA*
Norman Wilde, *University of West Florida, USA*
Linda Wills, *Georgia Institute of Technology, USA*
Horst Zuse, *Technische Universitat Berlin, Germany*

Keynote Address: Maintenance Environments: Tools for People or People for Tools?

Anneliesa von Mayrhauser

Session A:
Objects

A Greedy Approach to Object Identification in Imperative Code

B.L. Achee & Doris L. Carver

Louisiana State University

Abstract

The benefits of such recent innovations as object-oriented programming are not realized in most systems currently in use because they are, on average, over 10 years old. Additionally, they suffer from extensive maintenance. This paper addresses these concerns in the context of reverse engineering. It discusses the development of a method to identify objects in imperative code, specifically FORTRAN-77. An algorithm that uses a greedy approach to object extraction is presented. The imperative code is analyzed at the subroutine level and, using the concepts of graph theory, a set of objects is generated.

1. Introduction

Much of the scientific and commercial software in use today is an average of 10 to 15 years old. Most of these systems have undergone extensive maintenance and now suffer from poor structure and documentation. Moreover, the age of these systems prevents them from taking advantage of the benefits of such recent software innovations as object-oriented programming. [8]

The benefits of object-oriented programming include code reuse, modularity, deferred commitment, and a model that closely resembles the real world. In the object-oriented paradigm, the object is the primitive element. The object can be viewed as an abstract data type, encapsulating a set of data (i.e. attributes) and a corresponding set of permissible actions on the data (i.e. methods). Each object is an autonomous entity and interacts with other objects during the execution of the system. [9]

Reverse-engineering involves analyzing a system to "identify the system's components and their interrelationships" and to "create a representation of the system in another form or at a higher level of abstraction." [3] Therefore, by using techniques of reverse-engineering to achieve an object-oriented design, the benefits of current software technologies can be realized without discarding a working system.

As the connotations of reverse-engineering have changed from negative to necessary, there has been a concentration of research in the area. No longer is reverse-engineering clouded by the idea that it is an admission of failure because of the "get it right the first time" mentality. Today it is realized and widely accepted that a software system is dynamic. "It is not possible to predict what you will want the system to do five, or even two, years from now." [11]

The widespread appeal of object-oriented programming and the realization of the necessity of reverse-engineering as a software maintenance activity have motivated research on re-engineering procedural code to object-oriented code. The concept of the object module as a means of restructuring FORTRAN code into an object-oriented style is introduced in [12]. While code structured with object modules is not truly object-oriented, it marks the beginning of the research along that path. A method for identifying candidate objects based on routines which use global variables and the types of formal parameters and return values is discussed in [5]. The problem of object identification is approached by first developing a formal specification of the code and then proceeding to identify objects from the formal specification in [4]. Object identification based on hints from the user and information in common blocks is discussed in [8]. "A method for becoming familiar with a procedural system so that it can be converted into the object paradigm" is discussed in [10] but the issues of object identification are not addressed. The results of the RE^2 project, which includes criterion for identifying abstract data types in existing software systems are discussed in [1] and [2]. The development of the Ghinsu tool, which performs dependence analysis on a subset of ANSI C or Pascal programs is discussed in [6].

0-8186-5647-6/94 $04.00 © 1994 IEEE

This paper presents an approach to extract objects from FORTRAN-77 code. An algorithm for the definition of the objects is given. The input to the algorithm is structured FORTRAN-77 code and the output is a set of candidate objects. Throughout the paper, a simple example is used to demonstrate the algorithm.

2. Object Identification Algorithm

This method to identify objects analyses the program at the subprogram level and uses a data-driven, bottom-up approach to construct objects. An object, O, is defined as a two-tuple, (D,M) where D is the set of data items and M is the set of methods that act on those data items.

In the process of developing the algorithm, a choice was made between using a top-down approach or bottom-up approach to object construction. A top-down approach would begin with all data elements being contained in a single object and proceed to divide the object into smaller objects. Although this approach has some merit, the margin of error would be on the side of objects that are too small. The bottom-up approach constructs objects by determining the cohesive strength between two data items. We chose the bottom-up approach.

The algorithm, given in Figure 2.2, analyses a program to extract objects in a bottom-up manner. The functionality of a program is viewed at three levels. At the top-level, the functionality is that of the entire program. This view is, indeed, too coarse grained to aid directly in object extraction, however, it may provide some insight into the type of objects that may appear. The second level concentrates on the functionality of the individual subroutines. The third, and most fine-grained view, considers the functionality of each line of code. The view of a subroutine as the unit of functionality is the approach of the algorithm. By considering each subroutine as a unit of functionality, the actual parameters are then necessary to perform the function of the given subroutine. Based on these guidelines, the algorithm seeks to obtain the smallest set of parameters needed to obtain the strongest cohesive unit. The cohesive strength of a pair of parameters is measured by determining the frequency in which they are both necessary to perform various functions (where the unit of functionality is the subroutine). Measuring the cohesion of a pair of parameters for a given function results in the consideration of three cases: (i) both parameters are necessary , (ii) only one parameter is necessary, and (iii) neither parameter is necessary. The value of cohesion of a given pair of parameters is affected by each case as follows: case (i) increases the value, case (ii) decreases the value, and case (iii) leaves the value unchanged. The greedy approach taken by the algorithm results in a costing function that, for a given pair of parameters, weights the necessity of both parameters of the pair as a stronger condition than the necessity of only one parameter of the pair. Thus, the cost function for a pair of parameters i and j with respect to subroutine *f* is as follows:

c(i, j) = c(i,j) iff neither i nor j is necessary for the execution of *f*
= c(i,j) + .2 iff both i and j are necessary for the execution of *f*
= c(i,j) - .1 iff either i or j (not both) is necessary for the execution of *f*

A bottom-up approach is used to construct a graph that maintains the frequency of their common occurrence. This graph is represented as a weighted adjacency matrix M, where M[i,j] is assigned a real number based on the result of a costing function, c(i,j). Thus, data that appear functionally related only a small portion of the time are given a lower weight than those that appear functionally related a majority of the time. By setting a threshold on the weight necessary to be considered relevant, the set of data contained in an object is determined. One benefit of this representation is that it facilitates the consideration of various sets of objects based upon varying the threshold.

Let M be the set of all sets of methods where
$M = M_1 \cup M_2 \cup ... \cup M_i$
Let A be the set of all sets attributes, where
$D = D_1 \cup D_2 \cup ... \cup D_i$
Let O be the set of all objects, where
$O = O_1 \cup O_2 \cup ... \cup O_i$
and $O_j = (D_j, M_j)$ for each $j = 1 .. i$

If method m_k changes the value of data attribute d_i and $d_i \in D_m$ then $M_m = M_m \cup m_k$.

Figure 2.1
Method Priority Rule

Objects are determined by first grouping the data together to form attribute sets and then affixing methods to the sets of attributes. Thus, once the data elements have been partitioned to represent the attribute sets, the next step is to determine the corresponding methods. The process of determining

the methods involves two steps: defining the methods and assigning the methods to the appropriate attribute set. When defining the methods, it is important to realize that the attribute sets are comprised of the actual parameters of the original FORTRAN source code. Thus, the procedure for method definition evaluates the subroutine code exclusively. The methods are attached to attribute sets using the following heuristic to define priority: priority of attachment is to that set of attributes whose state is changed by the method in consideration (Figure 2.1).

The algorithm for identifying the objects is given in Figure 2.2. A simple program, given in Figure 2.3, is used throughout the paper to demonstrate the application of the algorithm. The program, STATISTICS, inputs two arrays of experimental data and two integers representing the number of data elements in each array. The function STD is used to compute the standard deviation of the data set. Finally the results are output using the subroutine PRINT. Following is the trace of the example program on the algorithm given in Figure 2.2:

P is the program STATISTICS. It has n = 4 subroutine calls and m = 6 distinct actual parameters and function resultants. Following step (i) we create the following sets of actual parameters and function resultants based on the subroutine calls in P:

$CALL_1$ = { m, expa, n, expb}
/* using CALL INPUT(m, expa, n, expb) */
$CALL_2$ = { expa, m, stda}
/* using stda = STD(expa, m) */
$CALL_3$ = {expb, n, stdb}
/* using stdb = STD(expb, n) */
$CALL_4$ = {expa, m, stda, expb, n, stdb}
/* using CALL PRINT(expa, m, stda, expb, n, stdb) */

Proceeding to step (ii) we initialize graph, G, by forming the sets V and E as follows:

V = { m, expa, n, expb, stda, stdb}
E = { }

Realizing that there is an arbitrary ordering imposed on the elements of V, we refer to the elements of V as v_1 = m, v_2 = expa, v_3 = n, v_4 = expb, v_5 = stda, and v_6 = stdb.

In step(iii) the weighted adjacency matrix is initialized. Since the algorithm is using a constructive approach to creating the graph, the matrix is initially the zero matrix representing a graph with no edges.

The matrix AP is a boolean matrix that is used to represent the appearance of actual parameters and function resultants in a subroutine invocation statement. Thus, AP[i,j] is set to 1 iff v_j is used in subroutine invocation statement i. This is determined by considering $CALL_i$. If v_j (as designated by V) is an element of $CALL_i$ then AP[i,j] is set to 1, otherwise it is set to 0. The matrix AP corresponding to program P is as follows:

AP:

j / i	1 m	2 expa	3 n	4 expb	5 stda	6 stdb
1	1	1	1	1	0	0
2	1	1	0	0	1	0
3	0	0	1	1	0	1
4	1	1	1	1	1	1

Once the matrices M and AP have been initialized the analysis begins. Following is a partial trace of the analysis using the first and last values of i.

i = 1:

k / j	1 m	2 expa	3 n	4 expb	5 stda	6 stdb
1 m	0	0.2	0.2	0.2	-0.1	-0.1
2 expa	0	0	0.2	0.2	-0.1	-0.1
3 n	0	0	0	0.2	-0.1	-0.1
4 expb	0	0	0	0	-0.1	-0.1
5 stda	0	0	0	0	0	0
6 stdb	0	0	0	0	0	0

i = 4:

k / j	1 m	2 expa	3 n	4 expb	5 stda	6 stdb
1 m	0	0.6	0.2	0.2	0.3	-0.1
2 expa	0	0	0.2	0.2	0.3	-0.1
3 n	0	0	0	0.6	-0.1	0.3
4 expb	0	0	0	0	-0.1	0.3
5 stda	0	0	0	0	0	0
6 stdb	0	0	0	0	0	0

Let **P** be a structured FORTRAN program
with **n** subroutine calls
and **m** (distinct) actual parameters and function resultants.

(i) For i = 1 to n do
    $CALL_i$ = set of actual parameters of subroutine call i
    if subroutine i is a function
    then $CALL_i$ = $CALL_i$ union with the function resultant of subroutine i
od

(ii) Let G = (V,E) be a graph
    V = the arbitrarily ordered set of actual parameters and function resultants,
        and denote the elements of V as $v_1, v_2, \ldots, v_m$ /*note |V| = m */
    E = { } /* initially E is empty */

(iii) M[1..m, 1..m] ARRAY of REAL; /* a weighted adjcency matrix */
AP[1..n, 1..m] ARRAY of BOOLEAN; /* n sets of actual parameters */
/* Construct the graph, represented as an weighted adjcency matrix */
/* Initially G consists of only a set of vertices with no edges */
For i = 1 to m do
    For j = 1 to m do
        M[i,j] = 0
    od
od
/* Initialize the sets of actual parameters; AP[i,j] = 1 iff v_j is an element of $CALL_i$ */
For i = 1 to n do
    For j = 1 to m do
        If v_j is an element of $CALL_i$
        then AP[i,j] = 1
        else AP[i,j] = 0
    od
od
/* Perform the analysis on the sets AP */
For i = 1 to n do
    For j = 1 to m do
        For k = j+1 to m do
            if AP[i,j] = 1 and AP[i,k] =1
            then M[j,k] = M[j,k] + .2
            else if AP[i,j] = 0 and AP[j,k] = 0 /* inconclusive */
                then skip
                else M[j,k] = M[j,k] - .1 /* only one is 0 */
        od
    od
od

(iv) The output is r < n connected graphs. The vertices of each connected graph represents the attributes for a distinct candidate object.

Figure 2.2
Greedy Algorithm for Object Identification

```
      Program STATISTICS
          INTEGER m,n
          REAL expa(14), expb(14), stda, stdb
          CALL INPUT(m, expa, n, expb)
          stda = STD(expa, m)
          stdb = STD(expb, n)
          CALL  PRINT(expa, m, stda, expb, n, stdb)
      END

      SUBROUTINE INPUT(m, expa, n, expb)
          INTEGER m, n
          REAL expa(14), expb(14)
          READ (5, 10, end = 15) m, (expa(i), i = 1, 14)
15        READ (5, 10, end = 30) n, (expb(j), j = 1, 14)
10        FORMAT /* excluded */
30    END

      FUNCTION STD(expx, x)
          INTEGER x
          REAL  mean, expx(14), ind(14), tot
          tot = 0.0
          sum = 0.0
          DO 60   i = 1, x
          tot = tot + expx(i)
60        CONTINUE
          mean = tot /x
          DO 70   j = 1, x
          ind(j) = mean - exp(j)
          sum = sum + ind(j) ** 2
70        CONTINUE
          STD = SQRT(sum/(x-1))
      END

      SUBROUTINE PRINT(expa, m, stda, expb, n, stdb)
          REAL   stda, stdb, expa(14), expb(14)
          WRITE(6,80) ' Experiment A    ', ' Measurements', ((expa(i)), i = 1, m)
80        FORMAT /* excluded */
          WRITE(3, 90) 'Standard Deviation' , stda
90        FORMAT /* excluded */
          WRITE(6, 100) ' Experiment B  ', 'Measurements', ((expa(j), j = 1, n)
100       FORMAT /* excluded */
          WRITE(6, 90) 'Standard Deviation', stdb
          RETURN
      END
```

Figure 2.3
Sample Program

At this point consideration must be given to the appropriate threshold value. If the threshold is set too low, the data sets of the candidate objects will be too big; however, if it is set too high, the data sets will be too small. Consider the following possible values for the threshold: 0, 0.2, and 0.3 (Figure 2.4).

Threshold Value	Data Sets
0	{m, expa, n,expb, stda, stdb}
0.2	{m, expa, stda} {n, expb, stdb}
0.3	{m, expa} {n, expb} {stda} {stdb}

Figure 2.4
Data Sets based on Threshold Values

As expected, the value of the threshold is inversely proportional to the size of the data sets. The chart above can be divided into three categories, based on the following two important characteristics of the data sets: (i) all data sets below a given threshold consist of a single data set containing all of the parameters, and (ii) all data sets above a given threshold contain at least one singleton, i.e. a data set of cardinality one. In Figure 2.4, all data sets below the threshold of 0.2 contain all parameters, and all data sets above 0.2 contain at least one singleton. Thus, the threshold value classes are as follows: class one is the set of threshold values { 0 }; class two is the set of threshold values { 0.2 }; and class three is the set of threshold values { 0.3 }. Once such a division is obtained, the threshold value 0.2 is chosen for determining a desired data set.

The choice of the threshold value as 0.2 gives two objects. Object one has D = {m, expa, stda} and object two has D = {n, expb, stdb}.

3. Method Definition and Assignment

After D is defined for each object, the next step is to define the methods, M, for each object. The methods for these high level objects consist of state changes to the data sets of the objects. The methods are determined using both the invocation statements and the bodies of the subroutines. The invocation statements are used to provide the proper mapping of formal parameters to actual parameters while the bodies of the subroutines are considered line-by-line to define the actual methods.

Rudimentary methods are identified using only the assignment and I/O statements of the subroutines. During this line-by-line analysis of the subroutine, an assignment statement is classified as one of the following types: incrementing, decrementing, computing or re-defining. An incrementing assignment statement takes the form X = X + c, where c is some constant and X is any variable. Such an assignment statement results in the method Increment_X(X). Similarly, a decrementing assignment statement takes the form X = X - c, and results in the method Decrement_X(X). When X appears only on the left hand side, as in X = EXPR, where EXPR is any valid expression but not a function invocation, the resulting method is Compute_X(var_list) where var_list is a list of all variables in EXPR. Finally, in all other cases where X appears on both the left hand side and right hand side of the assignment statement, the resulting method is Redefine_X(var_list) where var_list is a list of all variables on the right hand side of the assignment statement (including X). The I/O statements appear as methods virtually as is. A Read statement results in the method Read(u) var_list, where u is the unit number and var_list is the list of input variables appearing in the statement. Similarly, a Write or Print statement results in the method Write(u) var_list. The unit number is retained to provide the maximum amount of information about the original code. A line-by-line review of the subroutines generates a set of methods from each subroutine. These methods are referred to as "formal" methods because they are generated using the formal parameters. Once each subroutine has produced a set of "formal" methods, the invocation statements are used to provide the proper mapping of formal parameters to actual parameters resulting in "actual" methods. Every invocation statement is used to generate a set of "actual" methods. It is these "actual" methods that are assigned to the objects. The "actual" methods for an invocation statement are generated by substituting each formal parameter in the "formal" methods with the appropriate "actual" parameter (as determined by the invocation statement).

Using the example program, Figure 3.1 shows each subroutine and the corresponding "formal" methods. The subroutine Input generates two "formal" methods; Std generates five "formal" methods and Print generates four "formal" methods. Figure 3.2 shows each invocation statement and the

corresponding "actual" methods. There are a total of sixteen "actual" methods identified.

Subroutine	"Formal" Methods
Input(m, expa , n, expb)	Read(5) m, expa
	Read(5) n, expb
Std(expx, x)	Redefine_tot (tot, expx)
	Compute_mean(tot, x)
	Compute_Ind(mean, expx)
	Redefine_sum(sum, ind)
	Compute_std(sum, x)
Print(expa, m, stda, expb, n, stdb)	Write(6) expa
	Write(6) stda
	Write(6) expb
	Write(6) stdb

Figure 3.1
"Formal" Methods Generated from Subroutines

Invocation Statement	"Actual" Methods
Call Input(m,expa,n,expb	Read(5) m, expa
	Read(5) n, expb
Stda = Std(expa, n)	Redefine_tot (tot, expa)
	Compute_mean(tot, n)
	Compute_Ind(mean, expa)
	Redefine_sum(sum, ind)
	Compute_std(sum, n)
Stdb = Std(expb, m)	Redefine_tot (tot, expb)
	Compute_mean(tot, m)
	Compute_Ind(mean, expb)
	Redefine_sum(sum, ind)
	Compute_std(sum, m)
Call Print(expa, m, stda, expb, n, stdb)	
	Write(6) expa
	Write(6) stda
	Write(6) expb
	Write(6) stdb

Figure 3.2
"Actual" Methods Generated from Invocations

Once the "actual" methods are identified, the final step is attaching the methods to the data sets to form objects. Each method is attached to a data set where the priority of attachment is given to that object whose state is changed by the method in consideration (Figure 2.1). At this stage, human intervention is necessary to attach a meaningful name to each object.

The objects identified using the example are given in Figure 3.3. As expected, the two objects identified for the example are very similar. They have data sets of equal cardinality and type signature, as well as method sets of equal cardinality. Moreover, the methods are virtually identical, varying only in the variable lists. Such similarities are expected and are necessary for addressing the issue of class abstraction which will be considered in later work.

```
OBJECT 1:   D = {m, expa, stda}
            M = {
                  Read(5) m, expa
                  Redefine_tot(tot, expa)
                  Compute_mean(tot, m)
                  Compute_Ind(mean, expa)
                  Compute_stda(sum, m)
                  Write(6) expa
                  Write(6) stda
                  Redefine_sum(sum, ind)
                }

OBJECT 2:   D = {n, expb, stdb}
            M = {
                  Read(5) n, expb
                  Redefine_tot (tot, expb)
                  Compute_mean(tot, n)
                  Compute_Ind(mean, expb)
                  Compute_stdb(sum, n)
                  Write(6) expb
                  Write(6) stdb
                  Redefine_sum(sum, ind)
                }
```

Figure 3.3
Candidate Objects

4. Conclusions

The algorithm described in this paper evaluates the subroutines of a FORTRAN-77 program to determine a set of objects. The relationships among the actual parameters are evaluated to construct the attribute sets of the objects. By using the invocation statements of the program, a measure of cohesion is recorded in a weighted adjacency matrix. Using this

measure of cohesion, a threshold value is determined and the data is partitioned into disjoint sets each corresponding to an attribute of a distinct object. Methods are then generated by considering, line-by-line, the subroutine code. Finally, the methods are attached to data sets to form objects with an attachment priority based on state change of an object. We are continuing to expand the object identification process along with the method identification process.

5. References

[1] Canfora, G., Cimitile, A., Munro, M., & Tortorella, M. "Experiments in Identifying Reusable Abstract Data Types in Program Code" Proc. IEEE Second Workshop on Program Comprehension, July, 1993, pp. 36 - 45

[2] Canfora, G., Cimitile, A. & Munro, M. "A Reverse Engineering Method for Identifying Reusable Abstract Data Types" Proceedings of the Working Conference on Reverse Engineering, May 1993, pp.73 - 82

[3] Chikofsky, EJ & Cross, JH "Reverse Engineering and Design Recovery: A Taxonomy" In IEEE Software, Jan 1990, pp 13 -17.

[4] Gannod, G.C. & Cheng, B.H.C. "A Two-Phase Approach to Reverse Engineering Using Formal Methods" In Proc Formal Methods in Programming and their Applications Conference, June 1993, pp. 335 - 348.

[5] Liu, S.S. & Wilde, N. "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery" In Proc. Conference of Software Maintenance, 1990, pp 266 -271.

[6] Livadas, P.E., Roy, P.K. "Program Dependence Analysis" Proc. IEEE Conference on Software Maintenance, Orlando, November, 1992, pp. 356 - 265.

[7] Ong, C.L. & Tsai, W.T. "Class and object extraction from imperative code" JOOP Mar/Apr 93, pp 58 - 68.

[8] Osborne, WM & Chikofsky, EJ "Fitting Pieces to the Maintenance Puzzle" IEEE Software, Jan 1990, pp. 11 - 12.

[9] Pokkunuri, B. P. "Object Oriented Programming" In SIGPLAN Notices, Vol 24, No 11, Nov. 1989, pp.96-101.

[10] Sward, R.E. & Steigerwald, R.A. "Issues in Re-Engineering from Procedural to Object-Oriented Code" In Proc 4th Systems Reengineering Technology Workshop, 1994, pp 327 - 333.

[11] Waters, R.C. & Chikofsky, E. "Reverse Engineering: Progress Along Many Dimensions" In CACM Vol 37, No 5, May 1994, pp 22 - 24.

[12] Zimmer, J.A. "Restructuring for Style" In Software - Practice and Experience, Vol 20(4), Apr 1990, pp 365 - 389.

This work is supported in part by the National Science Foundation Grant No. CCR-9307917.

Program Comprehension Through the Identification of Abstract Data Types

A. Cimitile, M. Tortorella
DIS
Dep. of "Informatica e Sistemistica"
University of Naples
Naples 80125, Italy

M. Munro
CSM
Centre for Software Maintenance
University of Durham
Durham DH1 3LE, UK

Abstract

This paper presents the results of experiments carried out on identifying Abstract Data Types in existing code by an improved algorithm described in previous work. It presents a brief description of the improved algorithm and then describes the results of the experiments. It addresses issues in program comprehension from the perspective of software reuse.

Introduction

The comprehension of existing software systems plays a major role in many software engineering projects, encompassing such activities as testing and debugging, validation, migration, maintenance and enhancement, re-engineering, and reuse. Identifying the assets in an existing system requires techniques to decouple the components from the external environment. By doing this and splitting a program into simpler more cohesive modules the system becomes easier to comprehend and maintain.

There are a number of approaches to program comprehension, one of which is a systematic approach where the maintainer examines the whole program and works out the interactions between various modules that constitute the program. This task is made more difficult if the code is not modularized or has been modularized is an ad-hoc manner leading to high coupling between, and low cohesion of, the modules. Without these attributes of low coupling and high cohesion the modules, and thus the program, will be more difficult to understand and therefore more difficult to maintain.

The approach adopted here in this paper is to apply a method of identifying Abstract Data Types (ADTs), these being the basis of new, reusable, modules of the program. The premise of this work is based on the assumption that the identified ADTs will divide the code into modules that will have the necessary attributes of low coupling and high cohesion. The drawback of the approach is that it is not always possible to identify ADTs in the code because for example the original design of the system precluded the use of ADTs, or because the program has been so heavily modified that it is impossible to disentangle any coherent structure from it.

The Identification Process

Some criteria for identifying ADTs in existing code has already been defined in [1, 2, 3, 4]. The approach by Canfora *et. al.* has been applied in a series of experiments [5], where the method is based on the relationship existing between the user-defined data types and the procedure-like components (procedures or functions) that use them as formal parameters and/or as a return type of a function.

The method can be expressed simply by considering the set $STYP$ of couples (c,t) such that c represents a procedure-like component and t represents a user-defined data type used to define a formal parameter and/or a return type and such that c does not use a super-type of t. Given this set the following relations can be defined:

$$ABTYP = (trans(STYP)STYP)^*$$
$$CCTYP = (trans(STYP)STYP)^*trans(STYP)$$

where $trans(R)$ and R^* are the transpose and reflexive transitive closure of the relation R.

The relation $ABTYP$ defines the supporting structure of the ADT that is the user defined data types that contribute to the make-up of the candidate module. The relation $CCTYP$ defines the operators on the candidate ADT, that is the procedure-like components that are included in the candidate module. Software engineering knowledge and heuristics are necessary to eliminate coincidental and spurious connections possibly existing among the components [6].

A new, more precise method, has been proposed by Canfora *et. al.* [9], that extends the work described above. In this extended method the call graph together with the dominance tree is used to give a more precise set of modules and ADT. Before describing the extended method, some basic definitions are given.

The **call directed graph** (CDG) of a program can be described by the triple (s, PP, E), in which s is the main program, PP is the set of procedure-like components and E is defined by the *call relation* on $\{s\} \cup PP$ x PP. If recursion exists in the program, CDG is a cyclic graph and, then, can contain strongly connected subgraphs. By collapsing each subgraph into one node,the **call directed acyclic graph** (CDAG) can be generated [11]. CDAG is defined as the triple (s, PP', E'), where each element in PP' is a compo-

0-8186-5647-6/94 $04.00 © 1994 IEEE

nent of PP or a collapsed subgraph of PP, and E' is the *call relation* defined on $\{s\} \cup PP' \times PP'$. In a CDAG the *dominance relations* are defined: a node px *dominates* a node py if and only if every path from the initial node of the graph to py span px; a node px *directly dominates* a node py if and only if all the nodes that dominate py dominate px; finally, a node px *strongly direct dominates* a node py if and only if px directly dominates py and it is the only node that calls py. The **direct dominance tree** (DDT) is defined as the triple (s, PP', ED), in which ED describes the *direct dominance relations* on $\{s\} \cup PP' \times PP'$. The **strong direct dominance tree** (SDDT) is obtained from the DDT by marking all the edges that connect nodes having the *strongly direct dominance* relation. A method using the call-graph generated from a system and its transformation into a dominance tree has been proposed by Cimitile and Visaggio [7, 8].

The need to use the *dominance relations* to improve the quality of the ADTs identified with the method illustrated above arises from the observation that, in each module, the set of procedure-like components belonging to it is not complete. In fact, the method for looking for ADTs is exclusively based on the reality that some procedure-like components use some user-defined data types as formal parameters and/or as a return type of a function. It does not take into account that there are some procedure-like components, selected to constitute the set of operations of an ADT, that call other procedure-like components that, not referencing to any interesting user-defined data type, have not been selected. Moreover, the set of ADTs identified does not reveal any relation of USE between the various ADTs. In fact, it can happen that the components of an ADT call the components belonging to other ADTs, establishing an USE relation. The *direct* and *strongly direct dominance relations* offer an useful instrument for the completion of the ADTs and for the re-engineering them.

The extended method consists of an algorithm of seven steps and a set of rules and is described in detail by Canfora *et. al.* [9]. The aim of the new algorithm is to identify all and only the procedure-like components involved in the implementation of some of the ADTs. It is an iterative process that discards, in each iteration, the procedure-like components that will not be involved in the implementation of some of the operations of the ADTs. The identification of the components to eliminate comes from the following the observation that a procedure-like component *strongly direct dominated* from the MAIN program is called just from the latter, and it will not be called from other components belonging to some other module. The iterative operations of elimination in the CDGA of these components and re-definition of the SDDT gives the conclusive set of interesting procedure-like components. When the final set of interesting components is obtained, the set of rules establishes the USE relations that can be defined both between components and between modules. The main instrument, on which the rules are based, is the final SDDT composed of only the interesting components identified above and opportunely equipped with additional informations about the relations of *call* for the components that, not being strongly direct dominated, are called from more than one components. On the other hand, the components that are strongly direct dominated are called from the respective dominators. The rules indicate some guidelines that help to identify the procedure-like components not selected in the previous step belonging to some modules, the best way to nest procedure-like components belonging to the same module and how a module USES other modules. For example, if a component is strongly direct dominated and it belongs to the same module of its dominator or to no module, it will be nested in its dominator, otherwise, if it belongs to a different module of its dominator, a USE relation between the two modules is established. Analogously, a component not strongly direct dominated is called from more than one other components if it has not already been established which module it belongs to, it will belong to the same module of the calling components, otherwise, a multiple USE relation between modules is defined.

Results from Experiments

To show the validity of the method proposed above, a set of Pascal program are used in a series of experiments. These programs were analysed by Canfora *et.al.* [5] to test the strength of their criterion for looking for ADTs and, by comparing the actual results with that they obtained, it will be shown how improved solutions are obtained by applying the new process. Moreover, it will be shown how the method adopted facilitates to the splitting of a software system, no matter how complex, into more than one module, that are simpler and easier to analyse and to comprehend.

The analysis system uses a Prolog program dictionary to keep all the main information relative to the subject program. A dictionary is produced by static code analysis and it is composed of facts of arity 1 (*proc(procedure_name)*, *func(function_name)* and *user_def_type(type_name)*), to describe the kind of each software component, and facts of arity 2 (*proc_use_type_in_interface(procedure_name,type_name)*, *func_use_type_in_interface(procedure_name,type_name)*, *used_to_define(type_name_1,type_name_2)*, *proc_func_dec(procedure_name_1,procedure_name_2)*, and *proc_func_call(procedure_name_1,procedure_name_2)*), to describe the relations existing between two software components. By query of the program dictionary, a Prolog program generates the ADTs proposed for the reuse, and another Prolog program generates the dominance relations existing between the procedure-like components. The second program constructs a second Prolog dictionary by storing facts of arity 2, *dir_dom(procedure_name_1,procedure_name_2)* and *str_dir_dom(procedure_name_1,procedure_name_2)*, expressing if a procedure-like component is either *directly* or *strongly direct* dominated and the component dominator.

The data for the experiments consist of four Pascal programs, developed in different periods and by different people. Clearly, for the different expertise of the developers, the programs present different character-

istics, but the results obtained are comparable.

All the programs analysed have a size between 1000 and 2000 LOC. A brief description and the relative analysis for each program follows. For each program the structure of the identified modules are presented. The notation adopted here uses EXPORT as the *interface specification*, that is all the information, the names of the types involved and the operations that act on those types, that the module exports; and uses BODY for the *implementation*, that is the representation of the type, the local resources and the implementation of the operations on the types, that is, all the information that the module hides.

Editor.pas

This is a version of the Unix text editor. The program is augmented with functions that have been written for a particular environment to carry out operations such as opening files and detecting interrupts from the user.

The first results, obtained by applying the method for the extraction of ADTs, gave three modules. The first two modules were well-formed and easy to assign a meaning; on the contrary, the third appeared as a large 'pot pourri' module, composed of five user-defined data types and thirty-seven procedure-like components. This clusterisation was due to the use of an enumeration type. As discussed in [9], sub-range and enumeration types are often the cause of the clusterization of more than one simple module in a bigger one. The isolation or, if possible, the replacement in code of these types with other kinds of types, for example *integer* types, allows to the splitting of the module obtained in the simpler modules. The operation described above and the re-application of the ADTs method split the last module into three simpler modules. Definitively the modules identified are presented in the following table:

N. mod.	user-defined data types	procedure-like components
1	tracestring	etrace
2	lineptr	alloline,freeline,getind,getnew, getpak,gettxt,linkup
3	filenamestring	assignfile,doread,dowrit,getfn,open
4	linestring	addset,ctoi,esc,filset,inject,readcmd, readline,readterm
5	argstring, patternstring	amatch,catsub,dumppat,getccl,getrhs, locate,makpat,maksub,match,omatch, patsiz,stclos,subst

In reality, as cited above, the modules now presented are not complete. In fact, the procedure-like components involved in the constitution of a module, call, in the implantation, other components that do not use the user-defined data types, that are part of the subject module, in their interface, are not included in the same. Moreover, at this point of the analysis, the relation of USE existing between the various modules is not clear. To solve these problems the method based on the *dominance relation* is used.

In Fig.1a the **strongly direct dominance tree** (SDDT) is presented for the program Editor.pas; the solid lines represent relations of strong direct dominance, and the dashed lines the relation of direct, but not strong, dominance; the number in some of the nodes indicates the module that the procedure-like component represented from the subject node belongs to. As described above, Fig.1b shows the SDDT obtained from the CDAG of the program by deleting all the components not belonging to any modules and that are strongly direct dominated from the MAIN. In fact, for the definition of strongly direct dominance the above components will be called only from the MAIN and not from components belonging to some modules. The tree in Fig.1b has been obtained after a recursive process that in each iteration deletes from the CDAG the components with the above peculiarities and all the links in and links out of them, creates hypothetical links in from the MAIN with all the components that are remained without components calling them, and re-constructs the SDDT associated with the new graph. In addition, the informations placed inside the nodes not strongly direct dominated indicate from which module or which component (if this does not belong to some modules) the component mentioned in the node is called.

In the operation of creating the final SDDT, a considerable number of procedure-like components was discarded. Analysis of the code has revealed that they are either components never called, or initialisation components, or components referring to the user-defined data type *StatusRange*, before isolated, and checking the states codified from that type after a number of different operation.

Definitively, it can be said that the software components of the program Editor.ps can be grouped in the ADT *TraceString* with a debugging routine as the only operator; the ADT implements the type *Lines* with the operations to manage two lists of lines, the list of the used lines (the ones that currently contain text) and the list of free lines; the ADT *Files* with all the primitives to manage a file; the ADT *LineStrings*, with the operators for reading a line from either the terminal or a file buffer, modifying a piece of text and inserting an escape character; and the ADT *PatternMatcher* with the operators for searching and substituting strings, and of pattern matching.

The first two module do not present a particular structure, in fact, as can be seen in Fig.1b, they do not use other modules, and do not call components different from that discovered with the method looking for ADTs. The only observation that can be made is on the order of declaration of the components of *module 2*; for example, the component *getnew* has to be declared before, or nested in, the component *alloline*, since the former is strongly dominated from the latter and then is called only from it.

More interesting is the discussion of the other modules. For *module 3* the SDDT reveal, by means the information annotated under some nodes, that it uses *modules 1,2* and *4*. Moreover, *module 3* includes the component *skipbl*, because it is strongly dominated from *doread*. The structure of this module follows:

```
module MOD3              USES MOD1,MOD2,MOD4
type fileNameString;statusRange;                    EXPORT
  function open(fileName:fileNameString;dir:integer):integer;
  function dowrit(fromLine,toLine:integer;
                  fileName:fileNameString):statusRange;
  function doread(line:integer;fileName:fileNameString)
                                          :statusRange;
  function getfn(var fileName:fileNameString):statusRange;
const .....                                         BODY
type fileNameString=Packed array [1..MAXNAME] of char;
     statusRange=(NOSTATUS, OKSTATUS,
          ERRSTATUS, EOFSTATUS,INTSTATUS);
  function open(fileName:fileNameString;dir:integer):integer;
    function assignFile(var FILEX:text;var strname:
          fileNameString; dir:integer):integer;
      begin ..... end
    begin ..... end
  function intrpt(var x : integer) : boolean;
    begin ..... end
  function dowrit(fromLine,toLine:integer;
                  fileName:fileNameString):statusRange;
    begin ..... end
  function doread(line:integer;fileName:fileNameString)
                                          :statusRange;
    function injpak(var sts:statusRange):statusRange;
      begin ..... end
    begin ..... end
  function getfn(var fileName:fileNameString):statusRange;
    procedure skipbl;
      begin ..... end
    begin ..... end
```

Module 4 uses *module 2* as it can be established by observing the component *injpak*. The structure of *module 5*, appearing more complex, follows:

```
module MOD5              USES MOD1,MOD2,MOD4
type argString; patternString;                      EXPORT
  function amatch(lin:argString; from:integer;
                  var pat:patternString):integer;
  function makpat(arg:argString; from:integer;
          delim:char; var pat:patternString):integer;
  function match(lin:argString;var pat:patternString):boolean;
  function subst(sub:patternString;gflag:boolean):integer;
  function getrhs(var sub:patternString;
          var gflag:boolean):integer;
  procedure dumppat(pat:patternString; j:integer);
const .....                                         BODY
type argString = lineString;
     patternString = lineString;
  function amatch(lin:argString;from:integer;
                  var pat:patternString):integer;
    function omatch(lin:lineString;var i:integer;
          pat:patternString; j:integer):boolean;
      function locate(c:char; pat:patternString;
                      offset:integer):boolean;
    function patsiz(pat:patternString;n:integer):integer;
  function makpat(arg:argString; from:integer;
          delim:char; var pat:patternString):integer;
    function getccl(arg:argString; var i:integer;
      var pat patternString; var j:integer):boolean;
    function stclos(var pat:patternString;
          var j,lastj,lastel:integer):integer;
  function match(lin:argString;var pat:patternString):boolean;
  function subst(sub:patternString; gflag:boolean):integer;
    type statusRange=(NOSTATUS, OKSTATUS,
          ERRSTATUS,EOFSTATUS,INTSTATUS);
    function delete(fromLine,toLine:integer):statusRange;
      function previn(line:integer):integer;
    procedure catsub(lin:lineString; f,t:integer;
        sub:patternString; var str:lineString;
                    var k:integer; maxnew:integer);
  function getrhs(var sub:patternString;
                  var gflag:boolean):integer;
    function maksub(arg:argString; f:integer;
          delim:char; var sub:patternString):integer;
  procedure dumppat(pat:patternString; j:integer);
```

Module 5 uses *module 1*, *2* and *4*, includes components like *delete* and *prevl* that, in the structure of the module, will be nested in *subst*, and it contains more complex functionalities, composed of subfunctionalities belonging to the same module, like *amatch* that implements the sub-functionalities implemented by *omatch*, *patsiz* and *locate*, that can be nested in the dominant component.

ExamMarker.pas

The program implements a system for the evaluation of multiple choice examinations. It is particularly suitable for Universities that adopt a college organization like that of the University of Durham. The program inputs the number of questions, with the alternative answers and the exact answers, and the information about the students, with their answers, and produces the resulting marks in various orders.

The analysis of the program with the method described above splits the program into seven modules. A first incomplete picture of these modules follows:

N. mod.	user-defined data types	procedure-like components
1	strings	readstring
2	colleges	newcollege,readcollege, writecollege
3	listelement	alphaprecede,collegeprecede, highermark,swap
4	candidates,lines papers	checkavailablealternatives, checkcandidate,checkextradata, readandcheckanswers, writeparticulars
5	listsizes,markfudge, markschemes, questnos,title	getpreliminaries,readtitle, writetitle
6	lists	listbycollege,listbymark listbyname,listforstudents, permute,quicksort
7	exams	analyse,dumptofile, getparticulars,listsortedresults, mark,summarise,validate

The first five modules were obtained simply by applying the original algorithm looking for ADTs in software code. It also produced the last two modules clusterized in one big module. This clusterisation was due to the use of two subrange types. The operation to isolation of these types and the re-application of the method looking for ADTs gave the final situation as illustrated in the table above.

For the program ExamMarker.pas, in the operation of deletion of components not belonging to modules, only one component, *initialise*, has been deleted. The procedure *initialise* is a component, called from MAIN, executing the necessary initialisation at the start of the program. The final SDDT, obtained after the deletion, is presented in Fig.2

The program ExamMarker.pas can be seen as implemented by seven modules, interacting with each other on the basis of relation of USE. In particular *module 1*, implementing the type *String* and with the only operator *ReadString* to read a string to a maximum length from a file, is used only from the *module 4*. *Module 2*, implementing the type *Colleges* with the operators to write and to read the name of the college as well as the components already discovered, includes

also the components *lower* and *capital*, with the first nested in the operator *readcollege*. The structure of *module 2* is the following:

```
module MOD2
type colleges;                                              EXPORT
  procedure ReadCollege (var college:colleges;
          var unknown:boolean;var source,echofile: text);
  procedure WriteCollege (college:colleges;var outfile:text);
  procedure NewCollege (college:colleges;
                var lastcollege:colleges;var outfile:text);
type colleges = (grey,collingwood,marys,trevs,              BODY
                mildert,aidans,hatfield,chads,johns,
                cuths,castle,hildandbede);
  function capital (ch: char): char;
  procedure ReadCollege (var college:colleges;
          var unknown:boolean;var source,echofile: text);
    function lower (ch:char):char;
  procedure WriteCollege (college:colleges;var outfile:text);
  procedure NewCollege (college:colleges;
                var lastcollege:colleges;var outfile:text);
```

Module 3 implements the type *ListElements* with the operators to compare students records on the basis of alphabetical order of their name, their marks or the college which they belong.

Module 4 implements the type *CandidatePapers* and it contains all the operators for reading and checking the data of a student and his answers and printing out the resulting mark. As is revealed in Fig.2 from the number 4 under the name of the component *writecollege*, *module 4* uses *module 2*. Besides, it includes in its implementation the declaration of the components *seekchar* and *checkterminator*, whose declaration will be added also in *module 7*.

All the operations to read the title of an exam, the total number of questions, the marking scheme, and to produce a form echo-printed into a file are offered from *module 5* implementing the type *Script*, necessary to acquire the initial information for the examination. This module does not use other modules.

Module 6 implements the type *Lists* and contains all the operators to list the results of an exam in different orders on the basis of the college, the mark, the name, and the students. It also contains the operator to draw the histogram of the results, the operators quicksort, to order in different ways, and permute to exchange two elements. It will include the components *dice* and *getrandomnumber* nested in *permute*. For its implementation the module uses *module 2* and *3*.

The final module, 7, is the fundamental module. By using all the other modules, with the exception of *module 1*, it implements the type *exams*. It manages the whole exam, with operator to use *module 5* to input the information about the exam, *module 2* to input information about the colleges, *module 4* to input information about the candidates and their answers, *module 3* to verify the validity of the information acquired, *module 5* to list the candidates and their answers in different way, and the operator to analyse the exam, assign values and validate the legality of all the terms.

Minicalc.pas

This program implements a simple spreadsheet. It is provided with a simple portable interactive user interface, and shows the display divided into cells, labelled A to H vertically and 1 to 5 horizontally. The system takes as input the user commands that can be either be commands for the management of the spreadsheet or information to input into the cells.

The analysis of this program showed in [5], even showing interesting results, was not complete in terms of operations presented for each module. In fact, the application of the criterion to only the procedure-like components declared in the MAIN did not take into consideration operations that are fundamental for some ADTs but are implemented by components nested in others. Different results have been obtained by not considering the nesting between components in the first stage of the analysis. When compared with the previous results, the new results appear richer in terms of operations for each module. Also in this case, a real partition of the software components in more than one hierarchic module is obtained. To be precise, five modules are obtained, and they are shown in the following table:

N. mod.	user-defined data types	procedure-like components
1	usermsgs	errorhandler,writeuser
2	commands	getcommand
3	inputtype, token	writeuserinput,getcell,gettoken,getinp alphabetic,skipblanks,getchar,numeric getuserinput,ungetchar,postlabel
4	nodeptr	checkexprtree,evaluate,getexpression, factor,expression,makenode,subexpr, parseexpression,term,writeexpression
5	cellid	docellexpr,docelllabel,docellchange

To obtain the last three modules the human intervention was necessary. In fact, after the application of the method based on the ADTs, the last three modules were clusterized in one big module, comprehending in the structure more user-defined data types, sub-range, and more procedure-like components. The isolation of the sub-range types brought simpler module, but equally complex, composed of the set of all the components that appear in the last three modules. The splitting of the module was obtained by analysing the *call* relations between the procedure-like components and those existing between them and the user-defined data types. The clusterization was due to components, *makenode*, *checkexprtree* and *parseexpression*, that implement the characteristic functionalities to manage and to evaluate algebraic expressions, and are strongly dominated (Fig.3) from other components with the same purpose, but use in the interface the types *inputtype,token* and *cellid*. The insertion of the components causing the clusterization in *module 4* brought to the configuration presented above.

Fig.3 shows the final SDDT obtained after the deletion of uninteresting components. Only three components used to initialise the system, were discarded. As it can be seen in it, the components in *module 1* do not call other components, but are used from other modules. *Module 1* is very simple, it implements the type *User_Messages* managing the messages to the

user. The second module, *User_Commands*, contains the operator to read and to interpret the user commands. It uses *module 1* and operations of *module 3*.

Module 3 implements the type *Input* with the operators to manage all the different kinds of inputs, that is commands, or information to store in the cells of the spreadsheet. This information can be a label, or numeric values, or references to other cells. The module contains operations to verify the correctness of the information by analysing eventual links between cells. The final SDDT reveals that some components (*alphabetic*, *getcell*, *numeric*, ecc.) implement sub-functionalities of another component (*gettoken*), and that they can be nested in the latter. Also the component *simpletoken* can be nested here. This was not apparent from the first analysis and was thus not added to the module, but can since it is strongly dominated from *gettoken*. The routine *movetocell* called from *postlabel* will also be inserted in this module. It appears that the subject module uses *module 1*. These and other observations, that can spring from Fig.3, bring to the final structure of the module.

Module 4 is interesting in that it implements the ADT *Expression* with the operators to read and to memorize in the spreadsheet, to write, to check the correctness of and to evaluate an expression. In particular, the operation to read and to memorize the expression is implemented by the components *expression*, *subexpr*, *term* and *factor* that constitute a strongly connected component in the call graph. This component is represented by the node called *EXPRESSION*. It will be nested in *parseexpression*, that will be nested in *getexpression*. The module will include also the component *addtodependlist* nested in *checkexprtree*. *Module 4* uses *module 1* and *module 3*. In the module the type *counter* is also defined; this type is used from some components to consider the depth in the representation of the managed expression like binary tree. The above information is synthesized in the following schema:

```
module MOD4                                   USES MOD1,MOD3
type nodeptr;counter;                                  EXPORT
  procedure WriteExpression(Expr:NodePtr;Level:Counter);
  function GetExpression:NodePtr;
  function Evaluate(Expr:NodePtr;var Defined:boolean):real;
  function CheckExprTree(ExprTree:NodePtr;
                         var Count:Counter):boolean;
const .....                                            BODY
type Counter = 0..maxint;
     LineType = array[1..MAXLINE]of char;
     InputType = record
         Line : LineType;
         Length,Last : Counter;
     end;
  procedure WriteExpression(Expr:NodePtr;Level:Counter);
  function GetExpression:NodePtr;
    function ParseExpression(UserInp:InputType):Nodeptr;
      function MakeNode(NodeValue:real;
                Left,Right:NodePtr):NodePtr;
      function Expression:NodePtr;
        function Term : NodePtr;
          function Factor : NodePtr;
            function SubExpr : NodePtr;
  function Evaluate(Expr:NodePtr;var Defined:boolean):real;
  function CheckExprTree(ExprTree:NodePtr;
                         var Count:Counter):boolean;
    procedure AddToDependList(var Count:Counter);
```

Finally, the last module implements the ADT *Cells*. This module, by using all the other modules, manages the spreadsheet, with all the operators for the acquisition and the checking of labels, cell addresses and numeric expressions. In particular, for the last task, the component *docellexpr* declares the components *evaluatecells*, *buildgraph*, *sortcells* and *findzeroes*, whose names express the kinds of implemented operations.

Format.pas

This is an ancient public domain pretty printer program for Pascal that has been changed and added to by a number of different people.

The previous application of the method based on ADTs to the Format program produced very low results. Three modules were obtained with a very simple struture.

This time, the first incomplete modules obtained are the following:

N. mod.	user-defined data types	procedure-like components
1	alpha,symbols	checkfor,dostmtlist,insertsymbol
2	commenttext,width	doblock,dostatment
3	margins	changemarginto
4	symbolset	dodeclarationuntil,dofieldlistuntil
5	optionsize	bunch
6	params	readin

Fig.4 shows final SDDT of the Format program. From the original SDDT, only the initialization procedure-like components have been discarded; all the other components cooperate together to the implentation of the functionalities expressed in the identified modules. The six modules obtained look very simple, but in reality, are very complex. It appears that they cannot be re-engineered to be reusable ADTs because they are poor of operations, and, moreover, the large use of global variables would make this task very difficult. However they can offer an useful trace of the complexity of the code. They can be considered as *main* functionalities which is composed of the set of functionalities offered from the whole program. By using each other these modules cooperate in obtaining the goal of the program.

The study of the call graph reveals the existence of the following four strongly connected components:

```
SCC1:doblock,doprocedures
SCC2:dostatement, dostmlist
SCC3:dorecord,dofieldlistuntil,dovariantrecord
SCC4:readsymbol,skipcomment,docomment
```

The existence of the strongly connected components is the main cause of the recursion between the modules. In fact, *SCC2* is composed of a procedure-like component belonging to *module 1* and another one belonging to *module 2*. Since there are two components that are mutually recursive we should expect the two modules to be mutually recursive. On the other hand *SCC4* does not contain components belonging to some module, but it is used by *module 1,2* and *4*, and it uses some of this module, then its components cause the mutual recursion between *module 1,2* and *4*.

The other strongly connected components are *SCC1* and *SCC3*. The former, having a procedure-like component, *doblock*, belonging to *module 2* and the other one, *doprocedures* to no module, will implement a sub-functionality of *module 2* that will include both the procedure-like components. component, *SCC3*, will belong to *module 4* because it contains one procedure-like component, *dolistfielduntil*, belonging to this module and two to no other module. Analogously to the previous case, the last two components will be included in *module 4* to enrich the structure of it. Another cross use between modules, and between components can be seen in the final SDDT. The only modules that are not involved in the recursions and not used fromother modules are *module 3, 5* and *6*. The names of the procedure-like components, belonging to these modules, are eloquent enough to indicate the kind of functionalities implemented. For example, *module 3* is responsible for implementing one of the layout parameterizations, namely the width of the indentation. It is quite difficult to identify the way in which these functionalities are implemented. This is probably caused by the number of authors responsible for writing the program and the unavailability of the original program design.

Format is an important experiment even though poor results were achieved. This is because a partition of the program into simpler modules was obtained, and because it shows the importance of program design and that the loss of documentation is very often the main cause for not understanding a system.

Conclusion

The paper has presented a set of experiments to show the validity of a method to identify reusable assets in Pascal code thus making the programs easier to understand. By splitting the subject program into more than one modules, each of which implements an abstract data type or a group of functionalities, the method proposed can be used for the comprehension of the code. At the end of the application of the method, the program appears as a collection of simpler systems that, for their dimensions, are easier to comprehend than the full program. Also the interactions existing between the obtained modules to the pursuing of the program goal, are identified.

Unfortunately, despite obtaining better results than in past experiments, in some cases the comprehension of the implementation of some modules is difficult. This is due either to the original system design, that often does not follow guidelines useful for a correct programming, or to changes that weaken the validity of the program's specification and design.

The effectiveness of the method can be shown also on 'C' code, but the limited use of *typedef* in the 'C' programs so far studied produces results that are not really significant to be presented here.

The method and the experiments presented here have been developed in the RE^2 project, a two-years research project funded by the Italian National Research Council (CNR) and jointly developed by DIS (Dep. of Informatica e Sistemistica) at the University of Naples and CSM (Centre for Software Maintenance) at the University of Durham. The project addresses the wider issues of software reuse through the exploration of reverse engineering and re-engineering techniques to identify and extract reusable assets from existing systems.

References

[1] Liu S.S., Wilde N., **Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery**, *Proc. of IEEE Conference on Software Maintenance*, 1990

[2] Dunn M.F., Knight J.C., **Automating the Detection of Reusable Parts in Existing Software**, *Proc. of IEEE International Conference on Software Engineering*, 1993

[3] Canfora, G., Cimitile, A., and Munro, M., **RE^2: Reverse Engineering and Reuse Re-Engineering**, *Journal of Software Maintenance, Research and Practice, Wiley*, 1993

[4] Canfora, G., Cimitile, A., and Munro, M., **A Reverse Engineering Method for Identifying Reusable Abstract Data Types**, *Working Conference on Reverse Engineering, IEEE*, May 1993

[5] Canfora, G., Cimitile, A., Munro, M., and Tortorella, M., **Experiments in Identifying Reusable Abstract Data Types in Program Code**, *Workshop on Program Comprehension, IEEE*, 1993

[6] Canfora, G., Cimitile, A., Munro, M., and Taylor, C.J., **Extracting Abstract Data Types from C Programs: A Case Study**, *Conference on Software Maintenance, IEEE*, 1993

[7] Cimitile, A., and Visaggio, G., **Software Salvaging and the Dominance Tree**, *to appear on Journal of Systems and Software*, 1993

[8] Cimitile, A., Fasolino, A.R., Maresca, P., **Reuse Reengineering and Validation via Concept Assignment**, *Conference on Software Maintenance, IEEE*, 1993

[9] Canfora, G., Cimitile, A., Munro, M., and Tortorella, M., **A Precise Method for Identifying Reusable Abstract Data Types in Code**, *International Conference on Software Maintenance, IEEE*, 1994

[10] De Lucia, A., Di Lucca, G.A., and Fasolino A.R., **Towards the Evaluation of Reengineering Effort to Reuse Existing Software**, *International Conference on Achieving Quality in Software*, 1993

[11] Hetch, M.S., **Flow Analysis of Computer Programs**, *Elsevier North-Holland, New York*, 1977

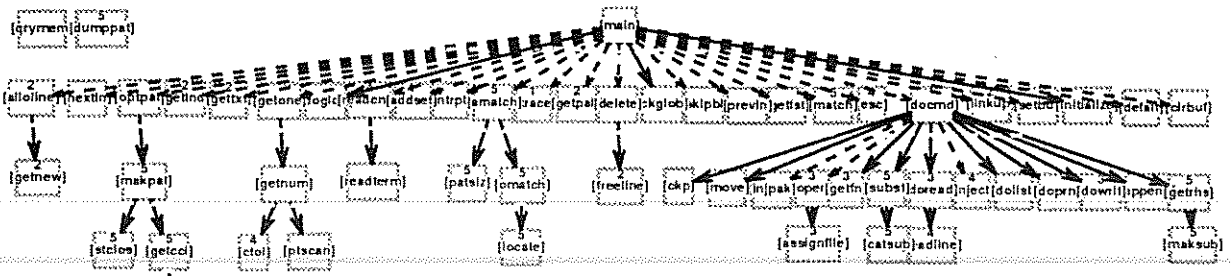


Fig.1a - Strong Direct Dominance Tree of Editor.pas.

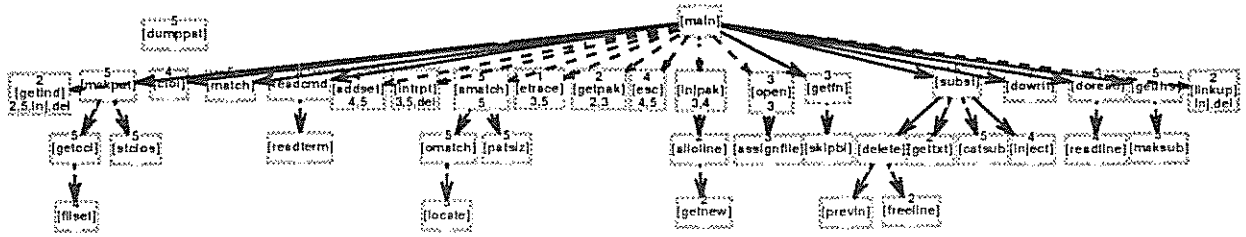


Fig.1b - Final SDDT of Editor.pas

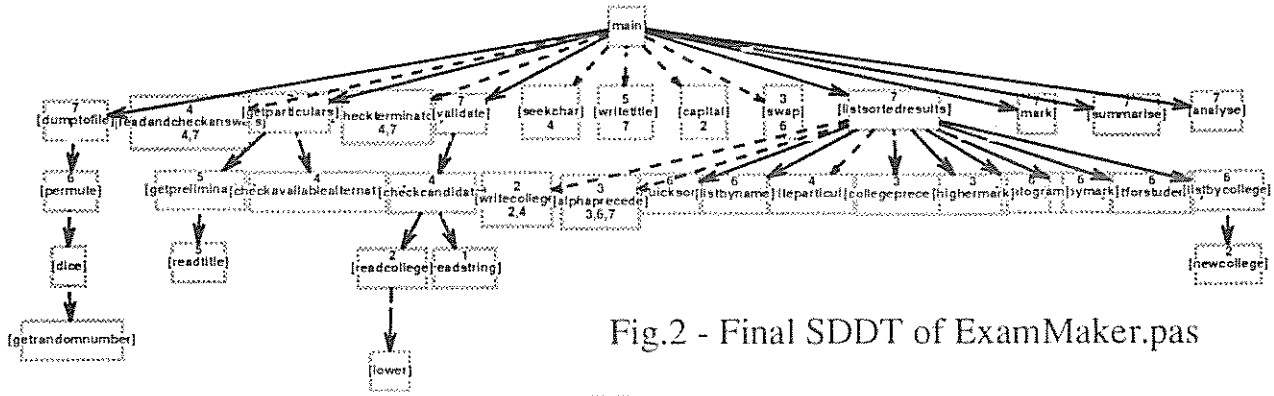


Fig.2 - Final SDDT of ExamMaker.pas

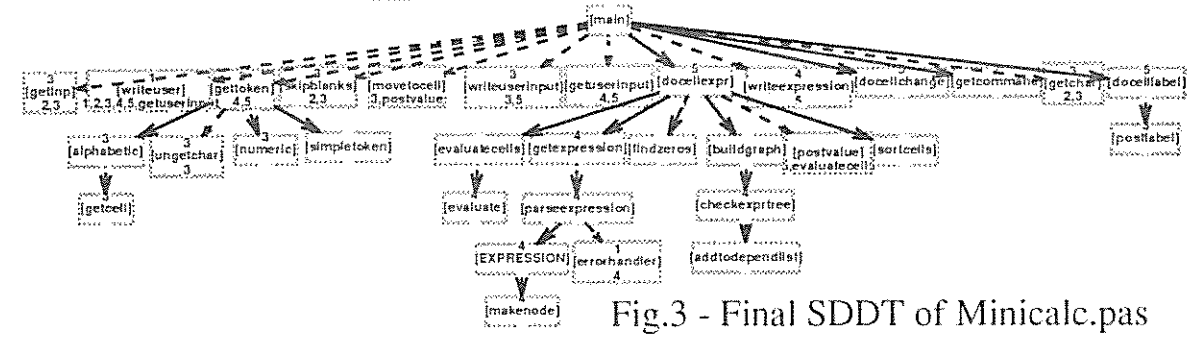


Fig.3 - Final SDDT of Minicalc.pas

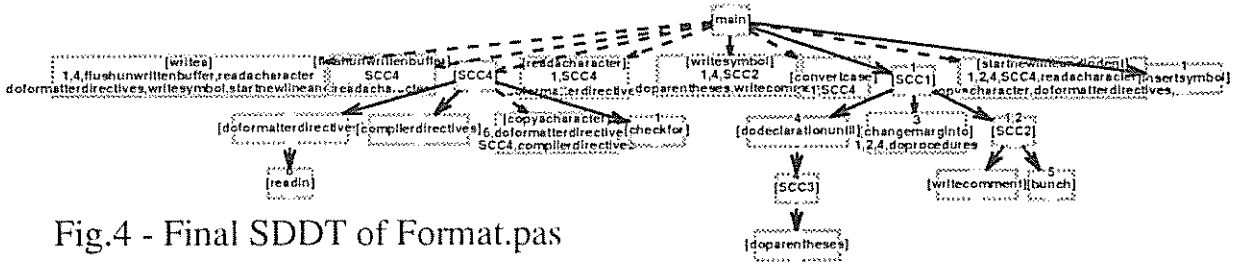


Fig.4 - Final SDDT of Format.pas

A Tool for Understanding Object-Oriented Program Dependencies

Panagiotis K. Linos
Tennessee Technological University
Computer Science Department
Cookeville, TN 38505, USA
email : PKL3678@tntech.edu

Vincent Courtois
Hautes Etudes Industrielles
13 Rue de Toul
59046, Lille
France

Abstract

In this paper, we present a tool for understanding and re-engineering C++ programs called OO!CARE (Object-Oriented Computer-Aided Re-Engineering). OO!CARE demonstrates some practical solutions to the problem of extracting and visualizing object-oriented program dependencies (i.e. data-objects and their relationships). It is an extension of an earlier tool for maintaining C programs called CARE (Computer-Aided Re-engineering). In this paper, we also discuss some early experiences acquired from using the tool. For instance, an important observation made during a re-engineering exercise is that some characteristics of the object-oriented programming paradigm such as *inheritance* and *polymorphism* contribute significantly to the complexity of understanding program dependencies. Moreover, in this paper, we discuss how object-oriented program dependencies differ from the procedural ones and explain how they can be visualized within the same environment.

1. Introduction

Program dependencies include information regarding the relationships between various components in computer programs such as the interaction between modules (i.e. files), the use of variables and their types, as well as the calls among functions. Today, there are many software environments which facilitate the comprehension of programs written in procedural programming languages by systematically extracting a large number of program dependencies contained in the source code. These dependencies are stored in a database and then visualized in various graphical representations. Taxonomies of software visualization environments and their characteristics can be found in Price [8] and Stasko [12]. These environments have been reasonably successful in helping to reduce the time and effort spent to understand existing programs written in procedural programming languages. However, a new problem arises from the fact that object-oriented languages are quickly emerging from the procedural ones. Although, the new features introduced by object-oriented languages offer flexibility some complications during program understanding can arise [16]. In particular, the use of *classes* and *inheritance* often leads to a plethora of small program components (e.g. *objects*) with many relationships (e.g. *message passing*) [14]. Consequently, locating and understanding object-oriented program dependencies becomes a difficult problem. Today, there are several commercial and academic tools available for understanding and re-engineering object-oriented programs. Examples of commercial tools include the *ObjectCenter* by Centerline Software Inc. [1], the *ObjectWork* by Parcplace Systems and the *Objective C* from the Stepstone Corporation. In addition, several research prototype tools for maintaining object-oriented programs are available today. A software environment described in Lejter [3] entails a relational database with an interactive interface which supports queries about programs written in object-oriented programming languages. Another software tool provides browsing features through the source code of object-oriented programs using hypertext techniques [10]. Finally, some visualization mechanisms for maintaining object-oriented software are described in DePauw [15] which are based on a language-independent approach.

Many of the above tools utilize graphical representations which become difficult to understand when medium-to-large programs are displayed. In addition, they support limited abstraction mechanisms and transformation tools for facilitating the re-engineering of object-oriented programs. In this paper, we address the above issues by developing a software environment for understanding and re-engineering C++ programs called OO!CARE (Object-Oriented Computer-Aided Re-Engineering). This effort focuses on the dynamic behavior of object-oriented programs and it supports the general hypothesis that visualization of program dependencies is most effective for program comprehension [6]. The OO!CARE environment evolved from an earlier tool for maintaining C programs called CARE (Computer-Aided Re-engineering) [5]. The new environment facilitates the understanding of existing C programs as well as the difficulties introduced by the C++ language. The rest of this paper is organized as follows : the second section explains how object-oriented program dependencies differ from the procedural ones. Then, a description of the OO!CARE environment is given in the third section. The fourth section presents a brief history of the OO!CARE project and finally in the fifth section we present our conclusions.

2. Object-Oriented Program Dependencies

Programs written using a procedural programming language (e.g. Pascal, C) consist of various components which embody *data-elements*, *data-types* and *sub-programs*. Examples of *data-elements* in procedural languages include variables,

0-8186-5647-6/94 $04.00 © 1994 IEEE

constants and parameters of the program. *Data-types* can be standard or user-defined and sub-programs represent functions, procedures or modules (i.e. a group of functions or procedures). These components are linked via various relationships such as *calls* between functions, the *use* of parameters by a procedure or the *definition* of a variable as of a data-type. We call these components and their relationships *Procedural Program Dependencies* (PPDs). More formally, a PPD can be represented as a triplet PPD=<X, Y, R> where X and Y can be *data-elements, data-types* or *sub-programs* and R depicts a relationship between X and Y. For example, the triplet <Variable, Type, *is-defined-as*> presents the *is-defined-as* relationship between variables and data-*types*. An instance of this program dependency is <counter, integer, is-defined-as> meaning that a variable called *counter* is defined as an *integer* data-type in the program.

On the other hand, programs written in an object-oriented language entail different kinds of components and relationships due to the different programming paradigm supported. Today, there are two families of object-oriented languages; the *pure* object-oriented ones where all computation is based on *message passing* (e.g. Smalltalk, Eiffel) and the *hybrid* object-oriented languages which have evolved from the procedural ones (e.g. C++) [11]. They usually include a mixture of features from both families of languages. Programs written in a pure object-oriented language consist of *data-objects, class-types* and *methods*. We call these entities and their relationships *Object-Oriented Program Dependencies* (OOPDs). Similarly, an OOPD can be represented by a triplet OOPD=<X, Y, R> where the entities X and Y can be *data-objects, class-types* or *methods* and R represents a relationship between X and Y. For example, the triplet <Class, Method, *implements*> defines the *implements* relationship between *Classes* and *Methods*. An instance of this relationship is <Shape, draw, *implements*> meaning that the class *Shape* implements (i.e. defines within its body) a method for this class called *draw*. Finally, programs written using a *hybrid* object-oriented programming language entail a combination of both procedural and object-oriented program dependencies. An example of such dependency is represented by the triplet <Class::Method, Function, *calls* > meaning that class methods *call* user-defined functions. An instance of this dependency is <Shape::draw, PrintLabel, *calls*> indicating that the *draw* method of the class *Shape* calls a regular function called *PrintLabel*.

In this work, we have selected C++ as a hybrid object-oriented programming language in order to study how object-oriented program dependencies can be visualized. C++ supports the object-oriented programming paradigm while maintaining the procedural features of the C language. In the object-oriented programming paradigm data and behavior of a program are strongly connected. C++ implements this concept by the use of *classes* whose instances are *objects* [2]. A *class* consists of a set of values (*data members*) and a collection of operations (*methods* or *member functions*) that can act on those values. For example, we can create a class called *GeometricFigure* where the data members are the *center* and *perimeter* and where methods are operations using those members such as *draw-figure*. *Inheritance* is a way of deriving a new class from existing classes called *base classes*. The derived class is developed from its parent by adding or altering code. Inheritance is said to be *single* if a class is derived from only one parent, or *multiple* if it is developed from several base classes. Moreover, access privileges to classes and their members can be managed and limited to whatever group of functions needs to access their implementation (this is accomplished in C++ by the use of the keywords *public*, *protected* and *private*). Also, functions and operators can be overloaded in C++. Overloaded functions or operators are also known as *polymorphic* entities because they can take many different forms (i.e. can have several distinct implementations). In particular, polymorphic functions are implemented by *virtual* member functions in C++.

A. Polymorphic Program Dependencies

The above mentioned features of object-oriented programming languages introduce some additional complexity towards extracting program dependencies from source code. In particular, the use of polymorphic entities create *dynamic* program dependencies in object-oriented programs. As we know, *static* program dependencies are extracted directly by analyzing (i.e. scanning) the source code and execution of the program is not necessary. However, *dynamic* program dependencies are only determined at run-time and they require program execution. *Message passing* to a *virtual* member function in a C++ program is an example of a *dynamic* program dependency. In order to demonstrate this dynamic behavior we present a C++ example shown in Figure 2.1. As we can see in this figure, the classes named *Box* and *Circle* are derived from the class *Shape*. Also, the class *Square* is a subclass of *Box*. Moreover, a variable P is declared as a pointer to a *Shape* or to any of its subclasses (i.e. *Box*, *Circle* and *Square*). When the program statement *P->draw()* is executed *a message* is sent to the virtual member function *draw()*. However, all four classes in the program of Figure 2.1 implement their own version of this method. Therefore, the version being called can only be determined at run-time, depending on what kind of shape P points to at the particular time of execution. This dynamic behavior is also demonstrated graphically in the same figure. When the while loop of the program in Figure 2.1 is executed for the first time the message is sent to the method *Shape::draw* because P points to a *Shape* object but on the next iteration P points to a *Box* so the message is sent to *Box::draw*. During the following loop iteration, the same message is sent to the *Circle::draw* method and then to *Square::draw*. This sequence continues until execution of the program stops. With this example, we demonstrate the fact that a statement such as *P->draw()* can have several different meanings (i.e. *polymorphic* entity) and can be

determined only at run-time. We call such dependencies *Polymorphic Program Dependencies*.

B. Implicit Program Dependencies

In addition, program dependencies can be *implicit* or *explicit*. Explicit program dependencies appear in the program and can be directly extracted from the source code. However, implicit program dependencies do not arise explicitly in the source code and thus some additional complexity is introduced for extracting them.

```
#include <iostream>
class Shape : { virtual void draw(void)
    { cout << "drawing a Shape"; } };
class Box : public Shape { virtual void draw(void)
    { cout << "drawing a Box"; } };
class Circle : public Shape { virtual void draw(void)
    { cout << "drawing a Circle"; } };
class Square : public Box { virtual void draw(void)
    { cout << "drawing a Square"; } };
main () {
Shape * P;  int i=0; int limit=10;
Shape Sh; Box Bo; Circle Ci; Square Sq;
while (i<limit)
{
   switch (i%4) {
   case 0: P=Sh;
      break;
   case 1: P=Bo;
      break;
   case 2: P=Ci;
      break;
   case 3: P=Sq;
      break;
   }
   P->draw();
   i++;
}
}
```

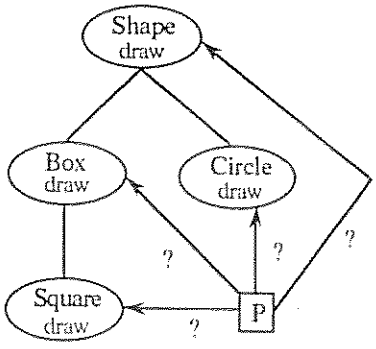


Figure 2.1 : A C++ program demonstrating dynamic program dependencies

In order to demonstrate some caveats regarding implicit program dependencies we consider a small C++ example shown in Figure 2.2. The output (with some explanations) generated by this program is given in Figure 2.3. Although only four objects (i.e. *ashape, abox, somesquare* and *othersquare*) are created in the *main()* function of the program as shown in Figure 2.2, nine *constructors* (i.e. member functions used to create objects) are executed (see their output in Figure 2.3). This happens because in the program, a *square* is defined to be a special case of *box* and a *box* a special case of *shape*. Thus, in order for the program to create a *square* object, it first builds a *shape* object, then constructs a *box* object of this shape, and finally turns this *box* into a *square*. This results in the execution of additional statements which do not appear explicitly in the source code. Moreover, in this example, there is no explicit call to *destructors* (i.e. member functions which deallocate space for an object). However, several destructors are invoked implicitly in the program for the four objects created (see their output in Figure 2.3). Notice that the program sends a message to three different destructors in order to destroy a *square*.

```
// Example of implicit operations performed by C+
#include <iostream.h>
class shape {
  private: double area;
  public :  shape(void)
   { cout << "I am a shape " << endl; }
   ~shape(void)
   { cout << "I am the shape destructor" << endl;}
};
class box : public shape {
  private: double l,L;
  public : box(void)
   { cout << ", i am a box " << endl; }
   ~box(void)
   { cout << "I am the box destructor "<< endl;}
};
class square : public box {
  private: double side;
  public : square(void)
   { cout << ", i am a square " << endl; }
   ~square(void)
   { cout << "I am the square destructor" << endl;}
};
main() {
shape ashape;
box abox;
square somesquare, othersquare;
somesquare = othersquare;
}
```

Figure 2.2 : C++ source code demonstrating implicit program dependencies

In addition, C++ provides default constructors and destructors (i.e. none defined by the programmer). Consequently, another version of the program in Figure 2.2 is shown in Figure 2.4. The two programs are functionally equivalent however, the program of Figure 2.4 depicts a higher abstraction of implicit program dependencies (i.e. constructors and destructors are not defined explicitly in the source code) than the program of Figure 2.2. Evidently, the above mentioned features of object-oriented programs can complicate the task of understanding program dependencies. In this paper, we focus on the above issues with respect to extracting object-oriented program dependencies from C++ programs. Also, we explore ways to efficiently visualize such dependencies in order to facilitate the comprehension of object-oriented programs. To this end, we have designed

and implemented a software tool for understanding and re-engineering C++ programs. An overview of its main features is given in the next section.

OUTPUT	PRODUCED BY
I am a shape	constructor of ashape
I am a shape	constructor of abox
, i am a box	constructor of abox
I am a shape	constructor of somesquare
, i am a box	constructor of somesquare
, i am a square	constructor of somesquare
I am a shape	constructor of othersquare
, i am a box	constructor of othersquare
, i am a square	constructor of othersquare
I am the square destructor	destructor of othersquare
I am the box destructor	destructor of othersquare
I am the shape destructor	destructor of othersquare
I am the square destructor	destructor of somesquare
I am the box destructor	destructor of somesquare
I am the shape destructor	destructor of somesquare
I am the box destructor	destructor of abox
I am the shape destructor	destructor of abox
I am the shape destructor	destructor of ashape

Figure 2.3: Output produced by the C++ program of Figure 2.2

in Figure 3.2. The *type* node in the figure represents C++ standard or user defined data types. The functions of the program are depicted by the *function* node. The *constant* and *variable* nodes in the graph represent constants and variables respectively. Moreover, the *object* node represents data-objects (i.e. instances of a class).

```
// A modified version of the C++ program
// shown in Figure 2.2

#include <iostream.h>
// constructors and destructors are not defined explicitly
// in the source code but provided by C++
class shape : { double area; };
class box : public shape { double l,L; };
class square : public box { double side; };
main() {
shape ashape;
box abox;
square somesquare, othersquare;
somesquare = othersquare;
}
```

Figure 2.4 : A C++ program with highly abstracted implicit program dependencies

3. The OO!CARE environment

This section explains how OO!CARE (Object-Oriented Computer-Aided Re-Engineering) evolved from the CARE (Computer-Aided Re-Engineering) tool. Then, it presents the architecture of OO!CARE and the data model used to construct its database. Finally, it describes the presentation model used to display data-flow and control-flow information graphically.

The OO!CARE environment evolved from an existing software tool called CARE which facilitates the comprehension and re-engineering of existing C programs [5]. The OO!CARE tool extends the existing features of CARE in order to facilitate the comprehension of C++ programs. Specifically, the original functionality and user-interface of CARE are extended in order to extract and visualize object-oriented program dependencies. The user-interface in OO!CARE is done through specially designed windows including the main panel as well as graphical and textual windows. Each window in OO!CARE is equipped with a group of typical operations for manipulating graphs or text (e.g. hide, highlight, zoom, code etc.). The architecture of OO!CARE is comprised of the *code analyzer*, the *dependencies database* and the *display manager*. The code analyzer extracts program dependencies from C++ source code and populates them into the database. In OO!CARE, eight kinds of program components are extracted from C++ code namely *data-types*, *user-defined functions*, *constants*, *variables*, *parameters*, *objects*, *classes* and *member functions* (*methods*). These components and their relations are populated in the database based on the data model shown

Information passed to a function or a method is depicted by the *parameter* node which can be defined as a class or as a type. A *class* encapsulates data with member functions. A *member function* is a function declared within a class and it can only be invoked by instances of this class. The relationships between these components are also shown in Figure 3.2 by connecting arrows. For example, the connecting arrow between *class* and *member function* means that a *class* implements (defines) a *member function* within its body. The program dependencies included in the database can be displayed using a presentation model designed for the OO!CARE environment. This model includes hierarchical displays for presenting class inheritance, control-flow program dependencies and file dependencies. In addition, an original display called *colonnade* (i.e. a sequence of columns displayed at regular intervals) is utilized in order to present data-flow program dependencies graphically. Such graphical displays are created and manipulated by the use of several graphical editors in OO!CARE. These include the *data-flow*, *control-flow*, *inheritance-hierarchy* and *file-dependencies* editors. Textual information (i.e. code) can also be manipulated in the OO!CARE environment using a traditional text editor (e.g. emacs, vi). The file dependencies editor presents the *include* relationships between files using a hierarchical display. Each file is represented by a node and included files are linked via connecting lines. The control-flow graph editor displays user-defined and member functions using different graphical notations. Table I includes a list of C++ program function types supported by the editor and their graphical notation. Function calls or message passing is denoted by connecting lines between

functions and methods (i.e. member functions). Specifically, a connecting line between two functions (or between a method and a function) depicts a call relationship. A connecting line between two methods (or a function and a method) represents a message being passed. Moreover, information regarding polymorphic program dependencies is also displayed in the control-flow graph.

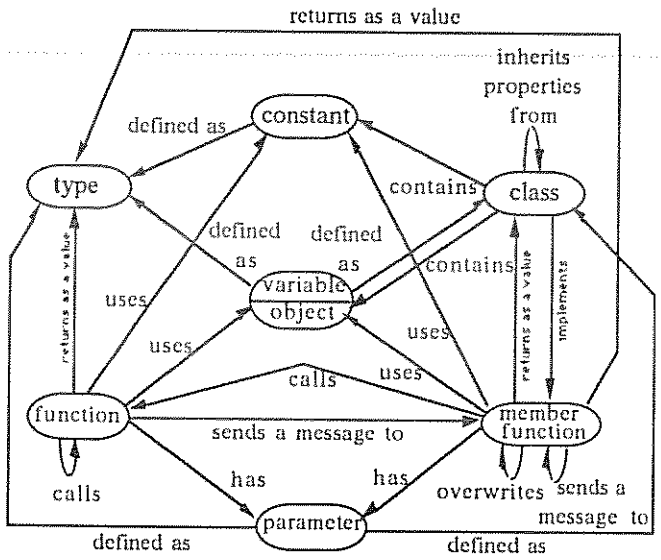


Figure 3.2 : The data model used in OO!CARE

For example, in the case of a message passed to a virtual member function (i.e. known at run time) the parser determines all possible entities where a message could be sent. In this case, OO!CARE creates a dummy member function of an unknown class (displayed as ?::member_function) which is connected to all possible member functions with the same name. This approach is illustrated by an example in Figure 3.5. In this example, a message to a polymorphic member function called *draw* is sent. The function *draw* is implemented differently within the classes *shape*, *box*, *circle* and *square*. OO!CARE displays a dummy node called *?::draw* which is linked to all the member functions named *draw* via connecting lines. An example of the control-flow graph of a C++ program is displayed by OO!CARE as shown in Figure 3.6. This example contains four user-defined functions namely *main*, *getname*, *getid* and *getsal* (all displayed by an oval shape according to Table I) and three virtual member functions called *student::print_virtual*, *teacher::print_virtual* and *person::print_virtual*. Moreover, we can see in Figure 3.6 that the *main* function of this program sends a message to a polymorphic function called *print_virtual* (all possible paths of this message are shown in Figure 3.6). In addition, the *main* driver function sends a message to three different types of constructors namely *student::student*, *person::person* and *teacher::teacher* and to their corresponding destructors namely *student::~student*, *person::~person* and *teacher::~teacher*. The above constructors call the user-defined functions *getid*, *getname* and *getsal* respectively. Finally, each of the three virtual member functions *student::print_virtual*, *person::print_virtual* and *teacher::print_virtual* sends a message to a *print* member function to their respective class.

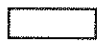
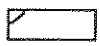
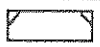
NOTATION	MEANING
	User-defined function
	Public member function
	Protected member function
	Private member function
	Virtual member function

Table I : Shapes used in the control-flow graph and their meaning

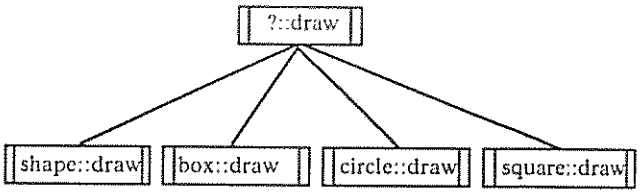


Figure 3.5 : Visualization of a message sent to a polymorphic function called draw

The data-flow program dependencies are presented separately in OO!CARE using a graphical display called *colonnade* (i.e. a sequence of columns drawn at regular intervals). Experimental data have shown that the colonnade display appears to be a promising graphical model for visualizing data-flow program dependencies [7]. It produces crossing-free, easy-to-draw and aesthetically pleasing layouts. The colonnade includes information about *Variables* or *Objects*, *Types*, *Parameters*, *Functions*, *Methods*, *Classes* and *Constants*. The relations between entities are represented by connecting lines between columns. Figure 3.7 gives an example of a colonnade display of a C++ program produced by OO!CARE. From this figure we can see that the first column displays the variables (or objects) of the program and the second column depicts the data-types. The next column contains the parameters and the following one embodies the functions (i.e. user-defined or member functions).

Finally, the fifth column displays the classes and the last one entails the constants of the program. Among others, Figure 3.7 depicts two constructor member functions for the classes *student* and *teacher* respectively. The *main* function is also displayed in the colonnade which uses an integer variable called *i* and six objects named *aperson*, *myperson*, *ateacher*, *myteacher*, *astudent* and *mystudent*. These objects are instances of their corresponding classes.

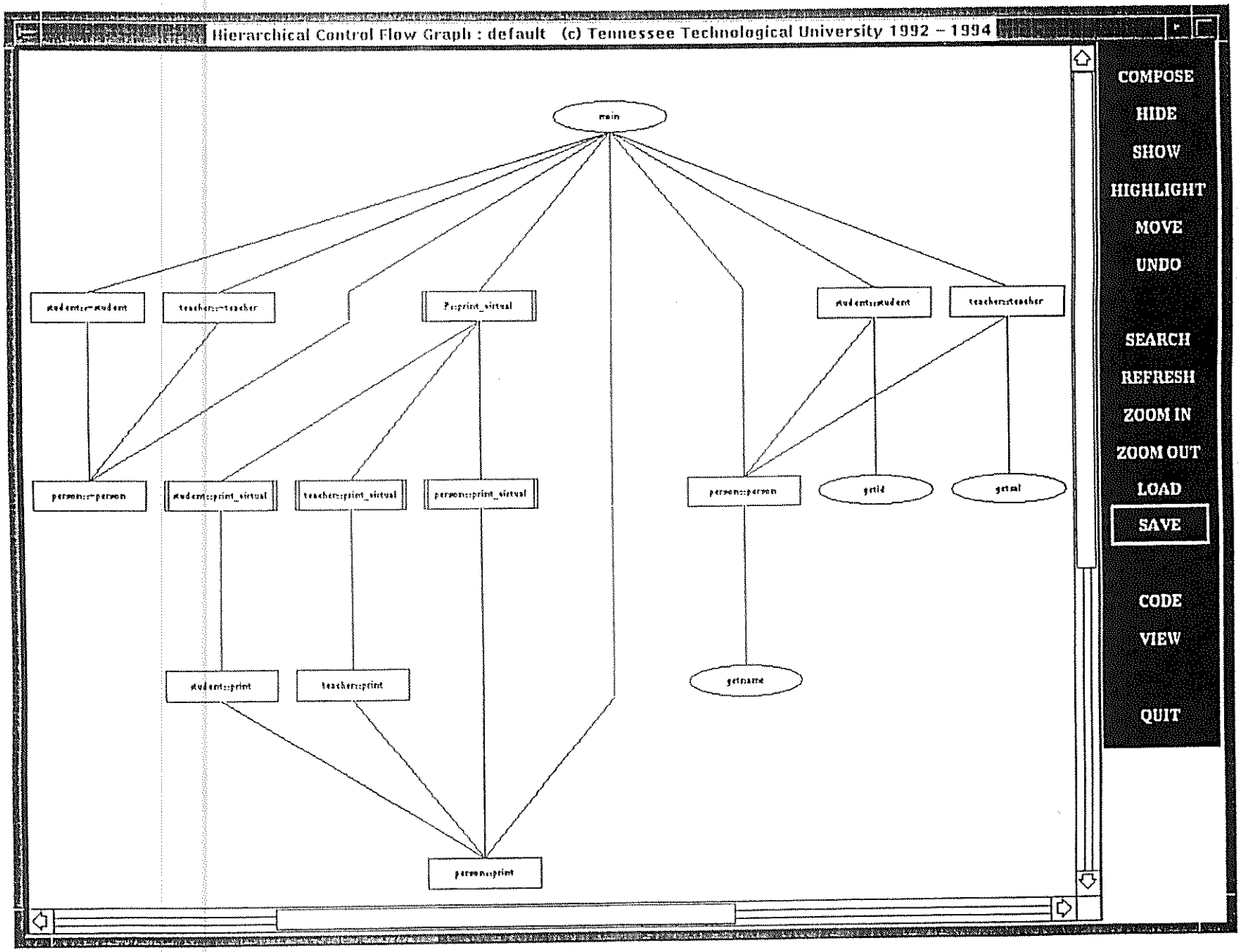


Figure 3.6: OOICARE displays the control-flow graph of a C++ program

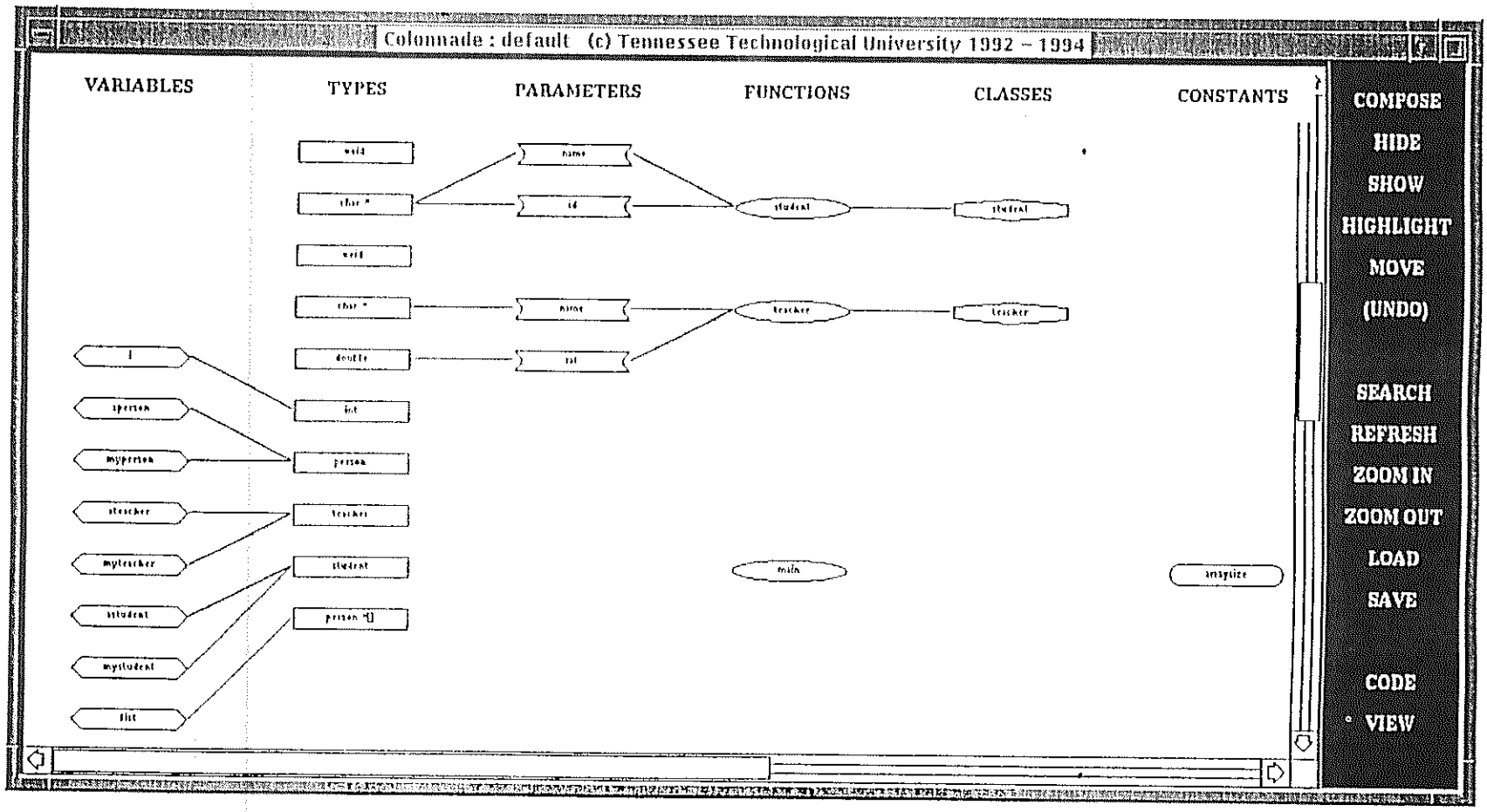


Figure 3.7: A colonnade display by OOICARE

Also, the *main* function references a variable called *list* which is an array of pointers to a *person* object. Finally, *main* utilizes a constant called *arraysize*. If two columns are not adjacent, the relations between their entities are hidden. In order to display such relations, the colonnade allows the user to change the placement of the columns using a *move-column* operation. For instance, if the relationships between functions and types need to be visualized, the move-column option can be used to create a different layout. Finally, the inheritance graph includes all the classes defined in a C++ program and their inheritance relationships (i.e. both *single* and *multiple* inheritance).

4. History

OO!CARE (Object-Oriented Computer-Aided Re-Engineering) is a software environment for understanding and re-engineering C++ programs. It is an on-going research project since 1990 and it evolved from an earlier effort towards maintaining C programs. It is partially funded by Tennessee Technological University (under grants #9211, #9423 and #9512) and it involves a faculty member and three research students. A prototype version runs on DEC stations under the Ultrix operating system and it uses the X window manager [9]. Finally, the *lex* utility available in the Ultrix environment is also used to implement the code analyzer [4].

5. Conclusions

In this work, we have designed and implemented a software tool for understanding and re-engineering C++ programs called OO!CARE (Object-Oriented Computer-Aided Re-Engineering). During the implementation of a prototype version of OO!CARE, we made some observations with respect to understanding object-oriented program dependencies (i.e. data-objects and their relationships). First, we have observed that the unique characteristics of the object-oriented programming paradigm such as *inheritance* and *polymorphism* can increase the complexity of understanding object-oriented program dependencies. In particular, tracing the control-flow program dependencies (i.e. message passing between objects) of a medium-to-large size object-oriented program becomes a difficult task because the maintenance programmer is forced to travel through a usually large hierarchy of classes. Moreover, *polymorphic* program dependencies contribute significantly to the complexity of object-oriented programs because their values cannot be known before the execution of the program (i.e. they are known only at run-time). Finally, the use of *implicit* object-oriented program dependencies (i.e. the ones that do not appear explicitly in the source code) adds some additional complications to the task of program comprehension. Although, we do not have any experimental data to support these observations at this point, our experience during the development of OO!CARE indicates that object-oriented program dependencies appear to be complicated enough in order to make tools for program comprehension a compelling area of research and investigation. These environments need to focus on efficient ways of visualizing object-oriented program dependencies, simple abstraction mechanisms and effective transformation tools. Such features can be useful to the maintenance programmer for understanding the difficulties introduced by object-oriented programs.

References

[1] Centerline Software, *Object Center Reference*, Centerline Software Inc., Cambridge, Massachusetts, USA, 1991.

[2] Gorlen K., Orlow S., Plexico P., *Data Abstraction and Object-Oriented Programming in C++*, Wiley and Sons.

[3] Lejter M., Meyer S and Reiss S., *Support for Maintaining Object-Oriented Programs*, IEEE Transactions on Software Engineering, Vol. 18, No 12, pp. 1045-1052, December 1992.

[4] Levine J., Mason T., Brown D., *Lex & Yacc*, O'Reilly & Associates, Inc.

[5] Linos P., Aubet P., Dumas L., Helleboid Y., Lejeune P., Tulula P., *CARE : An Environment for Understanding and Re-engineering C programs*, IEEE Conference on Software Maintenance, Montreal, Canada, 1993, pp. 130-139.

[6] Linos P., Aubet P., Dumas L., Helleboid Y., Lejeune P., Tulula P., *Visualizing Program Dependencies*, Software-Practice and Experience, Vol. 24(4), April 1994, pp. 387-403 .

[7] Linos P., Aubet P., Dumas L., Helleboid Y., Lejeune P., Tulula P., *Facilitating the Comprehension of C Programs : An Experimental Study*, 2nd IEEE Workshop on Program Comprehension, Capri, Italy, 1993, pp. 55-63.

[8] Price B., Baecker R., Small I., *A Principled Taxonomy of Software Visualization*, Journal of Visual Languages and Computing, Vol. 4, 1993, pp. 211-266.

[9] Reiss L., Rodin J., *X Window inside-out*, McGraw Hill.

[10] Sametinger J., *A Tool for Maintenance of C++ Programs*, IEEE Conference on Software Maintenance, San Diego, California, 1990, pp. 54-59.

[11] Sebesta R., *Concepts of Programming Languages*, The Benjamin/Cummings, 1993.

[12] Stasko J., Patterson C., *Understanding and Characterizing Software Visualization Systems*, IEEE Visual Languages Workshop, Seattle, Washington, September 1992, pp. 3-10.

[13] Stroustrup B., The *C++ Programming Language*, Addison-Wesley.

[14] Taenzer D., Ganti M., Podar S., *Object-Oriented Software Reuse : The Yoyo Problem*, Journal of Object-Oriented Programming 1989, pp. 30-35.

[15] Wim DePauw, Helm R., Kimelman D., Vlissides J., *Visualizing the Behavior of Object-Oriented Systems*, OOPSLA'93, pp. 326-337.

[16] Wilde N., Huitt R., *Maintenance Support for Object-Oriented Programs*, IEEE Transactions on Software Engineering, Vol. 18, No 12, December 92.

Session B:
Architecture

Recovering the Architectural Design for Software Comprehension

G. Canfora°, A. De Lucia*, G. A. Di Lucca*, A. R. Fasolino*
(canfora/delucia/dilucca/fasolino)@nadis.dis.unina.it

* Dep. of "Informatica e Sistemistica" - University of Naples "Federico II" - ITALY

° Dep. of "Ingegneria dell'Informazione ed Ingegneria Elettrica"
University of Salerno, Faculty of Engineering at Benevento - ITALY

Abstract

The work described in this paper addresses the problem of understanding a software system and focuses in particular on the comprehension of the system architectural design. A method is proposed to reconstruct the architecture of a system and represent it in the form of a structure chart. The method assumes the system was originally designed with a functional decomposition approach, and aggregates program units into modules whenever these implement a functionality of the system. A directed graph that describes the activations of program units is used to model the system, and the concept of node dominance on a directed graph is exploited to aggregate program units into modules and to derive intermodular relationships from the unit activations. Finally, the system data set is partitioned into sets of data items which are local to a given module and sets of data items which are global to the modules belonging to a subtree of the structure chart, and the interfaces of modules are identified.

1. Introduction

The steady increase in software production costs has caused the Engineering of Existing Software to emerge as a viable discipline. A number of experiences have shown that it is often cost effective to keep an existing system alive by adjusting it over time as the user requirements and/or operating environment change. Keeping existing systems alive is sometimes indispensable because they encode knowledge and expertise which is not available anywhere else than in the source code. Indeed, many pieces of software exist that perform valuable tasks for which no written documentation is available.

Software maintenance, evolution, re-engineering, reverse engineering, migration and reuse are a few examples of activities that aim to prolong a software system's lifetime. At the heart of all these activities is software comprehension, and in particular the ability to comprehend systems developed by other people.

For a software system to be completely understood three different aspects have to be analysed: the functional layer (what system modules do), the low level design layer (how modules work) and the architectural layer (how modules are organised to form the system).

Artificial intelligence techniques and knowledge-based systems have long been recognised as pioneers in automating some aspects of software understanding, mainly related to the functional layer [10, 12]. Common approaches feature: (i) an internal representation of the program control and data flow; (ii) a library of programming plans and clichés, and (iii) an algorithm to map program fragments onto these clichés. The main problem with these approaches is the size of the plan library, which must contain both domain independent and domain specific plans, and the size of the mapping algorithm search space. This greatly limits the length and complexity of the programs that can be tackled and, in fact, artificial intelligence techniques have not yet been realistically applied in any particular application domain.

Several reverse engineering tools have been proposed in the literature that aim to facilitate software comprehension at the low level design and architectural layers. Most of these tools are essentially browsers that provide users with different textual and graphical views of the code. While these tools have been seen to be effective in supporting comprehension at the low level design layer, they often fail to address the architectural layer. Notable examples of successes in understanding the low level design layer include the automatic generation of low level Jackson or Warnier/Orr documents from COBOL source code [2], the work of Sneed *et al.* on software recycling [13], and the GRASP/Ada project [5]. All these experiences show that understanding the low level design of a module essentially requires little more than re-mapping objects and the relations among them, which are already explicitly shown

This work has been supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of the C.N.R. (Italian National Research Council) under grant n. 9301633PF69.

0-8186-5647-6/94 $04.00 © 1994 IEEE

in the source code, into nicer and more readable forms. This is exactly what browsers do. Understanding the architecture of a system, on the other hand, involves abstracting views which may not be explicitly shown in the code. For example, many tools exist that produce structure chart-like views of a system. This is the case of VIFOR [11], which graphically displays the calls and references to global items in a FORTRAN system, and the VIASOFT toolset, which produces structure charts from PERFORM statements. Indeed, while resembling structure charts, the views these tools furnish differ considerably from the structure charts that can be produced in a forward engineering process. Reverse engineering tools often merely assimilate the concept of a module in a structure chart to a program unit of the coding language (a routine in FORTRAN, a procedure in Pascal, or a performed section/paragraph in COBOL), thus failing to aggregate program units into a module whenever these contribute to implementing a system functionality or to break a program unit into several modules whenever it implements more than one functionality. The identification of the module interfaces is also often performed simply on the basis of the language characteristics, for example by deriving the intermodular relationships from calls to routines or PERFORM statements and compiling the list of the data flowing in and out from each module according to the list of parameters exchanged in a routine call. The consequence is that the structure charts produced are a snapshot of the system structure rather than a representation of its architectural design.

Understanding the architectural design of a software system requires much more than just depicting relations already explicit in the code, such as a procedure call or the binding of an actual parameter to the corresponding formal parameter in the form of boxes, bubbles and edges. The type of decomposition, either functional or based on the ideas of data encapsulation and object orientation, which brought about the architectural design of the system must guide the definition of what the content of a module is, i.e. which kind of abstraction each module implements. Once the module content has been defined, the relative abstractions have to be localised in the source code, and the relationships between the resulting modules have to be derived. This requires the formulation of hypotheses on the way abstractions in the design have been mapped onto code. It is worth stressing that both the criteria and models adopted to localise the modules and the techniques used to derive the intermodular relationships vary considerably according to the kind of decomposition that drove the original development of the system, i.e. according to the type of abstraction to be identified in the source code.

This paper deals with the comprehension, at the architectural level, of a software system designed according to a functional decomposition approach. A method is proposed to reconstruct the architecture of a software system and represent it in the form of a structure chart. The structure chart nodes are not simply program units, but aggregates of units, each aggregate implementing a functional abstraction. Consequently, edges are not a mere representation of routine calls or PERFORM statements but, rather, depict the different kinds of relationship that can exist between modules and, in particular, the "uses" and "is composed of" relationships [6]. The concept of node dominance on a directed graph is exploited to aggregate program units into modules, and to derive intermodular relationships from the unit activations.

The architectural design produced by our method does not take into account the limitations of either the coding language or the operating environment: it is a sort of ideal architecture, i.e. the system architectural design as it could be implemented on an ideal platform. This means that all the design decisions deriving from considerations on the internal consistency of modules and the abstractions they implement are reflected in the structure chart, while the decisions dictated by limits of the implementation platform are filtered out. This is a matter of considerable importance as the platform limits usually corrupt the ideal design, thus making it difficult to understand the meaning of modules and their interconnections. Filtering out design decisions related to the implementation platform allows the architecture of the system as it was originally conceived by the designer to be reconstructed, and this is a valuable aid to comprehending the system. On the other hand, the structure chart we produce cannot be seen as a re-modularisation of the system itself, as tricks and shortcomings may have to be re-introduced in the architectural layer in order to make the modularisation feasible.

2. Recovering the system modular architecture

Several methods have been proposed in the literature to identify modules in existing systems with the aim of remodularising them, i.e. substituting a large program with a functionally equivalent collection of modules. A variety of reasons can trigger system modularisation processes, including the production of reusable assets, improving the quality attributes of a system or migrating it from a mainframe to a client/server platform.

For example, Sneed *et al.* [14] tackle the problem of remodularising large application programs in the context of migration from mainframe systems to distributed architectures. The paper suggests three different approaches to program downsizing (a procedural approach, a functional approach and a data type approach) and describes experience gained in practical applications of the methods. Markosian *et al.* [8] describe a tool for modularising large COBOL programs. The paper tackles the technical problems of constructing the modules but does not suggest any strategy for planning the new modular architecture of the software system. Ning *et al.* [9] propose program slicing as a tool for recovering functionally related code segments from legacy systems so that they can be packaged into independent reusable modules.

None of these methods, however, proposes the reconstruction of the system's original architectural design. As the objective is simply to remodularise code, these methods reconstruct an high level design documentation which is nothing other than a snapshot of the actual implementation, and make no attempt to distinguish the real design decisions from the choices imposed by the implementation platform.

In this section we propose a process for reconstructing the architectural design of large programs in order to get a better understanding of them. The process consists of two phases: first the system modules are identified and the functional relationships between them are derived; then the local data and interfaces are reconstructed for each module.

Next step consists of producing the specification of each module. This requires comprehending what modules do, what each intermodular relationship means and which concepts of the application domain data items implement. These activities cannot be completely automated because, as Biggerstaff *et al.* state [3], the way human-oriented concepts are associated with software components and related relationships is not precisely formalised. Knowledge of the application domain and human expertise play a fundamental role in program understanding.

2.1. Identifying modules and intermodular relationships

The method we propose identifies the system modules by aggregating program units that implement a functional abstraction, and produces the system architecture by deriving the functional relationships that exist between modules [4]. The method is based on functional dependency relationships, which are expressed in terms of dominance relationships [7] between the nodes of a graph describing the activations of program units. We will call such a graph the "call graph" of the program. Two different kinds of intermodular relationships are singled out, namely "uses" and "is composed of" relationships [6].

The identification of modules and their relationships is divided into the following steps:

Step 1. Production of the *Call Graph* and *Dominance Tree*

The Call Graph CG of a program P is a flowgraph CG= (N, E, s), where:

- N= s ∪ PP is the set of the program units and s is the main program;
- E describes the activation relation on (s ∪ PP) × PP.

The existence of direct or indirect recursion between program units produces cycles inside the CG. A cyclic CG can be reduced to a Call Directed Acyclic Graph (CDAG), by collapsing every strongly connected subgraph (i.e. each subgraph containing one or more cycles) into a single node. In fact program units linked by recursion contribute to the implementation of a single functionality and can, therefore, be regarded as a single module. Figure 2.1 shows a sample CG and its corresponding CDAG.

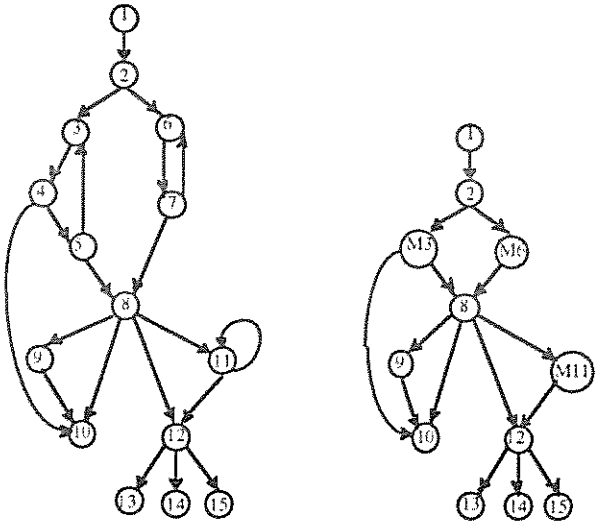


Figure 2.1. A sample CG and the related CDAG

From the call graph it is possible to obtain the dominance tree by computing the dominance relations(1) [7] between the nodes in the CDAG. More specifically, the reflexive and transitive closure of the dominance relation on the CDAG is a tree, called the Direct Dominance Tree (DDT). The dominance tree outlines the hierarchical functional dependences between program units: if a program unit p_1 dominates a program unit p_2 then each activation of p_2 is preceded by an activation of p_1.

The Strong and Direct Dominance Tree, SDDT, is obtained from DDT by marking those edges that link two nodes in a strong and direct dominance relation. The strong direct dominance concept captures a fundamental characteristic of a typical functional dependency between two program units in a software system: if a program unit p_2 is activated only by the program unit p_1, then p_2 implements a subfunctionality of a more general functionality defined by p_1. Figure 2.2 shows the SDDT of the call graph in figure 2.1. Production of the CG, the CDAG and the SDDT can be fully automated. The SDDT is a useful starting point in identifying functional abstractions clustered in whole or partial subtrees.

Step 2. Production of the *Module Tree*

CG and SDDT can be used to reconstruct the set of modules making up the high level architecture of the program P and the intermodular relationships between

(1) *Dominance Relation.* In a CDAG a program unit p_x dominates a program unit p_y if and only if all paths from the root of the CDAG to p_y go through p_x.
Direct Dominance Relation. A program unit p_x directly dominates a program unit p_y if and only if p_x dominates p_y and all the program units that dominate p_y dominate p_x too.
Strong Direct Dominance Relation. A program unit p_x strongly and directly dominates a program unit p_y if and only if p_x directly dominates p_y and p_x is the only program unit in CDAG that activates p_y.

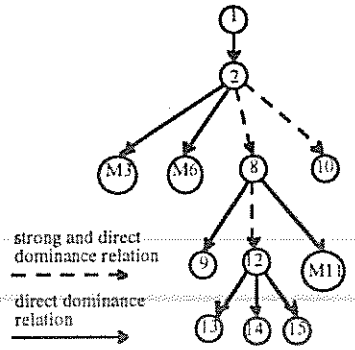


Figure 2.2. The SDDT of the CDAG in Figure 2.1

them. We have already outlined a first rule for the identification of modules on a CG:

a) a strongly connected component of the CG can be considered as a single module implementing a recursive functionality.

The second rule we use to identify modules consists of aggregating whole subtrees each of which implements a single functionality represented by its root:

b) each subtree of SDDT which holds only marked edges generates a module formed by all the program units belonging to this subtree.

Each node of such a subtree, except the root, is a program unit that implements a service that can be accessed only through its own strongly direct dominator node and can therefore be considered a subfunctionality of this dominator. The Reduction of the Strong Direct Dominance Tree, RSDDT, is the tree obtained from SDDT by collapsing each subtree of SDDT having only marked edges into one node. Figure 2.3 shows the RSDDT of the SDDT in figure 2.2.

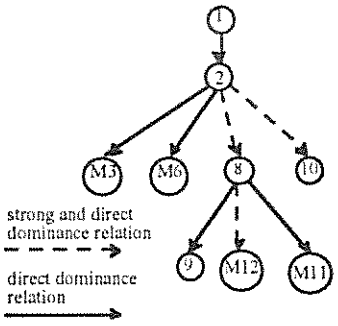


Figure 2.3. The RSDDT of the SDDT in Figure 2.2

Each subtree which holds both marked and unmarked edges may constitute a module implementing a functionality represented by its root. However the subfunctionalities implemented by the dominated nodes cannot be indiscriminately clustered and hidden in this module. In fact the nodes connected to the root by a marked edge represent subfunctionalities that may be considered as components of the more general functionality represented by the root. Vice versa, each node connected to the root by an unmarked edge represents a common service (a functionality used by more than one program unit) and therefore it must be clustered in an independent module:

c) each subtree of RSDDT which holds both marked and unmarked edges generates a module that aggregates the root of the subtree and the leaf nodes representing single program units which are linked to the root by marked edges. Each of the remaining program units defines a module.

The tree obtained by applying rule c) to the RSDDT is called the *Module Tree*. This tree partitions the sets of program units into a set of modules and identifies a hierarchical decomposition of the functionalities of the original system. Finally, direct and strong direct dominance relations between the modules identified on the Module Tree can be interpreted as follows:

d) each of the marked (unmarked) edges of the Module Tree is a candidate to generate an "is_composed_of" ("uses") relation between the software components represented by the nodes that the edge links.

Figure 2.4 shows the Module Tree obtained from the RSDDT in Figure 2.3. The Module Tree shows the functional relations between recovered modules. These relations now have to be completed with intermodular data relations.

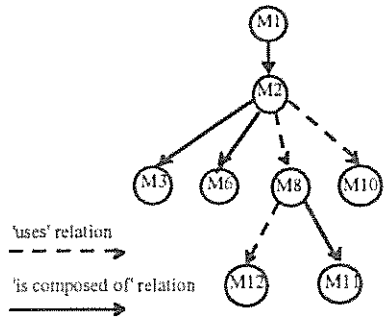


Figure 2.4. The Module Tree from the RSDDT in Figure 2.3

2.2. Identifying local data and interfaces of modules

Aggregating program units into modules and recognising the hierarchical functional relations existing between them is not sufficient to ensure full comprehension of the program, because information has to be gathered on module data coupling. As a first aid to reduce the comprehension effort, we partition the global data set of the program by singling out the set of data items local to each module and the set of data items global to the modules in a subtree of the Module Tree. Using this partition and the call relations between modules, the data items that potentially belong to the module interfaces are also identified. This preliminary decomposition does not take into account intramodular and intermodular data-flow, so we are unable to identify whether one and the same data item is used as local data in different

LOCAL(M1) = ∅	GLOBAL(M1) = {a}	INTERFACE(M1) = ∅
LOCAL(M2) = {c}	GLOBAL(M2) = {b}	INTERFACE(M2) = {a}
LOCAL(M3) = ∅	GLOBAL(M3) = ∅	INTERFACE(M3) = {a, b}
LOCAL(M6) = ∅	GLOBAL(M6) = ∅	INTERFACE(M6) = {a, b}
LOCAL(M8) = {c}	GLOBAL(M8) = {d, f}	INTERFACE(M8) = {a, b}
LOCAL(M10) = ∅	GLOBAL(M10) = ∅	INTERFACE(M10) = {a}
LOCAL(M11) = ∅	GLOBAL(M11) = ∅	INTERFACE(M11) = {b, d, f}
LOCAL(M12) = ∅	GLOBAL(M12) = ∅	INTERFACE(M12) = {d}

Table 2.1. The data sets of the modules of Example 2.1

modules. However, this first decomposition is a useful starting point for software comprehension because it allows for global variables to be dealt with as exchange parameters. Finally data-flow analysis [1] is exploited to refine data sets and interfaces.

Given a flowgraph G= (N, E, s), we denote with N(G) and E(G) the sets of nodes and edges in G, respectively. Let $M = \{m_1, ..., m_n\}$ be the set of modules of the Module Tree and let $P(m_i)$ be the set of program units clustered into the module m_i ($i = 1, ..., n$). We define the relation $MCALL \subseteq M \times M$ as follows:

$$MCALL = \{(m_i, m_j) \in M \times M \mid \exists\, p_h \in P(m_i), p_k \in P(m_j) \text{ such that } (p_h, p_k) \in E(CDAG)\}$$

It is worth noting that p_k must be the root of the module m_j because of the construction of the dominance tree and Module Tree. For each module $m_i \in M$ we also define the sets of immediate successors and predecessors on the relation MCALL:

$$ISUCC(m_i) = \{m_j \in M \mid (m_i, m_j) \in MCALL\}$$
$$IPRED(m_i) = \{m_j \in M \mid (m_j, m_i) \in MCALL\}$$

and the sets of successors and predecessors:

$$SUCC(m_i) = ISUCC(m_i) \cup_{m_j \in ISUCC(m_i)} SUCC(m_j)$$
$$PRED(m_i) = IPRED(m_i) \cup_{m_j \in IPRED(m_i)} PRED(m_j)$$

Similarly, for each module $m_i \in M$, let $MTSUCC(m_i)$ and $MTPRED(m_i)$ be the sets of successors and predecessors of m_i, respectively, on the Module Tree. Finally, we use $DREF(m_i)$ to denote the set of variables referenced in the module m_i and MREF(d) to denote the set of modules referencing the variable d.

For each module m_i we construct the following data sets:

- $LOCAL(m_i)$ is the set of variables local to m_i, i.e., the set of the variables which are only referenced in the module m_i:
  $d \in LOCAL(m_i)$ iff $MREF(d) = \{m_i\}$

Each variable d is considered global to the minimum subtree of the Module Tree that contains modules in which d is referenced:

- $GLOBAL(m_i)$ is the set of variables which are global to the subtree of the Module Tree rooted in m_i:
  $d \in GLOBAL(m_i)$ iff
  1) $d \notin LOCAL(m_i)$ and
  2) $MREF(d) \subseteq (\{m_i\} \cup MTSUCC(m_i))$ and
  3) $\forall\, m_j \in MTSUCC(m_i)$ not $(MREF(d) \subseteq (\{m_j\} \cup MTSUCC(m_j)))$

It is worth noting that each variable $d \in GLOBAL(m_i)$ can be considered local to m_i and can be propagated to each module m_j referencing it through the interfaces of the modules in the call chain that connects m_i to m_j:

- $INTERFACE(m_i)$ is the set of the variables potentially belonging to the interface of m_i, i.e., the variables that a module m_i export/import to/from another module m_j when $(m_j, m_i) \in MCALL$:
  $d \in INTERFACE(m_i)$ iff
  1) $d \in GLOBAL(m_j)$ for some $m_j \in MTPRED(m_i)$ and
  2) either $m_i \in MREF(d)$ or $\exists\, m_k \in (SUCC(m_i) \cap MREF(d))$

The interface of the module m_1, the root of the module tree, coincides with the interface of the original program, if this exists.

Example 2.1. Let us consider the Module Tree in Figure 2.4 and suppose that variables a, b, c, d, e are referenced in the original program:

MREF(a) = {M1, M2, M6, M10}
MREF(b) = {M3, M11}
MREF(c) = {M8}
MREF(d) = {M8, M12}
MREF(e) = {M2}
MREF(f) = {M8, M11}

Table 2.1 shows the recovered LOCAL, GLOBAL and INTERFACE module data sets.

Intermodular data-flow analysis techniques help in understanding the intermodular data relationships, by detecting the module that defines the value of a global variable and the module that uses it, thus enabling the data sets and interfaces of the recovered modules to be refined. In

fact, it is possible that a variable *v*, which is global to a subtree of the Module Tree, is assigned to the INTERFACE set of some module *m* in the subtree. However this variable could be used as a local variable in this module, i.e.:

- every time the value of the variable *v* is used in the module *m* it is also first defined in *m*;
- no other module uses a value of the variable *v* which is defined in the module *m*.

In this case, the variable *v* has to be moved from the INTERFACE to the LOCAL set of the module *m*. Let us consider example 2.1. The variable f has been assigned to the GLOBAL set of module M8 and to the INTERFACE set of module M11. However, if no definition of f in M8 reaches a use of f in M11 and no definition of f in M11 reaches a use of f in M8, then f is used as local variable in both M8 and M11. This means that f is the same name for two distinct data items and then it must be in the LOCAL sets of both modules M8 and M11.

3. Case study

The method for reconstructing the architectural design of a system has been validated by applying it in a set of case studies, the results of which are shown in this section.

The set of case studies consists of five COBOL programs selected from a university's halls and residence information system. The system consists of 103 programs and 90 more Copy and Screen files: the overall size of the system is approximately 200 K lines of code. The software system was designed by adopting a functional decomposition approach and respecting the software reusability requisite: for instance, functionalities that are frequently used in the system are encapsulated in independent programs activated a number of times through CALLS. Table 3.1 shows some structural characteristics for each of the five programs analysed and, in particular, the number of Lines of Code (LOC), the number of Paragraphs/Sections and the number of different programs activated with a CALL instruction.

NAME	L.O.C.	PARAGRAPHS	CALL
AUDIT	536	23	3
CHRGRENT	459	17	6
DAMAGE	1004	32	13
LETTGEN	552	42	6
ROOMADM	1067	43	12
TOTAL	3618	157	40

Table 3.1. Structural Characteristics of the analysed programs

Although our method has been introduced and discussed without making reference either to the coding language or the operating environment of the software systems, it now becomes necessary to make a number of considerations that are typically language-dependent. The characteristics of the language require certain points to be clarified regarding the proposed model but without diminishing the model's generality. Consequently, the first part of this section is devoted to customising the proposed method for COBOL systems, while the second part illustrates the results obtained.

3.1. The architectural design recovery method in COBOL environment

The model proposed for the reconstruction of the architectural design of existing software systems is based on the program call graph. The definition of CG introduced refers to the concept of program units that make up a program. However, while the identification of these program units for languages like Pascal, Fortran and C is almost immediate as they coincide with syntactic structures such as procedures, functions and subroutines, this is not true for COBOL. COBOL does not allow "procedure like" units to be declared inside a program, which is monolithic. Nevertheless COBOL makes it possible, through the Procedure Division segmentation mechanism, for a program to be functionally decomposed into several sections which can in turn be decomposed into a number of paragraphs. The PERFORM verb is used to activate a paragraph or section, or a sequence of paragraphs or sections. In agreement with these considerations, a COBOL program P can be modelled through a call graph defined as CG= (N, E, s), where:

- PP is the set of the performed Paragraphs, Sections and sequences of Paragraphs/Sections of P;
- E describes the activation relation of program units in PP by means of the PERFORM verb.

Defining the set PP as the set of performed paragraphs/sections means that the resulting call graph does not represent transfers of control from one paragraph to another due to the fall-through execution or GO TO statements. Consequently this model faithfully reproduces the execution of those programs in which syntactic mechanisms different from PERFORM have not been used to transfer control between program units, and in which GO TO statements have been used in a structured way, i.e. without producing any jumps outside the performed program unit.

Apart from this characterization of the call graph adopted, the modules and relative intermodular relationships are identified as described above. The way in which the data referenced by each module m_i is distributed among the sets LOCAL(m_i), GLOBAL(m_i) and INTERFACE(m_i) remains unchanged.

3.2. Experimental results

In this section the application of the proposed method is shown with reference to the "LETTGEN" program, one of the five programs analysed. LETTGEN consists of 552 LOC and is structured in 42 paragraphs. The call graph, produced by a commercial code static analyser, and the related dominance tree have been used to recover the architectural design. Figures 3.1 and 3.2 show the call graph and the recovered Module Tree respectively.

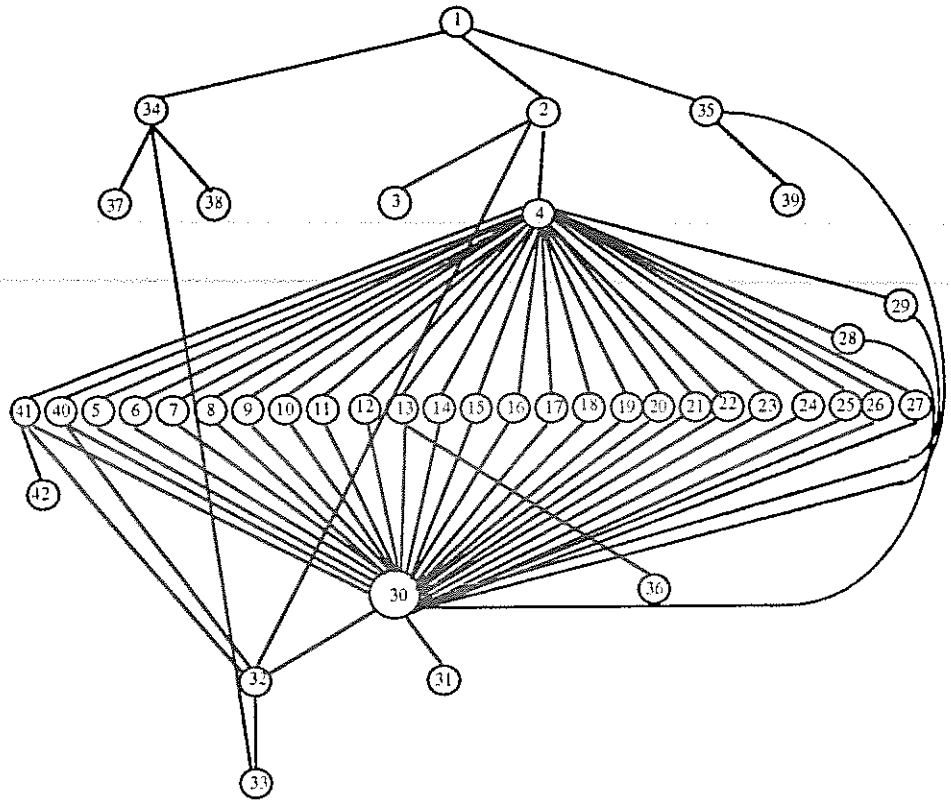


Figure 3.1. The CG for the LETTGEN program

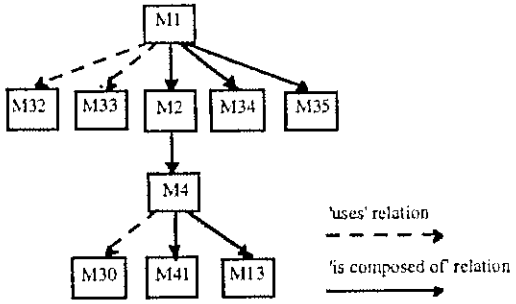


Figure 3.2.The Module Tree for the LETTGEN program

MODULE COMPOSITION
M1= (1)
M2= (2, 3)
M4= (4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,20, 21,22,23,24,25,26,27,28,29,40)
M13= (13, 36)
M30= (30, 31)
M32=(32)
M33=(33)
M34= (34, 37, 38)
M35= (35, 39)
M41= (41, 42)

Table 3.2. The composition of the modules in Figure 3.2

In Figure 3.1 each performed paragraph/section has been identified with a progressive number, while the number associated with each module in Figure 3.2 corresponds to the root of the subtree clustered in the module itself. In Table 3.2 the composition of each module is shown. The subsequent step is dedicated to the reconstruction, for each module m_i, of the sets DREF(m_i), LOCAL(m_i), GLOBAL(m_i) and INTERFACE(m_i). Table 3.3 shows the GLOBAL, LOCAL and INTERFACE data sets for each module.

This is followed by the reconstruction of a *data dictionary* which is achieved by assigning a meaning in the application domain to each data item. Meaning assignment is a process that cannot be easily formalised as it is founded on groups of clues and hints hidden in the code, such as the choice of variable names, the position of the declarations, the comments inserted by the programmer, etc.. It often happens that logically correlated data have declarations that are physically close in the code and their names are assonant with the nomenclature adopted in the application domain. For example, Figure 3.3 shows the declaration of the data item describing the identification number of a college room contained in the Working-Storage Section of the LETTGEN program. The use of names that are very close to the terminology of the application domain makes it easy to understand the meaning of the 'normalised-room-number' data item.

M	INTERFACE(M)	LOCAL(M)	GLOBAL(M)
M1	----	----	letter-ptr, letter-record, letter-file, text-file, master-record, input-room-number, error-parameter, input-address-lines, text-ptr, text-length, system-parameter-block
M2	letter-ptr, text-file, text-ptr, text-length	diff	----
M4	letter-ptr, letter-file, text-record, text-ptr, master-record, input-room-number, input-address-lines, error-parameter, text-length	text-parameter, word-buffer, full-offer-room, period-count, temp-date, temp-value, temp-amount, para-ptr	missing-record, assembly-buffer
M13	assembly-buffer, system-parameter-block	full-date	----
M30	letter-ptr, letter-record, assembly-buffer	buffer-ptr, len	----
M32	letter-record, letter-file	----	----
M33	letter-ptr, letter-record	----	----
M34	letter-ptr, letter-record, letter-file, master-record, input-room-number, text-file, error-parameter, input-address-lines, text-ptr, text-length, system-parameter-block	ls-text-file-name, eof-flag, text-file-name, letter-file-name	----
M35	letter-record, letter-file, text-file	----	----
M41	letter-record, letter-file, master-record, missing-record	----	----

Table 3.3. Data set partitions for the LETTGEN program

```
1     normalised-room-number.
 3    nrn-main.
  5   nrn-block                PIC XX.
  5   nrn-room-no              PIC 999.
 3    nrn-place                PIC 9.
```

Figure 3.3. A sample code fragment

MODULE	FUNCTIONALITY
M4	Prints a line of a letter
M13	Converts and prints the current date
M30	Deletes blanks from the tail of a string
M41	Reads the master file and prints the letter body

Table 3.4. Summary of the functionalities for module M4

It was now possible to tackle the problem of identifying the functionality performed by each module and to draft a series of functional specifications in natural language for each of them. This required the specification of the functionality performed by each subtree in the Module Tree, abstracting it from the ones performed by its component modules. This procedure was performed using a *bottom-up* technique: first the functionality performed by the deepest level subtrees were identified, starting from the leaf modules; this gradually made it possible to rise in level up to the root of the Module Tree. This reduces the comprehension effort as the process is carried out on a quantity of code that is smaller than the whole program; moreover, the comprehension process can also be carried out in parallel for the various subtrees. Table 3.4 describes the functionalities obtained for the modules belonging to the subtree with the root M4, and the global functionality associated to the M4 module itself.

The experiment effectively shows the validity of the proposed method. This can be seen by the fact that once the Module Tree has been obtained and the data has been partitioned among the various modules, the process for the assignment of meaning to each data item and module was considerably facilitated compared to the effort that would have been required if the process had been conducted on the whole system as a single entity. It was also possible to conduct this process in parallel by suitably pruning the Module Tree. The experiment showed that pruning the Module Tree vertically (i.e., depth first visiting several subtrees of the Module Tree in parallel) was more effective than horizontal pruning.

4. Conclusions

The work described in this paper addresses the problem of understanding a software system and focuses in particular on the comprehension of the system architectural design. A method has been presented to reconstruct the architecture of a system and represent it in the form of a structure chart which we call module tree. The method assumes the system was originally designed with a functional decomposition approach, and attempts to aggregate language program units into modules whenever these implement a functionality of the system. The system is represented in the form of a directed graph describing the activations of program units, and the concept of node dominance on a directed graph is exploited to aggregate program units into modules, and to derive intermodular relationships from the unit activations. Two types of intermodular relationship are singled out, namely the "uses" and "is composed of" relationships. Finally, the system data set is partitioned into sets of data items which are local to a given module and sets of data items which are global to the modules belonging to a subtree of the structure chart. The analysis of both these partitions and the activations of the program units forming a module allows the set of data items potentially belonging to the module interface to be identified.

The architectural design this method produces does not reflect either the characteristics of the coding language or the limitations of the operating environment. While this is an important feature in the comprehension of the system, it implies that the recovered architecture cannot be directly assumed as a basis to re-modularise the system. In fact, remodularising the system requires firstly mapping the module local data set and interfaces onto the programming language, and then possibly modifying the recovered architectural design to take into account the limitations of the operating environment.

For example, if the aim is to modularise a COBOL system on the basis of the architectural design recovered by our method, the sets of local, global and interface data items have to be mapped onto File Section, Working Storage Section and Linkage Section. The following two equations define the Working Storage Section - denoted $WS(m_i)$ - and the Linkage Section - denoted $LK(m_i)$ - of each module m_i belonging to the module tree of a COBOL system:

$$WS(m_i) = LOCAL(m_i) \cup GLOBAL(m_i)$$
$$LK(m_i) = (DREF(m_i) \cup_{m_j \in ISUCC(m_i)} LK(m_j)) \setminus WS(m_i)$$

Of course, each module will have its own Input Output Section and File Section in which the files will be described. To illustrate the problem deriving from the limitations of the operating environment, let us consider the case in which the recovered architecture includes modules that access one and the same file. This could be a problem whenever the operating environment does not allow files to be shared among modules, as in the case of OS/VS COBOL.

Acknowledgements

We wish to thank professors Aniello Cimitile and Ugo De Carlini for their precious suggestions.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, "Compilers, Principles, Techniques, and Tools", *Addison-Wesley*, 1986.

[2] P. Antonini, P. Benedusi, G. Cantone and A. Cimitile, "Maintenance and Reverse Engineering: Low-Level Design Documents Production and Improvement large application programs", *Proc. of Conference on Software Maintenance*, Austin, Texas, IEEE Comp. Soc. Press, 1987, pp. 91-100.

[3] T.J. Biggerstaff, B. G. Mitbander and D. Webster, "The Concept Assignment Problem in Program Understanding", *Proc. of the 15th International Conference on Software Engineering*, Baltimore, Maryland, 1993, pp. 482-497.

[4] A Cimitile and G. Visaggio, "Software Salvaging and The Dominance Tree", Int. Rep. PF-CNR "SICP" Sp6R77, 1992, Dep. of "Informatica e Sistemistica" - University of Naples. To appear on *The Journal of Systems and Software*.

[5] J.H. Cross II, "Reverse Engineering Control Structure Diagrams", *Proc. of Working Conference on Reverse Engineering*, Baltimore, Maryland, IEEE Comp. Soc. Press, 1993, pp. 107-116.

[6] C. Ghezzi, M. Jazayeri and D. Mandrioli, "Fundamentals of Software Engineering", *Prentice Hall Pub.*, 1991.

[7] M.S. Hecht, "Flow Analysis of Computer Programs" *Elsevier North-Holland*, New York, 1977.

[8] L. Markosian, P. Newcomb, R. Brand, S. Burson and T. Kitzmiller, "Using an Enabling Technology to Reengineer Legacy Systems", *Comm. of ACM*, vol. 37, no. 5, May 1994, pp. 58-70.

[9] J.Q. Ning, A. Engberts and W. Kozaczynski, "Recovering Reusable Components from Legacy Systems by Program Segmentation", *Proc. of Working Conference on Reverse Engineering*, Baltimore, Maryland, IEEE Comp. Soc. Press, 1993, pp. 64-72.

[10] A. Quilici, "A Memory-Based Approach to Recognising Programming Plans", *Comm. of ACM*, vol. 37, no. 5, May 1994, pp. 84-93.

[11] V. Rajlich, N. Damaschinos, P. Linos and W. Khorshid, "VIFOR: A Tool for Software Maintenance", *Software Practice and Experience*, vol. 20, no. 1, January 1990, pp. 67-77.

[12] C. Rich and L. M. Wills "Recognizing a Program's Design: A Graph-Parsing Approach", *IEEE Software*, vol. 7, no. 1, Jan. 1990, pp. 82-89.

[13] H.M. Sneed and G. Jandrasics, "Software Recycling", *Proc. of Conference on Software Maintenance*, Austin, Texas, IEEE Comp. Soc. Press, 1987, pp. 82-90.

[14] H.M. Sneed and E. Nyary, "Downsizing Large Application Programs", *Proc. of Conference on Software Maintenance, Montreal*, Canada, IEEE Comp. Soc. Press, 1993, pp. 110-119.

A Documentation-related Approach to Object-oriented Program Understanding

L.H. Etzkorn and C.G. Davis

The University of Alabama in Huntsville

Abstract

Object-oriented code is considered to be inherently more reusable than functional decomposition code; however, object-oriented code can suffer from a program understanding standpoint since good object-oriented style seems to require a large number of small methods. Hence code for a particular task may be scattered widely. Thus good semantics based tools are necessary. This paper describes an approach to object-oriented code understanding that focuses largely on informal linguistic aspects of code, such as comments and identifiers.

Keywords: program understanding, object-oriented software, software reuse, software maintenance.

1. Introduction

The use of the object-oriented paradigm should result in the goals of software engineering such as maintainability and reusability being more easily achieved. This is due to the encapsulation, inheritance, and polymorphism aspects of the object-oriented paradigm. Studies have tended to show that reusability [7], and maintenance [6] are better achieved with object-oriented code than with functional decomposition code. However, object-oriented code, due to the very aspects that make it desirable, tends to suffer from the wide scattering of the code that performs a particular (even a fairly simple) task. It is considered to be good object-oriented programming style to write small member functions [10]. This results in an object-oriented system consisting of a large number of small modules. Also, by the use of inheritance, a class may inherit one or more classes, with associated methods. Often, these inherited classes may be inherited by more than one derived class, and are not always defined locally. This tends to underline the need for good, semantically-based tools for object-oriented code [12]. Some structure-oriented tools for C++, for example, do exist. For example, Together/C++ (by Object International, Inc., Austin Texas) provides an object model in one window, with associated C++ code in another window. If one window is edited, then the other window is updated automatically. This type of tool has uses both in object-oriented software development, and in object-oriented software maintenance and reuse. However, there is still a need for domain-based understanding tools that would provide more than just the structure of the code, but also a domain-based understanding of the program concepts.

2. Background

Our primary research area is the metrics-based automated extraction of reusable components from object-oriented code. A workable program understanding approach is an integral part of the extraction process. There have been several knowledge-based approaches to program understanding. These approaches differ in the understanding methodologies, and in the program specifications produced. Some of these approaches are more automated than others -- most require user intervention at some point. Caldiera and Basili [4] provide an identification methodology that automatically extracts candidates for reuse. Then, in a following component qualification phase, a domain expert analyzes and records the meaning of each candidate component. An interactive tool (the specifier) employed by the domain expert assists in the formal specification of the candidate component. Our approach is similar to Caldiera and

0-8186-5647-6/94 $04.00 © 1994 IEEE

Basili's in that both approaches are metrics-based, and both include an automatic candidate component extractor. However, our understanding phase, instead of attempting formal specification, provides a heuristic, knowledge-based, concept-recognition approach to program understanding of the candidate components. Our system is called the PATRicia (Program Analysis Tool for Reuse) system. The concept-recognition module is known as the CHRiS (Conceptual Hierarchy for Reuse employing Semantics) module. A block diagram of the PATRicia system is provided in Figure 1. The rest of our discussion is related to the approach employed by the CHRiS module.

Typically, the more automated versions of the heuristic program understanding approaches consist of identifying typical program components as being comparable to those program component plans located in a program-knowledge base, and deriving a specification that is based on plans associated with the program component plans that were identified. Rich and Wills [13] take a graph-parsing approach to program understanding. In this approach, the program is first transformed into a flow graph. This flow graph is then parsed using programming-knowledge plans (grammar rules stored as graphs), and a design tree (consisting of program components) is produced. A natural language description associated with the programming-knowledge plans is then produced. Ning and Harandi [5] [11] take a heuristic-based concept recognition approach.This approach uses programming-knowledge plans, and a mechanism called interval logic, to recognize concepts at various levels (structure-level concepts, function-level concepts, etc.), and to formulate abstract concepts at higher levels. A natural language description associated with the programming-knowledge plans is then produced. More recently, Kozaczynski, Ning, and Engberts [8] used a transformation approach to formulate higher level concepts from lower level concepts. Low level concepts are represented by abstract syntax trees. Production rules are used to recognize language concepts. These program concepts are combined to form more abstract concepts. Note that all of these approaches (except Caldiera and Basili's, which depends on a human domain expert), look only at the code itself (or a version of the code translated into an intermediate form), and ignore the comments, identifiers, and related documentation. Biggerstaff, Mitbander, and Webster [2] [3] take a different approach that employs informal information -- keywords embedded in comments, identifier names, and design documents -- as well as code analysis. In this system a user would first identify a suggestive identifier name. Then the user requests a browser view of all functions related to this identifier. The identifiers from these functions give additional information. Various other tools provide a pattern of relationships that gives additional information. A concept assignment can then be applied. Further analysis can then occur using this identified concept. This approach can be applied easily to domain-related concepts, whereas the earlier approaches discussed (except Caldiera and Basili's, which relied on a domain engineer) tended to identify primarily programming-oriented concepts, or algorithms. Biggerstaff claims that there is a paradigm-shift when moving from programming-oriented concepts to domain- or human- related concepts, and not just a simple aggregation of programming concepts to form a higher level domain concept.

3. Object-oriented program understanding

The proposed approach for object-oriented code understanding employs methods derived from the Biggerstaff approach, as well as the programming concepts approach. We think that a better concept recognition engine could be built that would employ comment and identifier recognition initially to derive initial tentative concept understanding, then apply a programming-concept recognition approach to identify programming-concepts related information within the previously tentatively identified concept. (Note that Biggerstaff [2] [3] has recommended a similar approach). The secondary understanding cycle would serve multiple purposes. First, it would enable partial-filling of previously partially understood concepts. Second, it would serve as a check to make certain that the initial tentative concept understanding had not been mistaken. The primary goal is to achieve, as nearly as possible, a totally automated understanding system, with very little human input required. This is beneficial in our primary research area of interest, the automatic extraction and categorization of reusable code components. If good automation could be achieved, and intermediate results

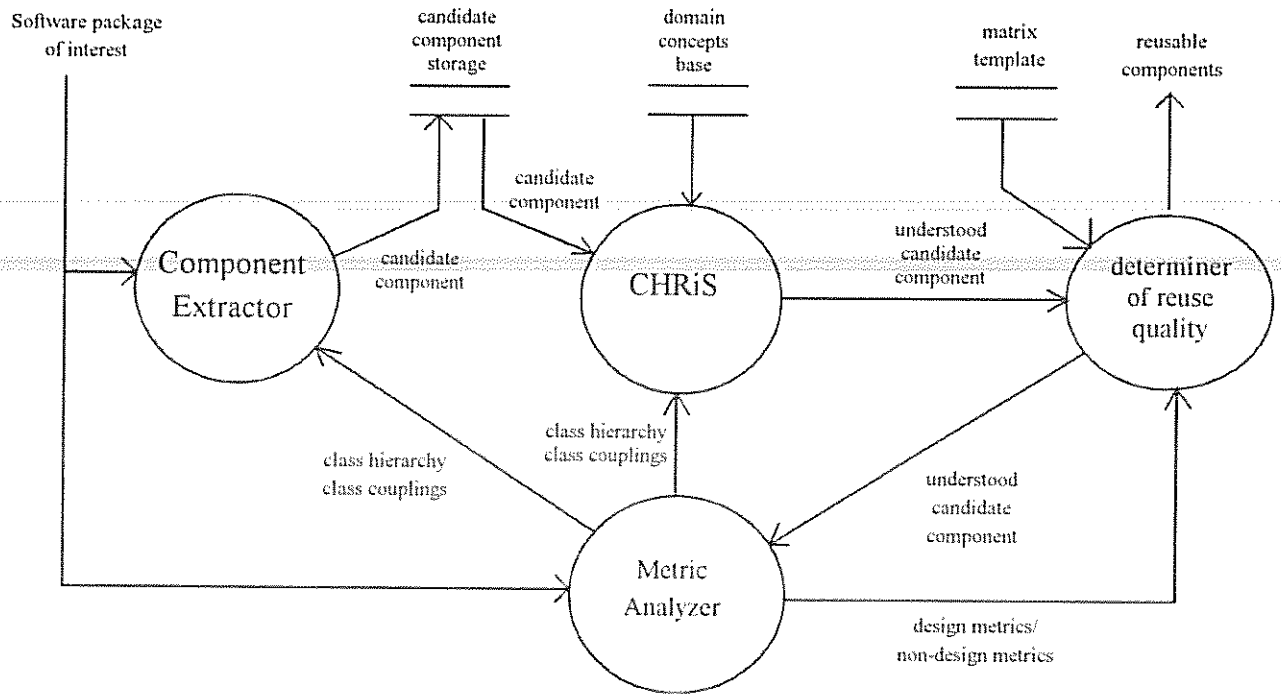


Figure 1. PATRicia System

could be reported, then this process could operate in the background, and any extra processing time required would not be as important. Also, when discussing object-oriented reuse, most of the programming-related concept understanding would be performed on member functions, which are typically short [10]. Thus some intermediate results could be reported fairly quickly. Some of the issues involved in combining the two understanding approaches are:

1) Comment/identifier understanding (in an object--oriented system, see later discussion) is inherently top-down, whereas programming concept understanding is inherently bottom up. What is the best way to combine the two approaches?

2) In the case of confusion or disagreement, how much weight should be given to one method over the other?

3) There is a conceptual leap between comment/identifier understanding and programming concepts understanding (human concepts versus programming concepts) [3]. What is the best way to bridge this gap?

However, in object-oriented code, more so than in functionally oriented code, much understanding can be performed simply by looking at comments and identifier names alone. This is true since object-oriented code is organized in classes, with everything required to implement a class at least mentioned (if not defined) in the class definition. Thus by a combination of the class hierarchy (easily derived), and identifier names and comments associated with each class, an at least partial understanding of what that class does can be derived more easily than in functionally oriented code. The primary reason for this is that a class itself is a more-or-less abstract concept. Thus, instead of saying "what possible (more or less) abstract concepts can I build from this set of understood low-level concepts", you say instead (in many cases) "here is a concept -- I think I understand it based on my domain knowledge -- do the low level concepts it contains agree with my tentative concept assignment?" This has two advantages. In the first place the set of concepts that make up the class is limited. You know that no concept associated with another class that is not in this class' hierarchy (unless a friend method or an attribute of a type which is an abstract data type that happens to be another class) could

possibly be part of the makeup of this class. In other words, some object-oriented design metrics, such as the inheritance hierarchy and various object-oriented coupling metrics (such as coupling through abstract data types and coupling through message passing [9]) could assist in the understanding process. In functionally oriented understanding the subconcepts that make up a concept would not be as easily identified. Dependencies between concepts would have to be derived in a more complicated manner, such as control flow graphs, etc. (This might still be necessary within portions of object-oriented code, primarily in the member function understanding, but it is not necessary, in many cases, to achieve a high level understanding of a class). Another advantage of object-oriented class hierarchies, is that after a particular class has been identified, a shrinking of the domain occurs when attempting to understand derived classes. This can result in a quicker and more accurate understanding of the derived classes. Thus, in many cases, an almost top-down approach to object-oriented understanding can be taken.

This paper will concentrate primarily on comment and identifier understanding.

3.1 Comment understanding

Comment and identifier understanding has the advantage that not so much time need be spent in setting up plans in a knowledge base as is required when setting up code recognition plans. For example, you need not concern yourself with iteration. Thus, instead of combining two nested iteration events (loops) and a swap nested within the two iteration events to form a recognizable bubble sort algorithm, you would recognize the words "bubble sort". Thus comment understanding would start at an abstraction level higher than code understanding. In Biggerstaff's system, primarily keywords were searched for in comments. In [2] he reported use of a "linguistic idiom", which represents the expected linguistic form of a conceptual abstraction. He mentioned that this linguistic idiom might be implemented as a set of regular expression patterns that match the previous natural-language forms in source code identifiers or comments. We have made an analysis of different types of common comments and identifiers, and based on this analysis, we believe that this idea could be taken one step further, and a full (though limited) natural language parser could in some cases (such as the analysis of header blocks of subroutines) give better results. After the parsing of the comments associated with the code, this could be useful in acquiring additional information from any external documentation (that happened to be machine-readable). This would take the form of a keyword and previously-identified abstract concept search through the external documentation, followed by application of the natural language parser to the applicable area within the external documentation.

Having looked at typical comments, we can conclude:

1) comments are almost always written in the present tense, with either indicative mood or imperative mood. For example:

Present tense, indicative mood:
This routine reads the data.

Present tense, imperative mood:
Read the data.

This restriction makes a natural language parser much simpler, since it need not handle past tense, present perfect tense, etc.

2) The set of verbs typically used for comments is much restricted over the set of all English verbs (although varying with domain).

For example:

Verbs often used in comments:
is, uses, provides, implements, accesses, prints, inputs, outputs, reads, writes, supplies, defines, retrieves, gets, etc.

Verbs seldom used in comments:
smiles, frowns, laughs, rides, flies, jumps, sings, fights, electrocutes, falls, punishes, hires, fires, pats, throws, pitches, calms, etc.

The set of comment verbs, while still very large, is still in general much smaller than the set of natural language verbs. Also, since the domain will be restricted, the

selection of verbs to support becomes much easier. Still, although our goal is to understand as much as possible automatically, it becomes clear that not all comments can be parsed.

3) Comments tend to be certain types:
A) Header block comments
1) operational description

a) This item does (or is used for) something.
For example:
This routine reads the data..

b) ItemName does (or is used for) something.
For example:
READ_DATA reads the data.

Both a) and b) are often followed by more sentences related to the operation of the item.

For example:

/* READ_DATA reads the data. It opens the file for read, then it reads the data a character at a time. */

2) definition

a) ItemName -- definition.
For example:
/* GeneralMatrix -- rectangular matrix class */

b) definition
For example:
/* A rectangular matrix class */

Both a) and b) are occasionally followed by more sentences related to the operation of the item.

Note that header block comments are normally considered to refer to the construct immediately following.

B) Inline comments

1) operational description
For example:

// Get matrix row
// Get matrix column

2) definition
For example:

/* index variable */
/* counter of incoming characters */

Inline comments also work by the principle of locality, in that they typically refer either to the code line they are appended to, or to a line or lines of code immediately following. Less commonly they refer to a previous line.
C) File header blocks
For example:
/* Routines for reading data from files. This includes ASCII routines and binary routines. */

File header blocks apply less of the principle of locality than other comments. If parsed correctly, they could be used in restricting the domain for analysis of the later comments in the file. However, often only the fact that this header block is located first in the file allows us to infer that it is a file header block. It can be difficult to distinguish this from a header block for the first construct within the file.

One of the primary requirements is to apply the comment to the correct section of code, so that the comment can be matched to the appropriate identifiers (and to the understood code, when the code understander is later written). Typically the principle of locality will be used for this. A check of the comment versus the identifier is usually necessary.

In several cases the natural language parser can be useful -- primarily in the header blocks, although some use could occur in inline operational description comments. In the case of poorly commented code, the identifier handler would have to operate largely alone. (We can make the observation that code, in order to be easily reusable, should be well commented, and thus the fact that this system will work better on better commented code is perhaps a reasonable assumption for our primary research area). We believe that a fairly simple natural language parser, such as the augmented transition network described in [14] will be sufficient.

The identifier handler will work similarly to that of

Biggerstaff [2] [3]. The trick here is to first, have the concept to be recognized in the knowledge base, then have as many reasonable variations (abbreviations) of the name as possible also in the knowledge base. For example, consider a bubble sort routine. Various identifiers might be: BubbleSort, Bubble_Sort, Bubble_sort, Bsort, B_sort, bubble, BublSrt, etc. In some cases, the identifier might be different from that expected, and a comment would be used (hopefully) to match the identifier to the concept. For example, consider the previously mentioned GeneralMatrix class (/* GeneralMatrix -- rectangular matrix class */). One might find, instead:

```
/* GMat -- rectangular matrix class */
class GMat : public Matrix
{
 public:
        GMat(); //constructor
        .
        .
        .
        real Trace();
        void GetNextRow();
        void GetNextCol();
}
```

In this case, the identifier used would have to be created from the preceding comment. This example also can illustrate some of the benefits (as well as difficulties) of this approach. For example, consider the above without any good identifier names:

```
/* JJ -- rectangular matrix class */
class JJ : public KK
   {
        public:
                JJ();     //constructor
                .
                .
                .
                real LL();
                void MM();
                void NN();
   }
```

This illustrates the importance of the comment/identifier approach for most code. A programming concept understanding system would basically have this type of code input. It would then have to interpret the LL(), the MM(), and the NN() routines by identifying the code in the functions. In the comment/identifier approach, we are able to arrive at a high level understanding without the necessity of even looking at the code for the Trace (LL), for example. However, this also shows some of the limitations of this approach. If poor comments and identifiers are used, the programming concept system would still work. However, the comment/identifier approach would not. This is one reason that a combined approach would be best, each providing different aspects of understanding. The combined approach is also closest to how a human expert would try to understand the code. First he would look at documentation, comments and identifiers, and mentally tentatively classify the code. Then he would examine the code and compare it with his mental image of what the code should be doing. Based on information from both code and documentation he would make a final code classification decision.

4. Summary

In summary, we believe that, in our environment (the extraction of reusable components from object-oriented code), this approach has definite advantages. First, the combined approach, both comments/identifiers and code, leads to a more certain understanding decision. Second, semantic-based tools in general are necessary for object-oriented code, due to the scattering effect possessed by object oriented code. The inheritance hierarchy of object-oriented code, due to the domain shrinking effect as one descends in the hierarchy, enables an automated understanding approach to succeed more easily. The comments/identifiers approach is very useful for object-oriented code understanding, since so much of what is necessary to understand a class is present in the class definition, whereas understanding a method alone (which is what would primarily be done by programming concept understanders) does not provide the full understanding of the class. A system based on this approach can provide an integral part of an automatic reuse extraction system. Additionally, it could provide a good maintenance tool for object-oriented systems.

References

[1] Basili,V. and Abd-El-Hafiz, S.K., "Packaging Reusable Components: The Specification of Programs", University of Maryland at College Park, CS-TR-2957, UMIACS-TR-92-97, September 1992.

[2] Biggerstaff,T. ,"Design Recovery for Maintenance and Reuse", IEEE Computer, Vol. 22, Iss.:7, p. 36-49, July 1989.

[3] Biggerstaff, T., Mitbander, B., Webster, D., "Program Understanding and the Concept Assignment Problem", Communications of the ACM, Volume 37, Number 5, pp.72-82, May 1994.

[4] Caldiera, G. and Basili,V. "Identifying and Qualifying Reusable Software Components", IEEE Computer, pp. 61-70, February 1991.

[5] Harandi, M.T., and Ning, J.Q., "Knowledge-Based Program Analysis", IEEE Software,pp. 74-81, Jan. 1990.

[6] Henry, Sallie M. and Humphrey, Matt "A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software", Proceedings of the IEEE Conference on Software Maintenance, pp. 258-265, 1990.

[7] Kernighan, B.W. "The Unix System and Software Reusability", IEEE Transactions on Software Engineering,Vol. SE-10, Number 5, pp. 513-518, September 1984.

[8] Kozaczynski,W., Ning, J., and Engberts, A., "Program Concept Recognition and Transformation", IEEE Transactions on Software Engineering, Vol.:18, iss:12, pp.1065-1075, December 1992.

[9] Li, W., and Henry, Sallie, "Object-oriented Metrics that Predict Maintainability", The Journal of Systems and Software, Volume 23, Number 2, pp. 111-122, November 1993.

[10] Lieberherr, Karl J. and Holland, Ian M., "Assuring Good Style for Object-Oriented Programs", IEEE Software, Volume 6, Number 5, pp. 38-48, September 1989.

[11] Ning, J.Q., A Knowledge-Based Approach to Automatic Program Analysis, doctoral dissertation, University of Illinois at Urbana-Champaign, Urbana, Ill., 1989.

[12] Rajaraman,C. and Lyu, M. "Reliability and Maintainability Software Coupling Metrics in C++ Programs", Proceedings of the Third International Symposium on Software Reliability Engineering, pp.303-311, October 7-10, 1992.

[13] Rich,C. and Wills, L., "Recognizing a Program's Design: A Graph-Parsing Approach", IEEE Software, pp. 82- 89, January 1990.

[14] Winograd, T., Language as a Cognitive Process, Addison-Wesley, 1983.

Layered Explanations of Software:

A Methodology for Program Comprehension

Vaclav Rajlich, James Doran, Reddi.T.S.Gudla

Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
vtr@cs.wayne.edu

Abstract

In dealing with the legacy systems, one often encounters poorly documented and heavily maintained software. Lack of understandability of these systems complicates the task of software maintenance, making it time consuming and limiting the possibilities of the evolution of the system. We present a methodology that helps the programmers to understand programs. Our approach is compatible with the "top-down theory" of software understanding, where the programmer creates a chain of hypotheses and subsidiary hypotheses, concerning the properties of the code. Then he/she looks for evidence (beacons) in the code. Our approach shortens the process of hypotheses creation and verification, and allows recording of successful hypotheses for the future maintenance. All information needed for understanding is recorded in layers of annotations. An experiment was conducted to investigate how the proposed methodology helps in program understanding. A tool supporting the methodology, is presented.

1. Introduction

The first step in software maintenance, which is often the hardest, is simply understanding the code as it exists today. The software engineering field has a number of methodologies and tools that help programmers in this task, but as code complexity increases, so must a methodology's capabilities. This paper describes a methodology and an associated tool that facilitates the comprehension of programs. It is based on the cognitive theories known from the literature. The paper also describes an experiment conducted to investigate how the proposed methodology facilitates the ability of programmers' to understand programs.

Before presenting our approach to program comprehension, we need to introduce some basic terminology and facts. First, program comprehension is the act of understanding a program on all levels; that is, understanding what it does conceptually as well as the fine details of the source code [1]. Some estimate the time spent on understanding a program alone to be 50 to 90 % of the total time to maintain it. Although significant progress has been made in this field, there is still a need for continued investigation.

Our work is related to the work presented in [4]. In [4], Brooks maintains that anyone who tries to comprehend a program makes certain assumptions or hypotheses based on both acquired and existing knowledge. The hypotheses are checked against the source code to prove their validity. The methodology presented in this paper, Layered explanation of software and the associated tool, the Tool for Layered Explanation of Software (or "TLES" for short), exploits the Brooks' theory of program comprehension. TLES identifies constructs of code that need to be be explained; for example, a class in a C++ program is such construct. It organizes explanations of these constructs into layers. Each layer provides a summation of the decisions that the original analyst used when designing and developing the system.

In Section 2, we present a discussion of the top down, bottom up, and flat comprehension theories. Section 3 presents ideas and decisions behind TLES. Section 4 presents the experiment and the results. Section 5 describes the TLES tool. The appendices present details of the experiment.

2. Program comprehension theories

In [4], Brooks postulates that the programmer constructs a series of hypotheses to arrive at an

0-8186-5647-6/94 $04.00

understanding [4]. He argues that the original developer builds a program by initially gathering facts from a problem domain. After a firm understanding of the problem domain, the developer starts to apply or refine that domain towards a computer program by making a series of decisions which add multiple layers (e.g., algorithmic, representation etc.). In short, the software engineer takes a top-down approach starting with the most abstract concepts, refines them again and again, and transforms them into a computer program. The maintenance programmer performs a similar task. He reconstructs the domains and mappings as he tries to comprehend a program.

Brooks based his theory on a related research in the cognitive science. He puts forth the idea that people learn by creating hypotheses which shrink the domain of the problem space (i.e., narrow down the problem). As an example, "sort numeric list" suggests a great deal about the structure of a program. From this requirement, we know that we must create a data structure that simulates a list (e.g. linked list), we know the type of list (numeric), and we make assumptions on the algorithm that could be used for the sort. We know if the list is not very long, a simple bubble sort would do. Longer lists suggest a binary sort or something more complex. Also, there should be an input and output of the list. All of these subsidiary hypotheses are based on the original requirements.

In the process of reconstructing the program, the maintenance programmer creates hypotheses based on what he or she has learned from the documentation and source code. These hypotheses lead to more and more refined hypotheses which create a tree-like structure where each node is an assumption. Naturally, the programmer will make incorrect hypotheses from time to time. The hypothesis tree is then traversed until the correct hypothesis branch is discovered. Hypotheses are verified through the use of segments of code known as beacons. Beacons are programming statements that help maintenance programmer prove hypothesis true or false. For example, if a programmer is tracing through a program which deals with a large database, he might assume the access to the database is indexed. Upon examining procedure names, the title "Read_Key_Field" is located. This procedure name serves as a beacon to the programmer by indicating to him that the access to the database is indexed as he thought. Beacons have been studied experimentally and proven beneficial when attempting to understand programs [10].

The concept of bottom-up comprehension is discussed in [9]. The idea presented is programmers learn by focusing on small pieces of code (perhaps as small as one line of source code) and later combine this information together. The result is the comprehension of a larger and larger part of code. For example, a maintenance programmer traces through a procedure named Binary_Sort and a procedure named Match_Item_In_List. He knows that another procedure calls Binary_Sort with a list and the output is passed to Match_Item_In_List. From this knowledge, he reasons that the procedure which calls the two subfunctions must perform a search on a sorted list.

The third type of comprehension (flat comprehension) does not attempt to organize itself in any direction (from top to bottom or vice-versa). Current implementations of hypertext, like Neptune [9], generally follow a flat comprehension structure. Neptune is a CASE tool that runs a Hypertext Abstract Machine (HAM). HAM provides analysts with definition, modification, or retrieval of basic hypertext units, i.e. nodes and links. A node stores elements of information and links connect it to other nodes. For example, HAM could link a paragraph in a requirements specification to the module of source code where the specific requirements are implemented. The nodes that link all of the information together are not organized in a top-down or a bottom up fashion. Similarly, Horowitz and Williamsons' hypertext-like application SODOS also follows a flat structure of organization [8]. SODOS stores documents from each phase of the software life cycle in a database and links them together.

3. Layering principle

TLES provides an evolutionary history of the constructs of the program. Every programming construct, whether a variable, class, statement, an object, etc. has its origins in decisions of the programmer. A variable, for example, may represent an entity of the problem domain. As a variable progresses through successive refinements and design decisions to the source code, it undergoes a transformation: its scope is limited, its type is defined, it is expanded into a structure, etc. TLES allows the maintenance programmer to view this series of transformations (known as knowledge domains) by linking a number of layers of explanation to each construct in the program. Each of these layers relates to one of the domains used to define it.

In comprehending a program's source code, Brooks [4] points out that the maintenance programmer constructs a hypothesis tree with the series of decisions representing branches on the tree. TLES develops this idea further. It looks at the tree and horizontally groups the nodes into layers.

Even though most of the constructs undergo a transformation in every layer, some of the constructs will

remain unchanged. TLES accommodates the fact that some variables and expressions in programs are created in layers closer to actual implementation or that certain constructs remain unchanged from one layer to another. As an example, say that a programmer created a variable for an application called "timeout_counter", that starts counting when the system is inactive. After it reaches a certain number, the system closes down intuiting that the execution is deadlocked. This particular variable would not be represented at a highly conceptual layer because it serves mostly as a security constraint on the system. If the application was a banking application, timeout_counter would not represent an actual entity of the financial world. Its concept is closer to the implementation layer; therefore, this variable would not exist in the abstract (i.e., "higher") layers of TLES.

The layering principle allows maintenance programmers quick access to necessary information. Some programmers prefer to get a high-level view of the system before searching through the details of the code. Others want the details first hoping to join information together in comprehending the system. TLES can accommodate both lines of thought. TLES also allows a person to examine a program construct within the code and bring up its full history -- its reason for existence, its transformation of scope or definition, and finally its implementation.

In order to verify the concept of TLES, we conducted an experiment. The experiment and its results are described in the next section. The results supply an evidence that layered explanation of software is a helpful concept in understanding programs.

4. The experiment

The experiment was conducted in the winter of 1994 at Wayne State University, Detroit, Michigan. The subjects were 39 first-year graduate students, who were proficient in C++ programming and Software Engineering principles. The experiment was administered to all subjects in one 35-minute session. The subjects were divided into 3 groups (A, B, C) of equal ability. The basis for the division is the performance of the students in a course on Software Engineering, with all Grade A, Grade B and Grade C students equally distributed into the 3 groups. Each subject was given a C++ program listing and documentation (different for each group) and a question sheet (identical for each group) containing several questions. All the questions were multiple choice questions, with the subjects selecting the correct answer from 5 different choices (a,b,c,d,e). Only one correct choice exists for each question, and there was no ambiguity in the questions.

The program chosen for the experiment was an implementation of a tool for the OODG (Object Oriented Decomposition and Generalization) methodology for object oriented software development [11]. The program consists of four classes, with each class having 1-4 data members and 3-5 member functions (for a total of 17 member functions). The size of the program is 520 lines of code. Both the domain and the programming language (C++ language) were familiar to the subjects. Each of the three groups worked with a different documentation approach, namely Layered annotations, Non-layered annotations, and Embedded comments. They are described in the following way:

Layered annotations:

A complete description of the various components of the source code was presented in three different layers, namely : domain, algorithm, and representation. The components of the source code that are annotated include: classes, member functions of the classes, and selected local variables of the functions.

Problem domain layer contains description which is understood by an end-user and problem domain expert. Programming skills and knowledge of the implementation programming language are not required to understand the problem domain layer. Algorithm layer describes the underlying algorithms used. Representation layer depicts the constrains and assumptions made by the developer during the coding. All language dependent and machine dependent issues are presented in this layer.

A document was provided to the subjects, containing the code and the above mentioned three layers of explanations. An index listing all the software components and their explanations was provided. An example of the layered annotations of the experiment program is shown in Appendix B.

Non-layered annotations:

Same information is provided in this scheme, but in a single annotation without any distinction between the three different layers.

Commented version:

The description of each component of the source code is provided as a comment just preceding the program component in the source code. All information as provided by the layered and non-layered versions is embedded in the source code as comments.

Results and interpretation:

While conducting the experiment, we tried to remove all other influences on program comprehension. Therefore all variable names were changed to meaningless names. For each question, the subject is expected to identify the relevant portion of the source code and documentation.

The scores of the three groups, each group comprising of 13 subjects is shown in Table 1. It lists the number of correct answers for each subject. Also shown are the cumulative scores, means and variances of each group.

Notice that the group receiving the layered version scored consistently higher than the other groups. Surprisingly, a significant margin does not exist between the scores for the non-layered and commented version.

In order to quantify the significance of these results, an analysis of statistical significance was conducted following the methodology of [6]. That is, we want to know whether the difference in understanding between the three groups of teams can be expected to be observed if we had access not only to our sample of three groups, but to all similar teams that we may encounter now and in the future.

Our null hypothesis is that in the population, the mean score for program understanding among the three groups (layered or nonlayered or commented) does not differ. To test this null hypothesis, we calculate the 't' statistic. A 't' statistic greater than 2.00 or less than -2.00 is a sufficiently compelling evidence to reject the null hypothesis [6]. The 't' statistic for the three different combinations of groups are:

t layered vs nonlayered	3.34
t nonlayered vs commented	0.8524
t layered vs commented	3.6187

Since from the experiment data, the t layered vs nonlayered and t layered vs commented are greater than 2.00, we can regard these data as sufficiently compelling evidence to cause us to reject the null hypothesis. In other words, we can conclude that there is a reliable difference between comprehension of programs with layered annotations as opposed to other styles of documentation (nonlayered or commented). On the other hand, the t nonlayered vs commented is only 0.8524, and hence, we cannot reject the null hypothesis. In other words, we cannot conclude, based on the experiment data, that there is a reliable difference between the comprehension of programs with non-layered annotation as opposed to programs with comments.

Based on the positive result of the experiment, a prototype tool for Layered Explanation of Software (called "TLES") has been developed at Wayne State University. A description of the tool follows.

5. TLES tool

TLES is an interactive implementation notebook. It allows a programmer to store and retrieve annotations describing program constructs. The layering feature of the tool supports records of an evolutionary history of the constructs of the source code. When a maintenance programmer selects a program construct (say a class in

SCORES	Cumulative score	mean	variance
Commented:			
3 2 2 3 4 4 3 2 3 4 4 3 2	39	3.0	0.2248
Non-Layered:			
3 3 3 4 4 3 2 4 3 3 4 3 3	42	3.23	0.3313
Layered:			
3 4 4 4 5 4 4 4 3 4 4 4 4	51	3.92	0.6153

Table 1

C++) a window pops up displaying the stored annotation, if any, for the class at the layer chosen.

An important feature of TLES is the ability to customize the layers. The definition and number of layers changes from system to system. For example, a program that backs up files for a customer is conceptually easy and would not require many layers of explanation. On the other hand, tax accounting software could benefit greatly from many layers because of all of the rules involved and the fact that the program is so apt to change from year to year. For this reason, TLES allows customization. The names of the layers as well as their number are specified by the user. The programmer using TLES is responsible for defining the intermediate domains the implementation passes through in order to become a program. This frees him from having to use any particular software process in order to use TLES.

Another feature of TLES is the ability to update/modify the documentation as a program is modified and constructs continue to develop. A full-length, detailed description of when and how a change was made may be added to the existing TLES documentation for future enhancements. These modifications could affect any layers and any program construct. For example, changing a variable from a string to an integer would effect only the layers closest to implementation. However, changing an account variable to an account structure to include savings accounts as well as IRAs would result in a change to the uppermost layers causing a ripple effect through all of the layers underneath it. If these changes are made consistently, the programmer reaps the full benefits of the tool.

TLES prototype was developed using AT&T C++ Language System Release 2.0 on SPARC workstations. The graphic user interface was developed using Xview toolkit release 3.1. The existing tool is used to document the C++ program constructs "class" in a C++ source code, at any number of layers.

When the tool is used, a menu appears allowing the user to select from two options: to view the documentation for an existing project, or to document a new project. When a new project is to be documented, the number of layers and their names at which the project is to be documented are to be specified first. For existing projects, class and layers have to be selected from menus. The particular annotation can be either read or updated, based on the circumstances.

6. Conclusions

In this paper we have presented a methodology that facilitates the understanding of programs. In this methodology, an evolutionary history of the basic constructs of the program is documented in a layered fashion, where decisions done on each layer are properly recorded. This allows a user to examine a program construct within the code and bring up its full history.

The experiment was conducted on a group of graduate students, reading the same program with three different styles of documentation: layered annotations, non-layered annotations, and comments in the code. The result of the experiment was that the group receiving the layered version scored substantially higher than the other groups.

A tool to support the layered annotation scheme is also presented. The tool supports layered explanation of C++ programs at different layers. The number and the definition of the layers is configurable.

In future, a more powerful tool supporting layered explanation of source code is to be developed. It will be interfaced with a complete browser for the source code, and it will allow full browsing based on the dependencies in both the code and among the annotations.

Acknowledgements.

The authors want to thank to Jagadish K. Kadekar who implementated one of the the first versions of TLES. We also would like to thank Vasik Rajlich for statistical analysis of the results.

References.

[1] Robson, D.J., Bennett, K.H., Cornelius, B.J., and Munro, M. "Approaches to Program Comprehension." Journal of Systems Software 14 (1991), 79-84.

[2] Bennett, K.H. "The Software Maintenance of Large Software Systems. " Reliability Engineering and System Safety 32 (1991), 135-154.

[3] Corbi, T.A. "Program Understanding: Challenge for the 1990s." IBM System Journal Vol.28 No.2 (1989) 294-306.

[4] Brooks, R. "Towards a Theory of the Comprehension of Computer Programs." International Journal of Man-Machine Studies 18 (1983) 543-554.

[5] Brooks, R. "Towards a Theory of the Cognitive Processes in Computer Programming". International Journal of Man-Machine Studies 9 (1977) 737-751.

[6] Judd, C.M., Smith, E.R., Kidder, L.H., "Research Methods in Social Relations", Sixth Edition, Holt, Rinehart and Winston, Inc., Fort Worth 1991, 396-405.

[7] Bigelow, J. "Hypertext and CASE." IEEE Software, Mar.1988: 23-27

[8] Horowitz, E. and Williamson, R.C., "SODOS: A Software Documentation Support Environment-- It's use." IEEE Transactions on Software Engineering 12 (11) (1986) 1076-1087.

[9] Basili, V.R. and Mills, H.D., "Understanding and Documenting Programs." IEEE Transactions on Software Engineering 8 (1982) 270-282.

[10] Wiedenbeck, S. "The Initial Stage of Program Comprehension." International Journal of Man-Machine Studies 35 (1991), 517-540.

[11] Rajlich, V., "Decomposition/Generalization Methodology for Object-Oriented Programming." Journal of Systems Software 1994; 24:181-186

Appendix A: Selected part of the source code:

```
void claa :: uildclas(classes& cmns, function& frmsn) {
    int i, j, k;
    int comum = 0;
    int claon, efolanc;
    FILE *fp;

    fp = fopen("memdoc","w");
    claon = cmns.couentc();
    efolanc = frmsn.etcno();
    for(i = 0;i<claon;i++) {
      fprintf(fp,"class %s {\n",cmns.etlacs(i));
      fprintf(fp,"\tpublic:\n\n");
      for(j = 0;j<efolanc;j++) {
          for(k = 0;k<3;k++)
            if(tramci[j][i].mmaetrin[k] == 'M')
              fprintf(fp, "\t%s\n", frmsn.tfunge(j));
      }
        fprintf(fp,"\n\tprotected:\n");
        fprintf(fp,"\t....\n");
        fprintf(fp,"};\n\n\n");
    }
    fprintf(fp,"\n\nERRORS IN MEMBERSHIP\n");
    fprintf(fp,"_____\n");
    for(i = 0;i<efolanc;i++) {
    for(j = 0;j<claon;j++) {
          for(k = 0;k<3;k++)
            if(tramci[i][j].mmaetrin[k] == 'M')
                comum++;
      }
      if(comum > 1) {
          fprintf(fp, "\n\n\t%s belongs to classes:",
              frmsn.get_func(i));
          for(j = 0;j<claon;j++) {
              for(k = 0;k<3;k++)
                if(tramci[i][j].mmaetrin[k] == 'M')
                    fprintf(fp,"%s ",cmns.etlacs[j]);
          }
          fprintf(fp,"\n");
      }
      comum = 0;
    }
    fclose(fp);
}
```

Appendix B: Layered annotations.

function: claa::uildclas

Domain:

This function uses the relations between the deferred functions and classes, and builds the skeleton declarations of each of the classes. It also checks for errors in membership. If a deferred function has "Class Membership" relationship with more than one deferred class, an error is reported.

Algorithm:

```
Open the output file.
for (each deferred class in the list)
    write class name to output file.
    for (each deferred function)
        if (function has a 'class membership' relation
                with the class)
        write the function as a public member function
                of the class.
for (each deferred function)
    if (function has a 'class membership' relation with
            more than one class)
        write error message.
```

Representation:

"memdoc" is the output file to which the skeleton declarations of the classes are written.

cmns : Contains the list of all class names given as input by the user. The maximum number of classes is 5.

frmsn : Contains the list of all function names given as input by the user. The maximum number of functions is 10.

claon : Contains the number of class names in the list 'cmns'.

efolanc : Contains the number of function names in the list 'frmsn'

comum : counts the number of 'class membership' relations for a deferred function.

Appendix C: Commented version

```
void claa :: uildclas(classes& cmns, function& frmsn) {

// This function uses the relations between the deferred
// functions and classes, and builds the skeleton
// declarations of each of the classes. It also checks for
// errors in membership. If a deferred function has "Class
// Membership" relationship with more than one deferred
// class, an error is reported.
        int i, j, k;
        int comum = 0;
// comum : counts the number of 'class membership'
// relations for a deferred function.
        int claon, efolanc;
// claon : Contains the number of class names in the list
// 'cmns'.
// efolanc : Contains the number of function names in the
// list 'frmsn'
// cmns : Contains the list of all class names given as
// input by the user. The maximum number is 5.
// frmsn : Contains the list of all function names given
// as input by the user. The maximum number of
// functions is 10.
        FILE *fp;
// open the output file. "memdoc" is the output file to
// which the skeleton declarations of the classes are
// written.
        fp = fopen("memdoc","w");
//          number of classes
        claon = cmns.couentc();
//          number of deferred functions
        efolanc = frmsn.etcno();
        for(i = 0;i<claon;i++) {
          fprintf(fp,"class %s  {\n",cmns.etlacs(i));
          fprintf(fp,"\tpublic:\n\n");
          for(j = 0;j<efolanc;j++) {
// Check for 'Class Membership' rlationship
                for(k = 0;k<3;k++)
                   if(tramci[j][i].mmaetrin[k] == 'M')
                      fprintf(fp, "\t%s\n", frmsn.tfunge(j));
            }
            fprintf(fp,"\n\tprotected:\n");
            fprintf(fp,"\t....\n");
            fprintf(fp,"};\n\n\n");
        }
        fprintf(fp,"\n\nERRORS IN MEMBERSHIP\n");
        fprintf(fp,"_____\n");
        for(i = 0;i<efolanc;i++) {
// Check for more than one 'Class Membership'
// relationship
        for(j = 0;j<claon;j++) {
                for(k = 0;k<3;k++)
                   if(tramci[i][j].mmaetrin[k] == 'M')
                        comum++;
            }
            if(comum > 1) {
// More than one 'Class Membership' ; Print error
// message
                fprintf(fp, "\n\n\t%s belongs to classes:",
                     frmsn.get_func(i));
                for(j = 0;j<claon;j++) {
                     for(k = 0;k<3;k++)
                        if(tramci[i][j].mmaetrin[k] == 'M')
                           fprintf(fp,"%s ",cmns.etlacs[j]);
                 }
                 fprintf(fp,"\n");
            }
            comum = 0;
        }
        fclose(fp);
    }
```

Appendix D: Experiment questions

1) A class/function pair has
   a) Only one coupling relationship.
   b) At least two coupling relationships.
   c) A maximum of two coupling relationships.
   d) Any number (0 thru 4) coupling relationships.
   e) At least one coupling relationship

2) The length of a function name is
   a) any number of characters.
   b) must be less than or equal to 10 characters.
   c) must be less than or equal to 5 characters.
   d) must be less than or equal to 30 characters.
   e) must be less than or equal to 35 characters.

3) The maximum number of deferred functions, that can be member functions of a class
   a) 1
   b) 2
   c) 5
   d) 8
   e) none of the above.

4) What happens if a function has a 'Class Membership ('M')' relationship with two classes.
   a) Prints an error message and quits.
   b) Prints an error message and continues execution.
   c) Makes the function a member of both the classes.
   d) Makes the function a member of both the classes and then prints an error message.
   e) none of the above.

5) How many times can a user input/change the coupling relationships between function/class pairs and view the results in the output file "memdoc".
   a) 1
   b) 5
   c) 10
   d) Any number of times.
   e) none of the above.

Session C:
Experience Reports

Experiences Using Reverse Engineering Techniques to Analyse Documentation

Graham Ewart
Centre for Advanced Studies
IBM Toronto Laboratory
844 Don Mills Road
Don Mills, Ontario M3C 1V7

Marijana Tomic
Centre for Advanced Studies
IBM Toronto Laboratory
844 Don Mills Road
Don Mills, Ontario M3C 1V7

Abstract

This paper discusses an approach taken to analyse IBM product documentation using reverse engineering technologies, which are normally applied to the analysis of system source codes.

Key words: *program understanding, information structure, Abstract Syntax Tree*

1 Introduction

Society's dependence on software maintenance and program understanding continues to grow. It is estimated[8] that 30 to 35 percent of the total life-cycle costs of a system are consumed in trying to understand existing software. Furthermore, these needs are very expensive: $30 billion is spent annually on software maintenance world-wide, $10 billion in the US alone, constituting 50 percent of most data processing budgets and 50 to 80 percent of the working hours of an estimated one million programmers. This software maintenance and program understanding is not an option.

"Program Understanding" is the process of developing mental models of a software system's intended architecture, purpose, and behaviour. One way of augmenting this process is through computer-aided reverse engineering. Although there are many forms of reverse engineering, the common goal is to extract information from existing software systems. The often-neglected partner in all this is the documentation that goes with a software system. The focus of this paper is the application of reverse engineering technology to system documentation.

2 Background

The Centre for Advanced Studies (CAS) at the IBM Toronto Laboratory is a small research group whose primary aim is to facilitate the transfer of research ideas into the various product development groups in the Laboratory. Interactions between these groups has been identified as critically important in the transfer of research results within organizations.

The CAS Program Understanding Project includes researchers at McGill University, the National Research Council of Canada, the University of Toronto, and the University of Victoria. It has been using computer-aided reverse engineering technology for the past three years to help the SQL/DS database system team improve their productivity and the quality of their product. The success of this work led to the establishment, in 1993, of related services in the Software Engineering Quality Consultants (SEQC) area of the Toronto Laboratory.

Recently, we have begun to examine ways in which reverse engineering technology can be applied to the understanding of the structure of the documentation that accompanies our products. There are many common characteristics between understanding challenges presented by source code and those presented by documentation that suggest to us that the same kinds of analysis are worthwhile. These include the requirement to understand complex structures and data references; both code and documentation can be complex and hard to manage.

[1]The following are trademarks or registered trademarks of International Business Machines Corporation: IBM, AIX, Risc System/6000, BookMaster, and SQL/DS.

Software Refinery, REFINE, DIALECT, INTERVISTA, and REFINE/C are trademarks of Reasoning Systems, Inc.

UNIX is a registered trademark in the United States and other countries licenced exclusively through X/Open Company Limited.

TeX is a trademark of the American Mathematical Society.

0-8186-5647-6/94 $04.00 © 1994 IEEE

Our hope is that our work will make the maintenance of product documentation easier and enable the discovery of structural anomalies in our documentation.

The methods we used to extract structural information from product documentation was described earlier in [1]. These methods are summarised in the next two pages. We then describe methods we have developed to enhance the visual presentation of this information, and discuss, in section 6, some preliminary work we have done with Software Refinery to analyse documentation.

3 Documentation Format

In the Toronto Laboratory, product documentation is written (for the most part) using BookMaster, an IBM product that supports a markup style of document editing using tags, control words, and macros.

Tags, which are the essence of markup, take the form:

:*tagname* [<*qualifiers*>].

These tags, as with SGML (ISO Standard ISO 8879, "Information Processing – Text and office systems – Standard Generalized Markup Language"), are used to specify the function that each fragment of text takes on in the document. Macros and control words, while important to the document author, are not used to specify the meaning of document elements.

In this environment, as with many others, a document has a beginning, a middle, and an end. The beginning contains the title page, preface, abstract, table of contents, and other introductory matter; the middle, introduced by the `:body` tag and typically the largest part of a document, the text of the document; and the end, introduced by the `:backm` tag, contains the appendices, glossary, bibliography, and index. The tags for such a document will be something[2] like [9]:

```
:userdoc ibmcopyr='1992, 1994'.
:prolog.
:title.C/C++ Tools: Standard Class
Library Reference
:topic.C/C++ Tools: Standard Class
Library Reference
:version.Version 2.0
:etitle.
:docnum.61G1180
:eprolog.
:cover.
:notices.
:enotices.
:toc.
:body.
:h1 id=notpg.Notices
:h1 id=lgintro.Introduction
:p.This book describes the
standard class library included
with the C/C++ Tools.
These class libraries provide
sets of predefined [...]
:h2 id=csc1006.Who Should Use
This Book.
[...]
:euserdoc.
```

As seen above, these tags are interspersed throughout the text.

Overlaid on this hierarchical document structure is a series of references from within one part of the document's text to other parts. For example, text in the documentation could refer to figures or tables that are often nearby, but could just as easily be in remote sections of the document. This is similar to the variable definitions and variable references found in source codes of a system.

As with traditional programming languages, the source for a programming manual can consist of many files, sometimes hundreds, which are "`#included`" using the "`.im`" control word.

In many respects the similarity in understanding challenges between source codes and documentation is a direct result of the similarity in structure between traditional programming languages and the sources associated with product documentation. This similarity is what led us to consider the application of the same reverse engineering technologies to this area.

4 Approaches used to Analyse Source Codes

Presently, we are using or investigating several approaches in the analysis of source code:

RIGI is a research project being carried out at the University of Victoria concerned with the visual representation of complex systems. It has not been used in production to date in the Toronto Lab.

Software Refinery, from Reasoning Systems, analyses sources, builds Abstract Syntax Trees representing the internal structure of the code, and

[2] We have removed the author's commentary and some tags that are used to format the same source for different products or different media as they would only serve to complicate this example

provides a framework for writing rules that can be used to uncover characteristics of interest.

It has been used in the Toronto Lab in support of several quality initiatives, including

1. Programming language violations (overloaded keywords, poor data typing),
2. Implementation domain errors (data coupling, addressability), and
3. Application domain errors (coding standards, business rules) [3, 4].

Our project is also doing work at the National Research Council of Canada on the pattern matching of source code at the text level, largely for the purpose of clone detection[3], and at the University of Toronto on the integration of the underlying repositories that support REFINE and RIGI[5, 2].

Both Software Refinery and Rigi operate on the IBM Risc System/6000 platform, in the AIX (UNIX-like) operating system environment. The work on text-base clone detection, while currently running under AIX, has far less dependence on platform-specific tools.

From our perspective, REFINE is more directed to fine-grained analysis, while Rigi offers a more coarse-grained view of the system under analysis[7]. This view is mostly opinion, and a reflection of our use of these tools.

5 Rigi

The Rigi system provides a versatile and extensible graph display and editing environment that is conducive to the discovery and analysis of the structure of large systems. As applied to traditional program understanding problems, it supports the following features[6]:

- Parsing for a variety of programming languages, including C, C++, COBOL, LaTeX, and PL/AS (an internal IBM language).
- A repository to store the parsed information, and
- An interactive graph editor that allows the user to manipulate the displayed information.

[3]The technique of identifying scattered fragments of code that are in some respects (in this case, textually) similar. This often points out instances of "cut and paste" programming.

Presently, the Rigi parser has no support for the BookMaster "language". Nor do we suggest that it be extended to provide this. Rather, we are using an important feature of the graph editor, its tight coupling with a command language (rcl) that allows us to write scripts to create and manipulate Rigi graphs.

This command language is based on TCL; it gives Rigi an open-ended ability to extend its interface mechanism by providing for scripts (either supplied with Rigi or user-written) that embody the definition of commonly used operations. We use this interface to create a Rigi graph from a stream of tuples that describe the interesting structural features of the subject documentation. These tuples, which take the form

<relationship-name, element1, element2>

describe how identified document elements are related to each other. For example, we might use the triple

<LOCATION, FIG2, HEAD8>

to indicate "document element fig2 appears inside the document element head8".

We are considering the following relationships:

- The "is a part of" structure of the document elements, namely subsections within sections within chapters within parts,
- The "data references" within one document part to other document parts (figure references, heading references etc.), and
- The relationship between the logical parts of the document and the files that make it up.

5.1 Creating the Tuple Stream

Our approach requires us to provide our Rigi scripts with a flat file containing a stream of tuples describing the structure of the documents of interest. We create this file using the facilities of BookMaster itself.

BookMaster consists of two layers. At the low level is an "engine", called SCRIPT/VS, which reads the source files for the documentation and processes very rudimentary control words that specify what formatting is to be done. One such control word is used to define macros which, although made up of these rudimentary control words, can contain logic and accomplish very sophisticated formatting tasks.

The higher-level layer is a complex library of macros that are "attached" to the tags described earlier, and accomplish the real work of formatting a document.

Normally when a document is formatted SCRIPT/VS reads two files: the initial file for the document to be formatted, and a profile file that "bootstraps" the macro library containing all the tag definitions.

We replaced the standard BookMaster profile file with one of our own design in which we redefined the processing macros for the BookMaster tags that were of interest to us. Our macros, instead of formatting, keep track of the current "position" (part, chapter, section, subsection etc.) in the document and log the structural information discovered. This log is the tuple-file we read from Rigi rcl scripts.

5.2 Laying out the Graph

When Program Understanding or Reverse Engineering tools are used to analyse the structure of an application system, the order of functions within modules or modules within executables is often of little concern.

This is because this ordering is seldom of any significance to the understanding of the system's operation or, in fact, the actual operation of the system.[4] Contrast this to the "operation" of many books, where the desired use is one of "straight through" operation with as little branching as possible.

With this in mind, we felt it necessary to preserve the ordering of book parts in our graphical display of the structure of a book. We considered adding a "follows" relationship to the sequence of tuples we generated when first analysing our books, and continuously returning to this relationship as we transformed the graph. We abandoned this in favour of laying out our structure graph in a way that preserved (in some sense) the input order of the tuples as they were placed on the screen, and then applying only transformations that preserved a sense of this order. Our initial attempt at this laid out the graph representing the book's structure with the high-level sections (front matter, body, back matter) across the top of the display, with their immediate components ranged beneath horizontally. This placed the leaf nodes of this graph next to each other three or four rows below the top level.

This proved unworkable: we were faced with the choice of either spreading the leaf nodes out so that their text (the titles) did not overlap, thus reducing the number of nodes we could see on the screen at once to the point where very little structure was evident; or reducing the font size of the text to the point where it was unreadable.

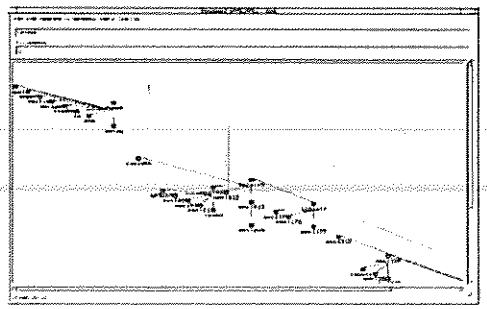


Figure 1: Sample document: results of automatic layout

To address this problem, we decided to lay the graph out on an angle, starting at the top left of the screen and proceeding down and to the right. This is close to the layout presented in most of the examples shown in this paper, although those examples were "touched up" somewhat to to improve their readability, proving to us that our search for a good layout is not complete. This approach allowed us to place neighbouring nodes much closer together without having their text overlap.

The layout algorithm we developed is this:

```
do while not a end-of-file
- read the input file
- if this tuple describes a ''contains''
  relationship between two document
  sections:
  - locate the ''parent'' node
  - if found:
    - move it and all of its
      ancestors to the right R units
      and down D units,
  - otherwise:
    - create a parent node R units
      to the right of the last node
      we created at any level
      and on the top line
  - endif
  - add the child node B units below
    the (perhaps moved) parent node.
- endif
enddo
```

We experimented with different values of R, D, and B to get the graphs shown in this paper.

In the actual construction of the graph, this loop was followed by a similar loop which drew the non-structural (that is, all but "contains") relationships.

We have applied this algorithm to several documents, including the C/C++ Tools: Standard Class

[4] There are, of course, exceptions to this: for example, locality of reference is a very important area of concern for those doing performance analysis of systems.

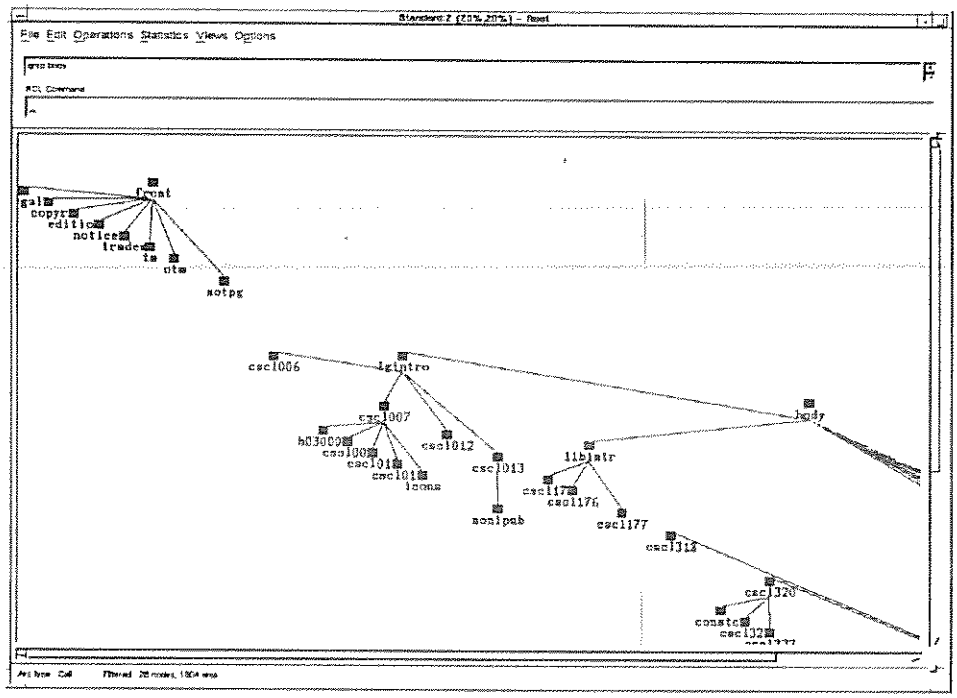


Figure 2: Sample document: The C/C++ Tools: Standard Class Library Reference

Library Reference with similar results. The result of this layout, showing only the structural elements (that is, after the first loop) can be seen in Figure 1.

As is evident, this algorithm is far from perfect. One fairly simple enhancement we are considering is, after the initial layout, to visit each node, moving it to the halfway point between its current and its initial position. This mimics one of the activities we invariably do manually when we "touch up" the graph to make it more readable. We continue to analyse the various other activities that seem to make the graphs more presentable while preserving the match between the actual structure of the book and the mental model conveyed to us by the layout.

The top left corner of the graph presented in Figure 1 as touched up manually, is shown in Figure 2.

5.3 Viewing Structural Information

One of Rigi's important features is its ability to allow the user to manipulate a graph. An important aspect of this is the user's ability to filter out nodes and/or arcs to enable viewing of the aspects of the graph of interest. The graphs shown in Figures 1 and 2 were not made by running only the first loop of the algorithm described in the previous section. Rather, they were produced by running both (the structural and non-structural) loops and filtering out the "file" nodes and all but the "structural" arcs.

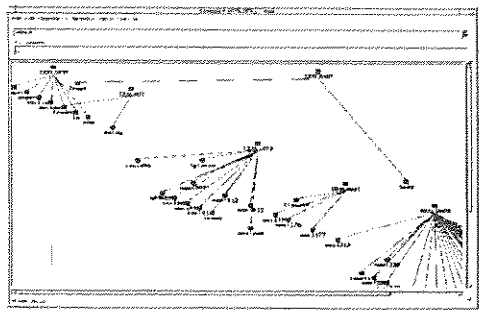


Figure 3: The relationship between the sample document and the files used to build it

When these nodes and arcs are returned and the "structural" arcs are filtered out, the result is as shown in Figure 3. We can see (barely) in this figure that the front matter and the body are both introduced in file CXXLMAST but the contents of the front matter are contained in file CXXLOINT. This view of the

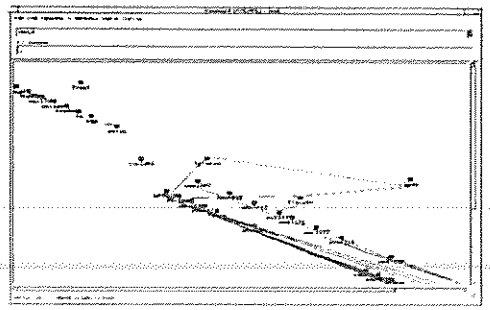


Figure 4: References within the sample document

document allows the user to view the overall structure of the document and determine which file contains the source text for any given part of the document.

Figure 4 shows the references between parts of the document. It is not intended that the user be able to analyse all the inter-section references in a document by simply looking at this graph. The power of this lies in the ability to use the graph to select regions of the document or clusters of references, and then apply the power of Rigi's scripting language to select its arcs and/or neighbouring nodes for further analysis.

We have not shown the graph of the entire document resized to fit within the boundaries of the user's screen. This graph, even when displayed on the user's screen, is a mass of indistinguishable nodes and arcs occupying the main diagonal of the window.

6 REFINE

Our work with Software Refinery is still in its beginning stages.

Software Refinery consists of three parts (products) as shown in Figure 5:

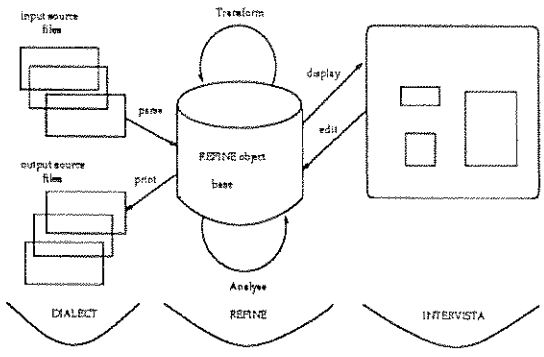


Figure 5: Software Refinery structure

REFINE[11], a high-level programming language with facilities that can create, analyse and transform the Abstract Syntax Tree representation of the system being analysed;

DIALECT[12], which is used to produce parsers and printers from specifications written in a high-level languages; and

INTERVISTA[13], which provides specialised tools to create user interfaces for REFINE applications, such as window management, mouse handling, menus,

etc.

6.1 Defining the domain model and grammar

The first step in using REFINE to analyse source code is to understand the language model that defines the set of object classes and the attribute definitions that define the structure (nodes and edges) of abstract syntax trees (AST). This is the domain model; it defines how the elements of a system will be modelled in REFINE's object base.

To define a domain model for a language, it is important to:

1. Determine the basic structure of the Abstract Syntax Tree,
2. Define object classes for each type of node in AST,
3. Define attributes that capture the structure of AST, and
4. Define attributes for any annotations needed.

A domain model is based on object classes, attributes, and objects.

Object classes model the kinds of objects in the domain (programs, files, declarations, statements, ...);

Objects model individual objects in the application domain (the inventory program, its first file, a function in the file, ...);

Object classes are arranged into an inheritance tree via specialization. The general form of an object class definition is:

```
var class-name: object-class subtype-of parent-class
```

The object classes we chose for analysis of BookMaster are:

```
var   DOC-OBJECT:    object-class
                     subtype-of user-object
var   IDENTIFIER:    object-class
                     subtype-of doc-object
var   FRONT-BODY:    object-class
                     subtype-of doc-object
var   FRONT-ELEM:    object-class
                     subtype-of doc-object
var   BODY:          object-class
                     subtype-of doc-object
var   BODY-ELEM:     object-class
                     subtype-of doc-object
```

The general form of an attribute definition is:

var *attribute-name*:map(*object1* , *object2*) = {||}

Attributes model the relationships among objects and also the features of objects (e.g., a program with three files). In REFINE language terms, an attribute is a global variable of type map whose domain type is an object class.

The attributes we chose (edges in the Abstract Syntax Tree) for our BookMaster objects are:

```
var DOC-NAME:
    map(identifier, symbol) = \{|\}
var FRONT-LINK:
    map(doc-object, front-body) = \{|\}
var FRONT-SEQ:
    map(front-body, seq(front-elem)) = \{|\}
var BODY-LINK:
    map(doc-object, body) = \{|\}
var BODY-SEQ:
    map(body, seq(body-elem)) = \{|\}
```

In the domain model we define the classes and attributes used to model the AST. More is required to actually parse sources and populate an Abstract Syntax Tree. A grammar that defines the characteristics of the language under study must be contructed, using DIALECT, so that the Software Refinery can build a parser for the language. This parser is then used to populate the AST.

The body of a grammar contains a set of productions and other clauses specifying the surface syntax of a language. Productions specify the relationship between text strings that are statements in the target language and AST that are stored in the object base.

The general form of syntax for a production is:

<*nonterminal-name*> ::= [<*syntax-for-class*>] <*action*>

DIALECT's syntax description language extends BNF in two ways:

1. Attribute names can be used instead of terminals and nonterminals on the right-hand side of a production;
2. Regular expressions can appear in the right-hand side of a production. In this way much more complex surface syntax description can be written, within a single production than in standard BNF.

Using DIALECT, we have built a BookMaster grammar:

```
grammar BookMaster}
file-classes doc-object
productions
identifier::= [doc-name]builds identifier,
front-body::= [''':frontm'' front-seq * ''.''']
              builds front-body,
preface::= [''':preface''doc-object-label''.''']
              builds preface,
:
:
```

Our use of REFINE has not progressed past defining its grammar and building the domain model. We anticipate using the work we have done to date to investigate ways these powerful facilities can be used to perform further analysis of our documentation.

7 Conclusions

At this stage we feel that both Rigi and REFINE can be of great help in understanding the structure of our product documentation. We have progressed much more quickly with Rigi than with REFINE, mostly because REFINE is far more demanding in its requirements to formally specify the problem at hand. We have to date seen no real results from our use of REFINE but we feel that this approach, if successful, has the possibility of allowing much more sophisticated analysis.

Rigi, with very little cost, has provided views of documentation that can help the Information Developer understand the structure of a particular document. We believe that the view provided by Rigi can be used as a front end to an editor so that selecting a node (document section) will position the user at the correct line in the correct file for editing.

Furthermore, we feel that Rigi provides a platform for measuring usability characteristics, such as the amount and kind of cross references, of a document.

About the Authors

Graham Ewart *Centre for Advanced Studies, IBM Toronto Laboratory, 844 Don Mills Road, Don Mills, Ontario M3C 1V7.* ewart@vnet.ibm.com.
Mr. Ewart is a Senior Development Analyst in the Centre for Advanced Studies at the IBM Toronto Laboratory. He is currently the Principal Investigator for the Program Understanding project at CAS. Prior to joining the Centre he was the Lead Architect for the IBM C/370 family of compilers and runtimes. His research interests include software maintenance, program understanding, document understanding, and reverse engineering.

Marijana Tomic *Centre for Advanced Studies, IBM Toronto Laboratory, 844 Don Mills, Don Mills, Ontario M3C 1V7.* mtomic@vnet.ibm.com.
Mrs. Tomic is a post-doctoral student from the University of Victoria, working in the Program Understanding project at CAS. Her research interests include software engineering in general, and software maintenance, program understanding, reverse engineering, re-engineering, and re-structuring in particular.

References

[1] Graham Ewart, Marijana Tomic, "Applying Program Understanding Technology to IBM Product Documentation", CASCON'94, (Toronto, Ontario; 31 October - 4 November 1994).

[2] M.Bernstein, *et al.*, "Towards an Integrated Toolset for Program Understanding", CASCON'94, (Toronto, Ontario; 31 October - 3 November 1994).

[3] Erich Buss, John Henshaw, "Experiences in Program Understanding", IBM Canada Lab., Technical Report TR-74.105, Centre for Advanced Studies, July 1992.

[4] E.Buss, *et al.*, "Investigating Reverse Engineering Technologies: The CAS Program Understanding Project", IBM Systems Journal **33**, No.3, 477-500 1994.

[5] R.De Mori, H.Müller, and J.Mylopoulos, "A Reverse Engineering Environment", 1992, Proposal submitted to NSERC's CRD programme.

[6] S.Tilley, M.Whitney, H.Müller, M.Storey, "Personalized Information Structures", SIGDOC'93: The 11th International Conference on Systems Documentation, (Waterloo, Ontario; 5-8 October 1993), 325-337.

[7] H.Müller, S.Tilley, K.Wong, "Understanding Software Systems using Reverse Engineering Technology Perspectives from the Rigi Project", CASCON'93, (Toronto, Ontario; 25-28 October 1993) 217-226.

[8] Kathleen Melymuka, "Managing Maintenance: The 4000 - Pound Gorilla", CIO, 4, No.6, 74-82, March 1991.

[9] "IBM BookMaster Reference Summary Release 3.0", IBM Corporation, 1990.

[10] Software Refinery Training Manual, Reasoning Systems 1993.

[11] REFINE User's Guide, Reasoning Systems 1993.

[12] Dialect User's Guide, Reasoning Systems 1993.

[13] Intervista User's Guide, Reasoning Systems 1993.

Analyzing the Application of a Reverse Engineering Process to a Real Situation

Fabio Abbattista (*), Gregorio M.G. Fatone (**), Filippo Lanubile (*), Giuseppe Visaggio (*)

(*) Dipartimento di Informatica, University of Bari, Italy
(**) Basica S.p.A., Potenza, Italy

Abstract

A reverse engineering process model was applied and, on the basis of the data collected, some modifications were made aiming to improve its efficacy.

The experience gave rise to various considerations of interest, first among them being the clear interaction between the quality of the product and the quality of the process. A method of synergetic application of static and dynamic analysis to improve understanding of the program was consolidated. The experience enabled modifications to be introduced connecting the reverse engineering process more closely with the understanding of the programs and information deriving from the application domain.

Finally, the problem of the efficacy of the tools used to obtain the reverse engineering products was made evident during the experimentation on the field.

1: Introduction

We present an experience in which process quality and product quality interact and mutually improve one another. The process is reverse engineering while the product is the documentation of programs necessary to exploit the program better. The salient point to be gained from the experience are, in general, the model as it appears after improvements stemming from trial on the field and, in particular, the method for integrating static and dynamic analyses to improve the process.

The paper describes the application of a process model to a real situation. The scenario is a set of programs with the following characteristics:

language:	COBOL
operating system:	BS2000
total no. of programs:	653
no. of on-line programs:	348
no. of batch programs:	305
no. of files:	70
no. of data:	9 000
no. of Instructions:	900 000

(including comments but not copy books)

These programs constitute a "large" software system as they are integrated by a database and cover various banking business areas. Their long life (average 12 years, reaching peaks of 23 years) was mirrored by their "old" structure, so that they were difficult to maintain and had inconsistent, invariably poor documentation. The system is so diffuse within the user bank that substituting it would have been impracticable. It was therefore necessary to rejuvenate the software system and, as the only reliable component was the source code, it was judged necessary to start by reverse engineering to understand the programs, while acknowledging that reengineering would then be required.

The paper includes a brief description of the process model (section 2), the results of its application (section 3), the modifications made to the process model (section 4) and the final conclusions (section 5).

2: The reverse engineering process

Our reverse engineering process had two main objectives: (1) to increase the ease of maintenance of the software system, and (2) to improve its usability by the final users and the ease of knowledge transfer among different users [5].

The first involved reconstruction of the project design documentation and restoration of the most degraded parts while the second required reconstruction of the user documentation and the data conceptual model. This helps users to understand their own information system better from the point of view of the data processed [4], [6].

Figure 1 shows the process model, which is briefly described underneath. Further details may be obtained from [10].

1. Inventory Software System. The following cross references are extracted from the old software system: call dependence X-ref, copybook X-ref, and file access X-ref.

0-8186-5647-6/94 $04.00

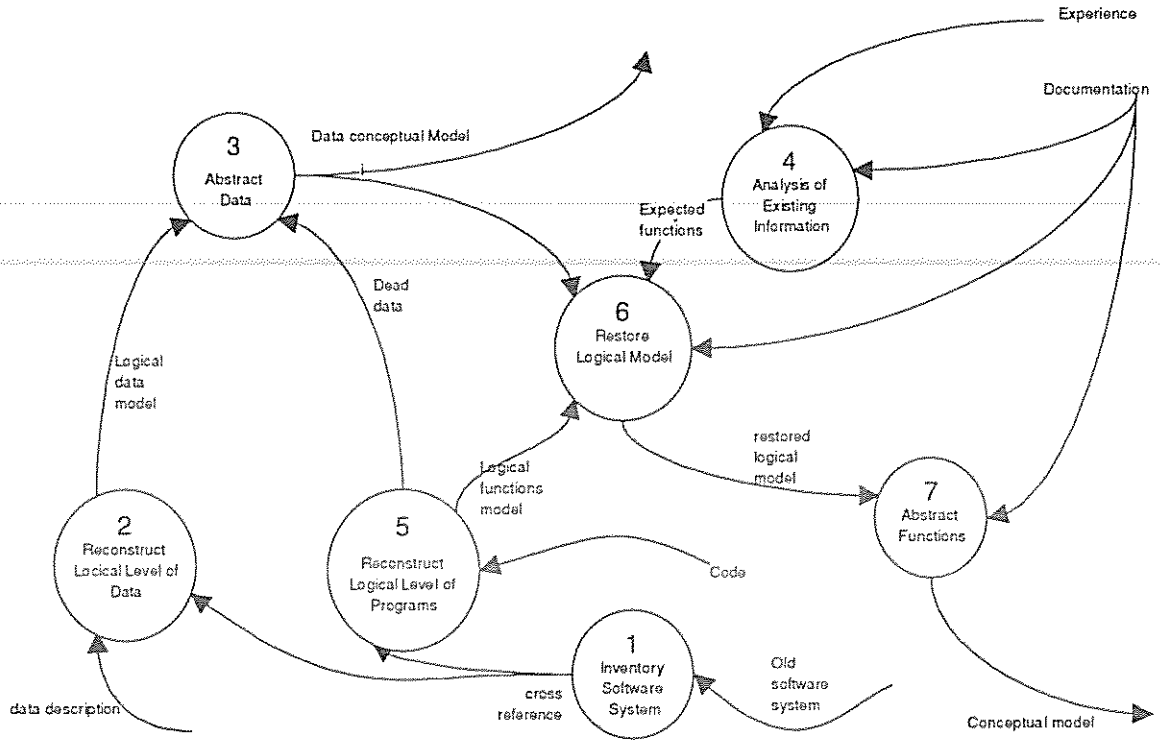


Figure 1. The process model

2. Reconstruct Logical Level of Data. From the data description, the hierarchical structure constituting the logical data model is reconstructed. The aliases are recognizable entities in the application domain. Figure 2 shows a record declaration, as input to the phase, and the relative hierarchical diagram, as output from the phase. The arrows represent the hierarchical relationships between the substructures in the record; elementary data are not shown. The data are classified as application domain data, control data, and structural data. Application domain data are the attributes of recognizable entities in the application domain. For example, the field named MT02-02 representing the "amount" is also an attribute of MORTGAGE and is therefore recognizable in the application domain. Control data have no correspondence with the application domain but are used to record the occurrence of an event during the execution of a program, so that other programs can adapt accordingly; flags validated by one program and used by others, asynchronously, to determine their behavior according to the previous history of the software system, are typical examples of control data. For example, the field MT02-33 is preset to indicate the existence of an agreement for taking out the mortgage, which must form the basis of the variation in interest rate to be applied when calculating the inctalment; it is a typical example of control data. Structural data are data necessary for managing the organization of data bases or files. The field MT01-05 is a typical example of structural data, because it identifies the record type inside the MORTGAGE file.

3. Abstract Data. All the data in the application domain which belong to the logical model and are not dead are associated with the corresponding meaningful concept for the application domain [1].

4. Analysis of Existing Information. This activity involves identifying the expected functions in the program being reversed using two types of information. The first is static knowledge, i. e. the internal and external documentation of the rules governing the application domain of the function. The second is dynamic, derived from the experience of the programmers and users who interact with the working programs.

```
01 REK-MT.
    02 N  PIC X.
    02 M PIC 9(5).
    02 M REDEFINES MT00
        03 MOT01-1.
            04 MT01-01 PIC 99.
            04 MT01-02 PIC 99.
            04 MT01-03 PIC 99.
            04 MT01-04 PIC 9(5).
        03 MT01-05 PIC 999.
    02 MT00-R.
        03 MT00-1 PIC 9(5).
        03 MT00-2 PIC 9(5).
        03 MT00-3 PIC 9(5).
        03 FILLER PIC X(145).
    02 MT02 REDEFINES MT00-R.
        03 MT02-01 PIC X(16).
        03 MT02-02 PIC 99(13) USAGE COMP-3.
        03 MT92-03 PIC 99V99.
        03 MT02-04 PIC 999.
        03 MT02-05 PIC 999.
            04 MT02-061 PIC 999.
            04 MT02-062 PIC 9(6).
        03 MT02-07 PIC 9(6).
        03 MT02-08 PIC S9(13) USAGE COMP-3.
        03 MT02-09 PIC S9(13) USAGE COMP-3.
        03 MT02-10 PIC S9(9) USAGE COMP-3.
        03 MT02-11 PIC 9.
        03 MT02-12 PIC 9.
        03 MT02-13 PIC 99.
        03 MT02-14 PIC X(4).
        03 MT02-15 PIC 9.
        03 MT02-16 PIC 9.
        03 MT02-17 PIC 9.
        03 MT02-18 PIC X.
        03 MT02-19 PIC 99.
        03 MT02-22 PIC 9.
        03 MT02-23 PIC 9.
        03 MT02-30 PIC 9.
        03 MT02-24 PIC 9.
        03 MT02-25 PIC 99.
        03 MT02-26 PIC 9.
        03 MT02-27 PIC S9(7) COMP-3.
        03 MT02-28 PIC S9(7) COMP-3.
        03 MT02-29 PIC S9(7) COMP-3.
        03 MT02-03 PIC 99V9999.
        03 MT02-31 PIC 99V99.
        03 MT02-32.
            05 MT02-321 PIC 99V9(4).
            05 MT02-322 PIC 99V9(4).
        03 MT02-33 PIC 9.
        03 MT02-34 PIC X(4).
        03 MT02-35 PIC XX.
        03 FILLER PIC X(18).
        03 MT02-36 BAS PIC 9.
        03 MT02-37 BAS PIC 9(16).
    02 MT03 REDEFINES MT00-R.
        03 MT03-01 PIC 9(5).
        03 FILLER REDEFINES MT03-01.
            05 MT03-01AA PIC 99.
            05 MT03-01MM PIC 99.
            05 MT03-01GG PIC 99.
        03 MT03-02 PIC S9(11) USAGE COMP-3.
        03 MT03-03 PIC S9(11) USAGE COMP-3.
        03 MT03-04 PIC S9(11) USAGE COMP-3.
        03 MT03-05 PIC S9(11) USAGE COMP-3.
        03 MT03-06 PIC S9(11) USAGE COMP-3.
        03 MT03-07 PIC S9(5) USAGE COMP-3.
        03 MT03-08 PIC S9(11) USAGE COMP-3.
        03 MT03-09 PIC 9(6).
        03 MT03-10 PIC S9(9) USAGE COMP-3.
        03 MT93-11 PIC S9(6) USAGE COMP-3.
        03 MT03-12 PIC 9.
        03 MT93-13 PIC 99V99.
        03 MT93-14 PIC 99V99.
        03 MT93-15 PIC S9(7) USAGE COMP-3.
        03 MT03-16 PIC 99.
        03 MT03-17 PIC 9(6) COMP-3.
        03 MT03-18 PIC S9(7) COMP-3.
        03 FILLER PIC X(02).
        03 MT03-19 PIC S9(7) COMP-3.
        03 MT03-20 PIC S9(7) COMP-3.
        03 MT03-21 PIC 9.
        03 MT03-22 PIC S9(7) COMP-3.
        03 MT03-23 PIC 9(6).
        03 MT03-13 PIC 99V9999.
        03 MT03-14 PIC 99V9999.
        03 MT03-24 PIC X.
        03 MT03-11 PIC S9(7) USAGE COMP-3.
        03 FILLER PIC X(40).
        03 MT03-25 PIC 9(4).
    02 MT04 REDEFINES MT00-R.
        03 MT04-01.
            05 MT04-011 PIC 99.
            05 MT04-012 PIC 99.
            05 MT04-013 PIC 99.
            05 MT04-014 PIC 9(6).
            05 MT04-016 PIC 99.
        03 MT04-02.
            05 MT04-021 PIC 99.
            05 MT04-022 PIC 99.
            05 MT04-023 PIC 99.
            05 MT04-024 PIC 9(6).
        03 MT94-03 PIC 99V99.
        03 MT04 PIC 9(6).
        03 MT94-05.
            05 MT94-051 PIC S9(5) USAGE COMP-3.
            05 MT94-052 PIC 99V99.
            05 MT94-053 PIC S9(5) USAGE COMP-3.
            05 MT94-054 PIC S9(5) USAGE COMP-3.
        03 MT04-06 PIC 9(6).
        03 MT94-07 PIC 99V99.
        03 MT04-08 PIC S9(11) USAGE COMP-3.
        03 MT04-09 PIC 9(6).
        03 MT04-10 PIC 99.
        03 MT04-12 PIC 9(9) USAGE COMP-3.
        03 MT94-11 PIC S99V99.
        03 MT04-13 PIC S9(9) COMP-3.
        03 MT04-14 PIC 9(6).
        03 MT04-15 PIC S9V9999.
        03 MT04-16 PIC 9(6).
        03 MT04-17 PICS9(9) COMP-3.
        03 MT04-18 PIC 9(3).
        03 MT04-03 PIC S9V9999.
        03 MT04-07 PIC S9V9999.
        03 MT04-11 PIC S99V9999.
        03 MT04-05.
            05 MT04-051 PIC S9(6) USAGE COMP-3.
            05 MT04-052 PIC S9V99.
            05 MT04-053 PIC S9(7) USAGE COMP-3.
            05 MT04-054 PIC S9(7) USAGE COMP-3.
        03 FILLER PIC X(31).
        03 FILLER PIC X(11).
        03 MT04-056 PIC 9(4).
    02 MT05 REDEFINES MT00-R.
        03 MT05-01 OCCURS 05 TIMES.
            05 MT05-011 PIC S9(7) COMP-3.
            05 MT05-012 PIC S99999 COMP-3.
            05 MT05-013 PIC S9(9) COMP-3.
            05 MT05-014 PIC S9(9) COMP-3.
            05 MT05-015 PIC S9(9) COMP-3.
            05 MT05-016 PIC S9(6) COMP-3.
        03 FILLER PIC X(34).
    02 MT06 REDEFINES MT00-R.
        03 MT06-01 OCCURS 08 TIMES.
            08 MT06-011 PIC S9(6) COMP-3.
            08 MT06-012 PIC S9(6) COMP-3.
            08 MT06-013X.
                07 MT06-013 PIC S9V9999.
            08 MT06-014 PIC S9(7) COMP-3.
            08 MT06-016 PIC S9(7) COMP-3.
        03 FILLER PIC X(32).
```

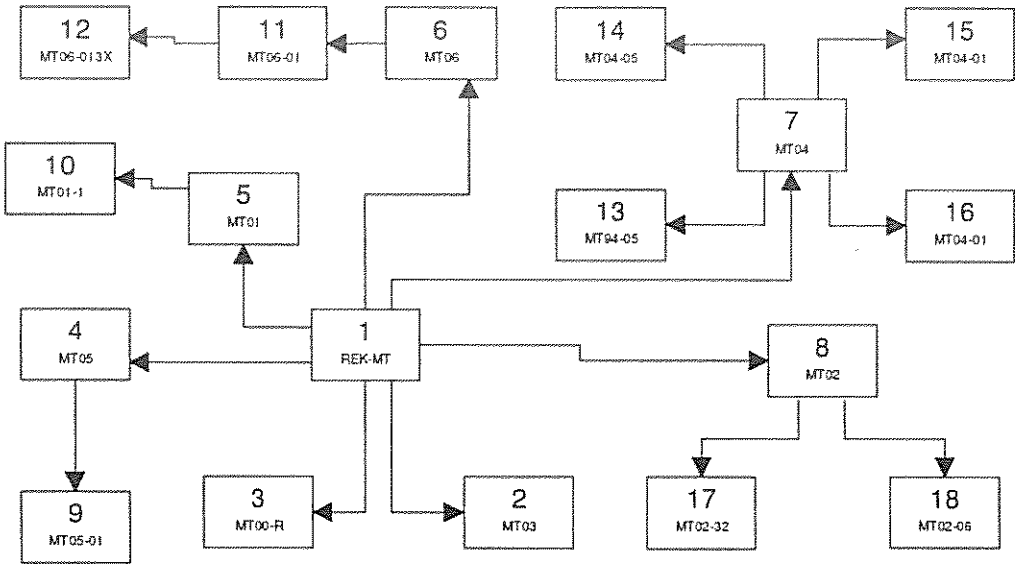


Figure 2. Input and output of "Reconstruct Logical Level of Data" phase

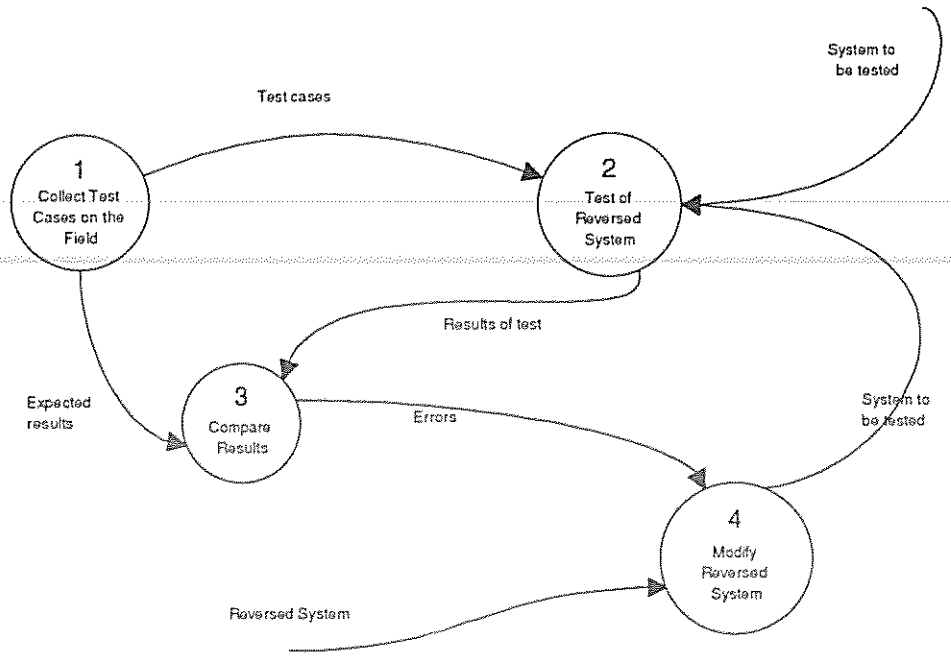


Figure 3. Equivalence test of the system before and after reverse engineering

5. Reconstruct Logical Level of Programs. Each program is associated with a structure chart in which each module corresponds to a SECTION or an external subroutine of the program. In this phase, both dead data and dead instructions are identified. These are data not used by the program and instructions which cannot be run, respectively. The former are communicated to the "Abstract Data" activity while the latter are erased from the structure chart , which thus constitutes the logical model of the functions.

6. Restore Logical Model. Restoration involves introducing changes to improve the structure of the programs and make them easier to maintain, without causing repercussions on the data or interfaces with other systems. Some examples of modifications are renaming of variables, making their identifiers more meaningful; extracting modules with high internal cohesion from those with low cohesion and isolating them in the structure ([7], [8], [12], [13]); externalizing modules which, in the present process, are in line with the main; localizing variables declared to be global but used locally in both existing modules and in processes extracted during restoration. Execution of these activities is facilitated by the expected functions derived from the phase of analysis of existing information. In fact, thanks to this knowledge, the operators can extract the functions from the modules present in the logical model. This makes the logical model more readable and its modules less complex.

7. Abstract Functions. The functions abstracted during restoration are documented. The aim of each function is described in textual form. The relationships between functions are also documented by means of data flow diagrams. The latter, together with the description of each function, constitute the conceptual model [9].

The reverse engineering process is not symmetrical because the programs are restored while the data are not, because any interference with the latter would affect the procedures and make the whole restoration process very expensive. In fact, restoration of the programs is confined to the instructions of each single program, a much simpler and more economical process.

The reverse engineering process described modifies code, so that it is necessary to verify that the working programs are equivalent to those produced by the process. A test process is used and, as the only reliable component in the working system is the code in question, it is only possible to test the equivalence between the actual and the reversed program. Test cases obtained during normal working of the actual system are used. The equivalence test is modeled in Figure 3.

R.E.of Processes	Total Effort (h.)	Total Locs.	Locs/h.	Total Procedure	Effort for	Effort for	Effort for Normalized	Other Effort (h.)
Resource				Division locs	Physical Level (h.)	Logical Level (h.)	Logical Level (h.)	
1°Reverse-Operator	386	55504	143	4439	89.5	37.5	196	63
2°Reverse-Operator	406	65201	160	10728	51	42	310,75	25
3°Reverse-Operator	377	49588	131	5292	86	18	186	87
4°Reverse-Operator	372	64304	172	6496	66	10	221	75
5°Reverse-Operator	306	27089	88	3157	80.5	9.5	147	69

Table I. Measurements in process reverse engineering

R.E. of Data Resource	Total Effort	Redocumented	Productivity	Effort for	Effort for	Other effort (h.)
		Data		Physical Level (h.)	Conceptual Level (h.)	
1° Reverse-Operator	1966	3838	1.95	145	1523	298
2° Reverse-Operator	1278	1671	1.31	51	1146	81
3° Reverse-Operator	1744	2904	1.66	101	1513	130

Table II. Measurements in data reverse engineering

3: Operative results

The planned process was put in production in the scenario described earlier and after seventeen calendar months of work on a production line, the first results were obtained.

It should be noted that a production line refers to an organizational unit which has all the resources required for executing the process autonomously. In this case, the production line is composed of eight reverse operators and one reverse engineer. The former execute the procedures according to the defined process models while the latter coordinates activities and takes all the decisions necessary for solving all indeterminate points in the execution procedures. The production line shares an expert in the application domain with other organizational units in the company.

These first operative results can be analyzed from the point of view of both efficiency and efficacy. Although efficiency was not the main aim of this work, the data on the activities for reverse engineering of the programs and of the data are summarized in Tables I and II, respectively. Two important considerations can be made.

The productivity of the operators for reverse engineering the programs is correlated with their experience in the application domain; this explains the differences seen in Table I. In reverse engineering the data, the difference in productivity has less correlation with experience because there is very little automation of the activities and so the man time required is very high in any case.

The second point is that commercially available tools are often inadequate for large projects. For example, the tool used in extracting the data structure becomes unacceptably slow when access to previously inserted information is required, if the data are more than a thousand or if access is to an entity with more than one hundred attributes. The tool used for the programs, on the other hand, shows an abrupt drop in performance (answers to questions on data and control flow slow down) as soon as the threshold of 8000 lines of code is passed. Clearly, inefficient automatic tools require more man time to attain the objectives.

As regards the efficacy of the process, the following points can be made.

The logical data model is not very useful to the maintainer because he can read the information he needs more easily from the record layout described in code. This is due to the complicated structure of the files in the old system, which is difficult to clarify and represent. It is necessary, however, to know the relationships existing between the files that manage the system; there are many of these, so maintenance is a high risk procedure. For example, the relationships between the CUSTOMERS' INFORMATION file and the MORTGAGE: CLIENT TAKES OUT MORTGAGE is expressed in the field MT02-01 which represents the customer's number and univocally identifies in the CUSTOMERS' INFORMATION file all the information on the above client.

ProgramName	Module Name	mccabe	halstead	ProgramName	Module Name	mccabe	halstead	ProgramName	Module Name	mccabe	halstead
AA0000		713	141139	VA0000		516	187404	ER0000		235	67562
	AA0000	28	4832		VA0000	10	2088		ER0000	3	364
	M0001042	3	800		M0000015	1	145		M0002030	4	2380
	M0001041	5	1792		M0000017	1	1969		M0002029	5	3472
	M0001040	20	5463		M0000018	10	2504		M0002028	4	3176
	M0001039	42	9873		M0000019	20	4626		M0002027	5	2600
	M0001038	10	4617		M0000020	4	1316		M0002026	12	4048
	M0001037	16	4653		M0000021	4	1309		M0002024	1	1408
	M0001036	6	2408		M0000022	8	1900		M0002023	1	812
	M0001035	11	3616		M0000023	15	7083		M0002022	7	1848
	M0001034	7	2152		M0000024	25	5216		M0002021	5	990
	M0001033	9	3368		M0000025	7	1544		M0002020	1	726
	M0001032	22	5922		M0000026	75	38908		M0002019	2	1414
	M0001031	12	3672		M0000027	4	357		M0002018	1	708
	M0001030	14	3592		M0000028	3	252		M0002017	1	750
	M0001029	7	2240		M0000029	3	238		M0002016	1	744
	M0001028	1	1113		M0000030	3	165		M0002014	9	1386
	M0001027	28	12132		M0000031	6	1312		M0002013	18	4280
	M0001026	14	4472		M0000032	8	1224		M0002012	7	1253
	M0001025	9	5615		M0000033	7	1352		M0002011	9	2233
	M0001024	3	1894		M0000034	23	10512		M0002010	11	2030
	M0001023	6	3984		M0000035	5	448		M0002009	6	1211
	M0001022	2	1421		M0000036	13	1771		M0002008	10	2002
	M0001021	51	6872		M0000037	166	32230		M0002007	8	1393
	M0001020	8	1365		M0000038	39	6344		M0002006	29	6678
	M0001019	13	3040		M0000039	3	2698		M0002005	9	2226
	M0001018	10	3080		M0000040	4	1944		M0002004	24	5168
	M0001017	6	2128		M0000041	65	34710		M0002003	10	3744
	M0001016	4	654		M0000042	58	33410		M0002002	6	2401
	M0001015	176	31050		M0000043	55	16830		M0002001	1	220
	M0001014	15	2490								
	M0001013	28	9207								
	M0001012	3	2200								
	M0001011	44	6300								
	M0001010	22	8766								
	M0001009	4	480								
	M0001008	77	15606								
	M0001007	2	384								
	M0001006	10	1086								
	M0001005	5	642								
	M0001004	3	336								
	M0001003	3	434								
	M0001002	1	135								
	M0001001	1	2296								

Table III. Complexity of some programs and their extracted modules

During restoration, thanks to BACHMAN's tool [2] and VIASOFT's tool [14], and to the techniques used, the reverse operator extracts a lot of information which cannot be expressed in any of the documents produced. In particular, for many modules in the restored program, he will know not only their description but the algorithms themselves contained in the module. In fact, referring back in Figure 1, the document including the description of the behavior of the modules belongs to the flow named "restored logical model". Only a textual, not a formal, description of the aim of the module is provided for.

The data in the files formalize many design decisions taken during the past life cycle of the system, whose reasoning has been lost. As they affect the actual structure of the programs, their inadequate use prevents the reverse operator from being able to extract some functions implemented by the code, derived from these decisions which have left no traceable reasoning. For example, the field MT02-08, which represents the total amount the client must pay in instalments to pay off the mortgage, is calculated from the mortgage amount (MT02-02), the total number of instalments (MT02-05) and the interest rate (MT02-03). The design decision has decreed that this datum be stored rather than calculated each time. The main consequence is that many modules identified during the reverse engineering process have low cohesion and high complexity and will obviously be difficult to maintain. Table III shows the complexity of sets of modules extracted from some programs, which is, in some cases, still very high. Performing test cases helps to understand very complex modules.

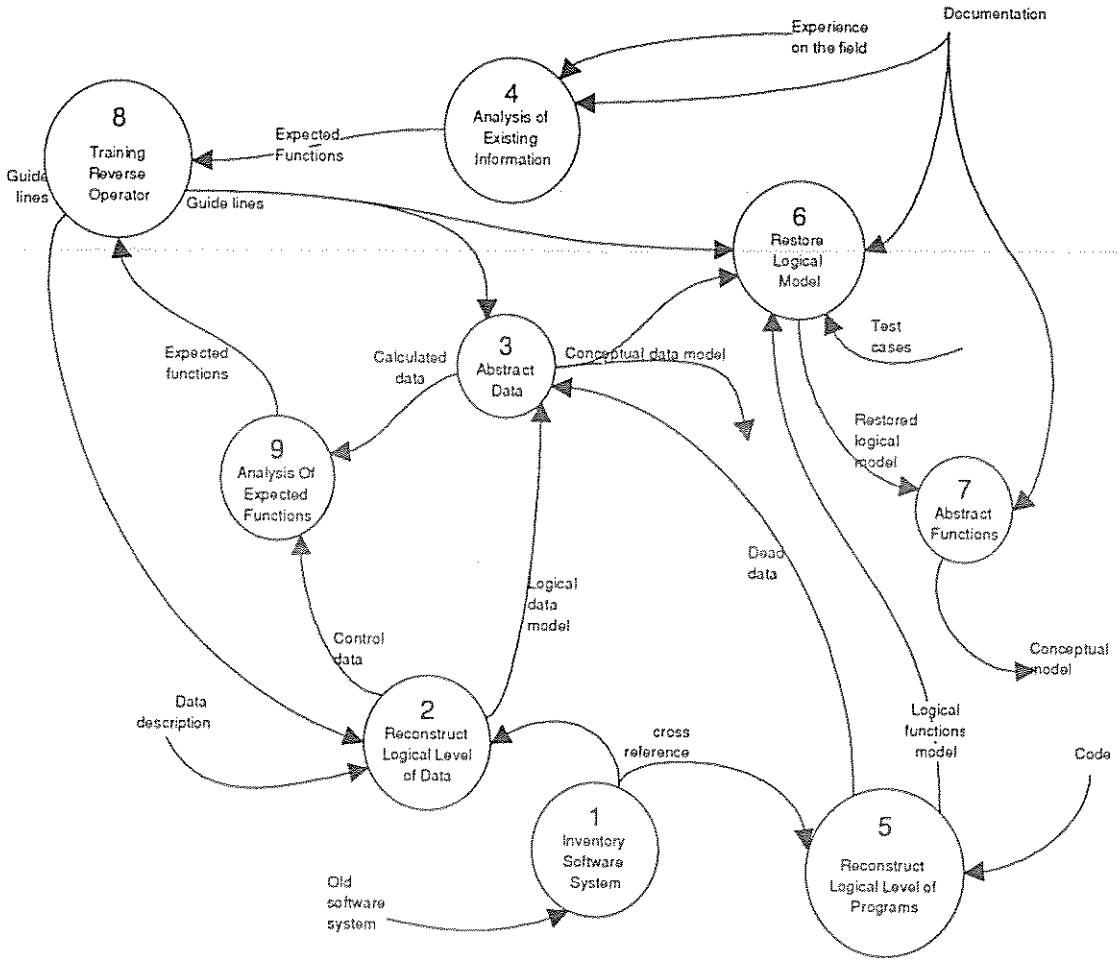


Figure 4. The modified reverse engineering process

4: Modifying the process

After the first experimental period on the field, some opportune modifications were made to the process model and to the product.

To increase the reverse operator's efficiency, depending on prior knowledge of the application domain, systematic training is necessary. It is not possible to have specialized reverse operators for each domain because this would make the "system to be reversed" - "operator to be used" pairing far too rigid. We therefore decided to alter the process, as can be observed in Figure 3, inserting training activities which, by using the expected functions, explain to the operators what they should find in the programs and in the data they process. This activity is carried out by the expert in the application domain and aims to provide the operator with guide-lines for performing the operative procedures detailed in the process. It is flexible, because the less the operator's prior knowledge of the application domain, the more exhaustive the training will be. It can be repeated as backup each time greater experience is required in the reverse operator.

It is not possible to formalize in the process the information that the tools, even when commercially known, must be carefully assessed for their efficiency, not only for their efficacy. Fortunately, in this case, the most inefficient tool was the one which derived the structure of the files, so reconstruction of the logical level of the data was modified (see Figure 2) to produce the classification of the data as conceptual, control or structural, and the description of the relationships between files.

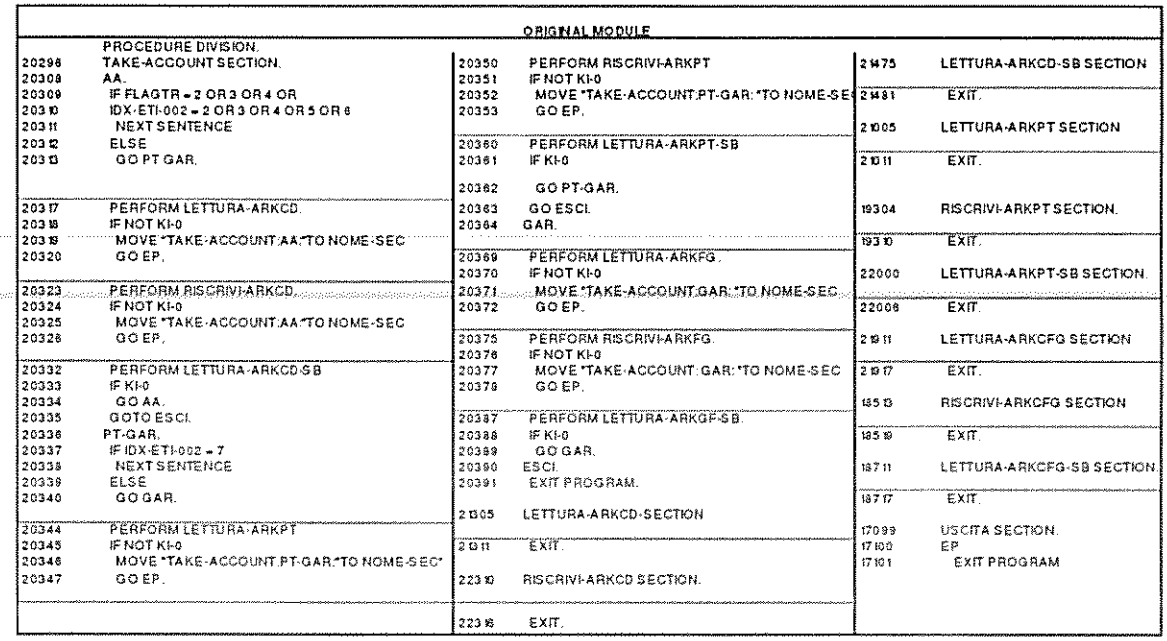


Figure 5. Original module for dynamic slicing

The formalization of the modules is included in the structure of the logical model deliverable, whose compilation standard changes. To increase the abstraction of the information on the decisions affecting the data, it is best that the control and structural data, first, and the calculated data, successively, be analyzed by the application domain expert so that he can complete the set of expected functions, where necessary. Thus, in the process model, changes have been made to increase the communication between the data and the process reverse engineering (see Figure 4).

Finally, the experience with the test cases suggested using them to perform dynamic slicing of too complex modules, during the restoration phase. A formal description of this dynamic slicing can be found in [11].

Given a module M and supposing it contains a set of functions (F_1,..., F_n); in the set of test cases used to verify the functions of M, some subsets of test cases (T_1,..., T_n) must be identified, in which the generic T_i contains the equivalence classes of function F_i. Thus the test cases T_i will activate in M all and only the instructions that implement the function F_i. These instructions constitute the module M_i corresponding to the function F_i. Once the modules (M_1,...,M_n) corresponding to (F_1,...,F_n) have been extracted, a complementary module M_c to the modules (M_1,...,M_n) can be found in M. M_c will be the module managing (M_1,...,M_n).

For example, for the module shown in Figure 5, the data for which the test cases have been constructed are:

IDX-ETI-002 :{1,...,20}

Index of types of transaction table

FLAGTR-WS :{0,...,9}

Work area flag whose value is paired with the index of types of transaction table.

The combined values of these two variables specify the exact nature of the transaction the client intends to make with the bank.

KI-0 :{true, false}

Control variable which states the result of access to the file. If true, access exists, if false, then access to the file was not possible.

The value of this variable is tested (in the cases in Figure 6) only on exit from PERFORM, which enables reading of the files.

Thus the original module was substituted by the structure shown in Figure 7.

The complexity is modified as shown in Table IV. This is formalized in the reverse engineering process model by modifying the procedure corresponding to the activity "reconstruct logical level of programs". In fact for all modules exceeding a predefined level of complexity, dynamic slicing will be performed using the test cases relative to the module in question.

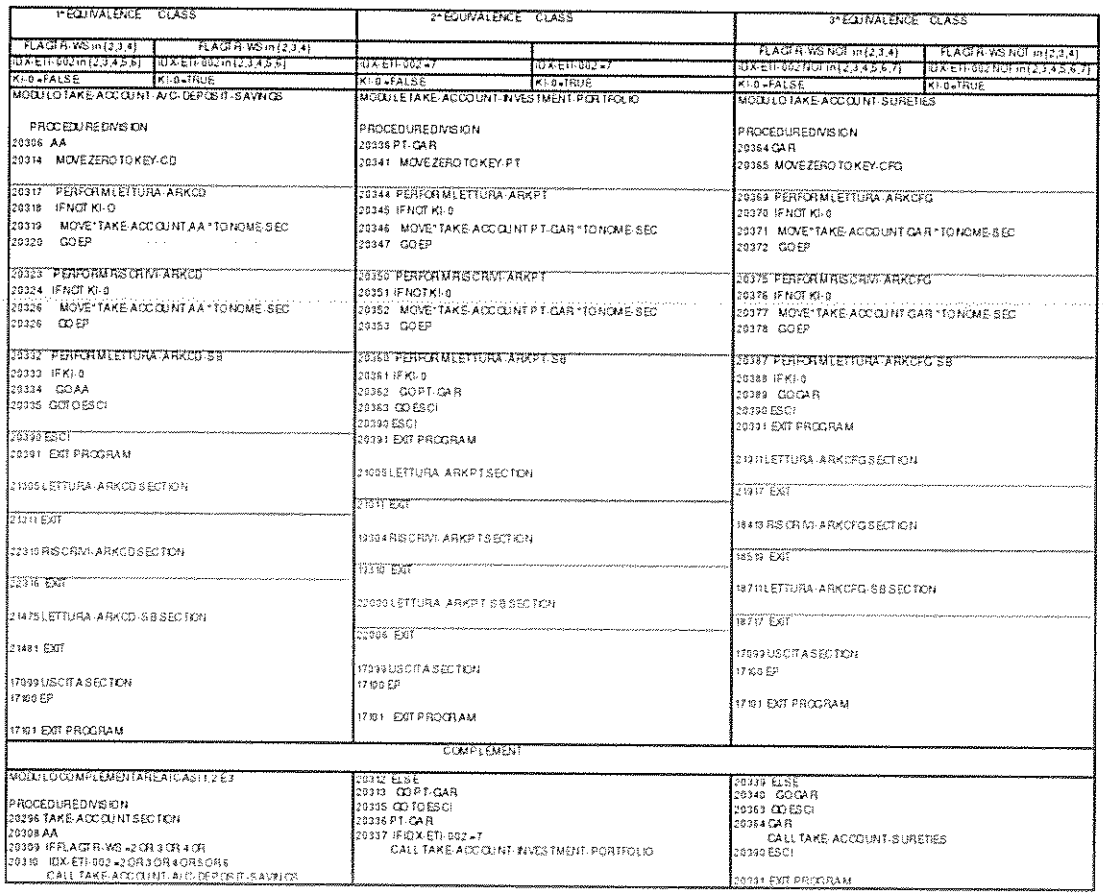


Figure 6. Results of dynamic slicing

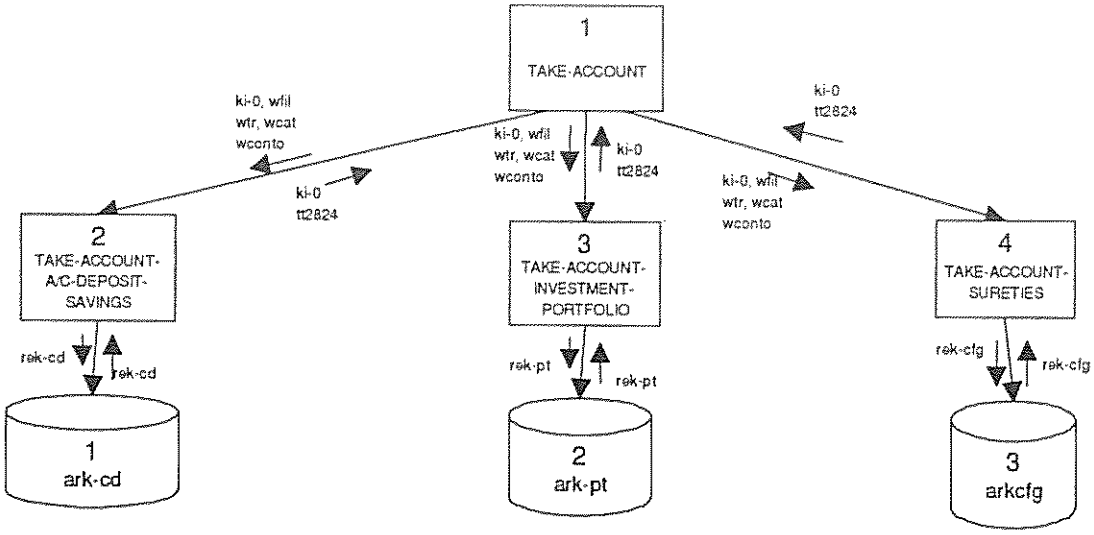


Figure 7. Program structure after dynamic slicing

Program/Module	Halstead	McCabe
Original program	3276	12
TAKE-ACCOUNT-MODULE	1512	3
TAKE-ACCOUNT-A/C-DEPOSIT-SAVINGS-MODULE	1920	4
TAKE-ACCOUNT-INVESTMENT-PORTFOLIO-MODULE	1873	4
TAKE-ACCOUNT-SURETIES-MODULE	1813	4

Table IV. Module complexity before and after extraction

5: Conclusions

This paper describes the results of experimentation on the field of a reverse engineering process and the improvements made on the basis of the data obtained.

The feedback gained from the quality of the product and the quality of the process is particularly enlightening. In this case, the usability of the documentation to understand the programs better suggested some improvements to the process. This feedback enabled closer connection to be made in the process model between the static and the dynamic information which can be extracted from the application domain and the user context of programs.

In addition, a synergetic analysis in the process of learning about the programs was defined. The programs to be documented were so complex that dynamic analysis alone would have been inadequate: in fact, in the original programs, each test case would have activated so many instructions that a great deal of man time would have been required to understand them. Instead, using prior static analysis, the programs were decomposed and classified essentially into two types: modules whose aim and behavior are clear, and modules whose aim is clear but behavior is not. The latter, more complex modules, can be understood better after dynamic analysis with the test cases used to verify the results of the reverse engineering process.

Unfortunately, the modified process has been operative for too short a time for data to be available to assess its quality after the modifications.

Finally, the experience highlighted the need to improve testing and validation of the functionality and efficacy of the tools used. Even well known tools which have been on the market some time may be inadequate because, for the most part, they have been used in pilot projects rather than effective production. Hence, the reverberations of their actions are still unknown.

References

[1] F. Abbattista, F. Lanubile, and G. Visaggio, "Recovering conceptual data models is human intensive", Fifth International Conference on Software engineering and Knowledge engineering, San Francisco, California, 1993.

[2] BACHMAN, *Bachman/Analyst Reference Manual*, Release 4.10.

[3] G. Canfora, A. Cimitile, and U. De Carlini, "A logic based approach to reverse engineering tool production", *Conference on Software Maintenance*, Sorrento, Italy, 1991.

[4] G. Canfora, A. Cimitile, and M. Munro "RE²: reverse engineering and reuse re-engineering", *Computer Science Technical Report 8/92*, University of Duhram, School of Engineering and Computer Science, 1992.

[5] E. J. Chikofsky, and J. H. Cross II, "Reverse engineering and design recovery: a taxonomy", *IEEE Software*, January 1990.

[6] G. Como, F. Lanubile, and G. Visaggio, "Design recovery of a data-strong application", *3rd International Conference on Software engineering and Knowledge engineering*, Illinois,USA,1991.

[7] F. Cutillo, F. Lanubile, and G. Visaggio, "Extracting application domain functions from old code: a real experience", *2nd Workshop on Program Comprehension*, Capri, Italy, 1993.

[8] F. Cutillo, P. Fiore, and G. Visaggio, "Identification and extraction of domain independent components in large programs", *Working Conference on Reverse Engineering*, Baltimora, 1993.

[9] F. Cutillo, F. Lanubile, and G. Visaggio, "Using program slicing for software comprehension", *IEEE Workshop Notes on Software Comprehension*, Orlando, Florida, 1992.

[10] FORMATICA, "Definition of the production line. reverse engineering: process model", *Int. doc. no. 69*. September 1992

[11] R. Gopal, "Dynamic program slicing based on dependence relationships", *Conference on Software Maintenance*, Sorrento, Italy, 1991.

[12] F. Lanubile, and G.Visaggio, "Function recovery based on program slicing", *Conference on Software Maintenance*, Montreal, Quebec, Canada, 1993.

[13] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, vol.SE-10, n°4, July 1984.

[14] VIASOFT, *VIA/Renaissance User Manual, VIA/Insight User Manual, VIA/SmartDoc User Manual.*

Session D:
Non-Traditional
Analysis Techniques

Dynamic Code Cognition Behaviors For Large Scale Code

A. von Mayrhauser

Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

A. M. Vans

Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

Abstract

This papers describes code cognition behaviors when maintenance engineers try to understand large-scale code. It reports on low level and higher level aggregate comprehension processes, hypotheses, and strategies. Results are based on the integrated meta-model of code cognition and use protocol analysis of code cognition sessions.

1 Introduction

Existing program understanding models agree that comprehension proceeds either *top-down*, *bottom-up*, or some combination of these two. Our observations [11] indicate that program understanding involves both top-down and bottom up activities and led to the formulation of a model that integrates existing models as components. This integrated code comprehension meta-model consists of (1.) *Program model*, (2.) *Top-down model*, (3.) *Situation model*, and (4.) a *Knowledge base*.

The basis for the top-down (also known as *Domain model*) component is Soloway and Ehrlich's [7] top-down model while Pennington's [4] program and situation models are reflected in the program and situation model components of the meta-model. These three model components reflect mental representations and the strategies used to construct them. They represent views of the code at various levels of abstraction. The knowledge component is necessary for successfully building the other three models. Thus the complete meta-model describes program, situation, and top-down model building together with the appropriate knowledge for construction of a mental model of the code.

Each model component (as well as the meta-model) builds up knowledge using what is already known about the domain, the architecture or environment, and the code. Each model component represents both the internal representation of the code (or *short-term memory*) and a strategy to build this internal representation. The knowledge base furnishes related but previously acquired information. During understanding, new information is chunked and stored into the knowledge base (or *long-term memory*) for future use.

The *Top Down* model is typically active if the code or type of code is familiar. The top-down representation consists of knowledge about the application domain. When code is completely new to the programmer, Pennington found that the first mental representation programmers build is a *program model* consisting of a control flow abstraction of the program [4]. Once the program model representation is constructed, a *situation model* is developed. This representation, also built from the bottom up, uses the program model to create a data-flow/functional abstraction.

The *knowledge base*, also known as long-term memory, is usually organized into *schemas* (or *plans*). Schemas are knowledge structures with two parts: slot-types (or templates) and slot fillers. Slot-types describe generic objects while slot fillers are customizations that fit a particular feature. Data structures like lists or trees are examples of slot-types and specific program fragments are examples of slot-fillers. These structures are linked by either a *Kind-of* or an *Is-A* relationship. Schemas are grouped into partitions specifically related to the comprehension processes. For example, knowledge of algorithms is used by the *program model building* process.

A key feature of the integrated meta-model [11] is that any of the three model components may become active at any time during the comprehension process. Structures built by any of the three model components are accessible by any other; however, Figure 1 shows that each model component has its own preferred types of knowledge. [11] contain thorough discussions of the integrated meta-model and its component models.

The integrated code cognition model includes knowledge and information. Previously, we described information needs of maintenance engineers as they understand large scale code and how tools and maintenance environments could support resolving these information needs [11].

Code understanding, however, is dynamic. We need to explore further what aspects of dynamic behavior exist and whether we can describe such behavior with common cognition processes. Preliminary results were reported in [10]. Section 2 describes the objectives of this study. The first step to describe cognition dynamics is by exploring how maintenance programmers switch between components of the meta-model. Next, we investigate whether low level processes exist and whether they aggregate into higher level processes. We

0-8186-5647-6/94 $04.00

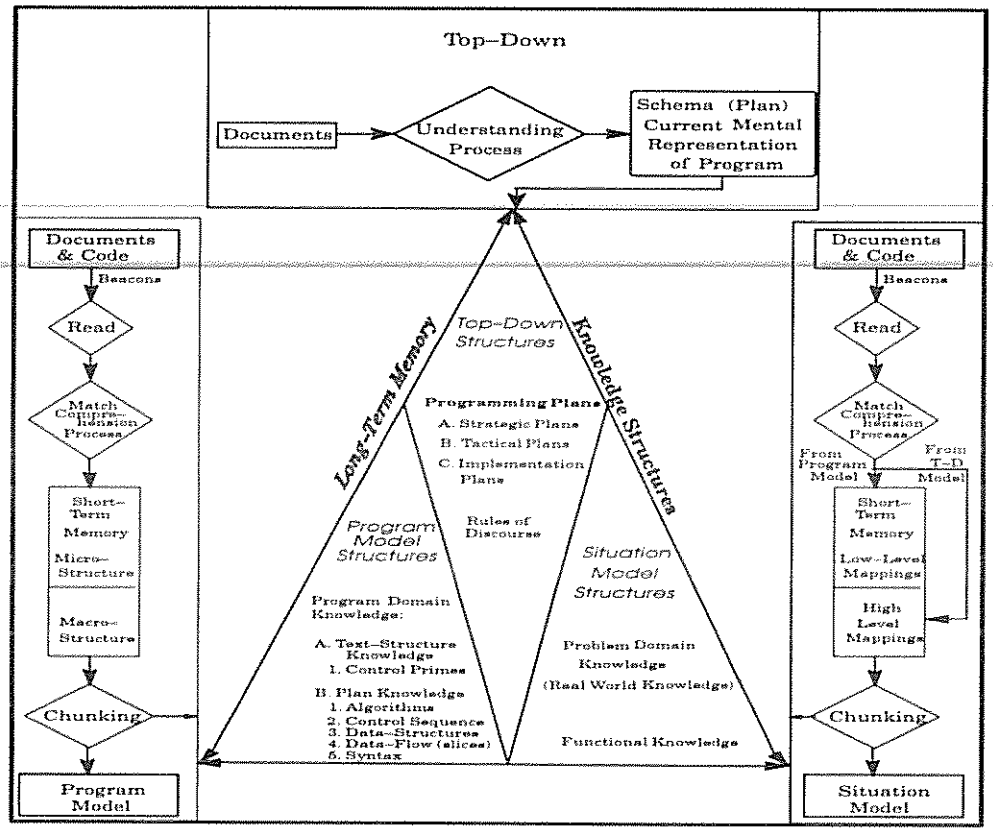


Figure 1: Integrated Code Comprehension Meta-Model

also determine the role of hypotheses and strategies as part of the meta-model.

Section 3 details the experimental design and analysis method. We used protocol analysis for cognition process discovery. Section 4 reports on the results of the analysis. This includes the nature of dynamically occurring switches between the components of the meta-model during understanding, episode (low level) processes, and their aggregation to higher level (session level) processes. Hypotheses and strategies play a major role in these processes. Section 5 offers conclusions and further work.

2 Study Objectives

1. *The role of Model Components in the Integrated Meta-Model.* Subjects frequently switch between all model components (i.e. understanding is built at all levels of abstraction simultaneously) [11]. Is there a difference in working at levels of abstraction based on the size of the code under consideration? Answers to this question affect type of knowledge, cognition process, and expertise best suited to large scale code understanding.
2. *Episodes.* We have seen the types of actions engineers execute while working on maintenance tasks [11]. Are there repeated action sequences (episodes) representing lowest level strategies? How similar are they? Which types of episodes occur most frequently? What information does the engineer need to complete an episode? Do episodes represent understanding at only one level of abstraction or do they span all levels?
3. *Aggregate Processes.* How are episodes used in higher level understanding strategies? Are there repeated episode sequences (aggregate processes)? How similar are they? What *triggers* the end of one episode and the beginning of another? Are some of these triggers more common than others?
4. *Session-Level Processes.* Do aggregate processes help in defining a maintenance task process (session-level)? Are there repeated sequences of aggregate processes? How do programmers switch from one aggregate process to another? If we find one process for each type of maintenance task, what are their similarities?
5. How are *hypotheses* used in the understanding process?
6. What *strategies* do maintenance engineers use during comprehension? How do they relate to hypothesis generation, processes, and meta-model construction?

3 Experiments

3.1 Experimental Design

The purpose of our study was to find a code comprehension *process model* as part of the Integrated Com-

prehension Meta-model. We also wanted a high-level preliminary validation through observation. Each observation involved a *programming session*. Participants were asked to think aloud while working on understanding code. We audio and video taped this as a thinking aloud report. Sessions were typically two hours long. Table 1 defines three major variables for our study. The columns represent *expertise*, the rows represent the amount of accumulated knowledge subjects had acquired *prior* to the start of each observation. The *type of maintenance task* is listed as an entry in the matrix. Each square represents specific observations that are characterized by the row, column, and maintenance task. Abbreviations, for example *C2*, are used in the rest of this paper to identify individual subjects. As the matrix shows (Table 1) these eleven

Expertise⇒ Accumulated Knowledge⇓	*Language Expert*	*Domain Expert*	Language & *Domain Expert*
Never Seen Before		**C2**: Understand Bug	
File Structure Call Graph		**C3**:Fix Reported Bug	**G1**:Program: General Understand **EN2**:Add Function
Requirement & Design Documents	**C1**:Fix Reported Bug		**G2**:Understand one Module
Worked some with code, style familiar	**L1**:Leverage Small Program	**C4**:Track Down Bug	
Prior code enhancement, debugging, adaptations		**AD2**:Add Function, Prototype Assess	**AD1**:Port Program across Platforms
Worked with code several years			**EN1**:Add Functionality

Table 1: **Programming Sessions – All Maintenance Tasks**

subjects represent good coverage in terms of a varying degree of knowledge about the task and expertise. All tasks represented actual work assignments.

3.2 Protocol Analysis

Protocol analysis proceeded in three steps (see Table 2). The following describe the criteria used to classify statements, identify information needs, and analyze protocols for discovery of processes.

Enumeration	*Segmentation*	*Process Discovery*
Utterance | Action	1. Abstraction level 2. Action types 3. Information Needs	1. Episode level processes 2. Aggregate level processes 3. Session level processes

Table 2: **Protocol Analysis Steps**

1. The first analysis on the protocols involved *enumeration* of *action types* as they relate to the integrated cognition model of [11]. Action types classify programmer activities during a specific maintenance task. Examples of action types are "generating hypotheses about program behavior" or "mental simulation of program statement execution". We began with a list of expected actions [9] and searched for them in the transcripts of the protocols. We also analyzed for possible new action types. Table 3 contains example protocols to show action type classification.

2. The next step in the analysis combines *segmentation* of the protocols and identification of information and knowledge items. Segmentation classifies action types into those involving top-down, situation, or program model and can be thought of in terms of different levels of abstraction in the mental model. *Information Needs* are information and knowledge items that support successful completion of maintenance tasks.

3. We discovered *dynamic* code understanding *processes* by classifying and analyzing *episodes*. Episodes are single instances of a sequence of action types. An episode starts with a *goal* and embodies the actions to accomplish that goal. For example, determining the function of a specific procedure or routine may entail a sequence of steps that include reading comments, following control flow, and generating questions when a concept is not understood. *Processes* are defined at three different levels; *episodic*, *aggregate*, and *session* levels. Episodes containing common action types with similar goals are defined as a single episodic process. Likewise, common sequences of episodic processes are defined as a single aggregate level process. Finally, the session level process is established by a sequence of similar aggregate level processes.

   A process is a sequence of action types, episodes, or aggregates whose purpose is to satisfy a specific goal. Specifically, each episode is determined by discovering the goal and cataloging all subsequent action types until reaching closure on the goal due to goal satisfaction or goal abandonment. Once episodes are identified, we analyze each, abstract out commonalities, and designate the resulting sequence an episodic process. An aggregate level process emerges from similar episodic processes. Similarly, sequences of aggregate processes are analyzed for commonalities and abstracted into higher-level processes. Once again, common sequences of aggregate level processes produce a single session level process representing a two hour programming session. Section four illustrates these processes using state diagrams.

4 Results

In this section we present the components that constitute the dynamic behaviors we identified during the programming sessions: 1) switches between meta-model elements, 2) processes, 3) hypotheses generation and confirmation, and 4) strategies. We focus on one subject's dynamic behaviors to illustrate the results.

Analysis Type	*Tag*	*Action Type*	*Example Protocol*
Action-Type Classification	Sys8	**Generate Hypothesis (Program Model)**	"..and my assumption is that nil with a little *n* and nil with a big *N* are equivalent at the moment."
	Sys7	**Chunk & Store knowledge (Program Model)**	"So clearly what this does is just flip a logical flag"

Table 3: **Example Protocol Analysis – Action Types**

Our example subject (G2) was in the process of understanding one module in a system for which he recently took over responsibility. The program consists of non-standard PASCAL of approximately 90,000 lines of code. During the programming session he was interested in thoroughly examining the main procedure which controlled a majority of the system. The engineer was an application expert (six years professional experience). He was very familiar with three different versions of Pascal as well as MS-DOS assembly language.

The subject preferred to work with a hard copy of the code and used a computer for tasks like searching for variable use/definitions and for writing mail questions to other experts. He annotated the hard copy with information which he frequently referred to during the session. He was very systematic, following the code, line by line and attempting to understand everything before going on to the next line. At the end of the two hours he had successfully understood the main procedure with very few unanswered questions.

4.1 Switches

A switch is a change of focus from one model of comprehension to another. For example, if an engineer is currently focusing on building a program model of the code and reads a comment that refers to a higher level concept, a switch to the situation model may occur causing the new information to be incorporated into the current situation model.

Table 4 summarizes the total number of references to meta-model components and frequency of switches between them for G2. There are a significant number of references to the program model and there are more switches between Program (Prg.) and Situation (Sit.) models than between Program and Situation or Program and Top-down (T-D) models. This is expected since the subject was building his mental model from the bottom-up. When he worked at the situation model level, he tended to switch back to the program model. When he worked at the program model level, he spent large amounts of time before moving on to the top-down or situation model. This accounts for the significant number of references to the program model as compared to the number of references at other levels. Finally, there are significantly more switches between the top-down and program model than between the program and situation models because our subject was unfamiliar with the X-25 protocol and spent most of his time trying to connect chunks of program level statements to higher level functional descriptions, relating to this type of protocol.

There are significantly more references to the program model than to either the situation or top-down models. Since our subject was studying a single module of an application and lacked some domain knowledge, he concentrated mostly on the program model.

Number of References		*Model Switches*		
	Model	*T-D Model*	*Sit. Model*	*Prg. Model*
56	T-D	N/A	6	24
34	Sit.	7	N/A	22
201	Prg.	23	25	N/A

Table 4: **References & Switches Between Models**

4.2 Processes

Processes are dynamic code understanding activities consisting of *Episodes*, *Aggregate Processes*, and *Session Level Processes* as discussed in subsection 3.2. Here we illustrate the types of processes discovered in G2's program understanding protocol.

4.2.1 Episodes

Episodes are sequences of action types carried out to accomplish a goal. Episodes containing common actions with similar goals emerge as the lowest level processes. To illustrate, in our example protocol, episodic process *P1 – Read Block in Sequence* starts with the overall goal of understanding a specific block of code, e.g. "I'm going to read the description and see if it gives me some good clues as to what's going on." Some of the observed actions that support the original goal are: generating hypotheses while reading comments, chunking information, making note of interesting aspects, and postponing investigation of them.

Table 5 lists seven episodic processes and how often they occurred in the transcript. The subject spent the majority of his time reading the code, determining the behavior of a variable, and incorporating this knowledge into his mental model of the program module. The engineer applied a *systematic strategy* of reading each line of code in approximate sequence. Figure 2 presents this process in graph form as a state machine. Arcs indicate action types while states represent level of understanding. Processes P1 and P3 were the most frequent. These two processes are preferred during detailed understanding of one module. Based on Table 5, we expect to find other processes, such as P2, more frequently referenced when understanding larger code segments.

Individual episodes can vary greatly, because their goals are very different. An episode may use the same action types as another but occur in a different order.

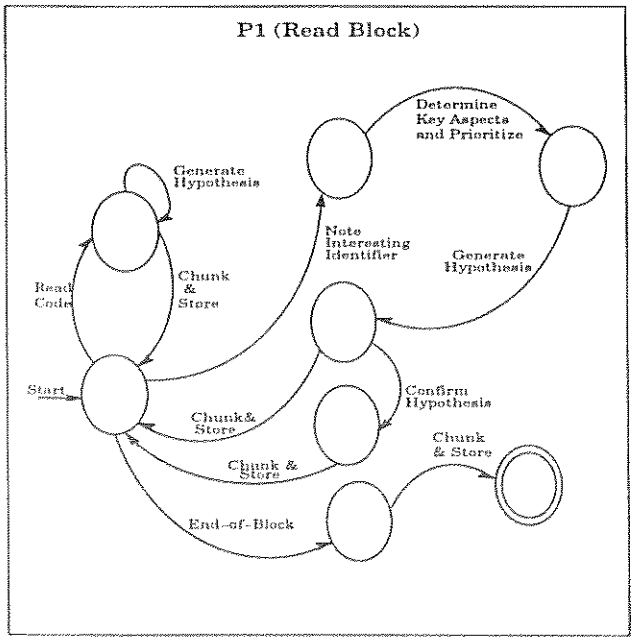


Figure 2: **Episodic Process – Read Block**

During our analysis we were able to associate information needs (and their frequencies) with action types and thus with episodic processes. Table 6 shows information needs for process P1 for subject G2. The three most frequently needed information types for P1 directly relate to the activities shown in the state diagram. E.g. determining the end-of-block condition requires code block boundary information.

Interestingly, we could not find processes that occurred on a single level of abstraction and therefore classified as purely top-down, situation, or program model processes. Many episodes contained actions that were associated with all three integrated model components. This supports the idea that programmers switch between model components (levels of abstraction).

Episodic Process Name	*Code*	*Number*
Read Block in Sequence	P1	7
Integrate Not Understood	P2	4
Determine Variable Def/Use	P3	7
Incorporate Acquired Program Knowledge	P4	5
Identify Block Boundaries	P5	2
Resolve Deferred Questions	P6	2
Understand a Procedure Call	P7	1

Table 5: **Episodic Process Frequency Count)**

4.2.2 Aggregate Processes

Three aggregate processes were discovered in G2's protocol and we illustrate one below in the form of a state diagram. Table 7 shows that aggregate processes consist of episodic processes.

Table 8 shows frequencies of aggregate level processes. At the aggregate level, processes PA, PB, and PC begin to look very similar. One conjecture is that these aggregate processes represent instances of a similar higher level strategy.

Episodic Process	*Information Needs*	*Number*
P1: Read Block	Code Block Boundaries	4
	Data Type definitions & location of identifiers	3
	Call Graph Display	2
	History of past modifications	1
	Data structure definitions tied to domain concepts	1
	Location of called procedures	1
	History of browsed locations	1
	Beacons tied to situation model or program model	1
	Description of system calls	1
	Location of documents for program & domain	1

Table 6: **Information Needs for Process P1**

Triggers cause state changes between processes. They can be code induced (e.g. end of code block) or an action type (e.g chunk and store knowledge). Table 9 lists the triggers found in the example protocol and their frequencies for Process PC and the total for all the aggregate processes. Beacons and end-of-block triggers were the most frequent triggers. Again, this could be a by-product of the systematic strategy used by this subject. E.g., a jump out of episodic process P1 (Read Block in Sequence) into process P7 (Understand Procedure Call) is caused when G2 encounters an unrecognized procedure call (a beacon). He decides to understand what the procedure does. After investigating it he reaches its end (End-of-block) which triggers the end of P7 and resumption of P1 where he last left off.

Episodic Process	*PA*	*PB*	*PC*
P1:Read Block in Sequence	X	X	X
P2:Integrate Not Understood	X		
P3:Determine Variable Def/Use	X	X	X
P4:Incorporate Acquired Program Knowledge		X	
P5:Identify Block Boundaries		X	
P6:Resolve Deferred Questions			X
P7:Understand a Procedure Call			X

Table 7: **Aggregate Processes – Episodic Composition**

4.2.3 Session Level Processes

The state diagram in Figure 4 was derived in the same way as the aggregate-level processes by tracking the sequences of aggregate-level processes. This diagram represents a general understanding maintenance task. At the highest level, only "End-of-block" and "Chunk & Store" cause switches from one aggregate-level process to the next.

The session-level process (for Understanding a single module) shows that all the aggregate-level pro-

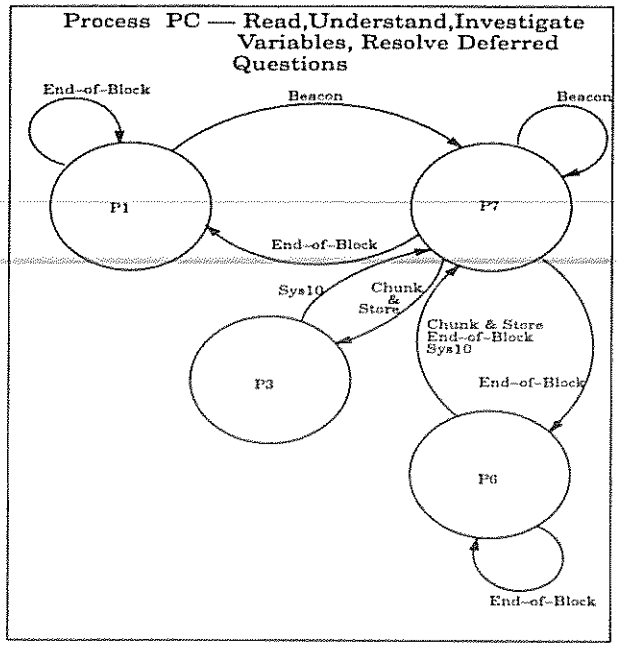


Figure 3: **Process PC – Aggregate-Level**

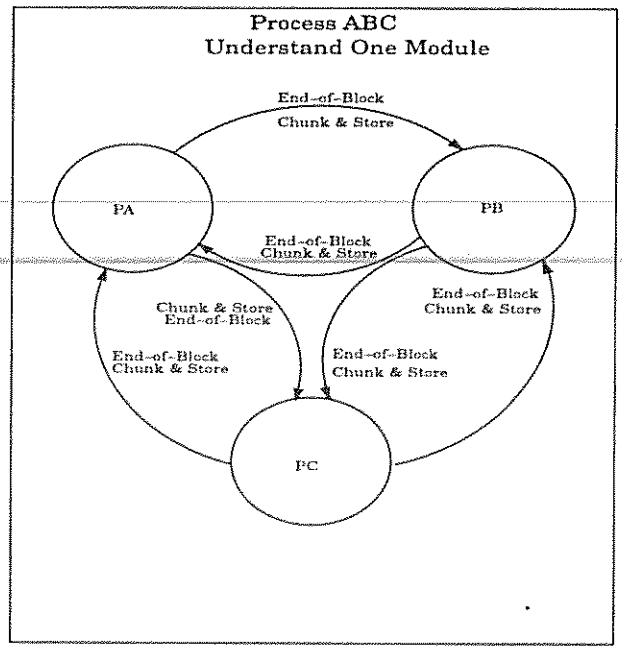


Figure 4: **Process ABC – Session Level**

cesses represent investigation towards building chunks [4, 6]. Thus, at the session-level the purpose of each aggregate process is to understand a block of code (using different detail steps and information) and then to chunk and store the learned information.

Aggregate Process Name	*Code*	*Number*
Read, Integrate, Investigate variables	PA	4
Read, Incorporate acquired program knowledge, Investigate variables, Identify Block Boundaries	PB	3
Read, Understand, Investigate variables, Resolve deferred questions	PC	3

Table 8: **Aggregate Process Frequency Count**

Process Trigger	*Frequencies*	
	PC	*All Aggregate Procs*
Beacon	7	14
Chunk & Store Knowledge	2	8
End-of-Block	7	20
End-of-Stack	0	2
Understanding strategy determined (Sys10)	2	7

Table 9: **Process Trigger Frequencies**

4.3 Hypotheses

Hypotheses are important drivers of cognition. They help to define the direction of further investigation. Generating hypotheses about code and investigating whether they hold or must be rejected is an important facet of code understanding. In our example protocol the programmer generated 63 hypotheses. Only 11 of these were confirmed. The programming session generated 13 top-down model hypotheses, 10 situation model hypotheses, and 40 program-model hypotheses. Again, this is expected since subject G2 spent the majority of his time building the program-model.

Hypotheses are related to model switching since they often drive a switch to another component model. Table 10 contains the total number of switches caused by a hypothesis related to a different comprehension model component. The row indicates the starting model and the column represents the ending model. The labels *T-D* represent the top-down model, *Sit.* the situation model, and *Prg.* the program model.

Number of References Enhancement	*From Model*	*To Model*		
		T-D Model	*Sit. Model*	*Prg. Model*
6	T-D	N/A	0	6
8	Sit.	2	N/A	0
5	Prg.	5	6	N/A

Table 10: **Hypotheses Switching Behavior – By Meta-model Component**

A majority of the switches occurred from either the top-down or situation model into the program model. Again, this is expected when focusing mainly on program model building. Table 11 contains type of hypotheses by meta-model component. The first column indicates the model component, the second column

Meta-Model Component	*Hypothesis Type & Frequency*	
Top-Down	Domain routine functionality	6
	Domain variable functionality	2
	Based on Rule of Discourse	1
	Type of I/O facility	1
	Unclassified	3
Program Model	Variable function	11
	Routine function	5
	Routine call function	3
	Variable structure	3
	Where routine called from/type of routine	2
	Statement execution order	2
	Variable default value	2
	Existence of construct	1
	Two variable equivalent	1
	Syntax meaning	1
	Existence of routine call	1
	Location of variable definition	1
	Two routines are similar	1
	Function of code block	1
	Unclassified	5
Situation Model	Variable function	3
	Function execution order	2
	Routine-call function	1
	Unclassified	4

Table 11: **Hypotheses – Types by Meta-Model Component**

contains the hypothesis type. For example, Top-Down model hypotheses include hypotheses about the functionality of a routine at the domain level. The most frequent types of hypotheses generated by G2 concern variable or routine functionality. Hypotheses about variable function for the program model were the most frequent (42% of program model hypothesis, 17% of total hypotheses, all models). For example, during the programming session, G2 came across a variable called **TMODE** which he did not understand. He created several unresolved questions and hypotheses. The following excerpt from the protocol illustrates an unconfirmed hypothesis about **TMODE**:

So, what we want to do here is put in a little comment that says, "Hypotheses T-MODE is a variable used to cause output to be echoed on screen either to STANDARD-OUT or STANDARD-ERROR."

At the program model level we find hypotheses related to micro-structure [4] and to macro-structure [4]. Micro-structure hypotheses are concerned with a single statement or variable definition. The higher level macro-structure hypotheses are concerned with *chunks* of statements, e.g. routine (control-flow) or variable (data-flow) behavior. Subject G2 primarily generated micro-structure hypotheses.

4.4 Strategies

A strategy is defined as the high-level or overall approach to comprehension. It is a *style* or method of understanding. A strategy can affect processes at the episode level, determining the sequence of actions to take. G2 used a *systematic* strategy [2, 3] in which he read the code almost line by line. If he had instead used an *opportunistic* strategy [2, 3] in which only code thought relevant is looked at, the sequence of actions might be very different. This in turn affects the cognition processes at higher levels.

Cross-referencing [5] is a strategy that builds links between the program model and the situation model. For example, if a chunk of code is suddenly recognized as a search routine, a cross-reference from the program model (lines of code in the chunk) to the situation model (search routine) is established.

More than one strategy may be used during program comprehension. Overall, G2 applied a systematic approach during the two hour session. G2 switched frequently between program and situation models and between program and top-down models (see table 4). A close look at the transcript reveals that switches between program and situation or program and top-down models established links between the two models. However, there are also a considerable number of switches between situation and program and between top-down and program models. These switches were not intended to establish a link between models, instead they were a return to a previous position in the program model. 58% of the hypotheses caused a switch to a different model component while the remaining 42% were returns to the model under construction prior to switching.

5 Conclusion

Program understanding is a key factor in software maintenance and evolution. This paper reported on an experiment with industrial programmers to discover the dynamic comprehension processes and the supporting information that programmers use when trying to understand production code. There are four major components of dynamic code cognition: switches between meta-model components, processes, hypotheses, and strategies. A protocol from one subject provided examples of these components and we have shown some interactions between them.

While our sample was small, this exploratory experiment showed a variety of interesting results:

- Programmers use a multi-level approach to understanding, switching between program, situation, and domain (top-down) models. The focus of meta-model component construction is influenced by the size of the code to be understood.
- Maintenance programming activities can be described by a distinct small set of cognition processes. These can be aggregated into higher level processes.
- Several types of hypotheses are generated during code understanding. There is also a link between hypotheses and switching behavior, i.e. hypotheses can cause switches between meta-model components.
- Strategies are important elements of cognition. We saw evidence of two, systematic and program-model to situation model cross-referencing, throughout the entire session.

However, this analysis only scratches the surface of dynamic program cognition. There are still many

unanswered questions which require additional research. In particular:

- We hypothesize that switches represent one of the *strategies* maintenance engineers use during understanding activities. If the size of the component is small enough to understand at a low level of detail, then it makes sense to spend most of the time in the program model. [2] showed that code changes are more successful if the engineer takes a systematic approach and understands every line of code. On the other hand, if the component size is more on the order of 85,000 to 90,000 lines of code (as with a few of our subjects) then understanding must necessarily occur at a higher level of abstraction. At the highest level of abstraction is domain knowledge. One ideally would like to understand at this level if the component to understand is large.

- Our example demonstrates that strategies affect the dynamic cognition process and switching behavior between model components. If an opportunistic approach was used instead of the systematic approach we saw, the sequence of actions comprising the episodes might be very different. This may result in very different aggregate level processes which may in turn define different session-level processes.

- We reported on process discovery for one transcript. We expect that with more experiments, additional episodes will be identified.

- It is unclear how hypothesis generation is affected by choice of strategy. It may be that hypothesis generation may affect choice of strategy. What is clear is the link between hypotheses and switching behavior, i.e. hypotheses cause switches between meta-model components. Whether this behavior is driven by a strategy or drives a strategy needs further investigation.

- We hypothesize that the cross-referencing strategy applies between all meta-model components and that both component size and style play a part in determining use of a particular cross-referencing strategy. If the code to be understood is an entire system, we would expect a cross-referencing strategy between the situation and top-down models. On the other hand, if the component is a small module, then a cross-referencing strategy between the program and situation models would be more appropriate. Switching behavior between meta-model components is a good indicator of the type of preferred cross-referencing strategy.

Acknowledgements

This research was partially supported by the Hewlett-Packard Co.,Inc.

References

[1] Edward M. Gellenbeck and Curtis R. Cook, **An Investigation of Procedure and Variable Names as Beacons during Program Comprehension**, Tech Report 91-60-2, Oregon State University, 1991.

[2] Jurgen Koenemann and Scott P. Robertson, **Expert Problem Solving Strategies for Program Comprehension**, In: CHI'91, March 1991, pp. 125-130.

[3] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway, **Mental Models and Software Maintenance**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 80 - 98.

[4] Nancy Pennington, **Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs**, In: Cognitive Psychology, 19(1987), pp.295-341.

[5] Nancy Pennington, **Comprehension Strategies in Programming**, In: Empirical Studies of Programmers:Second Workshop, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 100 - 112.

[6] Ben Shneiderman, **Software Psychology, Human Factors in Computer and Information Systems**, In: Chapter 3, ©1980, Winthrop Publishers, Inc., pp. 39-62.

[7] Elliot Soloway and Kate Ehrlich, **Empirical Studies of Programming Knowledge**, In: IEEE Transactions on Software Engineering, September 1984, Vol. SE-10, No. 5, pp. 595-609.

[8] Elliot Soloway, Beth Adelson, and Kate Ehrlich, **Knowledge and Processes in the Comprehension of Computer Programs**, In: *The Nature of Expertise* , Eds. M. Chi, R. Glaser, and M.Farr, ©1988, ALawrence Erlbaum Associates, Publishers, pp. 129-152.

[9] Iris Vessey, **Expertise in debugging computer programs:A process analysis**, In: International Journal of Man-Machine Studies, (1985)23, pp.459-494.

[10] A. von Mayrhauser and A. Vans, *Comprehension Processes During Large Scale Maintenance*, In: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 39-48.

[11] A. von Mayrhauser and A. Vans, *From Program Comprehension to Tool Requirements for an Industrial Environment*, In: Proceedings of the 2nd Workshop on Program Comprehension, Capri, Italy, pp. 78 -86, July 1993.

Abstraction Mechanisms for Pictorial Slicing

Daniel Jackson and Eugene J. Rollins
School of Computer Science
Carnegie Mellon University

Abstract

Big programs tend to have big slices, so reverse engineering tools based on slicing must apply additional abstractions to make slices intelligible. We have built a tool that displays slices as diagrams. By confining the slice to the statements of a single procedure, by eliding all primitive statements, and by merging different calls of the same procedure, we eliminate local information that is easily seen in the code without the help of tools. And by labelling edges with the variables responsible for flows between procedure calls, global information about called procedures is represented locally. The resulting diagram gives a compact but rich summary of the role of called procedures in the slice.

Keywords

Reverse engineering, program comprehension, program slicing, program dependence graph, dataflow diagram, interprocedural analysis, modularity, abstraction.

1 Introduction

Understanding a program means at least knowing how variables at critical points in the program acquire their values. It is not surprising, then, that slicing—a mechanical analysis that marks all the statements that might influence the value of a variable at a given point in the program text [Wei84]—is widely viewed as a promising basis for reverse engineering tools.

In practice, though, slicing is not as useful as one might imagine. A slice is itself a program, after all, and, unless dramatically smaller than the original program, is unlikely to be much easier to understand. Unfortunately, big programs tend to have big slices, so some further form of abstraction is essential.

We have developed a tool called Chopshop that slices C programs. In addition to highlighting the code to show traditional slices, it generates diagrams that illustrate how called procedures might affect the chosen variable. The abstractions use to form these diagrams were motivated by our intuitions of how programmers understand code, and their appropriateness has yet to be determined. Our prelimary experiments are encouraging, however; in examining a Unix utility we have discovered features of the code by looking at these diagrams in minutes that we had overlooked in several hours of code reading.

The diagram is a directed graph with labelled arcs. The nodes represent program statements and the arcs dataflow dependences between them. The labels on the arcs indicate which variables are responsible for the dataflow. This diagram is derived from a representation similar to the program dependence graph [FOW87], but its spirit is closer to the dataflow diagram used (albeit with a variety of differing, informal interpretations) in a number of development methods [DeM78]. Indeed, our work may be viewed as an attempt to bridge the gap between representations that can be generated easily from code and architectural descriptions that appeal to developers.

A number of abstractions are applied in the creation of the diagram. Most vital is the modular treatment of procedures. Standard interprocedural slicing [HRB90] pays no respect to procedure call boundaries; a slice on a variable that appears in some procedure p typically includes statements occurring both in procedures that call p and in procedures called by p. This approach was designed for applications that use slicing internally (for pruning regression test suites, e.g., or integrating different versions of a program); for reverse engineering, more structure is called for. Programmers tend to con-

Address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. Phone: (412) 268-5143. Fax: (412) 268-5576. This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330.

0-8186-5647-6/94 $04.00

fine their analysis of a program to one procedure at a time, so it seems desirable that a reverse engineering tool should be capable of this too. A Chopshop user first selects a procedure; any subsequent slices treat this procedure as the entire program. Since the resulting slices mark relevant variables at the entry to this procedure and at the return of called procedures, the programmer can easily follow the slice into a calling or called procedure by slicing again on these variables .

Merely confining the slice to a single procedure does not, of course, solve the problem. Such a slice is rarely comprehensible; some summary of the role of called procedures is needed to explain why each call was included in the slice. The diagram shows what cannot easily be expressed textually: which statements affect which, and how they do so (that is, which variables carry the dataflow).

The abstraction of procedure calls summarizes global information so that it may be presented locally. The remaining abstractions eliminate local information that can easily be gleaned by reading the code (of the procedure being sliced, not the called procedures). Control dependences are eliminated first. They are usually evident from the syntactic nesting of the code, and increase the size of the diagram enormously because of their transitive effects. Second, calls of the same procedure at different sites are folded into a single node. Third, primitive statements (excluding the statement at the slicing point itself) are elided. Suppose, for example, that procedure p writes x and that, following the assignment $y = x$, procedure q reads y. The diagram would omit the assignment, but would connect node p to node q with an arc labelled y/x to indicate that the reading of y by q depends on the writing of x by p. All of these abstractions may be turned off by the user.

Presenting slices in this pictorial form appears to be novel. Most slicers, such as Andersen's Cobol/SRE [NEK94], are purely textual. A slicer being built at Microsoft Research uses an intermediate representation (the value dependence graph) that would allow similar kinds of abstraction [Ern94], and its developers have considered laying unlabelled arrows over text; so far, though, only text highlighting is used.

Many reverse engineering tools generate diagrams, but these tend to be derived from shallow semantic analyses. Refine/Cobol's set-use analysis [M+94], for example, can produce a diagram superficially similar to ours, but derived from a traditional cross-reference listing: an arc is shown connecting two procedures if there is a global variable written by one that is read by the other, whether or not a dataflow path is present. Rigi [M+92] lets the user impose structure on a program by applying various syntactic aggregation mechanisms to call-graphs; Chopshop, in contrast, is designed to expose semantic relationships between components chosen by the user. The idea of folding different calls of the same procedure comes from the star diagram of [BG94], which uses an analysis a bit like slicing to present candidate statements for encapsulation in an operation of an abstract type.

The next section gives an example of Chopshop's output and compares it to a traditional slice. The remaining sections explain the underlying dependence model, the slice computation and the abstraction mechanisms.

2 An Example

One of the procedures from the *more* utility of Berkeley Unix 5.22 is shown, without emendation, in Figure 1*. This procedure, *screen*, controls the basic cycle of *more*: displaying lines and prompting the user for input. From reading the code, we can guess what some of the called procedures do. Presumably *getline* reads a line from the file and *prbuf* displays it; the prompting of the user perhaps happens in *command*. But discovering the details is not easy. How does the line read by *getline* get passed to *prbuf*? Which file is being read? Indeed, is it even the same file for all executions of *getline*?

This procedure is hard to understand for many reasons, but two stand out. First, global variables are used pervasively; *screen* reads or writes more than 20, of which several appear only in procedures it calls. The variable *line*, for example, is a pointer to the string that is passed from *getline* to *prbuf*, but it appears nowhere in the call to *getline*. Second, the called procedures are big, and since their functionality is often as obscure as *screen*'s, attempting to understand *screen* by examining their code raises more questions than it answers. *Getline*, for example, is 109 lines long; *prbuf* is 38 lines and *command* a debilitating 235.

Let's now consider slicing the *screen* procedure. We notice that the variable *dlines* appears only once, about 10 lines from the bottom, so we slice on it to see where its value comes from. The resulting slice is shown underlined. Unfortunately, this does not help much: it is too large to assimilate and it conveys no explanation of why those statements were chosen. Following the slice into called procedures just makes matters worse.

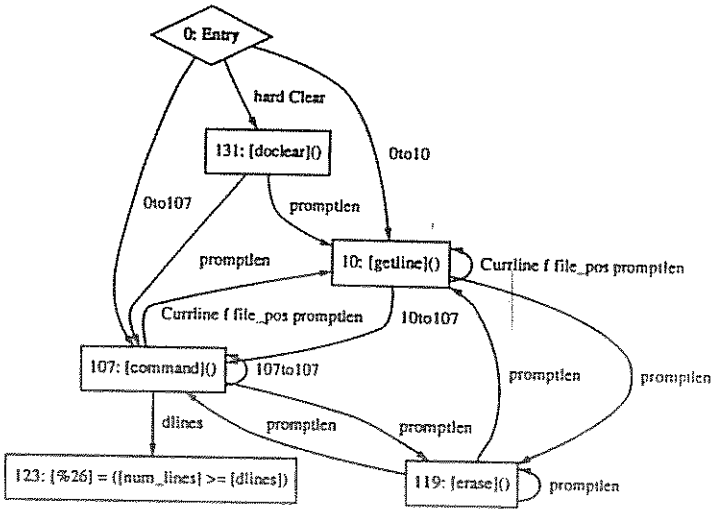
Chopshop generates, in addition, the diagram of Figure 2. Each node represents a procedure call appear-

```
screen (f, num_lines)
register FILE *f;
register int num_lines;
{
   register int c;
   register int nchars;
   int length;              /* length of current line */
   static int prev_len = 1;          /* length of previous line */

   for (;;) {
     while (num_lines > 0 && !Pause) {
        if ((nchars = getline (f, &length)) == EOF)
        {
          if (clreol)
             clreos();
          return;
        }
        if (ssp_opt && length == 0 && prev_len == 0)
          continue;
        prev_len = length;
        if (bad_so || (Senter && *Senter == ' ') && promptlen > 0)
          erase (0);
        /* must clear before drawing line since tabs on some terminals
         * do not erase what they tab over.
         */
        if (clreol)
          cleareol ();
        prbuf (Line, length);
        if (nchars < promptlen)
          erase (nchars);  /* erase () sets promptlen to 0 */
        else promptlen = 0;
        /* is this needed?
         * if (clreol)
         *     cleareol();  /* must clear again in case we wrapped *
         */
        if (nchars < Mcol || !fold_opt)
          prbuf("\n", 1);  /* will turn off UL if necessary */
        if (nchars == STOP)
          break;
        num_lines--;
     }
     if (pstate) {
          tputs(ULexit, 1, putch);
          pstate = 0;
     }
     (void) fflush(stdout);
     if ((c = Getc(f)) == EOF)
     {
        if (clreol)
          clreos ();
        return;
     }

     if (Pause && clreol)
        clreos ();
     Ungetc (c, f);
     (void) setjmp (restore);
     Pause = 0; startup = 0;
     if ((num_lines = command ((char *)NULL, f)) == 0)
        return;
     if (hard && promptlen > 0)
          erase (0);
     if (noscroll && num_lines >= dlines)
     {
        if (clreol)
          home();
        else
          doclear ();
     }
     screen_start.line = Currline;
     screen_start.chrctr = Ftell (f);
   }
}
```

Figure 1: Code of a procedure taken from Unix *more*, with slice on *dlines* (10 lines from the bottom) shown underlined.



0to10 = Currline Mcol dumb eraseln f file_pos fold_opt hard hardtabs pass_ctl promptlen stop_opt
0to107 = Clear Currline Senter Sexit clreol context dlines dum_opt eraseln errors f file_pos file_pos/f hard lastarg lastcmd no_intty noscroll otty promptlen screen_start shellp soglitch
10to107 = Currline f file_pos file_pos/f promptlen
107to107 = Currline dlines errors f file_pos file_pos/f lastarg lastcmd promptlen shellp

Figure 2: Slice diagram corresponding to textual slice of Figure 1

ing in *screen* that might be relevant to the slice criterion. Multiple call sites of the same procedure are folded together; *erase*, for instance, appears 3 times in the textual slice but only once in the diagram. No primitive statements are shown, except for the statement in which *dlines* itself appears. The arcs denote dataflow dependences; although control dependences within called procedures are accounted for, control dependences in *screen* itself are omitted. Each label on an arc indicates a variable defined by the procedure at the arc's source that is used by the procedure at the arc's target. Only variables relevant to the slice are shown, so, for example, there is no mention of *stdout* on the arc joining *doclear* to *command* even though *doclear* defines *stdout* and *command* uses it. When two variables appear in the form y/x as a single label, x is defined by the arc's source, y is used by the arc's target, and there are some elided primitive statements that cause the use of y to depend on the definition of x. A label list too long for its arc is given beneath the diagram, with a name giving the end points of the arc.

Examining the diagram reveals much more information than the textual slice about the interactions between the called procedures. We discover that *dlines* is set in *command* (since the arc from *command* carries *dlines* as a label). The initial value of *dlines* on entry to *screen* is relevant too (because of the label on arc *0to107*). The file pointer *f* (and the variable *file_pos*,

Size of *screen* procedure	74 lines
Sum of sizes of called procedures	1184 lines
Length of slice on *dlines*	21 lines
Size of *screen*'s PDG	138 nodes
Slice diagram, no abstraction	63 nodes
	67 dataflow arcs
	274 control arcs
...with control deps removed	15 nodes, 35 arcs
...and with primitives elided	8 nodes, 19 arcs
...and with calls folded	6 nodes, 15 arcs

Table 1: Effect of abstraction on slice of *screen* procedure

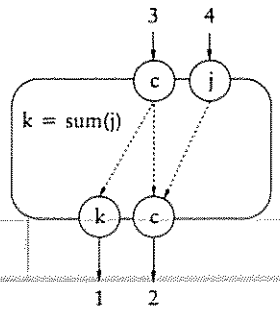


Figure 3: Summary dependences of called procedure. Slicing on the definition of k (1) follows through to the dependence edge incident on the use of c (3), while slicing on the definition of c (2) follows through on both edges (3 and 4).

which turns out to be a copy of it) is relevant, and, surprisingly, modified in both *command* and *getline*. Most puzzling is the influence of *promptlen*, which is responsible for bringing in the procedures *erase* and *doclear*.

The variable *dlines* appears to hold the number of lines to be printed on the screen in the next cycle. To understand how it might be affected by *promptlen*, the length of the last prompt displayed by *command*, we applied Chopshop to the bodies of *getline* and *command*. We discovered that, as a fresh line from the file is written over the prompt, when a tab is encountered, and the prompt has not been entirely overwritten, the characters to the tab mark are erased. This seems to affect the length of the line written, and thus (since long lines are folded) the value of *dlines*. It is tempting to regard the dependence as spurious, but this cannot be justified without a detailed verification of *getline*. Frequently, obscure dependences that the programmer believes to be absent are exactly the ones that signal the presence of real bugs.

To illustrate the effects of Chopshop's abstractions, Table 1 shows how each in turn reduces the size of the diagram. The remarkable shrinkage caused by eliminating control dependences might not be typical. It arises here because of conditional exits from the loop that bring in many of the global variables that have no dataflow links to *dlines*.

3 The Dependence Model

Chopshop's underlying model is similar to the program dependence graph (PDG) [FOW87] but accommodates interprocedural dependences more naturally. The standard way to extend a graphical program representation (whether of dependences, control flow, etc.) to account for called procedures is to form a single supergraph for the entire program by joining the graphs of the individual procedures at their call sites. Extending the PDG in this way [HRB90] introduces a host of complications, not only in the construction process but also in the graph itself. Each call site becomes a complex web of linkage nodes and mock assignments (to model the passing of parameters) that is not conducive to a modular analysis.

Our approach is simpler. A procedure call is modelled as a single node, as if it were a primitive statement. The effects of its internal dependences at the call site are represented by a def-use relation between the result and argument variables of the call. For example, the call

$$k = sum(j)$$

to the procedure

```
int sum (int i) {
    int t;
    t = c;
    c = c + i;
    return (t) }
```

would be summarized by the def-use relation

$$\{(c, j), (c, c), (k, c)\}$$

indicating that the definition of the global c depends on the use of the argument j and the use of c (that is, its old value), and that the definition of k depends only on the use of c. The local variable t is invisible to callers, so it does not appear.

It may be helpful to think of this relation as a set of edges inside the procedure call node. A slice on c after the call, for example, is found by following the two internal edges from c at the bottom to j and c at the top; the edges incident on the uses of these two variables are then followed to their definitions at preceding nodes, and so on. Note that a slice on k would traverse only a single internal edge to the use of c, and from there to its definitions, so that the edge bringing the definition of j to the call would be followed in the first case but not the second. This scenario is illustrated in Figure 3.

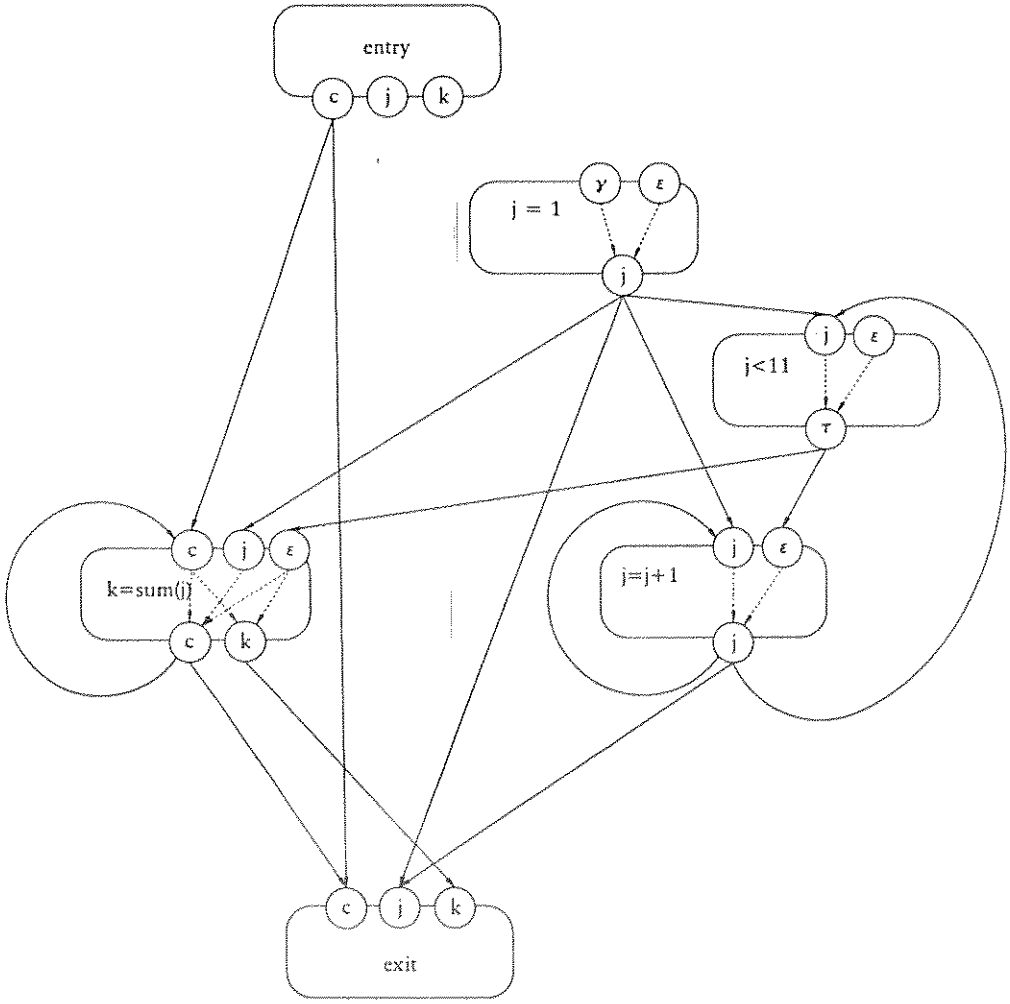


Figure 4: A dependence graph.
The *du* relation is shown by the dotted edges;
the solid edges belong to *ud* (when they link program variables),
and *cd* (when they link an execution ε to a temporary τ).

For uniformity, all nodes are treated in the same way, whether procedure calls or primitive statements. Since the caller of a procedure cannot observe any difference between control and dataflow dependences inside the call, there is no need to distinguish them in the internal edges. Between statements, however, the two kinds of dependence must be separated.

All dependences relate the use of a variable at one point to a definition at another. We call these points "sites" as they do not always correspond to program statements. Nesting of expressions causes a single source statement to be translated into several nodes connected by temporary variables.

Our model thus consists of three binary relations, *ud*, *du* and *cd*, over variable/site pairs (or "instances"). The dataflow dependence relation *ud* relates a use of a variable at one site to a definition at another; it contains the pair $((x, i),(x, j))$ if there is a use of x at site i that has a reaching definition at site j.

The control dependence relation *cd* relates the execution of one site to the evaluation of an expression at another; it contains the pair $((\varepsilon, i),(\tau, j))$ when execution of site i, modelled by the dummy variable ε, depends on the expression at site j, where τ is a temporary variable holding the result of the expression.

The summary relation *du* (which we referred to above as the "internal edges") relates the definition of a variable at a site to a use at the same site; it contains $((x, i),(y, i))$ when the effect of site i is to make x afterwards depends on y before. For every variable x modified at site i the pair $((x, i),(\varepsilon, i))$ is included too, to show that the definition of x depends on the execution of the site. If a variable x is modified but is not dependent on any use, the pair $((x, i),(y, i))$ is inserted to show a depen-

dence on some constant y.

Formally, then, we have:

$$Var = ProgramVariables \cup \{y, \varepsilon, \tau\}$$
$$Instance = Var \times Site$$
$$du, ud, cd: Instance \leftrightarrow Instance$$

Figure 4 illustrates these relations for this procedure, which calls *sum* to add the numbers from 1 to 10 to the global c:

```
total () {
  j = 1;
  while (j < 11) {
    k = sum (j);
    j = j + 1; }}
```

There are two special sites for every procedure: an entry and an exit. All variables are defined at the entry and used at the exit. The instances are drawn, as in Figure 3, as ports of the nodes, although this time the special variables are shown too. To see how control dependences work, follow the path back from j at the exit through $j = j + 1$; note how the definition of j depends on whether the increment is executed (ε), which depends on the outcome of the loop test (τ), which in turn depends on the value of j before the test. (Incidentally, the graph is not complete. One arc is missing: it proved to be beyond the authors' drawing abilities and is left as an exercise for the reader.)

Constructing the dependence relations is straightforward; the only subtleties arise in obtaining *du* from the bodies of called procedures, and handling recursion. The details, along with a more substantial justification of the model, may be found in our technical report [JR94a] and a paper to appear soon [JR94b].

4 Constructing Slice Diagrams

Rather than giving explicit worklist algorithms for constructing slices, we shall define slices as algebraic expressions in terms of the dependence relations. We find these much easier to understand and manipulate. As specifications, the expressions allow a variety of implementations, but the current version of Chopshop actually implements them directly. Altering the slicing algorithm or the abstraction mechanisms thus involves changes to only a few lines of code.

We use six relational operators. Union, composition, transitive closure and projection of a set are standard:

$$p \cup q = \{(a, b) \mid (a, b) \in p \vee (a, b) \in q\}$$
$$p \circ q = \{(a, b) \mid \exists z \in T.\ (a, z) \in p \wedge (z, b) \in q\}$$
$$p^* = I \cup p \cup (p \circ p) \cup (p \circ p \circ p) \cup \ldots$$
$$p[S] = \{b \mid \exists a \in S.\ (a, b) \in p\}$$

The domain restriction of a relation p to a set S is the relation containing all pairs in p whose first element is in S:

$$S \lhd p = \{(a, b) \in p \mid a \in S\}$$

Similarly, the range restriction of p to S contains pairs whose second elements are in S:

$$p \rhd S = \{(a, b) \in p \mid b \in S\}$$

The slice criterion is a set of variables V and a site i at which they are used, that is, a set of instances

$$C = V \times \{i\}$$

The reaching definitions that affect these uses directly are given by the projection of C under the *ud* relation. The uses affecting these definitions are found by projecting this set under the relation *du*. This process is repeated to find all uses that affect C. The set of relevant uses is thus the projection of C under the transitive closure of $ud \circ du$:

$$relUses = (ud \circ du)^* [C]$$

The basic diagram is now obtained by restricting the *ud* relation to these uses:

$$ud' = relUses \lhd ud$$

Now we apply two abstractions. Nodes corresponding to different calls of the same procedure are merged; this is easy and not worth formalizing. To elide primitive statements, we start by collecting together the instances

$$drop: \mathbb{P}\ Instance$$

associated with the nodes to be dropped, namely all the non-calls except the site of the slice criterion and the entry and exit, and the instances associated with nodes to be kept (all the rest):

$$keep: \mathbb{P}\ Instance$$

The elision happens in two stages. First we add transitive edges that replace dependences brought about by paths passing through dropped nodes:

$$ud'' = ud' \cup (ud' \circ (drop \lhd du \circ ud')^*)$$

Second, we restrict the diagram so that only nodes with instances in *keep* remain, giving

$$ud''' = keep \lhd ud'' \rhd keep$$

This relation, ud''' is the final, abstracted slice diagram. A pair $((x, i),(y, j))$ in this relation connects a use of a variable x at site i to a definition of a variable y at site j.

Only one edge is drawn between i and j, irrespective of how many pairs connect them. If x and y are the same variable, v say, the edge is labelled v; if x and y differ, the edge is labelled x/y to show that the use of x is indirectly due to the definition of y.

We have illustrated only a limited form of slicing. Most variants of slicing (such as slicing forwards instead of backwards, or slicing on a definition instead of a use) are easily expressed in our model [JR94a,b]. Chopshop implements a variant we call (predictably) "chopping", whose criterion is two sets of instances, *source* and *sink*; the chop shows how the definitions in *source* affect the uses in *sink*.

5 The Chopshop Tool

Chopshop is implemented in Standard ML. It runs as a subprocess under *emacs 19* (a new version that supports colour highlighting). The diagram is written to a file as an adjacency list, which is then converted to postscript by AT&T's *dot* program, and finally displayed by the *ghostview* previewer.

Selecting a procedure of interest initiates the analysis of the code. The def-use abstractions of procedures are cached, so they need be calculated only once. The tool also maintains a variety of closure relations, so that the chopping operations can be performed in linear time.

The performance of the current version is acceptable only for small programs. It takes about 15 minutes to analyze the entire *more* program (approximately 2000 lines), and about ten seconds to perform each chop. We are working on a number of improvements. By hardwiring the analysis (in contrast to direct implementation of the relational expressions), we expect to increase its speed dramatically. The next version will store the def-use abstractions of procedures in a library whose entries can be read and written by the user; this will allow Chopshop to handle calls to procedures with no code, either because they are low-level, or because they have yet to be written. It will also include an alias analysis to account for dependences caused by indirect accesses.

References

[BG94] R.W. Bowdidge and W.G. Griswold. Automated support for encapsulating abstract data types. *Proc. ACM Sigsoft '94 Symp. on Foundations of Software Engineering*. New Orleans, La., December 1994.

[DeM78] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.

[Ern94] Michael D. Ernst. *Practical fine-grained static slicing of optimized code*. Technical report MSR-TR-94-14, Microsoft Research, Redmond, Wa., July 1994.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3), July 1987, pp. 319–349.

[HRB90] Susan Horwitz, Thomas Reps and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1), January 1990, pp. 26–60.

[JR94a] Daniel Jackson and Eugene J. Rollins. *Chopping: a generalization of slicing*. Technical report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., July 1994.

[JR94b] Daniel Jackson and Eugene J. Rollins. A new abstraction of the program dependence graph for reverse engineering. *Proc. ACM Sigsoft '94 Symp. on Foundations of Software Engineering*. New Orleans, La., December 1994.

[M+92] H.A. Muller, S.R. Tilley, M.A. Orgun, B.D. Corrie, N.H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. *Proc. 5th ACM SIGSOFT Symposium on Software Development Environments*, 1992.

[M+94] Lawrence Markosian, Philip Newcomb, Russell Brand, Scott Burson and Ted Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5), May 1994, pp. 58–71.

[NEK94] Jim Q. Ning, Andre Engberts and Wojtek Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5), May 1994, pp. 50–57.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4), July 1984, pp. 352–357.

Understanding Code Containing Preprocessor Constructs

Panos E. Livadas and David T. Small
Computer and Information Sciences Department
University of Florida
Gainesville, FL 32611

Abstract

Understanding, debugging, and maintaining software is a costly and difficult task. The difficulties are exacerbated in programs written to take advantage of preprocessing facilities. This paper examines problems associated with source code containing preprocessor constructs—i.e., included files, conditional compilation, and macros. We define the useful mappings from tokens in the preprocessor's output to the source file(s), and propose that by capturing these correspondences an internal program representation can be built which will allow for the use of maintenance techniques including program slicing, ripple analysis, and dicing. The method presented is generic; to illustrate that the technique is feasible, we discuss ANSI C preprocessor constructs—in particular, macro substitution—and explain the modus developed to handle them in GHINSU *—an integrated maintenance environment for ANSI C programs.*

1 Introduction

Software maintenance is an expensive, demanding, and ongoing process. Lientz and Swanson [1] report that large organizations typically devote 50% of their total programming effort to maintenance. One US Airforce system is estimated to have cost $30 per instruction to develop and $4,000 per instruction to maintain over its lifetime.[2] These figures are perhaps exceptional; none the less, maintenance costs seem to be between two and four times higher than development costs for large, embedded systems. Our industrial affiliates in private communications estimate 60% of their maintainer's time is spent *looking* at code.

Recent research has focused on techniques to assist in program maintenace; among them are slicing, dicing, and ripple analysis, which will be discussed later in this paper.

Programs that take advantage of preprocessing facilities incur an additional maintenance burden. Typically, the maintainer interacts with the source code; however, the compiled program is based on the preprocessed code. Thus preprocessor commands can be a two-edged sword: they can increase code readability and programmer productivity while simultaneously obfuscating the program's mechanics and, consequently, maintainability.

We therefore aim to eliminate the penalties imposed by the use of preprocessor constructs, and thus reduce maintenance costs. This can be accomplished with an integrated environment that provides an assortment of tools "aware" of the correspondence between the preprocessed and source code.

1.1 GHINSU- An Integrated Software Maintenance Environment

GHINSU is an integrated software maintenance environment which facilitates code understanding, testing, debugging and program reengineering. It supports both static and dynamic slicing, ripple analysis and other program analysis functions on ANSI compliant C source code. C programs typically take advantage of the language's preprocessing mechanism; thus, a preprocessor is required for GHINSU to handle the gamut of valid ANSI C code.

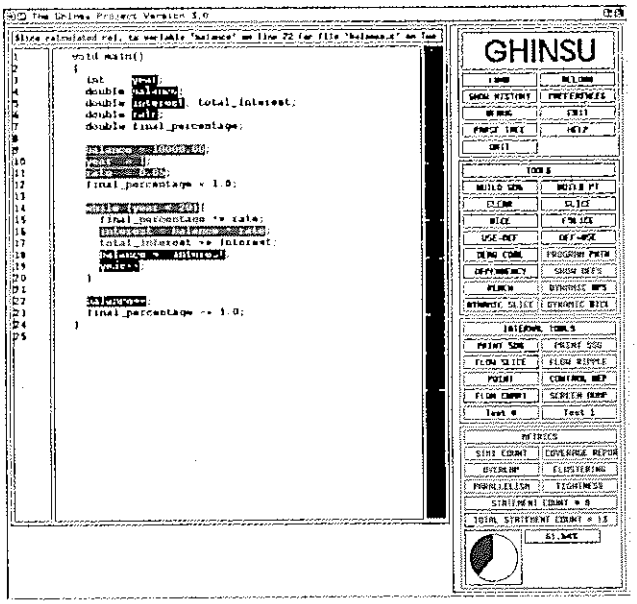


Figure 1: The GHINSU Environment

The highlighted code represents a program slice calculated relative to the variable balance on line 22.

One of GHINSU's key design goals is not only to aid program understanding, but do so in a user friendly fashion. GHINSU provides several easily apprehended graphical user interfaces to the information contained in an internal program representation. These include neatly formatted graphs depicting dependencies, flow,

0-8186-5647-6/94 $04.00 © 1994 IEEE

and control, and most importantly, a view of the original source code with "interesting" portions highlighted. An example is shown in figure 1.

2 Slicing, Dicing, and Ripple Analysis

Several powerful software maintenance techniques have been developed in the last decade; these include slicing, dicing, and ripple analysis. They use a common graph-based internal program representation, which will be discussed in the next section.

Program slicing is a useful tool for the maintenance programmer. let P be a program, p be a point in P, and let v be a variable in P that is either defined or used at p. A *static slice* of P relative to the *slicing criterion* (p, v) is defined as the set of all statements and predicates of P that *might* affect the value of the variable v at point p. This definition is less general than the one given in [3]; but, it is sufficient [4]. An example of a program slice was shown in figure 1. Program slices can be used in a variety of ways to aid in several software engineering activities such as the ones that are briefly discussed below.

Empirical studies have show that programmers use slices when debugging. Assume that during testing we discover that the value of a certain variable, v, is incorrectly computed at some statement, n, of our program. By obtaining a slice of v at n, we may extract a significantly smaller piece of code to examine, and thus allowing us to more readily locate the bug. In addition, slicing can aide in program understanding by decomposing a program into meaningful smaller components. Moreover, Horwitz [5] has used the concepts of slicing in integrating program variants and Badger [6] has demonstrated how slicing can be used for automatic parallelization. Furthermore, slicing assists with code resuability. If a large program computes the value of a variable, v, and the code associated with that computation is needed in another program, one may slice on the v and use the extracted program in the application at hand. Also, a number of metrics based on program slicing have been proposed [3] which include coverage, component overlap, functional clustering, parallelism, and tightness.

Dicing, a tool based on slicing, can be used to aid in debugging by *automatically* locating certain errors [7]. Dicing is a powerful heuristic which can further reduce the amount of code on which the programmer must concentrate. If, in addition to generating a slice of an incorrectly computed variable at a particular statement, and there exists another variable that is computed correctly, then the dicing heuristic may be employed; the bug is likely to be associated with the statements in the slice on the incorrectly computed variable minus the statements associated with the slice on the correctly computed variable. Dicing can therefore be used iteratively to locate a program bug [7].

Another analysis tool called the *ripple analyzer* can be thought of as a "forward" slicer. It finds all statements that are dependent upon a given statement. This tool can be used to visualize the statements that will be affected by a change made at any given statement. Additionally, ripple analysis may help the maintainer estimate the time and resources needed to effect a program modification based on the size and/or configuration of the resultant ripple.

3 The Internal Program Representation

In [8, 9, 10] we describe an internal program representation which is a parse tree based *system dependence graph* (SDG)—an extension of the statement based SDG proposed in [4]. Informally, the SDG is a labeled, directed, multigraph where each vertex represents a program construct—e.g., declarations, assignment statements, control predicates, and tokens. Edges are labeled to distinguish which of the several kinds of dependencies between connected vertices is represented. Since slicing, dicing, and ripple analysis can be reduced to a graph reachability problem, the SDG is a suitable internal program representation for an integrated maintenance environment based on those tools.

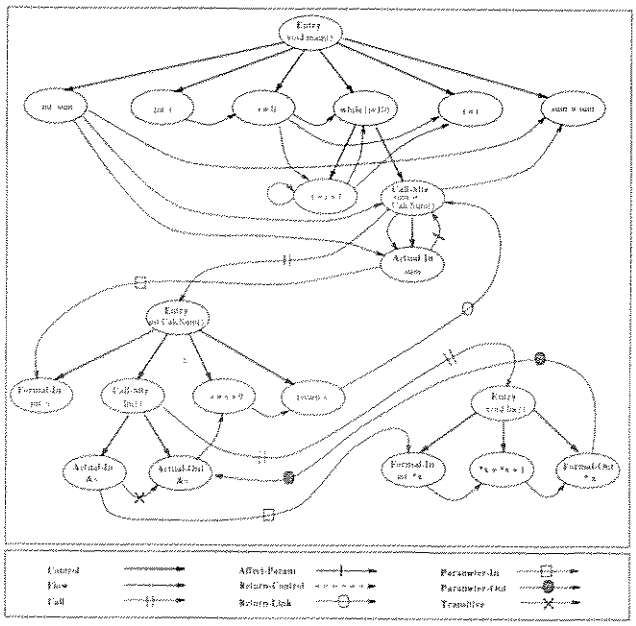


Figure 2: A System Dependence Graph

A program slice is most easily understood when the maintainer is presented with a view of the source code where those tokens which correspond to the vertices identified by the slicing algorithm have been highlighted. Therefore, a method is needed to capture the relationship between vertices in the SDG and tokens in the source code. In programs that do not use preprocessor constructs, this is a simple one-to-one correspondence. The situation is more complex for programs that utilize a preprocessor, since the SDG is no longer derived directly from the source code, but from the preprocessed code.

4 Typical Preprocessor Features and Associated Problems

Preprocessors usually feature mechanisms to include files, conditionally include a block of code, and perform macro substitutions. *File inclusion* is the ability to insert files into the current source file, and return the whole as a single file. Conditional code inclusion—more commonly referred to as *conditional compilation*—only adds a block of code to the preprocessed file if certain stipulations are met. *Macros* allow the programmer to associate a name with an arbitrary replacement value; the replacement value is substituted in place of the name in the output file, where ever the name is encountered.

The ability to include files is a powerful feature. It allows the programmer to break code down into smaller, more manageable units. Typically, functionally or logically related code is grouped into separate files. But, it can be difficult for the maintainer to determine in which of the included files a particular construct was defined. This problem is only exacerbated by nested include files.

Conditional compilation permits a single source file to generate different programs. This is useful when a program is targeted to multiple platforms or, for example, to include debugging code only in test versions. Complex conditional compilation blocks can be difficult to understand. Because the conditional expression only directs code inclusion, and does not itself appear in the preprocessed file, errors in the conditional expressions may be hard to detect.

Macros are tremendously flexible constructs which are often used as shorthand for the idioms appearing in a program. Unfortunately, errors which arise as the consequence of a macro substitution can be difficult to trace. Debugging can be complicated for several reasons:

- since macros are generally simple, there is a tendency to trust in their correctness, when in fact there may be an error in the macro definition,
- the macro was designed for one context and has been misapplied to another,
- the macro may not be readily identifiable as such, and
- the substitution value is not the one expected, because the macro has been redefined.

We have identified five distinct mappings from tokens in the preprocessed file to tokens in the original code. α-maps are the usual case, and occur where no macro expansions are involved; they are simple one-to-one mappings from tokens in the preprocessed file to tokens the raw source code. All macros with a replacement value have at least three mappings: each token in the macro expansion has a β-mapping to the macro occurrence, a γ-mapping to the macro definition statement, and δ-mapping to a token in the definition's replacement token sequence.[1] In addition, parameterized macros have a ϵ-map for each occurrence of a token derived from a supplied argument to the appropriate token in the argument list. Figure 3 illustrates the different mappings between a fragment of C code and its preprocessed equivalent.

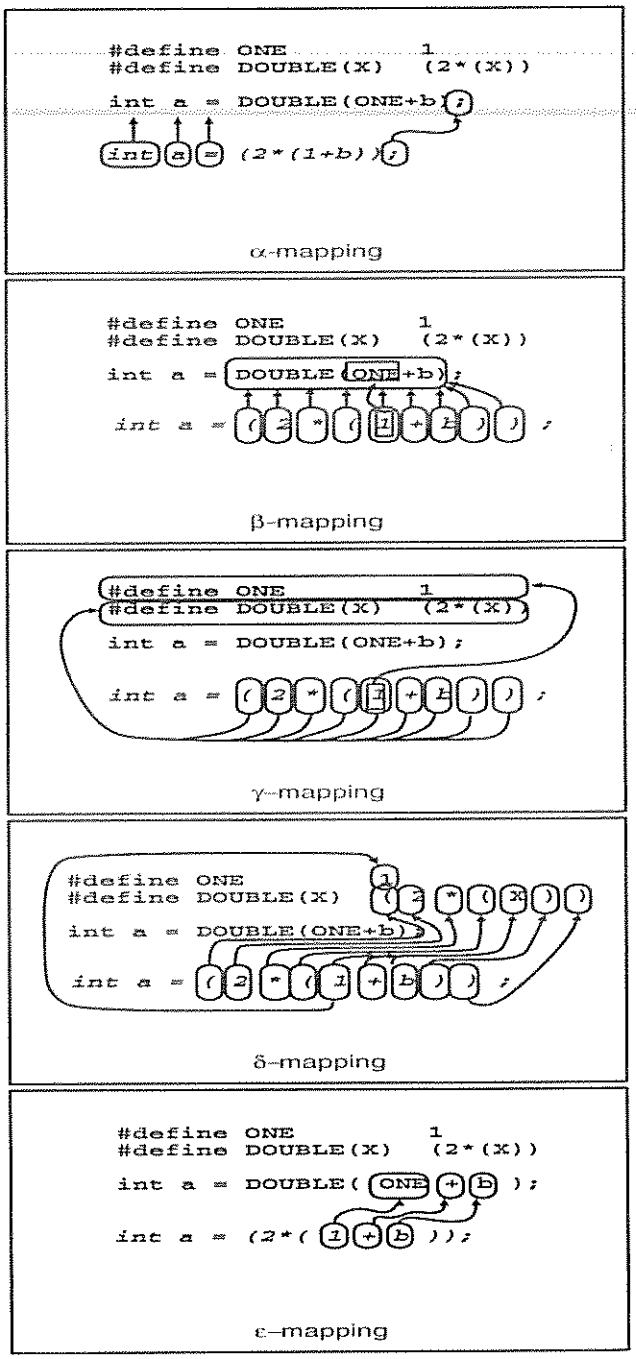


Figure 3: Mappings from Preprocessor Output to Source Code

[1] We mention δ-maps for completeness, but do not utilize them in our research.

It is important to observe that β, γ, δ, and ϵ maps can nest and span multiple files. Since we are mapping from the preprocessor's output to a source file, note that macros with null expansion values have no mappings. Likewise, parameterized macros that do not use a parameter will not have a mapping for the supplied, but unused argument.

5 Macro Preprocessing

The mappings identified in the previous section are applicable to source code written in any language that utilizes preprocessor constructs. We already had a working tool, GHINSU, for programs written in ANSI C; thus it was natural to use it as a testbed on which to develop routines to capture the token correspondences. Thus we restrict the remainder of our discussion to the ANSI C preprocessor.

ANSI C is endowed with a rich preprocessing facility. Tracking α-mappings for included files and conditionally compiled code is a trivial task[2] and will not be discussed here. Macro substitutions can be quite complex and will be examined shortly. Afterwards, Ghinsu and the approach taken to capture the various mappings will be discussed.

5.1 Ambiguities Inherent in C Macros

C macros have few rules governing their behavior—a complete description can be found in [11]—and can serve almost any purpose the programmer desires. The only constant feature is that a macro has name, which must be a valid C identifier. Optionally, it can have a replacement token sequence, which is expanded and substituted in place of the macro name during preprocessing. A macro may also parameter list—which may be empty—consisting of comma separated identifiers. If a parameter identifier appears in the replacement token sequence, then the corresponding argument supplied when the macro is used, will be expanded and substituted for that parameter.

A casual glance at the use of a macro might lead one to conclude that it is an instance of a variable, a constant, or a function call—appearances can be extremely deceptive. Often macros do serve as pseudo-constants:

```
#define PI      3.14
#define GPP     "Ghinsu preprocessor"
```

Frequently they act as mnemonics for a sequence of commonly used expressions or statements:

```
#define COST      (unit_price * number_units)
#define NEWLINE   (column = 0, line++)
#define BEGIN     state = 1 + 2 *
#define ABS(X)    (((X)<0)?(-X):(X))
#define INITIAL   0

subtotal += COST;
location = NEWLINE;
BEGIN( INITIAL );
ABS( a++ );
```

In the examples, `COST` and `NEWLINE` are behaving like inline functions, even though they appear to be used as variables. Even more deceptive is the parameterless macro `BEGIN` followed by what appears to be a parameter list—`BEGIN` is acting neither as a function nor a variable. The use of `ABS()` may look like a function call, but it has a side effect a true function call would not have: `a++` is evaluated twice. Macros can even be used to change the appearance of the C language:

```
#define BEGIN   {
#define END     }
```

These are just a few examples to illustrate the rich range of macro applications in C.

5.2 GPP- The GHINSU Preprocessor

The capability to highlight relevant code implies that the SDG must associate its vertices with physical locations in the source code. Earlier versions of GHINSU did not do any preprocessing; thus the task of mapping SDG vertices to tokens was trivial: each SDG vertex mapped directly to a single location in a single file. However, without a preprocessor, GHINSU was unable to accept real world programs—i.e., those with `#include`-ed files, macros, and conditional compilation directives. Support for file inclusion and conditional compilation could be implemented easily; the difficulty lies in macro substitutions.

Were we to generalize macros as pseudo-identifiers and pseudo-functions—as in [12]— we would be unable to account for many of their uses.[3] Having eliminated that approach, we are faced with the laborious process of relating each token produced by the preprocessor to appropriate points in the source code. As mentioned previously, an ordinary C preprocessor would lose the α, β, γ, and ϵ-mapping information; this fact is the motivation behind the development of our own preprocessor, GPP, the GHINSU preprocessor.

5.2.1 GPP's Output Conventions

The information GPP captures is best explained with an example. Several references will be made to the following code fragment which we will call `main.c`:

```
[1] #define ONE 1
[2] #define CPLUSONE ( c + ONE )
[3] #define DPLUS(X) ( d + X )
[4]
[5]   f = DPLUS( DPLUS ( CPLUSONE ) );
```

It would be expanded by a normal preprocessor to:

```
f = ( d + ( d + ( c + 1 ) ) );
```

GPP's output, shown in figure 4, retains the α, β, γ, and ϵ-mappings, and is consequently verbose. All of this information is required to make a slice highlight the germane macro `#define` statements. Figure

[2] Many preprocessors insert `#line` directives in their output, which are used by the C compiler to embed debugging data within an executable program; we have adopted an analogous approach.

[3] Additionally, support for the `#` (stringification) and `##` (concatenation) operators would probably be impossible.

5 shows a program slice computed using methods described later in this paper to retain information typically lost during preprocessing.

The Results of Running GPP on main.c

```
#L 3
   f =
#P DPLUS "main.c" 2 4 7 4 12 4 33
#0
( d +
#P DPLUS "main.c" 2 4 14 4 19 4 31
#0
( d +
#M CPLUSONE "main.c" 1 4 22
#0
( c +
#M ONE "main.c" 0 -1 -1
#0
1
#W
#0
 )
#W
#0
 )
#W
#0
 )
#W
;
```

macro name
defined in file
defined on line
used on line
used at column
opening paren on line
opening paren at column
closing paren on line
closing paren at column

GPP Formatting Codes

- #F change to file
- #L skip to line
- #M macro - no parameters
- #P macro, parameterized
- #0 following token(s) are not visible in the source file
- #T following token is visible in the source file
- #W end of macro expansion
- #S special built-in macro

Figure 4: An annotated example of GPP's output

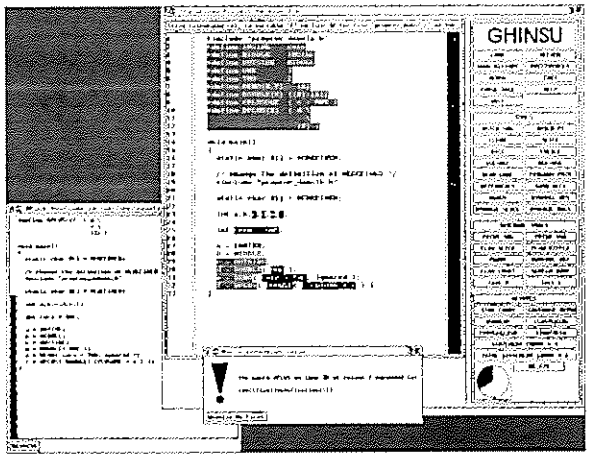


Figure 5: GHINSU Slice on Program Containing Preprocessor Constructs

The large window multi-paneled window shows the result of a slice on the variable f on line 32. Clicking on the use of macro DPLUS on that same line pops up two more windows. The medium sized window at the lower left is a text browser which shows the file in which the macro was defined scrolled such that the #define line is automatically at the top. This provides the maintainer not only the definition location but the context as well. At the lower center a small window identifies the macro selected and displays its expansion value. This allows the programmer to verify that the macro was expanded to the expected value.

5.3 Algorithms and Data Structures

5.3.1 Macro Definition

A dictionary associates the macro's name with the information required to generate its replacement: a pRTT (prototype Replacement Token Table), the parameter count, and the file name and line number where the macro was #define-ed. The pRTT contains, for each token in the replacement sequence, the literal token, the parameter number to which it corresponds (if any), and a token type code. The later is derived from the token type returned by the lexer, and serves to identify those tokens which are potentially significant in the expansion process. These relationships are illustrated in figure 6, and an example based on main.c is given in figure 7.

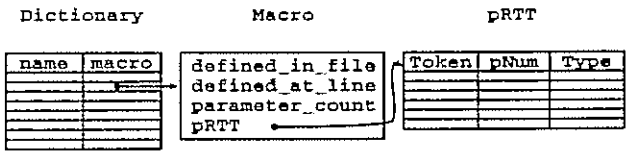


Figure 6: GPP's Macro Definition Representation

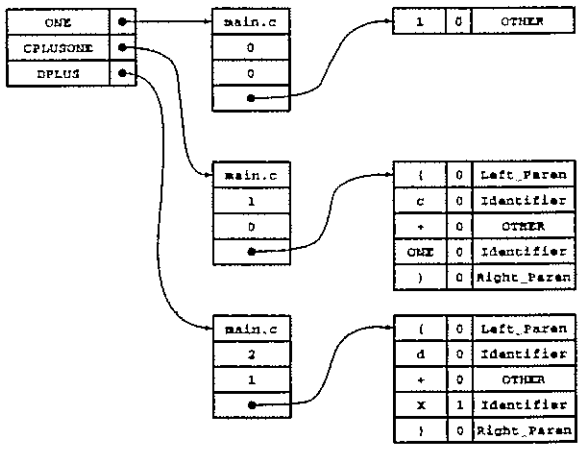


Figure 7: Representation of Macros Defined in main.c

The definition process is straight forward: parameters, if any, are extracted from the parameter list, and the remainder of the logical line—excepting comments, which are discarded—is converted to tokens which are added to the pRTT. The process is outlined in the following pseudo-code.[4]

[4]For clarity's sake, error checking and recovery are not accounted for in the algorithms described in this paper; both are an integral part of the GPP implementation.

```
define( macro_name )
{
  macro = allocate_macro;
  macro->defined_in_file = file_name;
  macro->defined_on_line = line_number;

  if ( LEFT_PAREN == ( token_type = get_token() ) )
    {
      while ( ! RIGHT_PAREN ==
                        ( token_type = get_token() ) )
        {
          if ( token_type == IDENTIFIER )
            add_to_parameter_list( token );
        }
    }

  macro->parameter_count = parameter_list_size();

  while ( ! EOL == ( token_type = get_token() ) )
    {
      add_to_pRTT( token,
                   matches_parameter_number( token ),
                   token_type );
    }

  add_to_dictionary( macro_name, macro );
}
```

5.3.2 Macro Expansion and Substitution

Whenever a macro is used, its expansion value must be calculated and substituted for the macro and its parameters, if any. Macro expansion and substitution is a more complex than the definition process. As shown in the code fragment `main.c`, macros can have other macros embedded both in their replacement token sequence and in a supplied argument list. GPP tracks every macro encountered during an expansion with a node structure.

The stages in the expansion of `DPLUS( DPLUS( CPLUSONE ) )` would proceed as shown in figure 8. After the expansion is completed, the resultant structure is traversed to extract the macro's substitution value. Pseudo-code detailing the steps involved in expansion and substitution are given below.

The routine `substitute()` is called when a macro is encountered in the source file. It creates a macro node, builds an ATT (Argument Token Table) from the supplied arguments, if any, and constructs a RTT (Replacement Token Table) from the prototype contained in the macro definition structure. The tables are linked to the macro node. Then the macro is *reduced*, that is, all of the tokens in the RTT with replacement values are recursively expanded. Finally, the substitution structure is traversed and the substitution value is extracted.

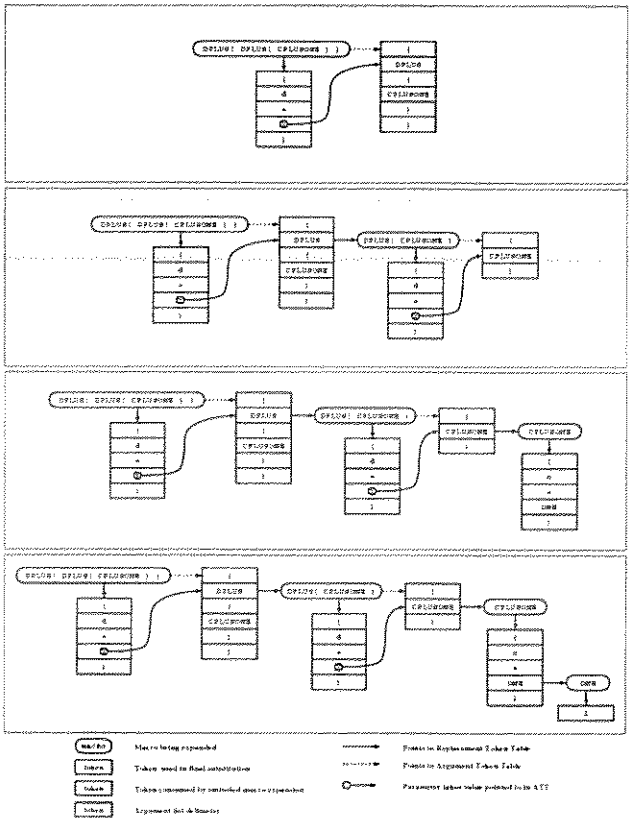


Figure 8: Stages in the expansion of `DPLUS( DPLUS( CPLUSONE ) )`

```
substitute( macro_name )
{
  node = allocate_macro_node();

  if ( HAS_PARAMETERS( macro_name ) )
    {
      node->ATT = allocate_ATT();
      token =
        opening_paren_of_argument_for( macro_name,
                                       source_code );
      do
        {
          if (DELIMITING_COMMA_OR_PAREN(token))
            add_to_ATT_marked_as_delimiter(token);
          else
            add_to_ATT( token );
        }
      while ( !CLOSING_PAREN( token++ ) );
    }
  else
    {
      node->ATT = nil;
    }

  node->RTT = build_RTT( token );
  reduce_macro( node );
  emit_table( node );
}
```

A macro node is processed by `reduce_macro()`. For every unused token in the RTT, first check if the token

has been expanded. If not, expand it. Then if the token has a replacement value—that is, it is a macro or a parameter—reduce it.

```
reduce_macro( node )
{
  for ( each !USED( token ) in node->RTT )
    {
      if ( !EXPANDED( token ) )
        expand( token );

      if ( HAS_REPLACEMENT_VALUE( token ) )
        {
          if ( PARAMETER( token ) )
            reduce_parameter( token );
          else
            reduce_macro( token.node );
        }
    }
}
```

Parameter tokens are slightly more complicated to process, as is seen in the code for `reduce_parameter()`. If the parameter is to be replaced with a stringified representation of the argument, then build the string. Otherwise, for all unused tokens in the corresponding argument contained in the ATT, expand unexpanded tokens, and reduce any tokens that are macros.

```
reduce_parameter( parameter )
{
  if ( CONVERT_TO_STRING( parameter ) )
    {
      for ( each token in the supplied argument )
        {
          if ( EMBEDED_ARGUMENT( token ) )
            append_to_buffer( get_embeded_arg( token ) );
          else
            append_to_buffer( token );
        }
      buffer_to_c_string();
      replace_parameter_token_with_c_string();
    }
  else
    {
      for ( each !USED( token ) in supplied argument )
        {
          if ( !EXPANDED( token ) )
            expand( token );

          if ( HAS_REPLACEMENT_VALUE( token ) )
            reduce_macro( token.node );
        }
      set_replacement_value_to( argument );
    }
}
```

The routine `expand()` is straight forward. If the token is a parameter, then reduce it. If the token is a macro, then build a macro node, and associated tables, for it. Finally, mark the token as expanded.

```
expand( token )
{
  if ( PARAMETER( token ) )
    {
      reduce_parameter( token );
    }
  else if ( DEFINED( token ) )
    {
      node = allocate_macro_node();
      node->ATT = build_ATT( token, token.host_table );
      node->RTT = build_RTT( token );
      set_replacement_value_to( node );
    }
  mark_as_expanded( token );
}
```

An ATT can be built one of three ways: from the argument supplied to the outer-most macro (from tokens extracted directly from the source file), from an argument embeded in the replacement token sequence, or from a combination tokens in the RTT and ATT. The later case occurs when a parameter is used as part of an embeded macro's argument. For example:

```
#define INNER(X)  (X)
#define OUTER(Y)  INNER( 3 + Y )

a = OUTER(2);
```

The ATT constructed for `INNER` would contain `3` and `+`, both extracted from `OUTER`'s RTT, as well as `2`, extracted from `OUTER`'s ATT. The function `build_ATT()` handles the later two cases; the first case is performed by `substitute()`.

```
build_ATT( macro_name, host_table )
{
  if ( HAS_PARAMETERS( macro_name ) )
    {
      ATT = allocate_ATT();
      token =
        opening_paren_of_argument_for( macro_name,
                                       host_table );
      do
        {
          if (DELIMITING_COMMA_OR_PAREN(token))
            copy_to_ATT_marked_as_delimiter( token );
          else
            copy_to_ATT( token );
          mark_as_used( token );
        }
      while ( !CLOSING_PAREN( token++ ) );
    }
  else
    {
      ATT = nil;
    }
}
```

5.4 Storing Maps on GHINSU's SDG

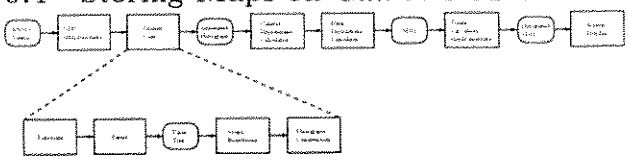


Figure 9: GHINSU's Architecture

As show in the conceptual overview of GHINSU's architecture, figure 9, the GHINSU core tokenizer reads the preprocessor's output. When the start of a macro

is encountered, the tokenizer begins to build a linked hierarchical representation called the *macro highlighting structure*, which retains the mapping information needed by the highlighting routines. It consists of identification, instance, and parenthesis nodes. *Macro identification nodes* store the name and definition's location—file name and starting line number. There need only be one identification node for each macro used, regardless of how many times, as the information it contains is constant. However, there is a unique *macro instance node* for each occurrence of a macro. It stores the position where the macro is used, a link to an identification node, and, if the macro use is embedded inside another macro, a pointer to that other macro's instance node. If the macro is parameterized, then a link is made to a *parenthesis location node* which stores the position of the argument list's delimiting parenthesis. An example, again drawn from `main.c`, illustrating the structure built for `DPLUS( DPLUS( CPLUSONE ) )` is given in figure 10. As each token resulting from a macro expansion is processed, it is added to the SDG with a link to the appropriate point in the macro highlighting structure. This is shown in figure 11.

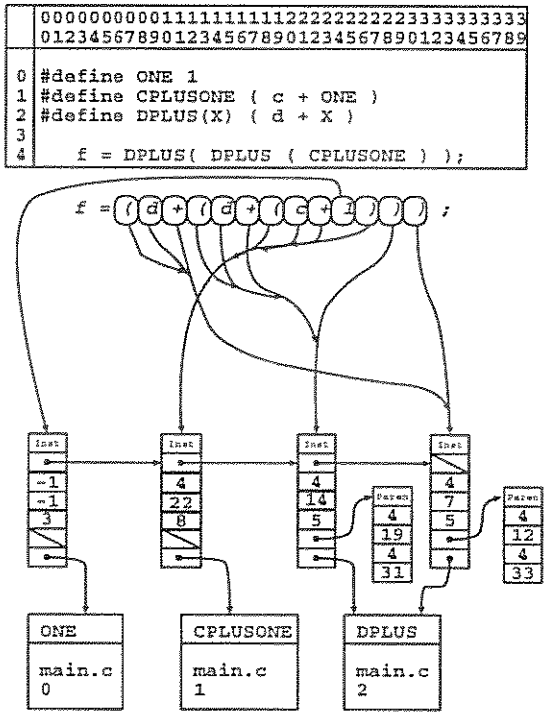


Figure 10: The Macro Highlighting Structure Relating Preprocessor Output to the Source Code

This method has several advantages. It saves memory, as mapping information is not duplicated at each SDG vertex. The highlighting routine is relatively simple; it has only to follow the pointer to the macro highlighting structure to determine the source code that needs to be marked. Had we tried tracking this data with a macro vertex added to the SDG, not only the internal program representation but the highlighting routine would be more complex.

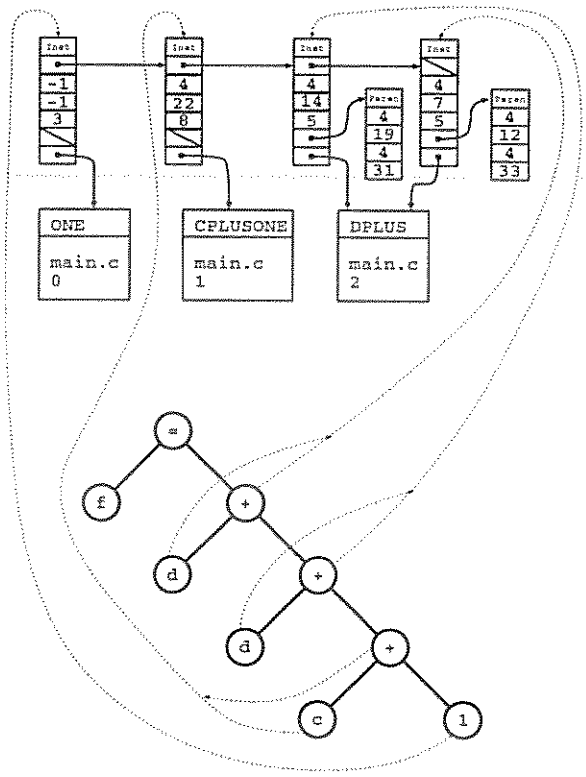


Figure 11: Links from SDG Vertices to a Macro Highlighting Structure

6 Conclusions

GPP produces more information than GHINSU currently uses. Our future work will be directed towards the display of multiple files and improving the highlighting routine. This would allow highlighting conditionally excluded code and viewing slices across files. We have yet to determine an elegant method to display an expanded macro with those tokens captured in the current slice highlighted.

Programs written using preprocessor constructs have an additional layer of abstraction which can increase their difficulty to maintain. In this paper five mappings from the preprocessed code to the source were identified. To illustrate the problem, ANSI C macros were discussed and details of the solution we developed for GHINSU were given.

GPP is an example of a customized preprocessor which retains mappings typically lost. Capturing the correspondence between tokens in the preprocessed and source files permits the use of powerful maintenance tools like slicing, dicing, and ripple analysis.

Acknowledgements

This research was supported, in part, by the Florida/Purdue Software Engineering Research Center funded by the National Science Foundation, the Center's 15 industrial affiliates, and the Florida High Technology and Industry Council, as well as an enhancement grant from BNR (Bell Northern Research).

References

[1] B. Lientz and E. Swanson, *Software Maintenance Management*. Reading, MA.: Addison-Wesley, 1980.

[2] B. Boehm, "The high cost of software," in *Practical Strategies for Developing Large Software Systems* (E. Horowitz, ed.), Reading, MA.: Addison-Wesley, 1975.

[3] M. Weiser, "Programmers use slices when debugging," *Communications ACM*, vol. 25, July 1982.

[4] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, January 1990.

[5] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," in *15th ACM Symposium on Principles of Programming Languages*, (New York), ACM Press, 1988.

[6] L. Badger and M. Weiser, "Minimizing communications for synchronizing parallel dataflow programs," in *Proceedings of the 1988 International Conference on Parallel Processing*, (University Park, PA.), Penn State University Press, August 1988.

[7] J. Lyle and M. Weiser, "Automatic program bug location by program slicing," in *2nd International Conference on Computers and Applications*, 1987.

[8] P. E. Livadas and S. Croll, "A new algorithm for the calculation of transitive dependences," *Journal of Software Maintenance*, (accepted).

[9] P. E. Livadas and S. Croll, "System dependence graphs based on parse trees and their use in software maintenance," *Journal of Information Sciences*, vol. 76, February 1994.

[10] S. Croll, "Towards an internal program representation: The ghinsu core," Master's thesis, University of Florida, 1994.

[11] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, second ed., 1988.

[12] M. Platoff, M. Wagner, and J. Camaratta, "An integrated program presentation and toolkit for the maintenance of c programs," in *Proceedings of Conference on Software Maintenance*, 1991.

Session E:
Parallelization

Parallelizing Sequential Programs by Algorithm-level Transformations

S. Bhansali J.R. Hagemeister
C.S. Raghavendra H. Sivaraman
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752

Abstract

We address a significant problem in parallel processing research, namely, how to port existing sequential programs to run efficiently on parallel machines (the "dusty deck" problem). Conventional domain-independent techniques are inadequate for solving this problem because they miss significant opportunities of parallelism. We present experimental evidence to support our claim, analyze why current techniques are inadequate, and propose a knowledge-based reverse engineering approach for attacking this problem.

1 Introduction

Parallel processing is increasingly being used for computationally intensive scientific and industrial applications, such as computational fluid dynamics, molecular physics and biochemistry research, modeling of environmental effects, graphics and image processing, computer vision, computer aided design and manufacturing. A variety of parallel machines are commercially available like the Intel Paragon, CM-5, KSR, and IBM's SP1 to name a few. With the increased computing power of parallel machines, it is now feasible to use these them for larger and more complex applications. In fact, there is now a revolution in the scientific computing that is using massively parallel computers as a new experimental tool to discover new phenomena (the Federal Government's HPCC initiative).

The main goal of parallel processing research is to achieve efficient and scalable implementations for problems of interest on a parallel machine. One avenue of research is to design new and efficient parallel programs for a given parallel architecture. However, for many important applications, efficient sequential programs already exist. For example, there are many sequential scientific programs which study natural processes through detailed computer modeling. Most of these programs are written in FORTRAN and have been in use for many years. Thus, another approach in parallel programming research is to devise techniques for transforming sequential programs for efficient execution on parallel machines.

Our work is based on this second approach of transforming sequential programs to parallel. Automatic transformation of sequential programs to execute on a parallel machines has been the dream of investigators for several decades ("the dusty deck problem"). It is now generally believed that completely automatic identification of all the inherent parallelism in a sequential program is infeasible. In the past three decades, there has been extensive work on parallelizing tools which aid users in transforming a sequential program to execute on a given parallel machine. These tools analyze sequential programs for parallelism (profiling), analyze critical code sections, and help users port existing sequential programs to a parallel platform. Such tools can be extremely valuable in reducing the porting effort and achieving reasonable levels of performance with parallel programs.

Current techniques of parallelization, employed by the above tools, are based on a syntactic analysis of the source program. These methods work in two phases. In the first phase, based on a syntactic analysis, the source program is transformed to a semantically equivalent serial program. The purpose of this transformation is to expose the inherent parallelism in the source program. In the second phase this transformed program is rewritten so that those sections of the program that can be executed in parallel are marked using parallel constructs like *do-all*, and *do-across*. The transformations that are used during the first phase determine the efficiency of the parallelization. Therefore efficient parallelization requires aggressive transformation.

Our approach uses a novel technique, inspired by research in knowledge-based reverse engineering, to permit significantly more aggressive transformations than possible with existing techniques. The approach is based on recognizing algorithm-level concepts in a sequential program and replacing them by more efficient parallel programs. At present, we are in an early stage of our research and do not have an implemented system. The objectives of this paper are (1) to demonstrate that most of the current tools for parallelizing sequential programs are inadequate, (2) motivate the need for algorithmic level concept recognition for achieving greater parallelism, and (3) to propose a framework for building a knowledge-based

0-8186-5647-6/94 $04.00

parallelizing tool that can assist users to efficiently port their sequential programs to parallel platforms.

We begin by describing the current state of the art in parallelizing tools and describe the performance of a representative tool and why its performance is inadequate. We analyze the reasons for the poor performance of current tools, discuss some related work in knowledge-based reverse engineering, and propose knowledge-based program transformation as a promising approach for attacking the problem. We present an overview of a system that we are building and outline our ideas for future work.

2.0 Parallelization based on syntactic analysis.

Some of the well-known transformations provided by existing parallelizing tools are the following [6, 10]:

Loop fusion: This technique merges adjacent loops in a program provided both the loops iterate between identical loop boundaries, have the same level of nesting, and the statements in the two loops are data independent. This transformation helps reduce startup costs for the two loops that have been merged into a single loop.

Loop splitting: This technique splits a loop into two distinct loops. The purpose of this transformation is to eliminate the data dependencies between the statements in successive iterations. The result of this transformation is two loops both of which can then be parallelized.

Loop interchanging: In certain nested loops the order of nesting of the loop indices may obstruct parallelization. Loop interchanging corrects this by changing the level of nesting of the indices.

Eliminating scalar variables: In certain loops the programmer may have used a scalar variable as a temporary location. This variable obstructs parallelization of the loop. It may turn out that by stretching the scalar variable into an array the artificial data dependence can be eliminated. The resultant loop can then be parallelized.

Cycle Shrinking[11]*:* Consider a loop as shown in the example below:

```
do i=4, n
   a(i) = b(i-3) + a(i)          S1
   b(i) = a(i-4) + f(i)          S2
```

In this loop statement S1 is data dependent on statement S2. Statement S2 is also data dependent on statement S1. So this loop looks inherently sequential. But if we look closely at the subscript expressions of the array references on the 'rhs' of statements S1 and S2 we find that at least three successive iterations of this loop can execute in parallel in every step. Cycle shrinking transforms this loop as shown below so that this parallelism can be exploited.

```
do j= 4, n, 3
   do i= j, j+2
      a(i) = b(i-3) + a(i)       S1
      b(i) = a(i-4) + f(i)       S2
```

The inner loop can now be parallelized. It is important to note that a synchronization point needs to be inserted at the end of the inner loop. So all the processors need to synchronize between every two iterations of the outer loop.

There are several other similar techniques e.g. *loop-unrolling* almost of all these techniques analyze sections of a source code that involve an iterative construct and attempt to transform the code into a form that eliminates data dependence between successive iterations of the loop.

Our empirical experience suggests that these techniques fail to identify many significant opportunities of parallelism in sequential code that people write, or have written. To validate our hypothesis, we conducted some experiments based on both benchmark programs and real application code developed by researchers in different domains.

3.0 Empirical Evaluation of Parallelizing Tools.

There exist several tools marketed by commercial vendors and a few research prototypes that can assist users in parallelizing their code. For example, KSR-1 has a tool called KAP which attempts to "tile" parallel regions in various ways based on a series of switch settings to the tool; Parasoft Corp. offers an automatic translation tool called ASPAR as part of their EXPRESS environment; part of the FORGE tool set is the MIMDizer, an interactive tool that assists users in the creation, maintenance and modification of computer programs for distributed memory parallel machines. Among research prototypes many universities have tools such as the Parafrase-2from the University of Illinois, Urbana.

Most of the research prototypes are not available or are not robust enough to be evaluated, and we do not have the resources to acquire and evaluate many different commercial tools. Therefore, our experiments are based on three representative tools provided by leading vendors and research developers of parallelizing tools. We believe that these tools are a fair indicator of the current state of the art in automatically parallelizing sequential code.

The tools that we evaluated and report on here are KAP, Forge90, and Parafrase-2. To evaluate the tools we chose

eight sample programs consisting of both benchmark programs and real applications:

- *embar.f* embarrassingly parallel code distributed with the NASA parallel benchmark suite.
- *cgm.f* solves a unstructured sparse matrix using the conjugate gradient method from the NASA parallel benchmark suite.
- *appbt.f* solver for five coupled, partial differential equations from the NASA parallel benchmark suite.
- *mgrid.f* a simple multigrid solver in computing a three dimensional potential field from the NASA parallel benchmark suite.
- *buk.f* bucket sort demonstrations from the NASA parallel benchmark suite.
- *LWSI.f*, molecular dynamics simulation of flexible water molecules from the Perfect Club benchmarks.
- *drn5.f*, a solids modeling simulation program used at Washington State University.
- *monte_carlo.f* A monte carlo simulation of elastic acoustic scattering used at Washington State University.

We used each of the tools to process the eight source programs. Our results are based on using each of the tools with default settings as described in the user's manuals provided. We did not experiment with different switch settings or directives in order to optimize performance. The number and complexity of settings requires a fair degree of sophistication on the part of a user and our interest is in evaluating the tool for a novice or intermittent user (who is often a scientist or engineer and not a professional programmer).

The results of our experiments are summarized in Table 1.

The second column indicates the lines of code in each program. The next three columns are the results for each tool tested. Within each of these three columns we show the number of times that the tool transformed a segment of the code for parallel computation and the number of those transformations that would be counter productive to increasing the execution speed. The last two columns give a quantitative evaluation of the tools by indicating the total number of loops that the tool had to consider and an estimate of the number of opportunities with the code to increase performance through distribution of loop execution. (The numbers by themselves do not give a true picture because the performance depends not so much on the *number* of constructs that are parallelized as on the specific constructs being parallelized.) It can be seen that these tools select a limited number of loops to distribute and many of those selections, although syntactically correct, are poor choices for increasing performance. In other words, they attempt to parallelize code segments which would be better left to execute in a sequential manner. For example, both KAP and Parafrase distributed a loop which had only 4 iterations to initialize a four element array to zero. In such a case the communication overhead of parallelization can exceed the reduction in processing time resulting in a negative speedup!

Although it is not apparent from the table, we discovered that these tools miss many opportunities to parallelize code which are embarrassingly parallel. Tools such as KAP are designed to depend on the ability of a programmer to identify inherent parallelism within the source and mark it with directives and assertions which give it guidance. Parafrase, a research tool, tries to distribute every loop possible without consideration of processor number. Forge90 also only distributes loops and does not always make the best choices for distribution. Our study suggests that considerable user effort is needed to obtain more efficient transformations of sequential programs.

		Kap		Forge 90		Paraphrase			
Program	LOC	Loops Distrib.	Inapprop Distrib.	Loops Distrib.	Inapprop Distrib.	Loops Distrib.	Inapprop Distrib.	Num. of DO loops	Parallel opport.
embar.f	265	5	5	2	1	1	1	10	1
cgm.f	862	5	4	7	0	10	0	51	20
abbbt.f	4457	4	4	6	0	108	66	391	52
mgrid.f	676	9	6	3	0	33	18	45	18
buk.f	312	2	2	1	0	1	0	17	4
LWSI.f	1430	6	1	2	0	11	8	185	15
drn5.f	830	0	0	9	0	15	9	49	13
monte.f	1893	3	0	24	7	16	12	57	27

Table 1. Performance of three tools on eight sample programs

The reason these tools fail to optimally or effectively transform sequential code to parallel code is because their analysis of the source code is at a fairly low level of abstraction - called *language concepts* in [9]. They do not analyze the semantics of a program in terms of more abstract *domain-specific concepts* - which is essential in order to perform more powerful optimizations.

4.0 Algorithm-level transformation: An Example

To take a closer look at the kind of parallelizations that KAP (and other similar tools) is unable to recognize, consider the code in Figure 1 which is part of the Molecular Dynamics Simulation of the *Perfect club* benchmarks. The transformed code produced by KAP is shown in Figure 2.

```
C*********************************************
      SUBROUTINE KINETI(VAR,NMOL,SUM,HMAS,OMAS)
C*********************************************
C
C.....THIS ROUTINE EVALUATES KINETIC ENERGY
C
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
      DIMENSION VAR(1),SUM(1)
C
      JJ=1
      DO 100 K=1,3
      S=0.0D0
      DO 110 J=1,NMOL
        S=S+(VAR(JJ)*VAR(JJ)+
     -    VAR(JJ+2)*VAR(JJ+2))*HMAS
     -    +VAR(JJ+1)*VAR(JJ+1)*OMAS
  110  JJ=JJ+3
       SUM(K)=S
  100 CONTINUE
      RETURN
      END
```

Figure 1. Section of code from the Molecular Dynamics Simulation program in the Perfect_club benchmark.

In this example, the loop DO 110 J=1,NMOL controls a global combine over the data array VAR into the single element S. The data structure is indexed by the variable JJ which is also modified within the loop. KAP did perform the scalar optimization of removing the variable JJ and it's associated assignment statement. However, it did not exploit the parallelism within the global combine operation being implemented in the second DO loop. Based on a purely syntactic analysis, KAP finds that there is a data dependency within this loop structure and concludes that the loop cannot be parallelized. However, an experienced programmer would notice that the order of the summation is not important and can be computed in parallel. A programmer could exploit the parallelism by using a "parallel region" construct illustrated in Figure 3. This code will spread the work to a group of processors and then combine the final result.

```
C*********************************************
 SUBROUTINE KINETI (VAR, NMOL, SUM, HMAS, OMAS)
C*********************************************
C
C.....THIS ROUTINE EVALUATES KINETIC ENERGY
C
      IMPLICIT DOUBLEPRECISION (A-H,O-Z)
      DIMENSION VAR(1), SUM(1)
      INTEGER II1, II2
C
C*KSR*  ORIGINALLYLOOP ( 100 )
      DO 3 K=1,3
        S = 0.D0
C*KSR*  ORIGINALLYLOOP ( 110 )
        DO 2 J=1,NMOL
          S = S + (VAR(NMOL*(K-1)*3+J*3-2)
     X          * VAR(NMOL*(K-1)*3+J*3-2)
     X          + VAR(NMOL*(K-1)*3+J*3)
     X          * VAR(NMOL*(K-1)*3+J*3)) * HMAS
     X          + VAR(NMOL*(K-1)*3+J*3-1)
     X          * VAR(NMOL*(K-1)*3+J*3-1)* OMAS
    2       CONTINUE
        SUM(K) = S
    3   CONTINUE
      RETURN
    END
```

Figure 2. Section of the code from the Molecular Dynamics Simulation program in the Perfect_club benchmark after modification from KAP.

```
C*********************************************
SUBROUTINE KINETI (VAR, NMOL, SUM, HMAS, OMAS )
C*********************************************
C
C.....THIS ROUTINE EVALUATES KINETIC ENERGY
C
      IMPLICIT DOUBLEPRECISION (A-H,O-Z)
      DIMENSION VAR(1), SUM(1), S(numberprocs)
      INTEGER II1, II2, mynum
C
C*KSR*  ORIGINALLYLOOP ( 100 )
        DO 3 K=1,3
          S = 0.D0
C*ksr* parallel region (numthreads=numberprocs,
                        private=J, mynum)
C*KSR*  ORIGINALLYLOOP ( 110 )
          mynum = ipr_mid()
          DO 2 J=1,NMOL
            if(mod(J,numprocs) .eq. mynum)
            S(mynum)=S(mynum)
     X        + (VAR(NMOL*(K-1)*3+J*3-2)
     X        * VAR(NMOL*(K-1)*3+J*3-2)
     X        + VAR(NMOL*(K-1)*3+J*3)
     X        * VAR(NMOL*(K-1)*3+J*3)) * HMAS
     X        + VAR(NMOL*(K-1)*3+J*3-1)
     X        * VAR(NMOL*(K-1)*3+J*3-1) * OMAS
    2       CONTINUE
```

```
C*ksr* end parallel region
            for 3 J=1, numberprocs
              SUM(K) = SUM(K) + s(J)
    3       CONTINUE
          RETURN
        END
```

Figure 3. Parallel region pseudo code for the Molecular Dynamics Simulation program. The italicized parts are statements inserted manually by a programmer.

A global combine is a very common operation in scientific code and is often a sub-part of higher level operation or algorithm. It is a generalization of the more commonly used *global sum* operation shown below:

```
REAL X(NROWS)
SUM = 0.0
DO 10 I=1,NROWS
   SUM = SUM + X(I)
10 CONTINUE
```

Figure 4. Global sum operation

From this example we can identify two principle attributes of a global combine operation. First, the operation is contained within a loop and works over an indexed data structure (array). Secondly, and more specific, the body consists of a summation of all the data from the array. It should be noted that the combine operation within the loop body can be one or a combination of many different operations which combine the data in the structure into a single element.

We can generalize the various global operations and represent them as an abstract pattern as shown below:

```
GLOBAL-SUM:
DO  ?LABEL  ?I = ?START, ?FINSH
      ?COMBINED = ?COMBINED @OP
                  @h(?DATA[@f(?INDEX)])
      ?INDEX  =  @g(?INDEX)
@LABEL  CONTINUE
where
      Associative-op(@OP)
      Linear-function(@f,?index)
      Linear-function(@g,?index)
      One-to-one-function(@f)
```

Figure 5. Abstract pattern for the global combine operation.

Here, symbols preceded by a ? refer to a variable and symbols preceded by a @ refer to a function or operation. $@f(?i)$ denotes a function on *?i*. ?x[$@e_1$,$@e_2$,...$@e_n$] denotes an n-dimensional array indexed by expressions $@e_1$,$@e_2$,...$@e_n$.

The syntax of specifying an abstract pattern has been adapted from [9] and is:

Pattern-name(args):
  <pattern>
where
  <conditions>

A section of code C is said to be an instance of the above pattern if there exists a substitution σ such that $<pattern>\sigma = C$ and $<conditions>\sigma$ is true. Thus, according to Figure 5, the combine operation, @OP, in the global-combine pattern must be an associative operator, $@f\sigma$ and $@g\sigma$ must be some linear functions of *?index*[1], and $@f\sigma$ must be a one-to-one function. The last condition is needed to ensure that the computation of each component of the array $?data\sigma$ is independent and hence can be done in parallel. (Note that this my be too strict in some cases since it is based on the assumption that READ-READ conflicts inhibit parallelization).

The simple global sum example can be seen to be an instance of global operation since it matches with the following substitution:

?i = I, ?combined = SUM, ?data = X,
?index = I, ?label = 10, @op = +,
@h = *id*, @f = *id*, @g = *id*

where *id* is the identity function. Note that in the current form the global combine pattern would still not recognize the code in Figure 1 as a global operation. We need to generalize the pattern to be able to recognize such instances.

One possibility is to replace
```
?COMBINED  =  ?COMBINED  @OP
@h(?DATA[@f(?INDEX)])
```
with
```
?COMBINED = ?COMBINED @OP
```
$$@h(\$\overset{?size}{\underset{?i}{TUPLE}}(?DATA[@f_i(?index)]))$$

Here $\$\overset{?size}{\underset{?i}{TUPLE}}$ is a special constructor that returns a tuple of size ?size. The components of the tuple are indexed with ?i. In the global combine example, $\$\overset{?size}{\underset{?i}{TUPLE}}$ would match

```
( VAR(JJ),VAR(JJ),VAR(JJ+2),VAR(JJ+2)),
VAR(JJ+1),VAR(JJ+1))
```

[1] ?index can be bound to the same identifier as ?i

Program	*embar.f*	*cgm.f*	*appbt.f*	*mgrid.f*	*buk.f*	*LWSI.f*	*drn5.f*	*monte.f*
Global Combines	1	8	29	2	1	15	3	22

Table 2. Number of global combine computations in sample programs

To obtain preliminary evidence for our belief that a small number of patterns are used extensively in scientific computation, we counted the number of times the global combine operation occurred in the eight example problems in Table 1. The results are shown in Table 2 below. It can be seen that 89 different instances of the same abstraction were found in just eight observed programs! Our observations are corraborated by others who found that global accumulations appeard in nearly every code that they examined and were responsible for preventing most important loop nests from being parallelized [4]

The pattern matching can be done by using a standard unification algorithm together with some reformulation rules that rewrite the abstract syntax tree produced by the parser. A somewhat related approach to pattern matching has been used by researchers in other domains, e.g. [3]. The conditions can be checked by calling a set of service routines from a library. This is similar to the approach adopted in [9]. However, for the scientific computation domain the nature of the constraints to be checked are different and we will need to provide a different set of service routines.

Besides global combine, we have identified several other abstract concepts that occur frequently in scientific computation and are not optimized for parallelism by conventional tools. Some examples of these concepts are:

- *Relaxation.* This is a computation process used to determine the value of a data point based on neighbors within some bounding region. For example the *drn5.f* code performs a relaxation operation on the position atoms in a lattice based on the attractive and repulsive forces of other atoms near it.

- *Matrix operations.* These are any of the numerous numerical methods that operate on matrices and have been implement in code such as Gaussian Elimination and Matrix inversion. There exist libraries of very efficient implementation of these algorithms for parallel architectures and if one is able to recognize these concepts in sequential code, one could transform them to efficient implementations using these libraries.

- *Multigrid operations.* These involve the use of a neighbor to calculate a new value for a data point. Noise reduction in image processing is an example of a multigrid operation. (A pixel is considered noisy if its values differs from its neighbors by some pre-defined amount.)

- *Data Dependency Data structure Expansion.* This is a program fragment that has a multiply or singly nested loop which contains a variable that is a scalar along one or more loop dimensions. The presence of such a scalar introduces a data dependence that obstructs parallelization. In certain programs it may be possible, on the basis of a semantic analyses, to expand this variable so that it is not a scalar along any dimension. The program fragment can then be parallelized.

- *Scatter*: In a parallel machine, one processor has an array of data which should be distributed to other processors uniformly. It is the exact opposite of a global combine operation.

- *Histogram.* Histogram calculation is a frequently used operation in scientific computing, which can be implemented using efficient parallel algorithms on different architectures.

- *Approximations*: Quasi-Newton method and other techniques are used in finding approximate solutions through iterations for many problems. Such techniques are commonly used in scientific programs for finding fairly accurate local solution for unconstrained optimization problems.

- *Differential Equations*. Solution of differential and partial differential equations that have efficient parallel algorithms.

5.0 A Knowledge-based approach

It is generally believed that when parallelizing techniques are used on a sequential algorithm, the resultant code is usually not as efficient as a parallel algorithm. However, as the previous example shows, using a functional level understanding of programs one can translate a sequential program into equivalent programs with algorithms that will lend themselves to a higher degree of parallelism. The gobal combine that was demonstrated is implemented on some systems. We believe that there are several common operations, concepts, and data organizations within the

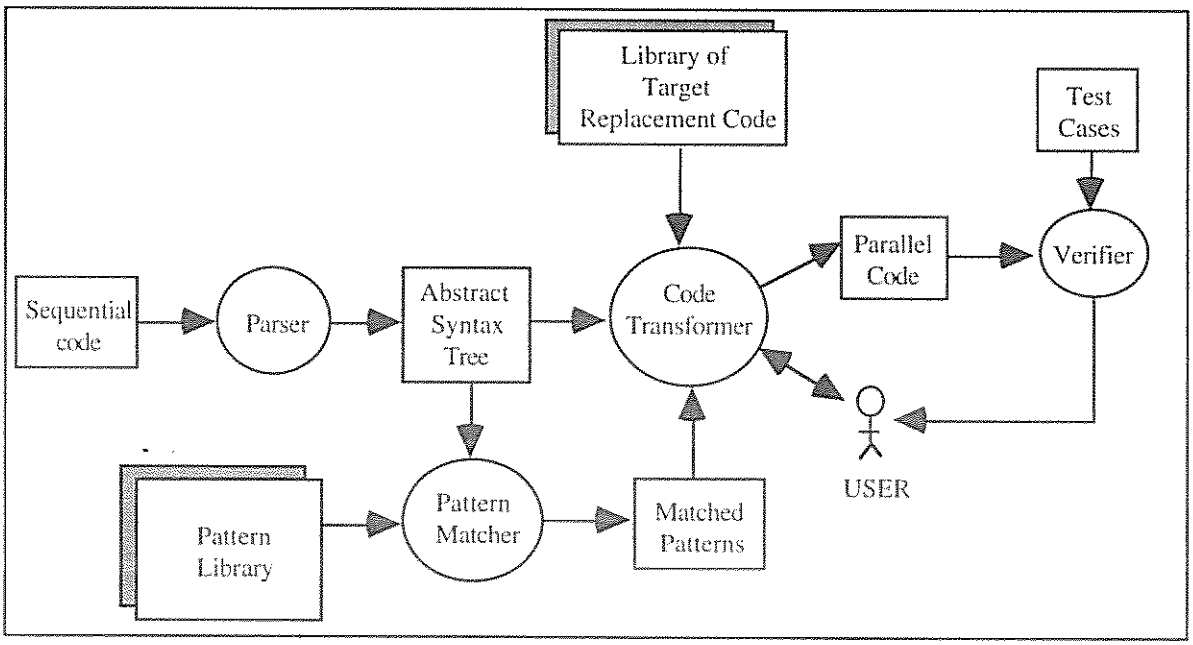


Figure 6. Architecture of a knowledge-based parallelizing tool

domain of scientific computation programs that can be recognized by performing a deeper analysis of the source code, and once recognized, transformed into a form that is more appropriate for the target environment. We now describe the architecture of a proposed system to implement our approach.

5.1 Overview of proposed environment

An architectural overview of our system is shown Figure 6. Initially we will focus on creating a *library of patterns* to identify commonly occurring parallelizable constructs. These patterns would be indexed on subdomains within the broad domain of scientific computation. Our initial efforts would be restricted solely to sequential programs written in FORTRAN. The code transformation would be accomplished in two stages. In the first stage the *parsing and pattern matching system* would be used to first create an abstract syntax tree for the source code which will then be matched against patterns in the pattern library. The matched patterns are potential candidates for parallelization and would be passed to the second stage.

The second phase utilizes a second *library of replacement code*, indexed by target architecture, for replacing matched patterns by efficient parallel code. We subscribe to the widely held belief that it is infeasible to identify all inherent parallelism in a sequential code completely automatically. The system's role is to identify potential regions in the code to be parallelized and suggest possible replacements for the code; the user will be responsible for making the final decision. Finally, the transformed code will be verified for correctness using a user-furnished suite of test cases. 6.0 Related Work

Current approaches to automatically transform sequential code into parallel code have been already described in Section 2. Here we describe some of the relevant work in the knowledge-based software engineering and Artificial Intelligence literature.

Our approach has been inspired by work in knowledge-based program understanding and reverse engineering , a relatively young area of research [1, 5, 8, 9, 13]. The goal of this research is to obtain a functional level understanding of a program by analyzing the source code. Our approach is most closely related to the work of Kozaczynski *et al.* [9]and Wills[13]. Both these approaches utilize a library of abstract concept representations, and program understanding proceeds by searching for the occurrence of these concepts in the source code. The abstract concepts are called *plans* and have been influenced by the Plan Calculus in the Programmer's Apprentice project [12]. However, the two systems differ considerably in the details of the plan representations and the plan matching algorithm. Wills uses a directed, acyclic graph representation for representing plans and uses flow-graph parsing to recognize plans. Kozaczynski *et al.* represent plans as an abstract pattern and seem to use standard unification and limited constraint checking to recognize instances of those patterns in the source code.

Both of these systems exemplify a bottom-up approach to program understanding by starting at the source code and

trying to recognize increasingly abstract concepts. An alternative approach, exemplified by the work of Johnson and Soloway [8] and Allemang [1], is to proceed top-down. In this approach a functional goal is assumed (in practice it is supplied by a user) and the task is to explain how the implemented program achieves or fails to achieve the functional goal. In our current work we are adopting a bottom-up approach since we are dealing with large programs with complex functional goals, which cannot be specified easily by an end-user. More importantly, it is not necessary to understand a complete program in order to parallelize it (although this is true for small programs and subroutines); in most cases, it is enough to identify sub-parts of a program. Thus, a top-down approach for this task would be an "over-kill" requiring a significant amount of domain knowledge without offering any increased opportunities for parallelism.

While work on program understanding is limited to plan recognition, the goal in reverse engineering is to use that understanding to modify the program for a variety of reason (e.g. performance enhancement, restructuring to improve code modularity or comprehension). The use of program transformational techniques to perform this task has been utilized by others e.g. [2, 5]. However, as far as we know, program transformation techniques have not been utilized to automatically restructure sequential programs into parallel programs.

7. Conclusion

The difficulty of porting sequential programs to parallel environments, in a manner such that they will perform efficiently, is a significant impediment to the use of high performance parallel computing by the scientific community. The solution of this problem is paramount to the use of high performance parallel computing by researchers and engineers.

Our solution represents a novel approach to this "dusty deck" problem facing computer users. The approach is based on using domain-specific knowledge about program fragments to identify parallelism within sequential source code. This would complement the domain-independent techniques on which the parallel programming community has been focusing for the past three decades. The feasibility of our approach has been demonstrated in reverse engineering and program understanding work. However, as far as we know, the application of these techniques to the "dusty deck" problem is quite novel.

Our future plans include an in-depth evaluation of other parallelizing tools and identification of commonly occurring patterns in scientific code. Once we have a prototype system we will evaluate its performance on several benchmarks as well as real applications developed by users in the scientific community. We believe that our work has the potential for significantly advancing the state of the art in parallelizing tools. In the long run, this could help enhance the usage of parallel computing and in turn the productivity of the scientific community.

References

[1] Allemang, D., *Understanding programs as devices.* 1990, Ohio State University.

[2] Arango, G., *et al., TMM: software maintenance by transformation.* IEEE Software, 1986. **3**: p. 27-39.

[3] Bhansali, S. and G.A. Kramer. *Planning from first principles for geometric constraint satisfaction.* in *Proceedings of AAAI-94.* 1994. Seattle, WA.

[4] Blume, W. and R Eigenmann, *Performance analysis of parallelizing compilers on the Perfect Benchmark Programs.* IEEE Transactions on Parallel and Distributed Systems, 1992. **3(6)**, p. 643-65

[5] Bush, E. *The automatic restructuring of COBOL.* in *IEEE Conference on Software Maintenance.* 1985.

[6] Callahan, D., *A global approach to detection of parallelism.* 1987, Dept. of Computer Science, Rice University.

[7] Etter, D.M., *FORTRAN-77 with Numerical Methods for Engineers and Scientists.* 1990, Redwood City,CA: Benjamin/Cummings Publishing Company.

[8] Johnson, W.L. and E. Soloway, *PROUST: knowledge-based program understanding.* IEEE Transactions on Software Engineering, 1985. **11**.

[9] Kozaczynski, W., J. Ning, and A. Engberts, *Program Concept Recognition and Transformation.* IEEE Transactions on Software Engineering, 1992. **18**(12): p. 1065-1074.

[10] Moldovan, D.I., *Parallel Processing from applications to systems.* 1993, San Mateo, CA: Morgan Kaufmann.

[11] Polychronopoulos, C.D., *Compiler optimizations for enhancing parallelism and their impact on architecture design.* IEEE Transactions on Computers, 1988. **37**(8): p. 991-1004.

[12] Rich, C. *A formal representation of plans in the programmer's apprentice.* in *Proceedings of the 7th International Joint Conference on Artificial Intelligence.* 1981.

[13] Wills, L.M., *Automated Program Recognition: A Feasibility Demonstration.* Artificial Intelligence, 1990. **45**: p. 113-171.

Towards Automated Code Parallelization through Program Comprehension

B. Di Martino G. Iannello

Dipartimento di Informatica e Sistemistica
Università di Napoli
v. Claudio, 21 - 80125 Napoli, Italy

Abstract

Currently available parallelizing tools are biased in favor of a particular parallel execution model for generating the output parallel program. This obviously limits the generality of these tools, since programs may be parallelized according to different programming paradigms. In this paper we propose a novel approach to automated code parallelization that tries to overcome these limitations. This approach consists in recognizing first the paradigm that is best suited to a given program to be parallelized, and then applying paradigm-specific transformation to generate the final parallel code. We argue that the recognition phase can be fully automatized using techniques developed in the framework of automated program understanding. With the help of a case study, we discuss how this new approach could be implemented and propose the basic structure of a paradigm-oriented parallelizer.

1 Introduction

So far, the focus of research on parallelization of sequential code by (semi) automatic tools has been on computational science and virtually all available tools are oriented to FORTRAN as a source language. The main goal of these tools is to make explicit the parallelism inherent to iterative control structures, through a sophisticated data-flow and dependence analysis [10, 12].

Currently available tools are all biased to a particular parallel execution model referred to in the literature as the SPMD (single program multiple data) model [6], allowing parallel code to be generated relatively simply. On the other hand, several authors have recently argued that parallel programs can be classified according to their *algorithmic skeleton* [4] or *programming paradigm* [3], i.e. to the way processes making up the parallel program are created, synchronize and communicate, abstracting from the details of their sequential code. According to this view, the SPMD model can be viewed as a family of paradigms, all characterized by common synchronization and communication patterns.

Unfortunately, the use of a fixed target paradigm, even one very flexible and quite general like the SPMD model, prevents from choosing the parallelization strategy that is best suited to the characteristic of the algorithm to be parallelized.

The main reason currently available parallelizers cannot choose among different parallelization strategies is that they perform a purely structural analysis of the sequential code. Conversely, knowledge about the most convenient parallelization strategy for a specific class of algorithms could be applied only if parallelizers were able to "recognize", at a proper abstraction level, *what* the sequential code does. In other words the program should be analyzed for discovering abstract concepts and assinging them to their realizations within the program itself. This problem has been referred to in the literature as the *concept assigning problem* [2].

While the concept assigning problem does not seem to be completely automatable in its general form, it could be fully automated if the recognition focused chiefly on programming-oriented concepts, such as searches, sorts, structure transformations, numerical integration, etc. For instance, in [8, 9], program understanding is viewed as a parsing process that looks for specific signatures representing concepts in the target program. The recognizer program uses a finite set of pattern templates (called clichés in [9]) that identify the concept signatures in a parsing process in which the less abstract concepts are recognized first, and then they become features of larger-grained composite concepts.

These results appear to be especially interesting with respect to parallelization since the abstract concepts that should be recognized in order to apply effective parallelization strategies can be classified as programming-oriented and are of the same kind as those recognized by the tool described in [9].

In this paper, we develop these ideas and propose a parallelization procedure based on automated program understanding that can overcome limitations of existing parallelizers. After a brief introduction to currently available parallelizers, we outline the main limitations of their structural approach and contrast it with a parallelization process based on a multitude of parallel programming paradigms. We then propose, with the help of a case study, a parallelization procedure that first tries to recognize those algorithmic properties of the sequential program concerning parallelization, then chooses the best suited parallelization strategy (the *paradigm*), and eventually generates the

0-8186-5647-6/94 $04.00

output parallel program starting from the input (sequential) code, from a skeleton code associated with the chosen paradigm, and from other output information produced during the recognition phase. Finally, the basic architecture of a tool implementing this parallelization procedure is presented and available techniques for its realization are discussed.

2 Tools for automated parallelization

Parallelizers are usually made up by a *front end*, that is substantially independent of the target parallel architecture, and a *back end* that is architecture-dependent and perform actual code generation [12].

The front end translates the program text in an internal representation suitable for program manipulation and performs code *normalization*.

The main goal of the back end is the generation of a parallel version of the program. This process relies on a catalog of transformations and on a strategy for their application which takes into account the results produced by the front end. These transformations can be classified according to the nature of the code they can be applied to.

The strategy controlling the application of these transformations obviously depends also on the architectural characteristics of the target machine and on the cost of its basic operations.

Since parallelization is so dependent on the target architecture, in our discussion we focus on distributed memory computers. These machines, also referred to as multicomputers in the literature [1], consist essentially of a number of autonomous processing units (*nodes*), each with a private local memory, connected by a message-passing network. The intrinsic scalability of this organization and its very favorable cost/performance ratio with respect to other alternatives make multicomputers the best candidate for future high performance machines and motivate the increasing attention which is being paid to improve their usability.

Transformations for DO loops are by far the most suited for parallelization in a distributed memory framework. When only parallelization of DO loops is performed, code generation is greatly simplified if a Single Program Multiple Data (SPMD) execution model is chosen for the output (parallel) program. According to the SPMD model [6], a parallel program consists of a number of processes started together and running in parallel. All processes execute the same sequential code that is divided into *sections*. All processes synchronize between subsequent sections. Provided that loop iterations are not constrained to be executed serially, in the SPMD framework parallelization of DO loops is easily performed by executing the loop in a *parallel* section and by distributing loop iterations among the processes.

Since in parallel sections loop iterations assigned to different processors operate on different data (typically elements of arrays), in a multicomputer environment a further *data distribution* problem arises from loop iteration distribution. Actually, data distribution and loop iteration distribution are tightly related problems and the solution for one of them implicitly induces a solution for the other [12]. Unfortunately, available parallelizers cannot solve the problem in the general case and user intervention is generally requested to perform either data distribution or loop iteration distribution.

From this brief overview on parallelization tools, it is apparent that they basically perform a structural analysis of the program to identify which loops can be distributed among the processors and which cannot. Since the final program is conforming to the SPMD model, the code produced consists of a single sequential program obtained by applying to the original program text local transformations such as: insertion of synchronization points between sections, conditional execution of code corresponding to serial sections, loop tiling to insert intra-loop communications, etc. Consequently, the parallelization process can be applied only to those computations for which an efficient data parallel version can be written. It is worth noting that not only automatic parallelization of algorithms that do not fit the data parallel pattern, but also the application to irregular data parallel computations of sophisticated parallelization strategies that would exhibit better performance than SPMD is prevented [7].

These considerations lead to the conclusion that we should do away with the constraint represented by the SPMD, as the only model output program have to conform to, if we want to improve the performance of parallelization tools. The first step in this direction would be to determine alternative execution models to SPMD that could be integrated in an automatic parallel code generator.

In our opinion, such alternatives do exist and they have been pointed out in the recent literature on parallel programming techniques, though in contexts other than automatic parallelization. In fact, more than a decade in parallel program development has shown that the "parallel structure" of most programs can be classified according to a limited number of patterns, that can be viewed as representative of the basic ways to organize a parallel computation. This idea has been explicitly expressed in the context of parallel programming methodologies [3, 4] and parallel programming languages [5].

Even though the notion of "parallel structure" of a program has not been well formalized yet, it can be informally defined as *the way processes forming the parallel program are created, synchronize and communicate, abstracting from the details of their sequential part*. Following [3], we will refer to this notion of parallel structure of a program with the term *Parallel Programming Paradigm*, or more briefly *Programming Paradigm* (PP).

From this informal definition it is apparent that a PP is just a template, parametrically defined with respect to the actual number of processes instantiated, the number of similar interactions between them during the computation, the length of messages exchanged, etc. In this respect, the SPMD model can be considered as a PP, or, more specifically, as a family of PPs, since synchronization and communication patterns of SPMD programs may slightly differ within

the constraint that all processes must participate to these events.

The availability of different PPs can be used to extend the capabilities of automatic parallelization tools. The idea is that parallelizers should be able to recognize which paradigm (or paradigms) are best suited for parallelization of a given sequential algorithm and to apply paradigm-oriented transformations to derive the final parallel program. Assuming that a catalog of PPs is available, to make effective the paradigm-oriented approach to parallelization just outlined, we have to solve essentially two problems: (a) how the best PP is selected and (b) how transformations are selected and applied to generate parallel code. We believe that the solution to these problems involves some kind of program comprehension, possibly integrated with traditional parallelization techniques based on structural analysis. In particular, automated program understanding techniques developed for maintaining sequential code appear to be especially suited for this task. In the following sections, through the presentation of case studies, we will further develop these ideas and we will informally propose a paradigm-oriented parallelization procedure amenable to being fully integrated in an automatic tool.

3 Parallelization using paradigms

In this section we discuss the parallelization process for two case studies that does not fit the SPMD model of parallel execution, in order to better understand how this process can be automated. We have chosen two programs that have the same basic structure, but differ in minor aspects that are otherwise relevant with respect to parallelization. We made this choice both to stress the recognition capabilities that a paradigm-oriented parallelizer should possess and to analyze how different parallel programs can be derived starting from structurally similar sequential code. Both programs have been coded by students with moderate programming experience, without having in mind any concerns about parallelization.

The first program, coded in Pascal, implements the iterative version of the quicksort algorithm and includes the inputting of the list to be sorted and the outputting of the sorted list. Figure 1a shows the complete program, with subprograms entirely macro expanded by hand. The second program, coded in C, implements binary branch-and-bound search, and includes inputing of initial data and outputting of the final result. Figure 1b shows the complete program, that did not contain subprograms.

Even though the two programs perform completely different functions, they both share a iterative *divide-and-conquer* resolution strategy corresponding to the traversal of a binary tree. Each node of the tree corresponds to a sublist to be sorted in the first program, and to a search subspace in the second program. Both programs share the basic control and data structures that are outlined in the two figures by means of annotations on the right of the code. More specifically, the tree traversal is performed by executing a loop that, at each iteration, processes a data structure representing the current problem and derives two subproblems from it. One subproblem become the current problem, while the other is pushed onto a data structure implementing a stack. Alternatively, if the current problem satisfies a given condition, some processing is possibly done and a new current problem is popped out from the stack. The loop terminates when the stack is empty. Note that the way the stack and the problems are represented in the two programs are quite different.

Once an expert programmer has recognized the algorithmic structure just described, he can select among two parallel paradigms to perform the tree traversal in parallel.

A first PP, that we call *tree computation*, consists of a set of processes connected by communication channels according to a binary tree structure. Each process receives a problem to be solved, tests for a condition and then either splits the problem into two subproblems that are sent to two child processes, or does some processing and returns a result to its parent. When a process sends over two subproblems, it remains waiting for a reply from each child, then combines these replies and sends back the new result. In practice a number of questions (how deep must be the tree of processes, if processes have to be created dynamically or the entire tree must be instantiated statically, etc.) have to be answered before a working program can be generated. However, they do not depend on the particular nature of the computation to be parallelized, but rather they are part of the PP and can be solved once and for all in the context of the paradigm itself. Figure 2 informally illustrates the paradigm through a graph specifying how processes communicate (fig. 2a) and a pseudocode describing the behavior of the component processes (fig. 2b).

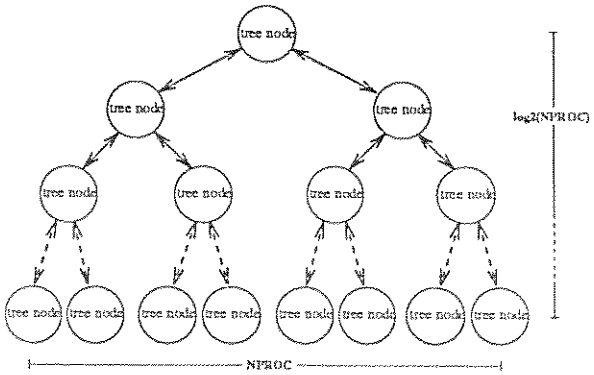
The second PP, that is usually referred to as a *processor farm*, consists of a *coordinator* process and a set of *worker* processes that act as slaves of the coordinator. In a processor farm, the coordinator decompose the work to be done in subproblems and assigns a different subproblem to each worker. Upon receipt of a subproblem, each worker solves it and returns a result to the coordinator. Again some details have to be defined before the PP can become a working program, and slightly different organizations can be selected for the processor farm (for instance workers may or may not be allowed to communicate each other). However, even in this case, these issues do pertain to the PP definition and can be entirely dealt with in the paradigm context. The processor farm paradigm is illustrated in figure 3.

Going back to the two programs to be parallelized, in principle both the tree computation paradigm and the processor farm paradigm (and perhaps others) might be chosen for the parallelization. In practice, however, an experienced programmer would choose the former PP for quicksort and the latter PP for branch-and-bound.

This because the split phase in the quicksort is the costliest part of the algorithm and it would be better to perform it in parallel as much as possible. In this respect, the tree computation is apparently more convenient. Moreover, unless a worst-case list is given

Figure 1: Case study programs: (a) Quick sort; (b) Branch & Bound.

in input, each split phase generates two subproblems with nearly the same associated work, leading to a well balanced tree computation that fit the process structure of the paradigm.



(a)

```
proc tree_node:
  depth_max = log2(NPROC)
  if root node then
    depth = 1
    <prologue>
    <initialize "problem" to "initial problem">
  else
    recv depth from parent
    recv "problem" from parent
  endif
  if depth < depth_max then
    <split "problem">
    spawn a_son
    send depth+1 to a_son
    send "a subproblem" to a_son
    spawn other_son
    send depth+1 to other_son
    send "other subproblem" to other_son
    recv "a subresult" from a_son
    recv "other subresult" from other_son
    join "result"
  else
    <tree traversal>
  endif
  if root node then
    <epilogue>
  else
    send "result" to parent
  endif
end proc
```

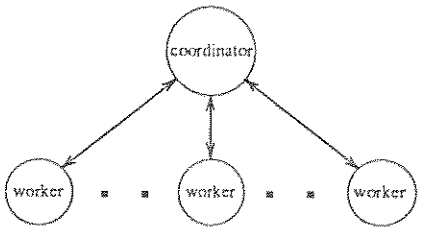
(b)

Figure 2: The tree computation paradigm: (a) communication graph; (b) pseudo code of a tree node process.

Conversely, in the branch-and-bound case the split phase is quite cheap, whereas the subproblems generated might correspond to very different amount of work owing to the presence of a pruning strategy to reduce the search space. This time, the processor farm paradigm appears to be best suited, since on one hand splitting subproblems sequentially does not represent a significant penalty, and, on the other, it is quite easy to embed a load balance policy in the coordinator that can keep workers busy most of the time. Finally, in branch-and-bound, pruning is partly performed on the basis of a global value (the current optimum) that is kept updated and made available to all workers. In the processor farm paradigm, this can be guaranteed if the coordinator collects local optima from workers, and returns them the updated values of the global optimum.

Before closing this section, we briefly discuss how the two problems are actually parallelized using the paradigms shown in fig. 2 and 3. In a paradigm-oriented methodology, parallelization corresponds to fleshing the skeleton of the chosen paradigm with problem-dependent sequential code. In the two cases considered, most of this code can be extracted from the original code shown in figures 1a and 1b, and inserted into the paradigm skeleton.

(a)




```
proc coordinator:
  <prologue>
  <initialize "threshold">
  <initialize "problem" to "initial problem">
  <initialize "stack">
  repeat
    <body of repeat>
  until <"stack" is empty>  or depth of "stack" == NPROC-1
  if depth of "stack" != 0 then
    spawn worker[1]
    send "prologue global data" to worker[1]
    send "threshold" to worker[1]
    send "problem" to worker[1]
    for i=2 to NPROC
      spawn worker[i]
      send "prolog global data" to worker[i]
      send "threshold" to worker[i]
      pop "subproblem" from stack
      send "subproblem" to worker[i]
    endfor
    repeat
      recv local_threshold from a_worker
      if local_threshold < "threshold" then
        "threshold" = local_threshold
        send "threshold" to all workers
    until all workers idle
  endif
  <epilogue>
end proc
```

(b)

```
proc worker:
  recv "prologue global data" from coor
  recv "threshold" from coor
  recv "problem" from coor
  <initialize "stack">
  niter = 0
  repeat
    <body of repeat,
     when <set "threshold" to "best pruning value">
     do send "threshold to coor>
    niter++
    if niter multiple of FREQ then
      non_blocking_recv "threshold" from coor
  until <"stack" is empty>
  send idle to coor
end proc
```

(c)

Figure 3: The processor farm paradigm: (a) communication graph; (b) pseudo code of the coordinator process; (c) pseudo code of the worker process.

Almost no change to the skeleton and to the sequential code to be inserted is required to generate the final parallel version of the quicksort algorithm. The only modifications concern operations such as `recv "problem" from partent`, which must be expanded in several simpler communications to cope with the

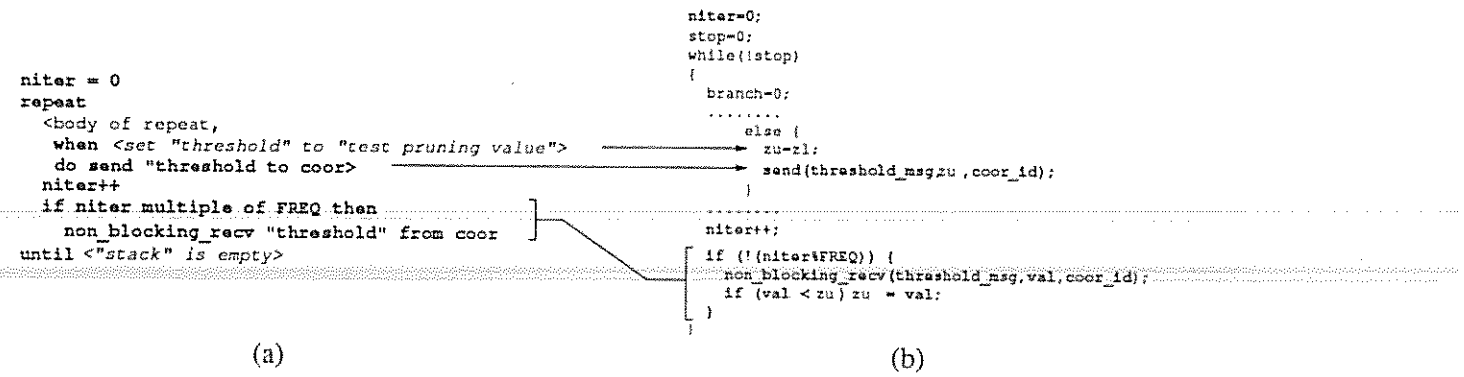


Figure 4: Skeleton instantiation for the worker process: (a) pseudo code; (b) final code.

structured nature of the problem description.

Similar considerations can be repeated with respect to the instantiation of the processor farm paradigm to the branch-and-bound algorithm of figure 1b. For instance, in the worker pseudo code, the when clause specifies that sequential code representing the body of the cycle must be modified so that a particular action takes place every time the "threshold" is set to "test pruning value". Figure 4b shows the final generated code and how it corresponds to the template (fig. 4a) contained in the paradigm skeleton of figure 3.

4 Paradigm selection through program comprehension

We examine now in greater detail the paradigm-oriented parallelization process outlined above to better understand how the PP selection phase could be performed automatically. We focus our attention here on the selection phase, postponing the analysis of the parallel code generation phase to next section.

In the examples presented, a first paradigm selection was performed after having recognized that both algorithms follow a particular pattern that is generally referred to as iterative divide-and-conquer. The features that identify this algorithmic pattern concern the way some abstract functions (split problem, push a subproblem onto stack, pop problem from stack, set current problem to other subproblem, etc.) are related and organized into a specific control structure.

Note that with the term "control structure", we cannot merely intend a particular composition of elementary constructs such as while, if-then-else, etc., but we have to give this term a more abstract and general meaning. For instance, every way to assemble the basic abstract functions making up the divide-and-conquer pattern, that can be proved equivalent to the one shown in figure 5, gives rise to a concrete divide-and-conquer algorithm. In other words, on one hand this notion of control structure should be independent to some extent of the specific control constructs chosen to implement it. On the other hand, this notion cannot be reduced to just structural properties of control-flow only, but it heavily depends on the existence of specific relations between the abstract functions making up the pattern and conditions in the control structure connecting them.

A similar situation has been encountered when we had to refine paradigm selection on the basis of the complexity of the split phase and of the presence of a pruning condition. In these situations too, what we have to do is to recognize that code implementing specific functions is present, and this code is characterized by a particular combination of functions and control.

From these considerations it is apparent that, in the general case, we have to recognize a mix of abstract functions and control structure in order to know that a given parallel paradigm can be applied for effective parallelization. While this kind of abstract structure cannot be formally defined once and for all, we will use in the following the term *parallelizable algorithmic pattern* (PAP) for referring to the set of algorithmic properties characterizing a group of concrete algorithms as parallelizable according to some common parallel paradigm.

Assuming now that a set of PAPs and associated PPs are given, the automatization of the paradigm selection phase basically requires:

- choosing a representation for the sequential program to be analyzed;
- choosing a representation for the PAPs to be searched in the input program;
- defining an effective recognition procedure.

Actually, the problem of recognizing algorithmic patterns is a kind of program recognition, that is the process of discovering abstract concepts in the input code and assigning them to their realizations within the code itself. This problem has been studied, for instance, in the context of reverse engineering to (partially) automatize maintenance and reuse of sequential software [11]. In its general form, the problem does not seem completely automatable, because recognition of human-oriented concepts embedded into the code appears to use informal, inherently ambiguous tokens and relies heavily on *a priori* knowledge from the specific problem domain [2].

However, if only programming-oriented concepts, such as searches, sorts, structure transformations, numerical integration, etc., have to be recognized, a more

```
<initialize "stack" to empty>
<initialize "problem" to "initial problem">
repeat
  if <"problem" is not "trivial problem">   then
    <split "problem">
    <push "a subproblem" onto "stack">
    <set "problem" to "other subproblem">
  else
    <solve "problem">
    if <"stack" is not empty>  then
      <pop "problem" from "stack">
    endif
  endif
until <"stack" is empty>
```

Figure 5: PAP of iterative divide-and-conquer.

pragmatic, automatable approach may be adopted. For instance, in [8, 9], program understanding is viewed as a parsing process that looks for specific signatures representing concepts in the target program. The recognizer program uses a finite set of pattern templates (called clichés in [9]) that identify the concept signatures in a parsing process in which the less abstract concepts are recognized first, and then they become features of larger-grained composite concepts.

In our opinion, from this previous work on automated program understanding, it is quite apparent that PAPs can in all respects be considered as ordinary clichés. More specifically they should be viewed as clichés at a specific abstraction level, that can be associated to one or more parallelization strategies (i.e. what we called parallel paradigms). Techniques like those reported in the papers just cited can therefore be integrated into a parallelization tool to implement the paradigm selection phase.

While details on how this tool could be actually organized will be discussed below, before closing this section we would point out a property of the parsing approach to program recognition that turns out to be very useful for our purposes. Given a program and a library of clichés describing functions and concepts at several abstraction levels, the parsing process can find all occurrences of the clichés in the program and build a hierarchical description of the program in terms of the clichés found and the relationships between them. The recognizer can therefore produce a hierarchical description of the program design with links from recognized concepts to their actual implementations within the code.

This kind of output can then be used during the code generation phase to selectively retrieve sequential code segments and integrate them into the template of the selected PP. More importantly, if code segments have to be transformed or partially substituted during the (parallel) code generation phase, thanks to the hierarchical description, code manipulation can be performed in a controlled way at different abstraction levels, and information about the functions implemented by any particular piece of code can be easily accessed.

5 Automated PAPs recognition and code generation

Figure 6 shows how can be structured a tool that automatically perform code parallelization according

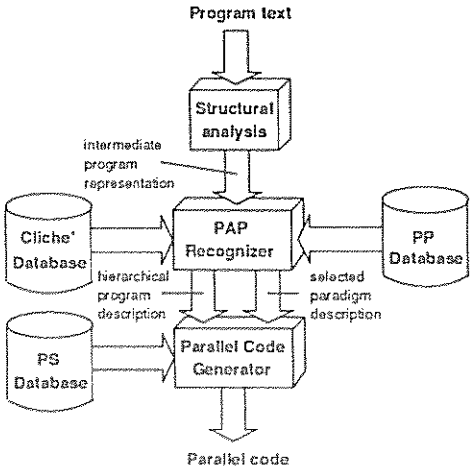


Figure 6: Parallelizer architecture.

to the paradigm-oriented approach outlined in previous sections.

The input program is first translated into a suitable intermediate representation. During this preliminary phase, control flow, data-flow and data-dependence analysis are performed and a number of normalizing transformations are applied to the program so that subsequent steps can be made independent of details such as the programming language, the specific control constructs used, etc.

The intermediate representation is then analyzed by a PAP recognizer implementing a parsing process that should not substantially differ from that described for instance in [9], even though structures other than plans and attributed flow graphs could be used to represent programs and clichés, respectively. This is represented in the figure by the access to a *cliché database*.

There is an important difference, however, between the PAP recognizer and a general-purpose recognizer such as that presented in [9]. In our case, when a cliché representing a parallelizable pattern is found, we can exploit information associated to the corresponding paradigm(s) to control subsequent steps of the recognition process. For instance, if we are analyzing the branch-and-bound program of figure 1b, after the iterative divide-and-conquer PAP of figure 5 has been identified, we know that two different PPs can be chosen to perform parallelization. We then try to find out if the split phase is cheap or expensive, or if some pruning condition is contained in the code (or both), in order to decide which paradigm is best suited for the concrete program given in input.

To exploit information associated to paradigms, a *PP database* has been added to the parallelization tool. Clichés corresponding to PAPs are marked and linked to the contents of the PP database. The latter contains a proper description of paradigms properties that could turn out to be useful during the recognition phase. For instance, PP descriptions could contain links to other PPs and to other clichés in the cliché

database representing, respectively, related (e.g. similar) PPs and algorithmic properties that might affect the choice and the application of a paradigm.

It is worth noting that the PP database behaves like a sort of control input with respect to the PAP recognizer. That reduces the complexity of the parsing process during the search towards identification of those clichés that represent meaningful patterns with respect to parallelization.

The second phase of the parallelization process to be automatized is parallel code generation. This is performed by a *code generator* that receives in input a description of the selected paradigm and the hierarchical description of the input program.

The code generator builds the output parallel program starting from a *parallel skeleton* (PS) of the selected paradigm. A parallel skeleton is a generic implementation of a PP that needs to be instantiated to a concrete parallel algorithm in order to become an executable program. Instantiation of PSs consists of extracting, from the original input, those code segments corresponding to parts of the skeleton that have been left generic.

As we have observed in section 3, in some cases this would require a certain degree of code transformation to integrate sequential code in a parallel, distributed memory framework. Once more, it is apparent the importance of exploiting paradigm information during the recognition process. Indeed, if some skeleton requires nontrivial transformations for sequential code integration, such as that reported in fig. 4, the corresponding information can be easily added to the paradigm description stored in the PP database. This information may help the recognizer to identify just those particular concepts that are needed to correctly perform the mentioned transformations.

As we have pointed out at the end of the preceding section, also the hierarchical description produced by the recognizer turns out to be particularly useful in the code generation phase.

6 Concluding remarks

In this paper we have proposed a novel approach to automated parallelization of sequential code, based on program comprehension. The approach tries to automatize a recently proposed parallelization methodology based on programming paradigms for building a parallel version of a given algorithm. We have discussed how this paradigm-oriented approach could be automatized and sketched the general structure of a tool based on these ideas.

While this paper chiefly deals with the paradigm selection phase, we are engaged in a research effort not restricted to this issue. In particular, we are working for characterizing the performance of parallel paradigms parametrically with respect to the underlying physical machine. This would provide, during the recognition phase, further criteria for paradigm selection taking into account performance. We are also exploring methods for implementing parallel skeletons with portable communication libraries, in order to make both the parallelizer and the generated code largely independent of a particular architecture.

Next step in our research will be the implementation of a working prototype that demonstrates the feasibility of the proposed approach. A first attempt in this direction has been the implementation in Prolog of a partial recognizer targeted to deal with several variations of the iterative divide-and-conquer algorithmic pattern. Even though this preliminary tool lacked any generality, it helped us to clarify the concepts of parallelizable algorithmic pattern and parallel paradigm, and provided several insight on the code generation phase.

References

[1] W.C. Athas and C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers", *IEEE Computer*, **21**, pp. 9–24, Aug. 1988.

[2] T.J. Biggerstaff, "The Concept Assignment Problem in Program Understanding", Procs. *IEEE Working Conf. on Reverse Engineering*, May 21-23, Baltimore, Maryland, USA, 1993.

[3] P. Brinch Hansen, "Model programs for computational science: a programming methodology for multicomputers", *Concurrency: Practice and Experience*, **5**(5), pp. 407–423, Aug. 1993.

[4] M.I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, Cambridge, MA, 1989.

[5] M. Danelutto et al., "High level language constructs for massively parallel computing", in *Computing and Information Science, IV*, Elsevier, October 1991.

[6] F. Darema et al., "A single-program-multiple-data computational model for EPEX/FORTRAN", *Parallel Computing*, **7**, pp. 11–24, 1988.

[7] G.C. Fox et al., *Solving Problems on Concurrent Processors*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[8] M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis", *IEEE Software*, Jan. 1990.

[9] L.M. Wills, "Automated Program Recognition: a Feasibility Demonstration", *Artificial Intelligence*, **45**, 1990.

[10] M.E. Wolf and M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", *IEEE Trans. on Parallel and distributed Systems*, **2**(4), pp. 452–471, Oct. 1991.

[11] Procs. *IEEE second Workshop on Program Comprehension*, July 8-9, Capri, Italy, 1993.

[12] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, NY, 1990.

Issues in Visualization for the Comprehension of Parallel Programs

Eileen Kraemer and John T. Stasko

College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280
E-mail: {stasko,eileen}@cc.gatech.edu

Abstract

Parallel and distributed computers are becoming more widely used. Thus, the comprehension of parallel programs is increasingly important. Understanding parallel programs is more challenging than understanding serial programs because of the issues of concurrency, scale, communications, shared resources, and shared state. In this article, we argue that the use of visualizations and animations of programs can be an invaluable asset to program comprehension. We present example problems and visualizations, showing how graphical displays can assist program understanding. We also describe the Animation Choreographer, *a tool that helps programmers better comprehend the temporal characteristics of their programs.*

1 Introduction

The comprehension of a program or its design is an important component of the coding, debugging, maintaining, testing, and reuse of software. Maintenance, in particular, is an expensive and difficult process. This difficulty is exacerbated by the fact that the maintainer is generally not the author, and that written documentation may be out of date, incomplete, or nonexistent.

A number of models, techniques, program representations, and tools have been developed to address the problems of program comprehension. In the top-down or problem-driven model[5], program comprehension is achieved through the formation, confirmation, rejection, and refinement of expectations of domain concepts. The user may search for confirming *beacons* [1] of these expectations. This method is thought to be common among expert programmers or if the code or type of code is familiar. In the bottom-up or code-driven model[23], comprehension is achieved through the detection of patterns or plans in the program code. This approach is common when the code is completely new to the programmer. In practice, comprehension may proceed top-down, bottom-up or a combination of the two.

All tools for program comprehension must perform three tasks: *extraction* or data collection, *abstraction* or analysis, and *presentation* or display of the result of the analysis. Various tools employ different methods for extraction, abstraction, and presentation, and place varying emphasis on the importance of each of these tasks. The result produced by these tools varies widely, from written documents, through intermediate forms passed on to CASE tools or code generators, to databases that may be browsed or queried, to static and/or dynamic visualizations of the program. Some of these tools produce predefined sets of reports or displays or furnish answers to fixed sets of queries, others allow the analyst to interactively select the analysis to be performed or the information to be displayed.

For example, TIBER(described in [22]) and Synchronized Refinement[21] focus on producing improved documentation. Charon[22] produces an intermediate result that can be passed on to a CASE tool. Ghinsu[15] allows the analyst to interactively explore the program, select target statements and variables, and perform analysis such as slicing, dicing, ripple analysis, and dependence analysis. DOCKET[12] creates a system model that may be interactively queried or browsed by the analyst.

A number of graphical representations of program information have been proposed. These graphical representations may be used internally to calculate results such as program slices or data flow information, or they may serve as the basis for displays that are presented to the analyst. See [7], [17], [8], and [9] for a more detailed discussion of program graph types. Use of these combined program representations allows a program comprehension tool to reference a single in-

[1]This research has been supported in part by a grant from the National Science Foundation (CCR9121607) and a graduate fellowship from the Intel Corporation.

0-8186-5647-6/94 $04.00 © 1994 IEEE

ternal representation, yet construct a variety of views, thus allowing the analyst to gain a better understanding of the program.

Systems such as CARE[14], VAPS[4], and VIFOR[20] visualize program dependencies. VAPS displays include a control flow graph, declaration and nesting trees, and a structure chart. VIFOR uses a two-column entity-relationship display. CARE uses a similar model, but presents a multi-column, *colonnade* display in addition to the traditional call-graph display. PUNDIT[16] combines statically collected semantic information with debugging capabilities, and provides both graphical and textual displays. Graphical displays include a dynamic call graph, an animated control flow graph, and data structure displays.

Other types of *software visualization*[26, 19] systems including BALSA[2], Zeus[3], and TANGO[24] focus on algorithm animation; they support the user in the design of arbitrary visualizations. The graphical displays described in preceding paragraphs, and those constructed using other visualization tools, can be much more intuitive and effective than textual representations, and thus can aid the analyst in assimilating the information produced by the program comprehension tool. These displays are most useful when they closely match the mental model[18] that the programmer/analyst forms through the process of program comprehension. The presentation of multiple views, and a facility that allows the viewer to hide, rearrange, or interact with objects in the display, all assist the analyst in achieving this match, and thus facilitate the comprehension of the program.

Thus, the role of visualization in program comprehension is that of *facilitator*. Visualization can help the analyst develop *intuition* about the functioning of the program under study. This intuition can help the "top-down" analyst form reasonable expectations about the program. The underlying program comprehension analysis tools can extract the information and perform some analysis or abstraction. Visualization can then present this information in such a way that the analyst will easily recognize its importance with regard to the expectation.

Similarly, visualization can help the "bottom-up" analyst proceed in an efficient manner. For example, by viewing an animated call graph of the program under study, the analyst can observe the order and frequency of execution of the various subroutines, and use this information to decide the order in which to study these pieces of code.

In the remainder of this paper, we will focus on a relatively new and extremely challenging program comprehension problem: the comprehension of *parallel and distributed* programs. We will discuss the role that visualization can play in facilitating their understanding. We will point out some of the issues that arise in the comprehension and visualization of parallel programs, and present our approach to addressing these issues.

2 Issues in the comprehension of parallel programs

The introduction of parallelism adds an additional twist to the model of program comprehension. Parallel programs are by nature large and complex. They often produce vast quantities of data. Programmers must understand and analyze large amounts of information describing complex relationships, including the states of each process and interactions among processes. Interactions include communication, synchronization, access to shared variables, and competition for shared resources. In addition to the control and data dependences, and control flow and data flow information that is essential to the understanding of a serial program, the user must address the added complexity of concurrency. The analyst now has additional questions to answer: Which pieces of code may execute concurrently? What type of synchronization must occur between these concurrent threads or processes? Are there any race conditions? Where and how is data shared? The analyst must decipher not only the portions of the code directly related to the purpose of the program, but also must wade through the code that creates and synchronizes processes, allocates tasks, makes intermediate results available to other processes, serializes access to shared memory, and determines some type of global state such as a termination condition.

Furthermore, performance is the primary motivating factor for the creation of parallel programs. Design decisions often hinge on obtaining the best possible performance on a particular architecture, rather than on producing a straightforward implementation of the underlying algorithm. Subsequent ports to other machines and additional "tweaking" (adjustment) can further obscure the original design.

We believe that visualization can assist the user in grasping the concurrency of the program, in managing the large number of objects, in understanding the interactions, and in analyzing the data describing the program's execution. We know that two-dimensional displays of information, such as bar charts and graphs,

give viewers insight into the data presented[28, 29]. Graphical animation can provide additional insight and allow the viewer to absorb more information by tapping into our well-developed visual abilities for detecting patterns, for tracking moving objects, and for spotting anomalies in patterns.

Visualization is a rich medium for communicating information about a program and its execution. That is, it allows program attributes to be represented by colors, shapes, sizes, locations, and motion, rather than merely as a text label and numerical data. When done well, visualization can improve understanding, and make obvious details that would have been obscure in a purely textual presentation. These "information-dense" displays can convey much more information than strictly necessary to confirm or reject a hypothesis. They can supply these answers *and* provide intuition about *why* the hypothesis is false or suggest additional refinement to the viewer.

Why is visualization superior to text for representing many aspects of parallel programs? Textual presentations are inherently serial. As stated earlier, displays are most useful when they closely match the viewer's mental model of the computation. A serial, textual presentation of program information is difficult enough to follow for the programmer who wrote the code and who thus should have a fairly well-established mental model of the concurrency in the program, and who has only to map the text back to this model. The reverse process, required of the analyst who must comprehend unfamiliar code, the *construction* of a mental model of this concurrent process from purely textual, serial data is surely more difficult. An animated, graphical display can more easily convey the concurrency of the program, and can more naturally deal with the temporal issues of parallel programs.

For example, suppose we are dealing with a program containing barrier synchronizations. The "master" process must check in to the barrier before any "slave" process may proceed past its "barrier check-in" statement. All "slave" processes must check out of the barrier before the "master" process may proceed past its "barrier check-out" statement. In the visualization shown in Figure 1, a new two row grid is displayed each time a barrier synchronization is invoked. As each participating process checks in to the barrier, the appropriate circle in the top row is filled in. If the process must wait, its color fades to indicate that it is inactive. When the master checks in, the processes are shown as active again. Similarly, as each participating process checks out of the barrier, the appropriate circle in the second row is filled in. If the master has to wait, it is shown going inactive. When the final slave process checks in, the master process is shown as active again.

Suppose, in our example, that the analyst knows that a barrier synchronization is in use. Do all processes participate in the barrier? If not all, then which ones? Which processes have to wait? Which processes keep the others waiting? An analyst might have to wade through a good deal of text or make a number of queries to answer this question. However, a quick glance at a display such as that shown in Figure 1 can answer these questions.

Similarly, an analyst attempting to understand the pattern of access to a shared variable could easily observe this from the display in Figure 2. In this display, the large circle represents a "mutex" - a critical section of code protected by a mutual exclusion variable. When a process obtains control of the mutex variable, its icon(a colored circle) is shown entering the circle. Processes that are waiting to gain access to that mutex are shown waiting outside the circle. Is there contention for this variable? That is, are there many icons waiting around the circle? How much time does a process spend inside the mutex relative to the time it spent waiting? This type of display would be useful in understanding design decisions such as the use of a distributed list rather than a centralized list. The analyst who must port this code to a new architecture can then make a more informed decision about whether to keep the distributed list or go to a centralized list in the new architecture.

Of course, this same information could be provided without visualization by an "ideal" program comprehension tool - a tool that could answer questions like "Why is there a barrier synchronization at this point?" or "If I add a routine that accesses data structure X, do I now need to enforce mutual exclusion on X?" In such a world, visualization would not be necessary. However, until the time that program comprehension tools have advanced to this state, visualization can serve the useful purpose of providing a rich medium for conveying information - information regarding data and control dependences and flow as in the serial world, and information regarding concurrency, distributed state, and shared variables in the parallel world.

Animated displays, in particular, are useful for conveying information regarding concurrency. They employ the very natural mapping of time to time, rather than the less natural time to space mapping, or the more obscure time to 12-digit timestamp value of a textual report. Events that were concurrent in the

program can be shown as concurrent in the display. In fact, events that *might have been* concurrent can be shown concurrently in the display. This leads us to a discussion of the importance of time and event order in the visualization of parallel programs.

3 Time and event order in the visualization of parallel programs

A number of systems providing visualizations of concurrent programs have been developed[10]. However, not all visualization systems are designed to deal with concurrency. Many follow a serial paradigm in which the visualization system receives some data from the executing program, animates the display to represent that event, and then processes another event (piece of data). Unfortunately, we then lose the concurrency inherent in the program.

The POLKA[25] animation system that we use to develop visualizations can support concurrent animations actions, however. POLKA allows designers to create graphical objects such as lines, text, circles, rectangles, etc., and then make the objects move, resize, change color, flash, and so on. A particular object can be performing many different actions at once, or multiple objects can be changing at the same time, thus reflecting the concurrency of a parallel program. POLKA is implemented on top of the X Window System and Motif.

Concurrent programs often consist of logical phases or rounds. The execution of these rounds or phases may be skewed in time across processors with different workloads or speeds. If valid timestamps are available, the user may wish to view the computation with the events ordered as they actually occurred. However, the calculation of a complete ordering on program events may not be possible - clocks may run at different rates, may have insufficient resolution, or may not be synchronized across processors. Techniques exist to minimize this problem, but they do not eliminate it and they often incur substantial overhead. Synchronization events - message sends and receives, barrier synchronization, serialized access to shared variables - can be used to calculate a partial ordering of events. Within the constraints of partial ordering, a number of feasible orderings exist. Visualizations that adhere to different feasible orderings can give the viewer different perspectives on the computation.

Several authors[27], [13], and [6], have emphasized the value of displaying alternate orderings of a program's execution. To truly comprehend a parallel program, the analyst must understand what these various orderings are and how they can affect the program under study. Of course, the many combinations of possible event orders makes it unmanageable to generate and display *every* feasible ordering. Instead, we utilize an *Animation Choreographer*[11] that provides several useful, canonical orderings, based on the synchronization events produced by the program under study. In addition, the Choreographer allows the user to interactively adjust these to produce any additional orderings.

POLKA and the Animation Choreographer are part of the PARADE (PARallel Animation Development Environment) system for the visualization of concurrent programs. A third component is an instrumentation or monitoring tool. The use of the instrumentation or monitoring tool, which varies between architectures and languages, helps identify the event records for a program.

Using POLKA, libraries of visualizations have been developed - synchronization, history, and callgraph views for Pthreads programs on the KSR(as shown in Figures 1 and 2), 3-D visualizations of communication on the MasPar, algorithmic and performance views of branch and bound algorithms in the iPSC hypercube, as well as a number of application-specific visualizations. Using PARADE, programmers may select visualizations from libraries such as these, or they may create their own new visualizations.

The Animation Choreographer allows the analyst to view program visualizations under a variety of orderings. As an example we will discuss a program that performs a parallel quicksort on an array of numeric values. The parallel execution follows a fork-join paradigm, implementing a tree-structured algorithm. That is, the computation begins with a single processor that reads in the data values to be sorted, and then makes a pass over the data in which it determines a median value and places all elements with values less than or equal to the median value on one side of the array, and all values greater than the median value on the other side of the array. It then forks off two child processes and waits for them to finish their work and join back to it, the parent process. Each of the child processes does the same thing to its portion of the array and forks off child processes of its own. This continues recursively.

The analyst attempting to comprehend the program might want to look at some of the same displays that are helpful with serial programs such as dependence graphs and animated call graphs. In addition, he might like to see application-specific or domain-

specific (in this case, we'll consider sorting as the domain) displays such as those shown in figures 3, 4 and 9 through 11.

Figures 3 and 4 are an animated view of the array. Each bar represents an element in the array (it could also represent a group of elements). The height of the bar indicates the value of the element, and the horizontal position indicates its index in the array. Color is used to show the processor that last touched that element. This use of color allows the viewer to easily detect and follow the actions of a particular processor across multiple displays. At the start of the visualization, the bars are arranged to represent the initial, unsorted array. As elements are swapped in the algorithm the bars are shown changing places in the display. From this display the viewer can observe the order in which elements are swapped, and see which processors act on each portion of the array, giving clues to the functioning of the underlying algorithm.

Figures 9 through 11 are animated swap histories of the parallel quicksort program. Again, horizontal position is used to indicate array index and color is used to indicate the processor performing the swap. Each time two elements are swapped, a horizontal line is drawn between them, in the color associated with the processor that performed the swap. Time runs upward on the display. Thus, in a sequence of swaps, the most recent swap is on top. The triangular patterns in the display indicate that the algorithm works by comparing the processor's first and last elements in its portion of the array, and then working in toward the middle of the subarray. The analyst can also determine the number of processors active at each stage (by counting the number of triangles in a row), and the depth to which the algorithm recurses in this execution (by counting the number of triangles in a column).

The order in which these swap events are displayed can greatly affect the appearance of the displays and the information that can be gained from them. The Animation Choreographer provides four orderings: *Timestamp*, *Adjusted Timestamp*, *Serialized*, and *Maximum_Concurrency*. The Choreographer reads in event records (in this example, each swap of array elements has an associated event record). It then displays an *execution graph*, an acyclic, directed graph in which the nodes represent the recorded program events, and the arcs indicate the temporal precedence relations between these events. The events produced by a particular process or thread are displayed in a column. Arcs between these nodes indicate the sequential relationship between the events of a single process. Arcs between columns are the result of synchronization events such as forks, joins, or barrier synchronizations.

Processors (or threads, etc.) are arranged from left to right. Vertical position in the graph represents execution time, with earlier times appearing above later times. Shape and color of node objects can be used to identify different event types. The execution graph reflects the program events as they were recorded. The user can examine the recorded events by scrolling through the graph, and by clicking on nodes of interest.

To begin viewing a program trace, the user selects an ordering. An initial ordering choice might be a *timestamp* ordering - the execution times are used to order the events for visualization. Figure 8 shows the appearance of the choreographer using events from parallel quicksort and a timestamp ordering. This ordering is frequently very useful to a user wishing to see the actual order of execution, when such information is available. This method relies on the existence of a global clock with adequate resolution, and will produce an essentially sequential visualization under these circumstances. Poor resolution, or timestamps that are not valid across processors, however, may produce visualizations that are misleading or incorrect.

Figure 4 shows the final appearance of the array view when timestamps are used to determine the order of events. As you can see, this does not appear to be a sorted array. In fact, there seem to be "holes" in the array. An examination of event records reveals that there are duplicate timestamps - the clock used was not of adequate resolution. The overlapping event symbols in the Choreographer display of figure 5 are a result of these duplicate timestamps. In a timestamp ordered visualization, all events with the same timestamp are animated concurrently. In this case, swaps that occurred sequentially are animated concurrently because they received the same timestamp. If a given element is involved in multiple swaps with the same timestamp, its final location is not correct, and it may seem to "disappear," resulting in a misleading visualization. Similarly, figure 9 shows the final appearance of the swap history view using a timestamp ordering. There appear to be very few swaps. Again, this is a misleading display resulting from inadequate clock resolution.

Within timestamp ordering we have several choices for scaling. We can use a 1:1 mapping from timestamp units to animation frames. However, this can result in an animation with long periods of inactivity, punctuated by short bursts of activity too rapid for the viewer to comprehend. Another option is to use an n:1 map-

ping from timestamp units to animation frames. This shortens the spans in which nothing happens, but intensifies the short bursts of activity. A third option is to use the timestamps to order the events for visualization, but to ignore them in determining the interevent waiting time. This eliminates the long waits, and allows the event activity to be visualized at a rate the viewer can understand. However, in this type of scaling we lose information about the relative timing of the program events. Ideally, it would be desirable for the mapping from execution time to animation time to behave like a "fun-house mirror." That is, we would like to compress long inter-event times, and stretch out periods of high activity, allowing viewers to discriminate between individual events, but preserving a perspective on the actual timing of events.

The choreographer display under an *adjusted timestamp ordering*, shown in Figure 7, represents a compromise on these goals. In this ordering the timestamps are adjusted just enough so that causal ordering is maintained (the displays are "correct"), but long interevent times are unaffected. This ordering is useful in obtaining a valid visualization without losing the perspective on the true sporadic nature of the program's execution behavior. The use of this ordering results in a "correct" final appearance of the array display, in which the elements are truly sorted, and a "correct" swap history view, in which no element is swapped more than once in any time period.

Figure 6 shows the choreographer under a *serial ordering*. For a serial ordering, we construct a complete ordering of events consistent with the partial order determined by the dependence relations. This method can produce valid, comprehensible visualizations in the absence of globally synchronized timestamps with adequate resolution, such as we have in this example. The array view is again "correct". The swap history view appears as in figure 10. If you look closely you will see that no two swap lines have the same vertical position; the visualization has been completely serialized. We have gained a "correct" ordering, but we have lost the concurrency in the display. In addition, we have eliminated the long interevent times.

Finally, we may wish to use a *maximum concurrency* ordering, as illustrated by the Choreographer display of Figure 8. (For this particular set of events and times, the maximum concurrency and adjusted timestamp orderings are nearly identical - this is not always the case.) In this ordering, we gather all events that *could* have occurred together, and animate them simultaneously. Essentially, this view shows the maximum concurrency possible given the partial order defined by the synchronization events. This ordering also produces a correct array view. The final appearance of the swap history view is shown in figure 11. It is correct, but you will notice that unlike the serial ordering of figure 10 there are multiple swaps on the same line. That is, they are visualized here and in the array view as concurrent actions. We get correctness and concurrency without the long interevent times. In other visualizations we have found this ordering type to be useful for identifying bugs by illuminating concurrent situations that were not imagined by the program's designer.

These various temporal perspectives can provide the user with insight into the program's execution, with each different ordering of the animation shedding light on a different aspect of the computation. The Animation Choreographer allows users to view program animations under the orderings described above, and to specify variations on these orderings.

4 Conclusions

Parallel and distributed programs present a number of challenges to program comprehension. The added complexity introduced by the multiple threads of control, the interactions between processes, the peculiarities of the various parallel programming paradigms and libraries and the tendency to optimize code for a particular architecture exacerbate the problems of program comprehension.

Visualization systems allow program attributes to be represented by colors, shapes, sizes, locations, and motion, rather than merely a text label and numerical data. When done well, visualization can improve understanding, and make obvious details that would have been obscure in a purely textual presentation. We believe that computer visualization, the graphical animation of the behavior and performance of computers and computer programs, can be an effective tool for understanding how programs work.

POLKA is an animation toolkit designed to support concurrent animations. The Animation Choreographer, which relies on POLKA, allows the viewer to easily explore the set of alternate feasible orderings. We believe that this ability to explore the set of possible event orders is essential to the comprehension of parallel programs. We do not claim that visualization will solve all the problems associated with the complexity of parallel programs. However, we do believe that visualization provides a rich means of communicating that can lead to better understanding of how large systems work.

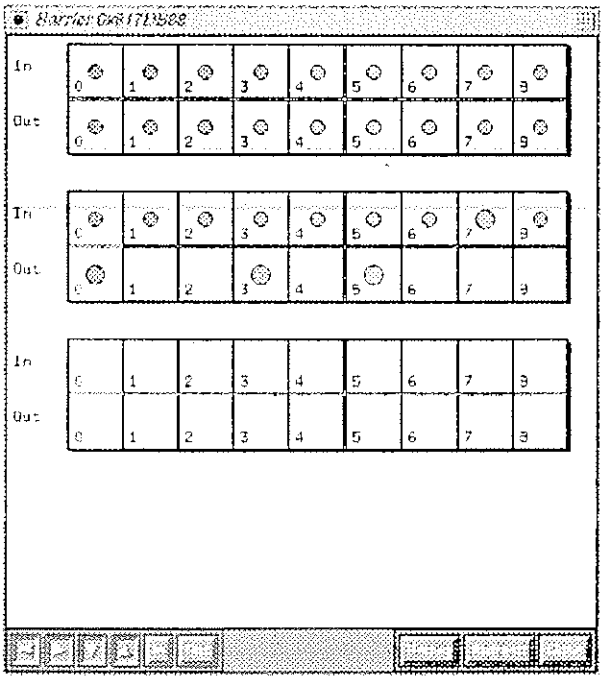


Figure 1: A snapshot of the animated Gthreads barrier display, created by Alex Zhao using POLKA.

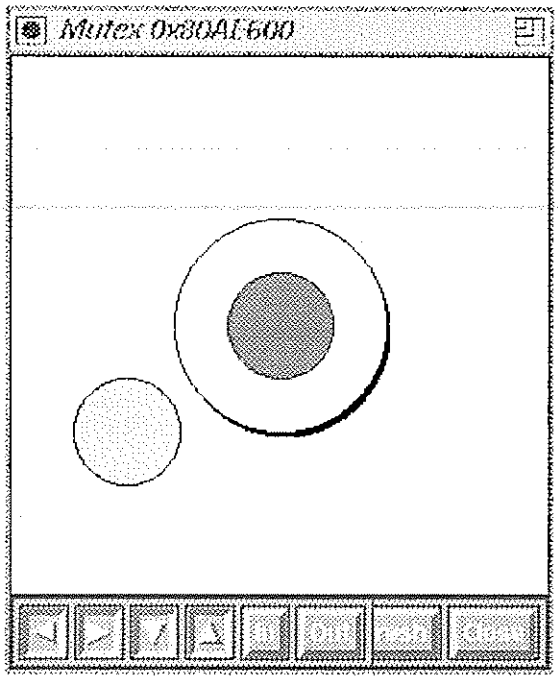


Figure 2: A snapshot of the animated Gthreads mutex display, created by Alex Zhao using POLKA.

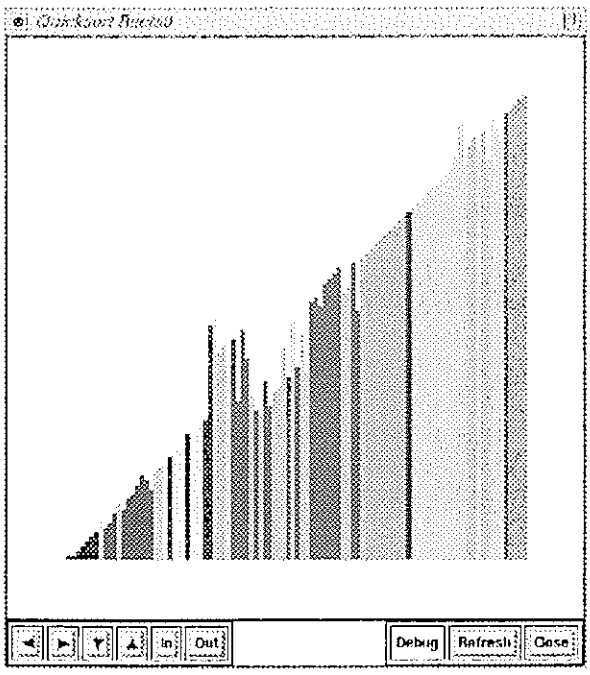


Figure 3: A snapshot of the array view of the parallel quicksort program under a correct ordering.

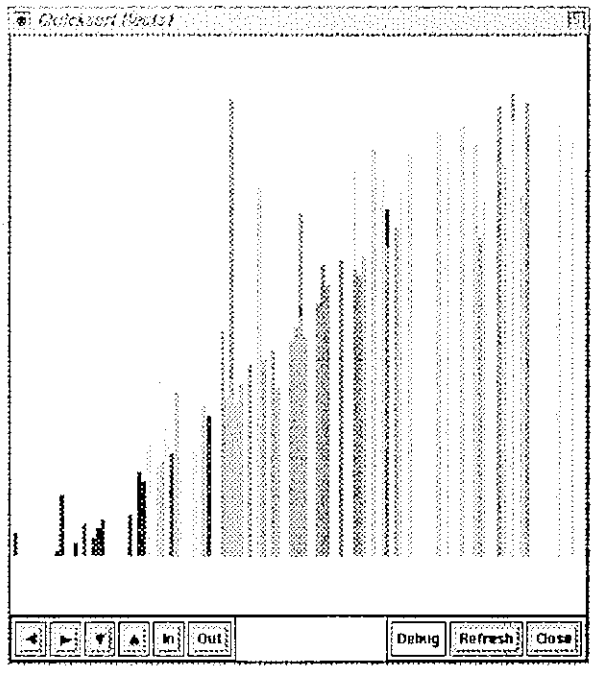


Figure 4: A snapshot of the array view of the parallel quicksort program under the timestamp ordering.

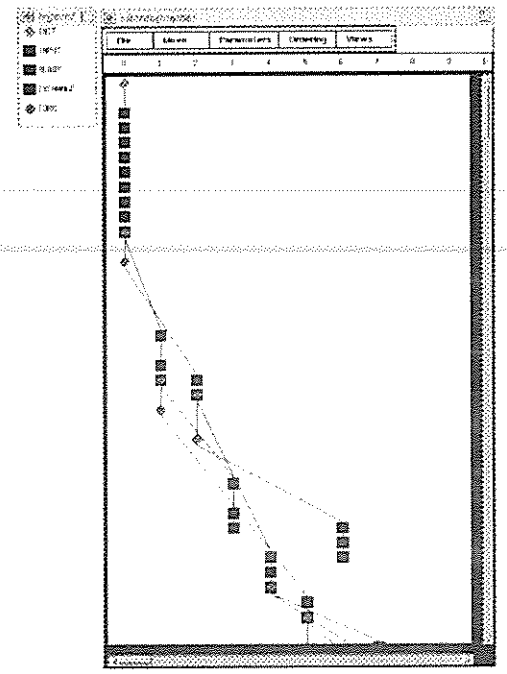


Figure 5: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is timestamp.

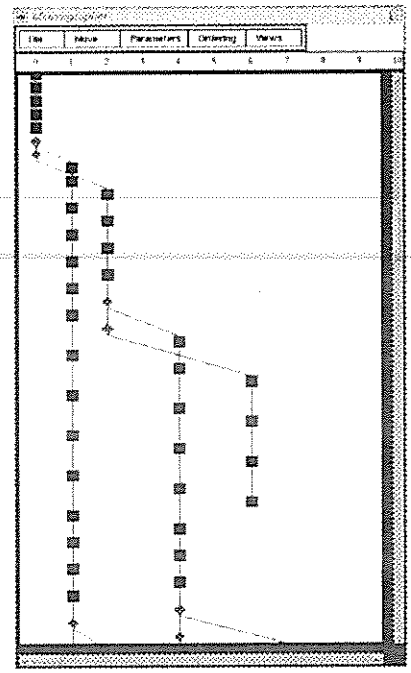


Figure 6: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is serialized.

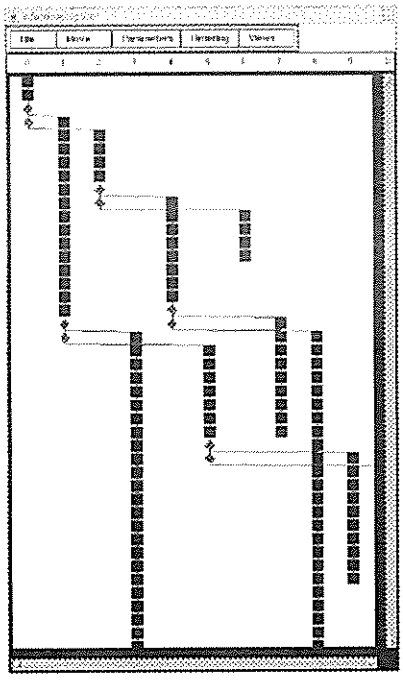


Figure 7: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is adjusted timestamp.

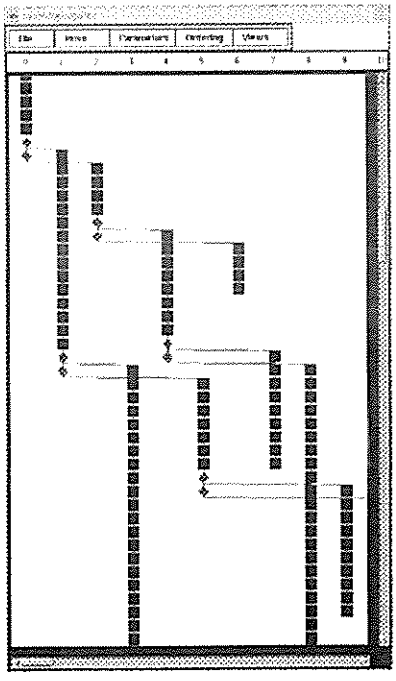


Figure 8: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is maximum concurrency.

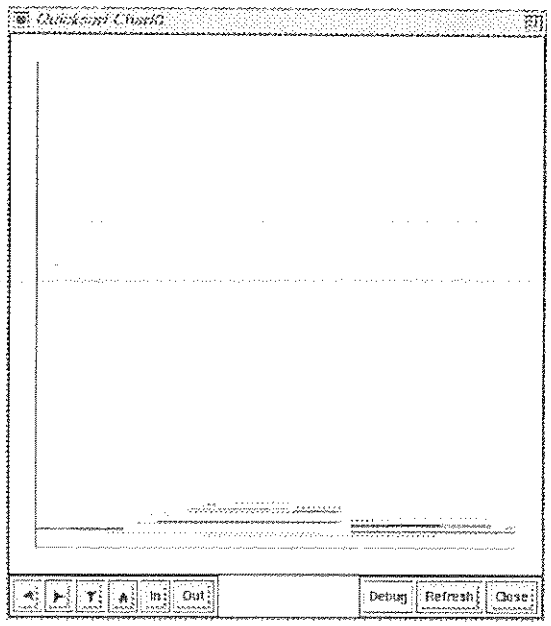


Figure 9: A snapshot of the swap history view of the parallel quicksort program under the timestamp ordering.

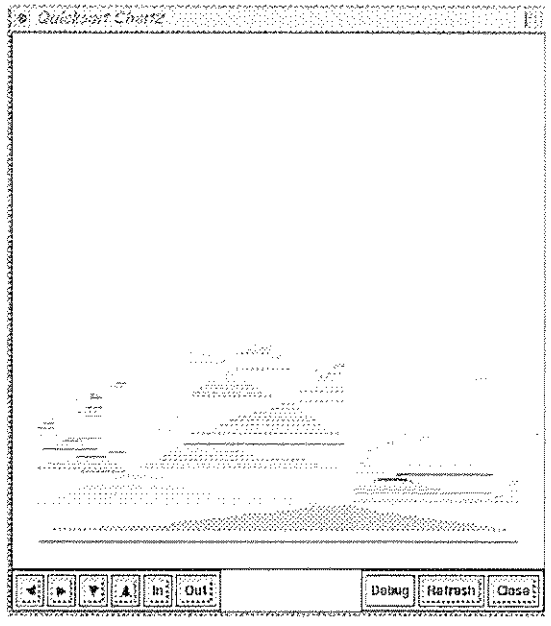


Figure 10: A snapshot of the swap history view of the parallel quicksort program under the serialized ordering.

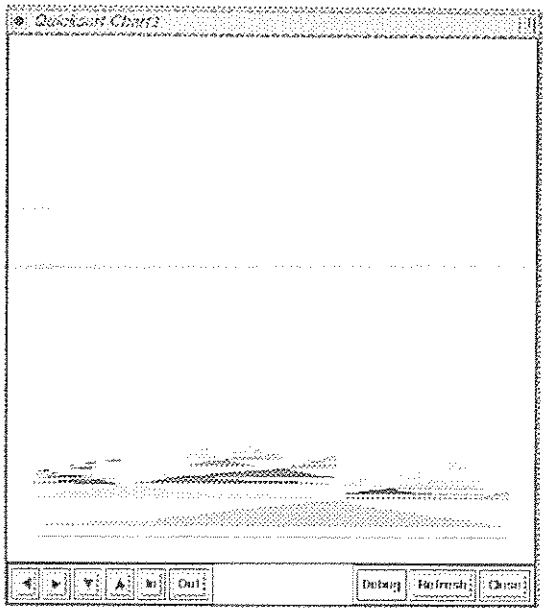


Figure 11: A snapshot of the swap history view of the parallel quicksort program under the maximum concurrency ordering.

References

[1] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[2] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[3] Marc H. Brown. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the IEEE 1991 Workshop on Visual Languages*, pages 4–9, Kobe Japan, October 1991.

[4] G. Canfora, A. Cimitile, and U. DeCarlini. Vaps: Visual aids for pascal software comprehension. In *Proceedings of the Program Comprehension Workshop*, pages 13–15, 1992.

[5] T. A. Corbi. Program understanding : Challenge for the 1990's. *IBM Systems Journal*, 28(2), February 1989.

[6] Janice E. Cuny, Alfred A. Hough, and Joydip Kundu. Logical time in visualizations produced by parallel programs. In *Visualization '92*, Boston, MA, October 1992.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its

use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.

[8] M. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. In *Proceedings of the Conference on Software Maintenance*, pages 138–147, Sorrento, Italy, October 1991.

[9] David Kinloch and Malcolm Munro. A combined representation for the maintenance of c programs. In *Proceedings of the Program Comprehension Workshop*, pages 119–127, 1993.

[10] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, June 1993.

[11] Eileen Kraemer and John T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations. *Proceedings Eighth International Parallel Processing Symposium*, pages 902–908, 1994.

[12] P.J. Layzell, R. Champion, and M.J. Freeman. Docket: Program comprehension-in-the-large. In *Proceedings of the Program Comprehension Workshop*, pages 140–148, 1993.

[13] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program execution using multiple views. *Journal of Parallel and Distributed Computing*, 9(2):203–217, June 1990.

[14] Panagiotos Linos, Philippe Aubet, Laurent Dumas, Yan Helleboid, Patricia Lejeune, and Philippe Tulula. Facilitating the comprehension of c programs: An experimental study. In *Proceedings of the Program Comprehension Workshop*, pages 55–63, 1993.

[15] Panos E. Livadas and Scott D. Alden. A toolset for program understanding. In *Proceedings of the Program Comprehension Workshop*, pages 110–118, 1993.

[16] David P. Olshefski. Position paper: Tools facilitating software comprehension. In *Proceedings of the Program Comprehension Workshop*, pages 32–34, 1992.

[17] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Notices*, 9:177–184, May 1984.

[18] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Proceedings of Supercomputing '89*, pages 627–636, Reno, NV, November 1989.

[19] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

[20] V. Rajlich, N. Damaskinos, P. Linos, and W. Khorsid. Vifor: A tool for software maintenance. *Software - Practice and Experience*, pages 67–77, January 1990.

[21] Spencer Rugaber. Reverse engineering by simultaneous program analysis and domain synthesis. In *Proceedings of the Program Comprehension Workshop*, pages 45–47, 1992.

[22] Oreste Signore and Mario Loffredo. Charon: a tool for code redocumentation and re-engineering. In *Proceedings of the Program Comprehension Workshop*, pages 169–175, 1993.

[23] E. Soloway and K. Ehrlich. Empirical studies of programming knowledgekk. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[24] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[25] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[26] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the IEEE 1992 Workshop on Visual Languages*, pages 3–10, Seattle, WA, September 1992.

[27] Janice M. Stone. A graphical representation of concurrent processes. *SIGPLAN Notices*, 24(1):226–235, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[28] E. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1983.

[29] E. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990.

Session F:
Database

Using Procedural Patterns in Abstracting Relational Schemata

Oreste Signore - Mario Loffredo - Mauro Gregori - Marco Cima

CNUCE - Institute of CNR - via S. Maria, 36 - 56126 Pisa (Italy)
Phone: +39 (50) 593201 - FAX: +39 (50) 904052 E.mail: oreste@vm.cnuce.cnr.it

Abstract

The rebuilding of the database conceptual schema from the physical database structure is a fundamental issue in the re-engineering and design recovery processes. In this paper we present an approach to the reverse engineering based on the identification of schema, primary key, SQL and procedural indicators that lead to the assertion of Prolog facts and, by some heuristic rules, to the rebuilding of the conceptual schema.

1. Introduction

In the last years, RDBMS have significantly increased their capabilities of handling constraints at the schema level. This means that a large part of the existing code can be "cleaned" from the statements that implement the constraints.

On the other hand, the mapping from the database conceptual schema to the physical structures is a sequence of transformations that induce a progressive degradation of the schema, that becomes less complete, simple, readable and expressive. Every database design methodology must face the problem of the difference between the semantic richness of the ER (or extended ER) model and the simplicity of the target relational model. In [9] the reader can find an analysis of the problems met in the mapping phase and a set of mapping rules. The application of the mapping rules and of the different semantic richness between the source and target models has the obvious consequence that many constraints must be implemented in a procedural way. Therefore it could be very difficult to capture the semantics of the conceptual schema by simply looking at the physical one.

2. Related work

In spite of his relevance in the RE area, very few research efforts have been put in the Database Reverse Engineering (DBRE) activities. At first glance, it seems that even commercial products are able to solve the problem, but we must stress that almost all existing approaches are conditioned by a set of restrictive hypotheses, namely:

- all the conceptual requirements have been translated into data structures and constraints;
- the mapping from the conceptual to the logical and physical schema has been done by strictly applying the mapping rules, without any "trick";
- additional requirements dictated by the user or the host environment did not give raise to further restructuring of the schema;
- the company defined some "naming policy" for the elements of the schema.

However, we must think that a satisfactory Relational DBRE (RDBRE) should consider several implementation solutions, both foreseeable and uncommon. Furthermore, we must be able to analyse schemata implemented in less recent DBMS environment, that show some well-known DDL limitations to the possibility of explicitly expressing the concepts of primary and foreign keys, referential integrity constraints, domain constraints, etc.

Batini, Ceri and Navathe ([1]) proposed an extremely simple and limited RDBRE process that sketched a series of steps for the analysis of the relations and the identification of the concepts a reverse engineer having a good semantic knowledge of the source relational schema should follow. It can represent a good starting point.

Another approach, described in [7], faced the problem of DBRE from a more experimental point of view by offering an enhanced set of methods, techniques and practical examples.

In [5] a short survey on the DBRE and data-oriented applications state of the art through is exposed. It synthesized some issues related to the difficulties met during a DBRE process that is independent from the adopted data model. According to the authors, the DBRE problem can be expressed as the research of a potential conceptual schema able to lead to the physical organisation constituted by the DDL, the host language data structures and the procedural specifications. Hence the process solving such a problem has two generally

0-8186-5647-6/94 $04.00

sequential phases. The first one is the *Data Structure Extraction* (DSE) that reverses the physical design phase by reconstructing the data structures from their DDL and host language representations. The subsequent phase is the *Data Structure Conceptualisation* (DSC), that reverses the logical design phase by the identification of a possible conceptual schema starting from the DSE output information.

In [2] the authors presented a methodology for extracting an extended ER model from a relational database. Their methodology analysed not only the data scheme, but also data instances. However we must note that the results of this technique not only are not completely decidible[1], but also are limited by the amount and the quality of the data stored in the database.

Moreover we have to remind that very often the knowledge is embedded in the code, without any up to date and reliable documentation, and none of these approaches take into account this peculiar aspect.

3. The methodology

The reverse engineering process is performed essentially in three phases: the identification of primary keys, the detection of the indicators, and finally the conceptualisation.

In the first phase we move from the facts represented in the initial knowledge base, including information taken from the catalog and the SQL procedural patterns used in the manipulation of the database (i.e. `FETCH` loops). In this phase, we automatically detect, if possible, the *primary keys* of the relations. Otherwise, we look for the usage of some subsets of the relations' attributes, detecting some *identification indicators*, that can help the user in the manual identification of the primary keys. We assume that all the information used in this phase is represented by a set of assertions, originating from the processing of both the catalog and the output of a static code analyser[2].

In the second phase, named *indicators' detection*, we search for some pre-defined indicators, i.e. a set of information detectable from one or more available sources (catalog, SQL code, output of a previous analysis phase), than can be significant to characterise, in the conceptual model, one or more relational schema items.

In the third and last phase, the *conceptualisation*, the user can formulate hypotheses on the conceptual meaning of the elements of the relational schema, evaluate them together with the conceptually not identified relations, and decide to restart some RE process phases, eventually making choices that are alternative to those previously taken.

4. Primary and candidate keys

The detection of a primary key is a trivial process if the RDBMS supports its explicit declaration; otherwise, if there is only one `UNIQUE` index defined, it is a reasonable assumption to take as primary key the set of attributes it is defined upon. If there are more than one `UNIQUE` indexes, we can take every set of attributes as *candidate* key, calculate their frequency of usage, and ask the user to choose.

If we don't succeed in the identification of a primary or candidate key, the analysis of some programming patterns can lead to the detection of some identification indicators that can help us in fulfilling the task.

For each table T, having neither an explicit primary or candidate key, nor any `UNIQUE` index, we consider the set P of the `NOT NULL` attributes, having `INTEGER`, `SMALLINT`, `CHARACTER`, `DATE`, `TIME` or `VARCHAR` type. For each subset $S=\{a_1, a_2,..., a_S\}\subseteq P$ of the attributes, we perform the control of some identification indicators. The underlying hypotheses are:

- at least one `WHERE` clause of a SQL statement must mention all them;
- the structure of the code must exclude that the selection returns a set of tuples (loop of fetches, aggregation operators, `ORDER_BY` or `GROUP_BY` clauses, etc.).

In table 1, we report the various patterns[3], their meaning, the corresponding assertions. In more detail, we can see that the condition (a) is based on the assumption that a candidate key should be referred by at least one `WHERE` clause in a SQL statement. However this assumption could be not true in some special cases, for example when a timestamp has been chosen as primary key. All the other conditions (b-g) occur when the programmer expects to select a set of tuples, so excluding the possibility that the attributes' set could be a primary key.

Every subset $S=\{a_1, a_2,..., a_S\}\subseteq P$ satisfying all the conditions, can be proposed as a possible key. It must be noted that the previous criteria cannot be considered by themselves sufficient to identify the primary key, while they can need some further refinements. Moreover, they are, in certain sense, depending on the quality of the software.

---

1 For example, the attribute A cannot be a candidate key for the relation T if we find two tuples having the same value for A. Otherwise, we cannot assert anything.

2 We are well aware of the limitations of a purely static analysis, of course.

3 The term `commalist` simply identifies a list of elements of the specified type, separated by a comma.

	Pattern
a	WHERE a_1=<scalar_exp$_1$> AND..AND a_S=<scalar_exp$_S$>
b	No declaration of a cursor like: DECLARE <cursor_id> FOR SELECT <selection> FROM T WHERE a_1=<scalar_exp$_1$> AND..AND a_S=<scalar_exp$_S$> followed by OPEN <cursor_id> and a loop containing: FETCH <cursor_id> INTO <list_of_host_var> or: No assignment of the selected tuples to an array.
c	No statement contains: SELECT ALL|DISTINCT <selection> FROM T WHERE a_1=<scalar_exp$_1$> AND..AND a_S=<scalar_exp$_S$>
d	No statement contains: SELECT <function-ref> FROM T WHERE a_1=<scalar_exp$_1$> AND..AND a_S=<scalar_exp$_S$> where function-ref::= COUNT(*) | distinct-function-ref | all-function-ref distinct-function-ref::= {AVG|MAX|MIN|SUM|COUNT}(DISTINCT column-ref) all-function-ref::= {AVG|MAX|MIN|SUM|COUNT}([ALL]scalar-exp)
e	No statement contains: SELECT <selection> FROM T WHERE a_1=<scalar_exp$_1$> AND..AND a_S=<scalar_exp$_S$> GROUP BY <column-ref-commalist> or SELECT <selection> FROM T WHERE a_1=<scalar_exp$_1$> AND..AND a_S=<scalar_exp$_S$> ORDER BY <ordering-ref-commalist>
f	No statement contains: SELECT <selection> FROM T GROUP BY a_1, a_2,..., a_S
g	No statement contains: WHERE <scalar-exp> [NOT] IN <subquery> or WHERE <scalar-exp><comparison> ALL|ANY|SOME <subquery> where <subquery> is like SELECT <selection> FROM T WHERE a_1=<scalar_exp$_1$> AND..AND a_S=<scalar_exp$_S$>

Table 1 - SQL patterns and corresponding assertions

Therefore, we think that it is necessary a user intervention to get the final decision by observing the clues detected in the previous phase.

Consequently, we show to the user the attributes' sets that satisfy all the (a)-(g) conditions, the sets of attributes that satisfy only some of them, the frequencies of usage. Finally, the user, making use of the information supplied and of his/her semantic knowledge of the application domain, can decide to identify a subset S as a certain primary key, or he/she can formulate a hypothesis on the primary key of T.

Anyway, at the end of this phase, for every relation we must have a primary key or we must formulate a hypothesis for the primary key and indicate the existence of candidate keys.

5. Detection of the indicators

In the first phase, we identified the keys by integrating and comparing some clues about sets of attributes, that derive directly from the constraint that a key is mandatory and unique. In the second phase, the reverse process must face difficulties arising from the different semantic richness between the ER and the relational model. Firstly, the mapping from an ER to a relational model is not unique, secondly, some factors can have affected the relational schema, namely optimisation choices, poorness of the DDL, unusual implementation techniques. Therefore, the process cannot rely on one-to-one correspondences between the ER concepts and the relational schemata's elements, as it was in the preceding case, when we made use of the equivalence between the entity identifier and the primary key. We must adopt a

"clued" approach, instead, while in the subsequent conceptualisation phase we will identify probable concepts based on suitable combination of indicators.

5.1. Domain identity

The first step is the identification of homonyms (attributes belonging to different relations, identified by the same name but defined on different domains) and synonyms (attributes having different names, but defined on the same domain). We can easily solve all the cases of homonyms just adopting the *extended name* of the attributes (`tablename.attributename`), while the detection of synonyms requires some additional processing. Indeed, SQL is a language having simple data types and weak type checking, and therefore the type checking based on the content of the catalog cannot provide reliable information. The underlying idea is that at the conceptual level the definition of the types is richer than at the relational level ([3]). Would the designer and the programmer take into account the constraints defined at the conceptual level, but non directly supported by the DBMS, we should find an evidence in the data manipulation procedures. Therefore, we detect the synonyms by some SQL indicators taken from the analysis of SQL statements (table 2)[4].

As an example of a nested query, and of the ambiguities coming from a simple comparison of the columns' names, consider the following relations:

```
BOOKS (ID, TITLE, AUT, PUBLISHER,...)
AUTHORS (ID, NAME,...)
```

and the query asking for the names of the Authors published by the publisher XYZ:

```
SELECT  NAME
FROM    AUTHORS, BOOKS
WHERE   AUTHORS.ID = BOOKS.AUT AND
BOOKS.PUBLISHER = 'XYZ'
```

or:

```
SELECT NAME
FROM   AUTHORS
WHERE  ID  IN
       (SELECT    AUT
          FROM    BOOKS
          WHERE   PUBLISHER = 'XYZ')
```

It is evident that `AUTHORS.ID` and `BOOKS.AUT` are defined on the same domain, while `AUTHORS.ID` and `BOOKS.ID` aren't, even if they can have the same SQL type (e.g. `CHAR(5)`).

The identification of the join of a relation with itself is syntactically equivalent to the join between two different relations, but, when it involves all the components of the key, can be semantically highly significant, as the first step towards the detection of unary or recursive relationships.

Other DML statements, as `INSERT`, `DELETE`, `UPDATE`, can help in the identification of synonymies. For example, an instruction like:

```
INSERT INTO <table> (<column-commalist>)
SELECT <selection-commalist>
<table-exp>.
```

clearly lead to the identification as synonyms of the attributes that has the same place in the two lists <column-commalist> and <selection-commalist>.

It must be noted that in some cases the semantic equivalence of the domains could be deduced from the usage of the host variables, as it happens when a join is implemented by separate `SELECT` on different tables, e.g.:

Type	Pattern
equijoin	SELECT ... FROM T_1, T_2 WHERE ...T_1.ATTR = T_2.ATTR'...
multiple join	SELECT ... FROM T_1, T_2, T_3,... WHERE ...$T_1.ATTR^{(1)} = T_2.ATTR_1^{(2)}$ AND $T_2.ATTR_2^{(2)} = T_3.ATTR^{(3)}$...
nested queries	SELECT ... FROM T_1, ... WHERE ...T_1.ATTR [NOT] IN (SELECT T_2.ATTR' FROM T_2,... WHERE ...) or: WHERE ...T_1.ATTR{=|≠} (SELECT T_2.ATTR' FROM T_2,... WHERE ...)
auto-join	SELECT A.STAFF_ID FROM STAFF A, STAFF B WHERE A.SALARY > B.SALARY AND A.SUPERVISOR = B.STAFF.ID

Table 2 - Query patterns

4 In table 2, $ATTR_i^{(j)}$ denotes the i-th attribute of the j-th relation. Obviously, when we use the options NOT IN or ≠, we are not in presence of an equi-join. Nevertheless, it is evident that the programmer intends to compare two values taken from the same domain. This form of statement is reported here for compactness.

```
SELECT NAME
FROM AUTHORS A, BOOKS B
WHERE A.ID = B.AUT AND B.TITLE = :book
```

is equivalent to:

```
SELECT AUT
INTO :aut_code
FROM BOOKS
WHERE BOOKS.TITLE = :book
        . . .
SELECT NAME
INTO :aut_name
FROM AUTHORS
WHERE ID = :aut_code
```

However, the detection of this kind of patterns requires the identification of the data dependences.

5.2. Foreign keys

Once the synonymies have been detected, we can look for the foreign keys, that model the associations between tables. The process takes place in three steps:

a) *Annotation of the explicitly declared foreign keys.*
This is the simplest case as the DDL possesses the appropriate mechanisms.

b) *Detection of non explicitly declared foreign keys.*
Given a relation T having a known primary key PK, we single out the synonyms of the components of PK that all belongs to a relation T'. They are the components of a foreign key, defined in T', referring T.

c) *Detection of foreign keys referring uncertain primary keys.*
For all the relations that only have an indication of possible primary key (PPK), we have to apply the same process described in the previous step, but we have to transfer the uncertainty of the information about the primary key to the result.

5.3. Referential integrity constraints

The uncertainty of the information gained in the previous step makes necessary to attempt to confirm the indications by looking at referential integrity constraints' checks in the code. There are several cases when we have to check the referential integrity constraints: insertion in the referencing relation or updating of its foreign key, deletion from the referenced relation or updating of its key. The older versions of SQL did not provide any mean either to explicitly define the primary key, or to guarantee the referential integrity constraints; hence, their check was entirely charged to the programmers. The more recent SQL standard has some special DDL constructs to define

```
PROFESSORS (LSTNAME, FRSTNAME, BIRTHDATE,
            ADDRESS,...)
COURSES (COURSE_ID, CLASSROOM, PROF_LSTNAME,
         PROF_FRSTNAME, PROF_BIRTHDATE,...)

EXEC SQL BEGIN TRANSACTION;
EXEC SQL
  SELECT *
  FROM PROFESSORS
  WHERE LSTNAME = :prof_lstname AND
        FRSTNAME = :prof_frstname AND
        BIRTHDATE = :date;
if (SQLCODE == 0)
  {
    EXEC SQL
      INSERT INTO COURSES (COURSE_ID,
       PROF_LSTNAME, PROF_FRSTNAME,
       PROF_BIRTHDATE)
      VALUES (:course, :prof_lstname,
              :prof_frstname :date);
    EXEC SQL COMMIT WORK;
  }
else <call of the error_handling routine>
```

Fig. 1 - A simple database and a referential integrity constraint pattern

primary and foreign keys ([4], [6]). Many commercial products make available triggers that can fire on insertion, update or delete on a table. The identification of the procedural patterns that implement the constraints can help in the simplification of the code and assure a more homogeneous implementation of the constraints, moving many actions directly at the schema level.

In fig. 1 we can see a simple procedural pattern assuring the referential integrity constraint at the insertion in a referencing table.

5.4. Analysis of the referential integrity constraints

Recognising foreign keys and referential integrity constraints can help in the detection of the associations. We have pointed out as the uncertainty about the identification of the primary key can affect that of the foreign keys. However, it seems reasonable to recognise simple programming patterns, or to proceed with a manual approach, guided by static analysis tools.

```
CUSTOMERS (CUSTOMER_ID, COMPANY, COUNTRY,...)
AGENTS (AGENT_ID, ..., ZONE).

EXEC SQL BEGIN TRANSACTION;
EXEC SQL
  SELECT *
  FROM CUSTOMERS
  WHERE COUNTRY = :zone;
if (SQLCODE == 0)
  {
    EXEC SQL
      INSERT INTO AGENTS (AGENT_ID,..., ZONE)
      VALUES (:agent,..., :zone);
    EXEC SQL COMMIT WORK;
  }
else <call of the error_handling routine>
```

Fig. 2 - A dynamic constraint check pattern

Schema indicators	Primary key indicators	SQL indicators	Procedural indicators	CONCEPTS
	• doesn't correspond to PK of other relations • doesn't contain FK			**Fundamental entity** - without multivalued attributes - non belonging to any hierarchy
• possible key dominance (PK + 1+2 attributes) • referential integrity constraint on the FK (the DDL supports their explicit declaration)	• is composed by some FK	• join with the related relations	• referential integrity constraint checks (not explicitly defined by the DDL)	**Relationship**
• key dominance (PK + a single attribute) • non key attribute unique and not null	• doesn't contain FK	• is referred in a number of relations (i.e. its PK is the FK in several relations) • a few or no insert, delete o update (or with DB administrator grants)	• referential integrity constraint checks (not explicitly defined by the DDL)	**Enumerative type relation (decoding table)**
• key dominance (PK + a single attribute) • non key attribute unique and not null	• corresponds to PK of other relations	• the I/O procedures on the relations including the related PK are executed using it always for coding/decoding operations • the selection and manipulation statements on the relation including the related PK can use it for coding/decoding a key		**Key coding relation**
• "full PK" • referential integrity constraint on the FK (the DDL supports their explicit declaration)	• PK includes a FK and a single attribute • usually FK and PK of the referred relations have the same name	• is referred by a single relation	• referential integrity constraint checks (not explicitly defined by the DDL)	**Simple type multivalued attribute**
• key dominance (PK + a single attribute) • non key attribute not null •referential integrity constraint on the FK (the DDL supports their explicit declaration)	• PK includes a FK and a single discriminating attribute • usually FK and PK of the referred relations have the same name	• is referred by a single relation	• referential integrity constraint checks (not explicitly defined by the DDL)	**Simple type multivalued attribute** (The single values for each entity are made distinct by the value of the discriminating attribute)
• "full PK" • referential integrity constraint on the FK (the DDL supports their explicit declaration)	• PK includes a FK and some attributes • usually FK and PK of the referred relations have the same name	• is referred by a single relation	• referential integrity constraint checks (not explicitly defined by the DDL)	**Complex type multivalued attribute**
• referential integrity constraint on the FK (the DDL supports their explicit declaration)	• PK includes a FK and a single discriminating attribute • usually FK and PK of the referred relations have the same name	• is referred by a single relation	• referential integrity constraint checks (not explicitly defined by the DDL)	**Complex type multivalued attribute** (The single values for each entity are made distinct by the value of the discriminating attribute)
	• sets of relations (#≥2) with related PK	• each relation belonging to the set has autonomous join with other relations	• decision statements are present to conditionally operate on the tuples of the subsets by using DML statements	**Disaggregate hierarchy of subsets or partitions** (It includes several relations)
		• in general is characterised by associations with several relations	decision statements are present to conditionally operate on the tuples of the subsets by using DML statements	**Aggregate hierarchy of subsets or partitions** (It includes several relations)
	• corresponds to PK of other relations	• has autonomous join with other relations	decision statements are present to conditionally operate on the tuples of the subsets by using DML statements	**Generalisation entity of a hierarhy**
	• corresponds to PK of other relations	• has autonomous join with other relations	decision statements are present to conditionally operate on the tuples of the subsets by using DML statements	**Subset entity**
• in general doesn't have key dominance	• contains FK	• in general has autonomous join		**Weak entity**

Table 3 - Matrix of indicators

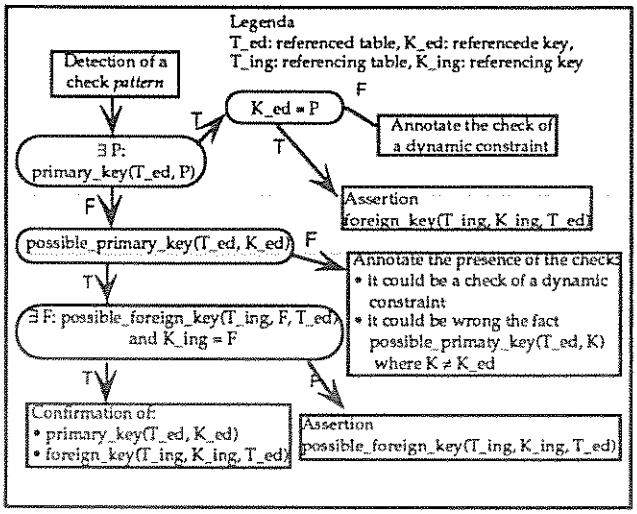


Fig. 3 - Algorithm for referential constraints' analysis

Anyway, the *procedural indicators* taken from the analysis of the control of the integrity constraints can give a valuable help in the identification of the foreign keys (if the primary key of the referenced relation is known), and in confirming or rejecting previously formulated hypotheses on the primary keys.

We must note that some patterns can be very similar, even if they have different purposes. In fig. 2 we report a simple example of the implementation of a constraint that adopts a pattern that could be confused with a referential integrity pattern, as it differs from this last one only because the existence check is performed on a non key attribute of the referenced relation. Fig. 3 sketches the algorithm for the check of the referential integrity constraints, with an obvious meaning of the assertions.

6. Conceptualisation

We have to refine the knowledge acquired in the previous phases, to identify the features of relations that implement some peculiar conceptual elements. As we have pointed out before, the indicators can be classified in four categories according to the source they derive from. Such a classification is the foundation of a simple and extensible paradigm, the *matrix of indicators*, aimed to the identification of the elements belonging to conceptual schema. The matrix of indicators (Table 3) has the following layout:

- the rows correspond to different ER concepts, including both those having a direct mapping to the relational model (strong and weak entities) and those who, on the contrary, can't be modelled directly with single relational objects (hierarchies, relationships with attributes, n-ary relationships, etc.);
- the columns correspond to the categories of indicators;
- every matrix cell contains the indicators of the related column we can start from to deduce the concept of the related row.

The phase of populating the matrix of indicators moves from both the theoretical knowledge about data models and the mapping rules, and the practical knowledge derived from implementation experiences. We cannot give general rules for the identification of the conceptual elements, as it depends on the quantity and quality of the indicators in the matrix. As an example, the structure of the primary key can be a "strong" indicator of an associative table, while a high join frequency with the associate tables can be taken as an indicator that confirms a hypothesis. If for a relation we are not able to detect any indicator for any row of the table, we have to:

1. show that it is impossible to clearly state the conceptual meaning of the relation;
2. display all the detected indicators;
3. ask the user to formulate a hypothesis.

Some inconsistencies can arise; in these cases, it can be necessary to re-execute some phases of the whole process.

6.1.The hierarchies

IS-A hierarchies can be mapped in different ways: creation of an aggregation table, that contains all the attributes of the generalisation and specialisation classes, or as many relations as are the classes and subclasses. In both cases, we can find some typical patterns, as shown in fig. 4-5 (with obvious meaning of the variable names).

```
EMPLOYEE (EID, D1,…,Dn)
MANAGERS (EID, M1,…,Mp)
TECHNICIANS (EID, T1,…,Tq)
SECRETARIES (EID, S1,…,Sr)

EXEC SQL
 INSERT INTO EMPLOYEE (EID, D1,…,Dn)
 VALUES (:id, :d1,…,:dn);
switch (role)
{
 case '01':
  EXEC SQL
   INSERT INTO MANAGERS (EID, M1,…,Mp)
   VALUES (:id, :m1,…,:mp);
  break;
 case '02':
  EXEC SQL
   INSERT INTO TECHNICIANS (EID, T1,…,Tq)
   VALUES (:id, :t1,…,:tq);
  break;
 case '03':
  EXEC SQL
   INSERT INTO SECRETARIES (EID, S1,…,Sr)
   VALUES (:id, :s1,…,:sr);
  break;
}
```

Fig. 4 - A typical insertion pattern for a disaggregate hierarchy

```
EMPLOYEE (EID, D1,…, Dn, M1,…, Mp, T1,…,
Tp,S1,…, Sr),

switch (role)
{
 case '01':
  EXEC SQL
  INSERT INTO EMPLOYEE (EID, D1,…,Dn, M1,…,Mp)
   VALUES (:id, :a1,…,:an, :m1,…,:mp);
  break;
 case '02':
  EXEC SQL
  INSERT INTO EMPLOYEE (EID, D1,…,Dn, T1,…,Tq)
  VALUES (:id, :a1,…,:an, :t1,…,:tq);
  break;
 case '03':
  EXEC SQL
  INSERT INTO EMPLOYEE (EID, D1,…,Dn, S1,…,Sr)
  VALUES (:id, :a1,…,:an, :s1,…,:sr);
  break;
}
```

Fig. 5 - A typical insertion pattern for an aggregate hierarchy

6.2. Associations

The cardinality of the associations can be derived from an analysis of the procedural patterns yet described in the paragraph about the primary keys. Some other indicators, and the features that can be derived from, are reported in Table 4.

7. Conclusions

In the reconstruction of the ER schema of a database, we can identify the constraints that are maintained at the procedural level, too. Therefore it becomes possible to re-engineer the applications getting all the advantages provided by the most recent DBMSs, that allow the definition of many constraints directly at the schema level. In this work we described the problematics and the guidelines of a RDBRE taking in input the DBMS catalog and the source code, and is able to deduce the database conceptual schema.

The proposed methodology shows some innovative aspects in respect with others presented in the literature. In fact, the particular "cognitive" approach recognises specific properties of the relations not only taking into account the knowledge that can be deduced from the *database structure*, but also attempting to interpret *how the applications make use of the data*. This is done searching for some SQL and *procedural patterns* that can be significant for the detection of the searched properties.

As the proposed approach requires a user interaction, it is evident that the recognition potentiality and the quality of the knowledge base depend on the user's experience in developping data oriented applications and his/her knowledge of the application environment.

A Prolog prototype of an "expert system" implements the methodology. As a future development, to make a more extensive test on a suitable number of applications, we foresee the integration of the tool in TROOP, a reverse engineering tool currently under implementation ([8]).

Type	Pattern	Feature
Schema	NULL <foreign_key> NOT ALLOWED IN <table>	*total association*
Schema	NULL <foreign_key> ALLOWED IN <table>	*partial association*
SQL	SELECT … FROM …, T,… WHERE …T.FK IS [NOT] NULL	*partial association*
SQL	Joins FK-PK have clauses: FROM T WHERE FK1=:host_var1 AND…AND FKn=:host_varn or FROM T, T' WHERE T.FK1 = T'.PK1 AND…AND T.FKn=T'.PKn	*multiple association*

Table 4 - Some patterns for association detection

References

[1] Batini C., Ceri S., Navathe S.B.: *Conceptual Database Design: An Entity-Relationship Approach*, The Benjamin/Cummings Publishing Company,Inc., 1992.

[2] Chiang R.H.L., Barron T.M., Storey V.C.: *Reverse engineering of relational databases: Extraction of an EER model from a relational database*, Data & Knowledge Engineeering, Vol. 12, N. 2 (Mar. 1994), pp. 107-142

[3] Codd E.F.: *Extending the Database Relational Model to Capture More Meaning*, ACM TODS 4, No. 4 (Dec. 1979)

[4] Date C.J., White C.J.: *A guide to DB2 - Second edition*, Addison-Wesley(1987)

[5] Hainaut J-L., Chandelon M., Tonneau C., Joris M.: *Contribution to a Theory of Database Reverse Engineering*, Proc. IEEE Working Conference on Reverse Engineering, Baltimora 1993

[6] *Information technology - Database languages - SQL2*; ISO standard N. 9075

[7] Premerlani W.J., Blaha M.R.: *An Approach for Reverse Engineering of Relational Databases*, Proc. IEEE Working Conference on Reverse Engineering, Baltimora 1993.

[8] Signore O., Loffredo M.: *Re-Engineering towards Object-Oriented Environments: the TROOP Project*, Proc. of The 8th Int. Symp. on Computer and Information Sciences (ISCIS VIII), Nov. 3-5 1993, Istanbul (Sponsored by IEEE)

[9] Teorey T.J., Yang D., Fry J.P.: *A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model*, ACM Computing Surveys, Vol.18, No.2, Jun. 1986.

Relational Views for Program Comprehension

Tim Jones, Warwick Allison, David Carrington
{tsj, warwick, davec}@cs.uq.oz.au

Software Verification Research Centre
Department of Computer Science
University of Queensland
Queensland, Australia 4072

Abstract

In this paper we describe UQ★, an integrated development environment that is currently under construction at the University of Queensland. Its architecture supports the definition of multiple documents and multiple document types, and allows the relationships that are implicit within the set of documents to be represented explicitly. We identify two techniques that aid program comprehension which require knowledge about the relationships that exist in and between documents. They are program dependency analysis and literate programming. Two simple examples are presented to illustrate the flexible definition of relations within such an architecture and the use of relations for presentation of, and navigation through, various views of a program and its related documentation. These examples highlight the application of such an approach to program dependency analysis and literate programming.

1 Introduction

Program comprehension, also referred to as program understanding, is the act of perceiving the meaning and structure of a program. When is program comprehension important? Software maintenance takes over 50% of the total expenditure that is allocated to a software system during its lifetime and program comprehension takes at least 50% of the time spent on the maintenance task [11, 12]. However, program comprehension is not solely a maintenance issue. Program comprehension is also a significant task during implementation. For example, individual members of programming teams continually review each others' code for both quality and corrective reasons. Even individual programmers reviewing their own code that was written over an extensive period of time will, for various reasons, find it necessary to investigate what various parts of their program do. Additionally, program comprehension is important during testing and debugging. Programmers must understand programs to devise complete and comprehensive test cases and to locate bugs. The need for tools to support program comprehension is well documented [19]. These tools should support program comprehension during implementation, maintenance, testing and debugging.

Wilde [32] emphasises that "*A key to program understanding is unravelling the interrelationships of program components*". In the context of a program source document, these relationships are more commonly known as program dependencies. Since the late seventies, much work has gone into the theoretical aspects of program dependencies [35, 34, 27, 28, 15, 24, 32, 21, 14] to pave the way for automating their extraction and presentation. This work has resulted in a wide variety of tools that address this issue. Large integrated environments such as Pecan [22], PV [7], MicroScope [1], ProDag in the Arcadia environment [23] all provide views of a limited set of program dependencies for a specific implementation language. Stand-alone program analysis and editing tools such as PUNS [12], Whorf [3], DgQuery and its associated tool-set [33], CIA [9] [10] and CIA++ [8] provide more comprehensive coverage of program dependencies, but are still language specific. Some program analysis tools such as LogiScope [18] are capable of analysing many different languages, but only provide views of a limited set of program dependencies.

Another important key to program comprehension is the documentation that accompanies the program. One well-known approach is the literate programming style [16]. However, Knuth's literate programming system WEB fell short of an ideal environment to encourage good documentation practice. Broom [5] improved on WEB by providing an interactive on-line presentation and simultaneous manipulation of both documentation and code as well as off-line presentation of a 'literate program' in an editor/browser called Sue. However interactive systems that support literate programming should not stop at this. Tools that support design and reverse engineering

0-8186-5647-6/94 $04.00

utilize graphical diagrams to convey ideas about a program. Literate programming and environments that support it should include appropriate graphical notations to allow enriched explanations of programs.

The comment by Teitelbaum that "*Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint*" [26] indicates the need to integrate programming tools into a single environment based on a uniform representation. This comment was associated with an environment based on a structure editor which emphasises a fixed view of the program. Welsh [29] indicates that environments not need be restricted to structural views to reinforce the fact that programs are hierarchical compositions of computational structures. Ambras extends this concept and indicates that "*Programming environments that support evolutionary software development must include tools that help programmers understand complex programs*" [1]. We believe that these ideas should be extended further so that the documentation associated with the design and development of a program's source code is manipulated in the same way as the program's source code. Thus a programming environment should reinforce the fact that programs and their associated documentation are hierarchical compositions and should provide facilities to edit, execute, debug and understand them.

In order to support these concepts, we are developing an environment, UQ★, whose architecture supports the definition of multiple documents and multiple document types, and allows the relationships that are implicit within the set of documents to be represented explicitly. The documents and views supported by such an architecture may be textual, graphical or both. The aim of this approach is to provide the ability to incorporate documents from the earlier phases of the software development life-cycle into the same architecture as the program, and to emphasise the inter- and intra-document relationships that allow more flexibility in defining and navigating through views of the software system than do conventional editors and environments. An important issue in the conceptual design of this architecture is that the program source is just another document whose language happens to be compilable into executable form. UQ★ provides the flexibility to define views of program dependencies through the definition, manipulation and use of relations. Furthermore it encourages literate programming. Programs which are not literate and their development documentation can be loaded into the environment and woven together by creating the appropriate relations as the programmer explores the documents.

There are three accepted categories of theories that describe the cognitive processes involved in program comprehension. Corbi [13] describes these as the *bottom-up*, the *top-down* and the *opportunistic* theories. Bottom-up theories are based on the notion that a programmer understands a program by iteratively abstracting and connecting together 'chunks' of code. Top-down theories are based on the notion that the programmer uses their own experience and attempts to confirm their expectations. Opportunistic theories are a mixture of the first two types of theories, where the programmer uses an *as-needed* rather than a systematic approach to understanding the actual code [17]. By supporting flexible views of program dependencies and the ability to peruse development documentation in a 'literate' style, UQ★ will support all three types of theories, particularly opportunistic theories.

In this paper, we discuss in more depth the importance of relations in program comprehension in section 2, and in the following sections illustrate how UQ★ takes advantage of this to provide an integrated development environment that addresses these issues. In section 3, an overview of the architecture of UQ★ is presented. An example is presented in section 4, showing the derivation and application of relations for program dependencies, and section 5 contains another example which shows the use of relations for navigation within and between documents. Section 6 presents an overview of the work presented in this paper and related work at the University of Queensland.

2 Relational Views for Program Comprehension

In this section, we extend the ideas presented in the introduction to show the important role that relations play in program comprehension. By treating relations as an underlying conceptual structure of a programming environment, all the views discussed in this section are available to the programmer. We confine the discussion to two areas in which relations form the foundation of views that contribute to program comprehension. The first area is program dependencies and the second area is relationships between development documents and program source documents.

2.1 Program Dependencies

Program dependencies are relationships between a set of program elements that are dependent either syntactically or semantically on another set of program elements. In an environment based on relational structures, these relationships can be made explicit within the structure in

which the program source document is contained. If inter- and intra-document relations are handled in the same fashion, the expression of program dependence relations is not restricted to being within a single program source document; they can be expressed between program source documents of both the same and different languages. This means that program dependencies can be traced through sub-programs that are written in different languages from the parent program under investigation.

Discussion of program dependencies in this section is limited to those found in conventional imperative programming languages. When considering such languages, we identify four classes of program dependencies: data-type dependencies; data-item dependencies; procedure/function dependencies; and module dependencies. Each of these is presented individually in the following sub-sections.

Data-Type Dependencies: In typed programming languages, data-type dependencies are the simplest type of dependency. Such languages have a finite set of predefined types; however in most of these languages the programmer can use these basic types as building blocks to create new, more complex types. Thus a programmer who has to understand a program, at some stage will be required to work out how these more complex types are constructed.

Data Item Dependencies: Data items are any components of a language that represent a particular value. Examples of data items are variables and constants. In most languages, data items are dependent on two things: their declaration which specifies what type of value can be stored; and other data items that are used to derive their value. In the first case, the programmer is interested in the type of a given data item. This type may be a complex type that requires further investigation (section 2.1.1). In the second case, the programmer is interested in the effects of operations on data items. There are two questions to consider in this case:

(1) What operations affect a particular data item at a given place in the program?

(2) What data items are affected by a change in a given operation?

The first question involves isolating the statements that have a direct effect on the given data item. Weiser [27] described a related technique, which he called *program slicing*, as a method used by experienced programmers for abstracting and understanding programs. This technique is also an excellent and widely used technique for debugging programs [28]. A program slice is formally defined as the minimal subset of a program that produces a selected subset of the program's original behaviour. Weiser [27] says, in general, automatically finding a slice is impossible. However dataflow algorithms can be used to approximate a slice where the behaviour subset is the values of a set of variables at a statement.

The second question involves isolating statements which use the result of the given data item to derive the values of other data items. Yau [35] describes this as *ripple effect analysis*. This technique is most applicable to maintenance [35, 34], and like program slicing, it relies on dataflow algorithms to locate the desired information.

Procedure / Function Dependencies: Procedures and functions[1] can be seen as functional black boxes, providing their full behaviour is known and does not need to be changed. However if this is not the case, programmers trying to understand various aspects of a function's behaviour are faced with the problem that a function may contain references to program elements that are not defined in the function. Such references are known as a function's global dependencies. In structured languages like C, Pascal and Modula-2, the global dependencies of a function may be any of the following:

- Global type declarations;
- Global variables;
- Other functions; and
- Constants.

As with data item dependencies, a programmer may be interested in the declaration of the dependency, or the operational effects of the dependency.

Module Dependencies: In this paper we consider modules to be separate files such as in Modula-2. By this definition, classes in object-oriented languages such as Eiffel can also be seen as modules. As with functions, modules can contain references to program elements that are not defined in the module. Such references are known as the module's global dependencies. In structured languages, the global dependencies of a module may be any of the following:

- Global type declarations;
- Global variables;
- Global functions; and
- Constants.

1. In this section the word function is used to mean procedure as well as function.

The global dependencies of a module are always contained in another module, but are not necessarily the entire module. Thus this type of dependency is a more abstract dependency than the ones presented in previous sections. As well as module dependencies, a programmer may be interested in the declaration of the dependency, or the operational effects of the dependency.

2.2 Relations Between Development and Program Documents

Development documents reflect both functional and structural aspects of a program's implementation, as well as reasons for design decisions. Typical development documents include requirement, specification, design and program description documents. These documents may be textual, graphical or both. Advocates of top-down program comprehension theories indicate that domain knowledge is the fundamental starting point for program comprehension. In many cases, particularly program maintenance, programmers will have a very limited knowledge of the domain. In such cases, programmers rely on the development documentation to gain this knowledge. This task is potentially time-consuming since the set of such documents associated with a program is often larger than the program itself. To further complicate matters, individual development documents usually only describe a few aspects of the program in question.

However development documents are often highly structural in nature. This structure results in explicit and implicit relationships both among development documents and between these documents and the program source documents. Understanding these relationships provides valuable insight into the functionality and structure of the program and issues that reflect why the program was implemented the way it was. Furthermore these relationships can be used to weave a 'literate program' that contains both text and graphics.

Unfortunately, the development documentation associated with large software projects is rarely indicative of the current state of the programs. Belady and Lehman [2] proposed three laws of program evolution, two of which indicate why it is important to spend time updating the development documentation so it is consistent with the current state of the program.

(1) "Law of continuing change. A system that is used undergoes continuing change until it is judged more cost-effective to freeze and recreate it."

(2) "Law of increasing entropy. The entropy of a system (its un-structured-ness) increases with time, unless specific work is executed to maintain or reduce it."

The first law indicates that maintenance is likely to be carried out in the future, and the second that this maintenance will be more difficult unless specific action is taken now. By making a program literate, its structure is captured and reinforced, easing the load on both current and particularly future activities that involve program comprehension.

3 Architecture of UQ★

The UQ★ system is based on a structured document model [30]. It allows for the representation of the syntactic and semantic structure of documents. It provides document construction facilities via textual and graphical views of document structures. Documents may also be constructed by integrated software-analytic tools and by importing text files.

The architecture providing this support is a persistent store of documents manipulated by *front-end* and *back-end* tools, as depicted in *figure 1*. *Front-end* tools are those which interact with users to present and modify the document store, while *back-end* tools are those which perform analytic operations upon the documents, producing results which become part of the document store.

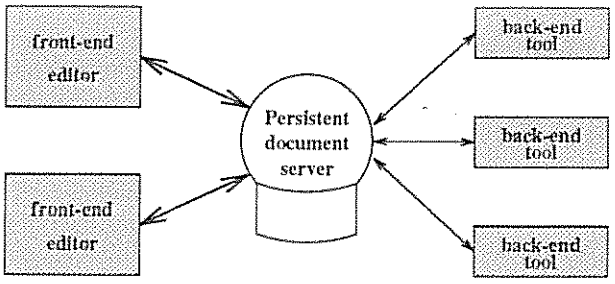


Figure 1. A high level view of UQ★ architecture

3.1 Documents

The syntactic structure of traditionally textual documents such as Pascal programs can be expressed via an EBNF document which is in the document store. From this, the system produces a state machine used for scanning and a grammar for parsing. These are used by the document construction facilities in textual views to build parse tree documents from text input in the language specified by the EBNF grammar. These parse trees are the document representation; textual views are formed by unparsing.

The document store consists of atomic elements and relationships between these elements, such as a *Statement* parse tree node and its parent-child relationship to its

syntactic constructs. Other document structures, such as the inter-relationship of the specification, design, implementation, and the user manual of a software system can also be expressed.

3.2 Relations

Relations exist between document segments. Relations can be classified by the mechanism by which they are created. The categories of relations supported by UQ★ are:

- User-Defined Values. The user creates individual links between document components. For example, a user might link portions of the implementation of an algorithm to points in a textual discussion of the algorithm in the design documentation.
- Pre-defined. The UQ★ parsing mechanisms create hierarchic links in parsed documents. Also, EBNF grammars are presented as relations among syntactic constructs.
- User-Defined Derived. The user expresses a relation in terms of other relations. For example, a user might define a call-graph from relations defined by tools and the parsing system. This example is illustrated in section 4.
- Tool-Defined. A tool defines new relations based on existing relations. For example, a back-end tool (*figure 1*) might generate a relation describing the scope of variables from the syntactic structural relations.

3.3 Back-end Tools

Back-end tools can be used to contribute to the document store. They may generate relations between document components, and generate document components for either inspection or further manipulation by the user. The nature and contribution of such tools are varied. A compiler may produce a list of errors, or an executable. A constraint verification tool may produce a single yes/no value based on its analysis. A program dependency analysis tool may produce a set of dependency relations. These are only a few examples of back-end tools that may be coupled with UQ★. A simple example of a possible application of a back-end tool is highlighted in the example presented in section 4.

3.4 Front-end Tools

The user interacts with the underlying documents via *views* of the structure provided by front-end tools. For interactive presentation and manipulation, the view can be some combination of text and graphics ranging from an automatically formatted unparsing to a graph of nodes and arcs. Similar view generation is able to generate publication-quality presentations, by generating LaTeX input.

4 A Simple Program Dependency Example

In this section we present an example to highlight the flexibility of definition and use of relations with UQ★. The example is a user-derived relation for a call graph in Modula-2. A call graph is a directed graph depicting the procedure call structure of a program. It is defined by the relation between each procedure or module in a program and each procedure it calls.

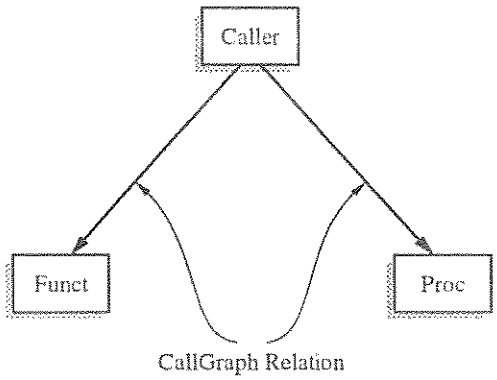


Figure 2. Call graph showing derived *CallGraph* relation

The example is illustrated using the Z notation [25] to show how a useful relation can be derived from other given relations. The relation described is the *CallGraph* relation (*figure 2*). *Figure 3*, provides an overview of the example under discussion.

We start by defining the set of all possible nodes in the parse tree as:

[NODE]

A relation is a segment that relates two other segments. We therefore define a relation as a mapping between two NODE elements. For this example, three relations are pre-defined:

$$GrammarNodeType : NODE \leftrightarrow NODE$$
$$Parent : NODE \leftrightarrow NODE$$
$$DeclarationUse : NODE \leftrightarrow NODE$$

The first two relations must exist for all documents. *GrammarNodeType* is a relation between a parse-tree node and its grammar-node and is inherent in the EBNF of the

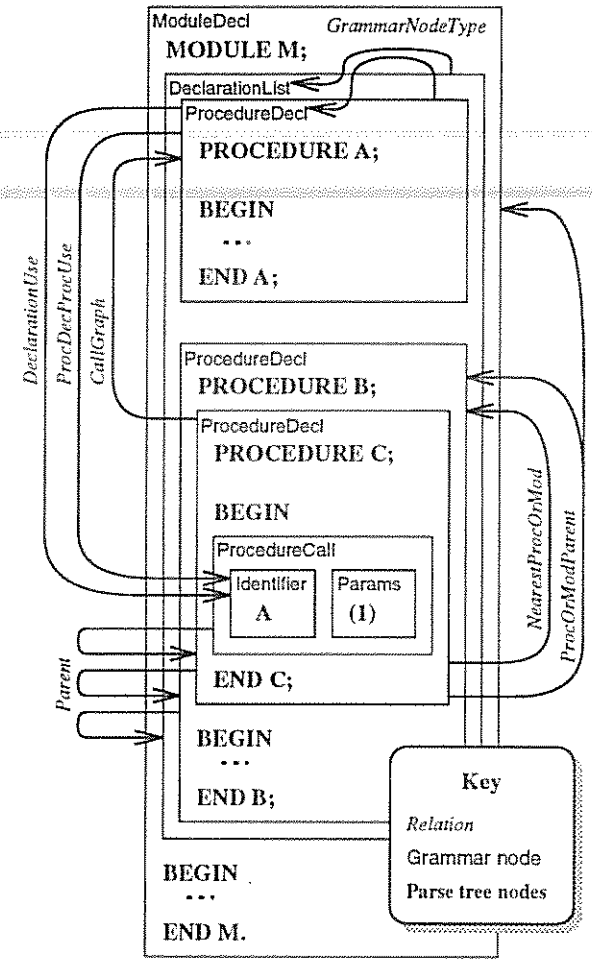


Figure 3. Relations involved in the derivation of the CallGraph relation

document. *Parent* is a relation between a child node and its parent node and is inherent in the structure of the parse tree. In this example *Parent* is defined as a partial function which is a specialised relation. *DeclarationUse* is a relation between the declaration of a program element and all uses of that element and is assumed to have been constructed by a back-end tool familiar with the semantics of Modula-2.

The *DeclarationUse* relation contains some information that is not relevant to call graphs since call graphs only consider procedure declarations, not all program elements. To extract the required information, a derived relation *ProcDecProcUse* is defined. This relation contains only those (declaration, use_id) pairs where the declaration is a procedure declaration and the parent node in the parse tree of the use_id is a procedure or function call.

$$ProcDecProcUse : NODE \leftrightarrow NODE$$

$$\begin{aligned}
&\forall\, declaration, use_id : NODE \bullet \\
&\qquad (declaration, use_id) \in ProcDecProcUse \Leftrightarrow \\
&\quad (declaration, use_id) \in DeclarationUse \wedge \\
&\quad (declaration, \text{"ProcedureDecl"}) \in GrammarNodeType \wedge \\
&\quad (\ (Parent\ (use_id), \text{"ProcedureCall"}) \in GrammarNodeType \vee \\
&\qquad (Parent\ (use_id), \text{"FunctionCall"}) \in GrammarNodeType\ )
\end{aligned}$$

Two further derived relations are required to define the *CallGraph* relation: *ModOrProcParent* and *NearestModOrProc*. *ModOrProcParent* relates each node to every enclosing procedure or module declaration.

$$ModOrProcParent : NODE \leftrightarrow NODE$$

$$\begin{aligned}
&\forall\, descendant, ancestor : NODE \bullet \\
&\qquad (descendant, ancestor) \in ModOrProcParent \Leftrightarrow \\
&\quad (descendant, ancestor) \in Parent^{+} \wedge \\
&\quad (\ (ancestor, \text{"ProcedureDecl"}) \in GrammarNodeType \vee \\
&\qquad (ancestor, \text{"ModuleDecl"}) \in GrammarNodeType\ )
\end{aligned}$$

NearestModOrProc is an additional specialisation that relates each node to the closest enclosing procedure or module declaration.

$$NearestModOrProc : NODE \leftrightarrow NODE$$

$$\begin{aligned}
&\forall\, descendant, nearest : NODE \bullet \\
&\qquad (descendant, nearest) \in NearestModOrProc \Leftrightarrow \\
&\quad (descendant, nearest) \in ModOrProcParent \wedge \\
&\quad \neg\, (\exists\, another : NODE \bullet another \neq nearest \wedge \\
&\qquad (descendant, another) \in ModOrProcParent \wedge \\
&\qquad (another, nearest) \in ModOrProcParent\ )
\end{aligned}$$

From these derived relations, the *CallGraph* relation can be defined. A callee is any procedure declaration in *ProcDecProcUse* that has a corresponding caller in *NearestModOrProc*.

$$CallGraph : NODE \leftrightarrow NODE$$

$$\begin{aligned}
&\forall\, caller, callee : NODE \bullet (caller, callee) \in CallGraph \Leftrightarrow \\
&\quad \exists\, use_id : NODE \bullet \\
&\qquad (callee, use_id) \in ProcDecProcUse \wedge \\
&\qquad (use_id, caller) \in NearestModOrProc
\end{aligned}$$

As with the other relations derived in this example, the *CallGraph* relation may be used as a starting point for deriving other useful program dependencies.

5 Presentation and Navigation through Documents

In this section we illustrate how relations can be used by front-end tools to allow the programmer to navigate through a collection of documents. Our illustration traces a enquiry session of a simple program that starts with a call graph and allows the programmer to browse the program progressively in more detail.

Figure 4 shows a call graph for a program that calculates the greatest common denominator for a set of positive integer pairs. Such a graphical view may allow the user to select a node and view the related code. The *CallGraph* relation was described in the previous section as a relation between the caller and the callee, which represented the arcs in the call graph. The nodes in the graph are the ProcedureDecl nodes in the parse tree (*figure 3*). Thus by selecting a node in the call graph, UQ★ is given a handle to the corresponding node in the parse tree.

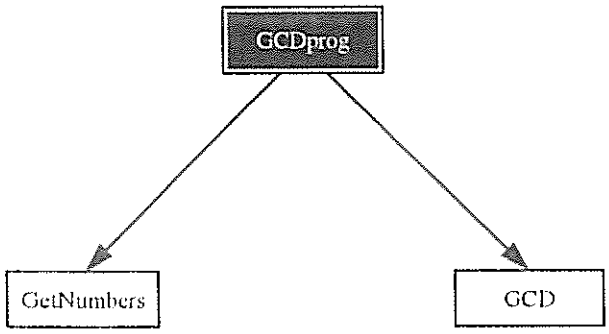


Figure 4. Call Graph of GCDprog

In *figure 4* the procedure GCDprog has been selected. GCDprog is the root module with the same name. *Figure 5* shows GCDprog as well as text and Z documents that describe it. A relationship exists between the GCDprog module and the text and Z documents. This relationship allows UQ★ to establish this view.

Additionally the detail of the procedure declarations in GCDprog is suppressed. The use of detail suppression [4, 29] supports program comprehension by reducing the amount of information the programmer has to contend with. However the programmer may at some stage be interested in these procedures.

View generation allows the programmer to expand the procedure declaration in situ, but this would make presentation of associated documentation more cluttered. Although the programmer may be interested in one of the two procedures, the information contained in *figure 5* may also be required. Opening another window to display a chosen procedure addresses both of these issues. An example of such a window for GCD is shown in *figure 6*.

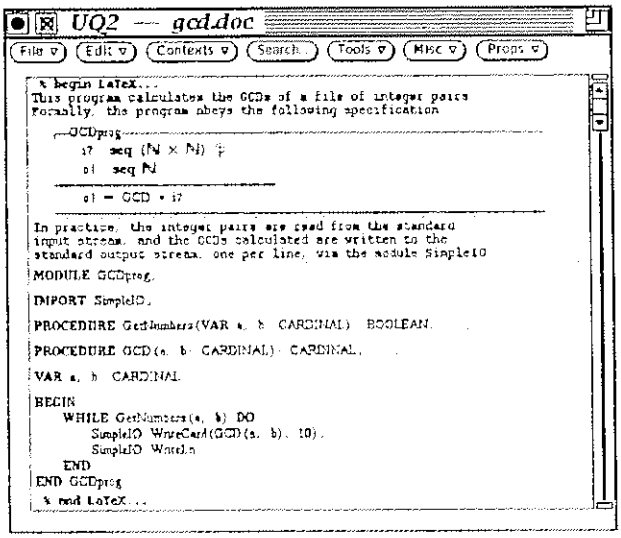


Figure 5. View if GCDprog and associated description documents

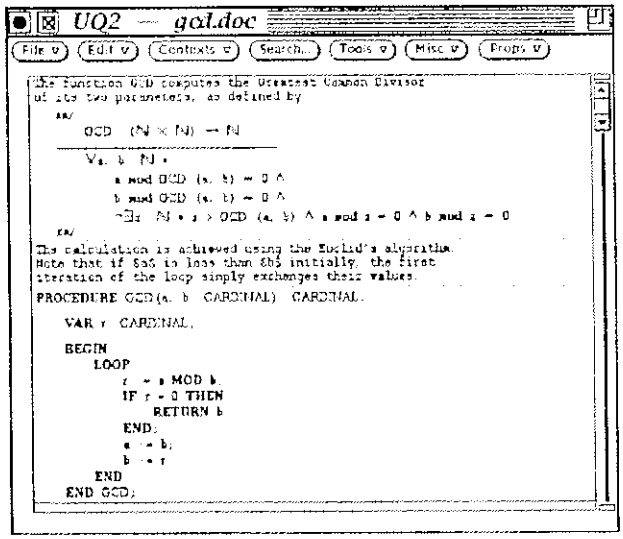


Figure 6. View of GCD and associated description documents

Oman [20] emphasises that off-line presentation of program sources are still a necessity even with the availability of multiple window system like UQ★, and proposes a book paradigm for the presentation of such documents. UQ★ is ideally suited to producing such documents through its ability to interleave multiple

document types, include LaTeX commands in text documents and generate views from cross-reference (relation) information.

For the example described in this section, a presentation promoted by the book paradigm may have the following contents page:

Contents

The structure chart is the view presented in *Figure 4*. The GCDprog and GCD would be the views presented in *figures 5* and *6* respectively and GetNumbers would be the equivalent view for that procedure. The index would contain page and possibly line numbers for the declaration and the uses of program elements occurring in the program. Note that this can be constructed from the *DeclarationUse* relation presented in the example in section 4.

6 Conclusion

Program comprehension is an important issue in software engineering. This paper has identified two techniques that can be employed, both to comprehend existing programs and to encourage the development of more comprehensible programs. They are, analysis of program dependencies and literate programming.

Documents that describe a software system are often highly structural in nature. This structure results in explicit and implicit, inter- and intra-document relationships. These relationships may be used in program dependency analysis or to link parts of various documents together to form literate programs.

We described UQ★, an integrated development environment which supports the use of multiple documents and document types and allows the relationships that are implicit in a set of documents to be represented explicitly. It will allow existing programs and related documentation to be imported for analysis and construction into a literate program. Furthermore it encourages the literate development of new programs, by providing the basis for hyper-textual literate views of programs. We illustrated a possible definition and use of relations within the UQ★ environment in two short examples. The first presented the derivation of a simple derived dependency and the second illustrated the use of relations for navigation through and presentation of multiple documents of differing descriptions. The relations and views presented in these examples are only a limited set of the views that UQ★ will be capable of presenting.

To conclude, we believe that an environment that treats relations as a fundamental part of document structure is both flexible in definition and nature.

7 Acknowledgements

The work described in this paper is an extension of work coordinated by Professor Jim Welsh and performed at the University of Queensland over recent years. The UQ★ environment is based on work done on UQ1 [31] and UQ2 [6, 29]. Early investigations of document based processing was performed by Jun Han [30] and contributed to the relational approach adopted in this paper. Current research is investigating coherent mechanisms for the graphical presentation and manipulation of software documents in the UQ★ environment. Much of this work, including the current research has received funding from the Australian Research Council.

We would specifically like to acknowledge Professor Jim Welsh for proposing the paper, Kelvin Ross for contributions to the definition of the Z schemas used in the example and Tim Mansfield for reviewing the paper.

References

[1] Ambras J. P., Berlin L. M., Chiarelli M. L., Foster A. L., O'Day V., Splitter R. N. Microscope: An integrated program analysis toolset. *Hewlett-Packard Journal*, 39(8):71–83, August 1988.

[2] Belady L., Lehman M. A model of large program development. *IBM Systems Journal*, 3:225–52, 1976.

[3] Brade K., Guzidial M., Steckel M. Soloway E. Whorf: A hypertext tool for software maintenance. *International Journal of Software Engineering and Knowledge Engineering*, 4(1):1–16, January 1994.

[4] Broom B., Welsh J. Detail suppression systems for interactive program display. In *Proceedings of the 9th Australian Computer Science Conference*, pages 83–93, January 1986.

[5] Broom B., Welsh J. Another approach to literate programming. In *Proceedings of the 11th Australian Computer Science Conference*, pages 257–68, Brisbane, February 1988.

[6] Broom B., Welsh J., Wildman L. UQ2: A multilingual document editor. In *Proceeding of the 5th Australian Software Engineering Conference (ASWEC '90)*, pages 289–94, Sydney, May 1990.

[7] Brown G. P., Carling R. T., Herot C. F., Kramlich D. A., Souza P. Program visualization: Graphical support for software development. *Computer*, 18(8):27–35, August 1985.

[8] Chen Y. F., Grass J. E. The C++ Information Abstractor. In *Proceedings of the Second C++ Conference*, pages 34–50. USENIX, April 1990.

[9] Chen Y. F., Ramamoorthy C. V. The C Information Abstractor. In *Proceedings of the Tenth International Computer Software and Application Conference*, pages 291–98. COMPAC, IEEE Computer Society Press, October 1986.

[10] Chen Y.F., Nishimoto M. Y., Ramamoorthy C. V. The C Information Abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–34, March 1990.

[11] Cleveland L. A user interface for an environment to support program understanding. In *Proceedings of the '88 Conference on Software Maintenance*, pages 86–91, Phoenix, Arizona, October 1988. IEEE Computer Society.

[12] Cleveland L. A program understanding support environment. *IBM Systems Journal*, 28(2):324–44, 1989.

[13] Corbi T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[14] Harrold M. J., Soffa M. L. Computation of interprocedural definition and use dependencies. In *Proceedings of the 1990 IEEE International Conference on Computer Languages*, pages 297–306. IEEE Computer Society, 1990.

[15] Keables J., Roberson K., Mayrhauser A. Data flow analysis and its application to software maintenance. In *Proceedings of the '88 Conference on Software Maintenance*, pages 335–47, Phoenix, Arizona, October 1988. IEEE Computer Society.

[16] Knuth D. E. Literate programming. *The Computer Journal*, 27(2):97–111, February 1984.

[17] Mayrhauser A. von, Vans A. M. From code understanding needs to reverse engineering tool capabilities. In *Proceedings: International Conference on CAiSE '93*, pages 230–39. IEEE Computer Society, 1993.

[18] Meekel J., Viala M. LOGISCOPE: A tool for maintenance. In *Proceedings of the '88 Conference on Software Maintenance*, pages 328–34, Phoenix, Arizona, October 1988. IEEE Computer Society.

[19] Oman P. Maintenance tools. *IEEE Software*, 7(3):59–65, May 1990.

[20] Oman P., Cook C. The book paradigm for improved maintenance. *IEEE Software*, 7(1):39–45, January 1990.

[21] Podgurski A., Clarke L. A. A formal model of program dependencies and its implications for software testing, debugging and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79, September 1990.

[22] Reiss S.P. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–84, March 1985.

[23] Richardson D. J., O'Malley T. O., Moore C. T., Aha S. L. Developing and integrating ProDAG in the Arcadia environment. In Herbert Weber, editor, *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, volume 17 of *Software Engineering Notes*, pages 109–119, Tyson's Corner, Virginia, USA, December 1992. Special Interest Group on Software Engineering, ACM Press.

[24] Schwanke R. W., Platoff M. A. Cross references are features. In *ACM 2nd International Workshop on Software Configuration Management*, pages 86–95, Princeton, N. J., October 1989. ACM Press.

[25] Spivey J. M. *The Z notation: A reference manual*. Prentice Hall International, 1989.

[26] Teitelman W. A display oriented programmer's assistant. *International Journal of Man-Machine Studies*, 11:157–87, 1979.

[27] Weiser M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–49, San Diego, California, March 1981. IEEE Computer Society.

[28] Weiser M. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–52, July 1982.

[29] Welsh J., Broom B., Kiong D. A design rationale for a language-based editor. *Software–Practice and Experience*, 21(9):923–48, 1991.

[30] Welsh J., Han J. Software documents: Concepts and tools. Technical Report TR-93-23, University of Queensland, Brisbane, Australia, 1993.

[31] Welsh J., Rose G. A., Lloyd M. An adaptive program editor. *Australian Computer Journal*, 18:67–74, 1986.

[32] Wilde N. Understanding program dependencies. Curriculum Module SEI-CM-26, Camegie Mellon University, August 1990.

[33] Wilde N., Huitt R., Huitt S. Dependency analysis tools: Reusable components for software maintenance. In *Proceedings of the '89 Conference on Software Maintenance*, pages 126–31, Miami, Florida, October 1989. IEEE Computer Society.

[34] Yau S. S. Methodology for software maintenance. Technical Report RADT-TR-83-262, Rome Air Development Centre, Griffis Air Force Base N. Y., February 1984.

[35] Yau S. S., Collofello J. S., MacGregor T. Ripple effect analysis of software maintenance. In *Proceedings of COMPSAC '78*, pages 60–65. IEEE Computer Society, 1978.

Object Data Models to Support Source Code Queries:

Implementing SCA within REFINE

Santanu Paul Atul Prakash

Dept. of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI-48105

Abstract

The REFINE [1] object base is being used widely for code analysis and reverse engineering. From the perspective of program querying and interactive program analysis however, REFINE-like object bases offer only general-purpose programming languages in which users must code their program queries. In contrast, **Source Code Algebra** *(SCA) is an object algebra designed to serve as an applicative source code query language. We are currently implementing an SCA-based query processor within the REFINE environment. This paper provides insights into some object data model features which are currently absent in the REFINE framework, and argues that their incorporation will enable certain source code queries to be handled more efficiently. We also argue that the inclusion of these features will greatly simplify the implementation of the SCA query processor.*

1 Motivation

Good systems to support querying on source code are an important part of a toolkit that aids program comprehension and reverse engineering. The purpose of a source code querying tool is to help human reverse engineers indulge in *plausible reasoning* or *domain bridging* [2] — an iterative process of guesswork and verification that leads them to a better understanding of what the source code is doing.

The choice of a suitable data model for programs is a central question in the design of source code querying systems. The most rudimentary "querying" systems include string-searching tools such as `grep`, which treat program source code as a stream of characters (represented in files). Files are also the chosen representation of cross-referencing tools which keep track of identifiers and their corresponding references. Progress in database theory has had significant impact on the evolution of program data models. The relational data model has been used to model program information in the CIA [5] and CIA++ [8] systems. Subsequently, the graph data model has been adopted in SCAN [1], Rigi [10], and other graph parsing-based systems [15] as the "natural" model for representing software structure and resource flow information. The syntax tree-based model (a restricted version of the graph data model) has been used in systems such as SCRUPLE [12]. At the present time, the object-oriented data model, a logical extension of the graph data model reinforced with strong type semantics, has emerged as the more appropriate data model for program information. In particular, the REFINE object database built into the Software Refinery [14] commercial product has been successfully employed in the analysis and reverse engineering of legacy software systems [3, 9].

However, this increase in modeling power is somewhat compromised by the absence of formal as well as flexible query mechanisms in the sophisticated models. For example, in `grep`, a query is specified declaratively using a simple regular expression. In a relational framework, a query is either a relational calculus (or SQL) expression (declarative) or a relational algebra expression (applicative). Either way, a query is essentially non-procedural, i.e., it doesn't have to be programmed explicitly by the user in some elaborate programming language. In contrast, in the graph-based and object-oriented models, there are no simple, flexible query languages. This can be ascribed to the absence of sufficient formalism in these models. Consequently, systems based on these models provide either a pre-programmed menu of query features, or a

[1] Software Refinery, REFINE, DIALECT, REFINE/C, REFINE/COBOL, and INTERVISTA are trademarks of Reasoning Systems

0-8186-5647-6/94 $04.00

general-purpose programming language in which users must explicitly program their queries.

To alleviate the problem of formalism, we have already proposed a *Source Code Algebra* (SCA) as the foundation for building program query systems [13]. The algebra defines an object-oriented model for representing program or source code information and gives a well-defined set of operators that can be used to make queries on the information. The analogy is the use of *relational algebra* [6] as the foundation for relational database systems. As in relational algebra, queries are expressed by writing expressions using the given operators. The benefits of using an algebra as the basis for a query language include the ability to provide formal specifications for query language constructs, the ability to use the algebra itself as a powerful applicative query language, and opportunities for query optimization.

This paper deals with an experimental effort to extend REFINE with a SCA-based non-procedural query language and processor. Specifically, we are interested in querying and analyzing C programs. In our view, this is an exercise with interesting possibilities. On one hand, REFINE/C, a commercial product based on REFINE tailored to handle C programs, comes with a predefined C data model. REFINE also comes equipped with a powerful, multi-flavored database programming language that can be used for diverse purposes ranging from code transformation to metrics analyses. On the other hand, SCA offers the possibility of an applicative, optimizable query language which REFINE currently lacks. A combination of the two approaches can be achieved by implementing an SCA query processor within the REFINE framework. To achieve this, two steps are necessary. First, the SCA data model must be mapped into a corresponding REFINE data model. Second, an SCA algebraic evaluator must be implemented within the REFINE environment to serve as a query processor. We have already implemented substantial parts of the SCA-based query processor within REFINE. The prototype is operational and many complex program queries can be processed with it.

Some important issues related to object-oriented modeling arose in the context of mapping the SCA data model into REFINE. In particular, we found that the REFINE data model lacks *first-class citizen* support for collections such as sets and sequences. It also does not support an inheritance hierarchy of attributes. Furthermore, we also found that unlike some other object-oriented databases, REFINE does not contain automatic support for extending class schemas at runtime. Admittedly, these shortcomings do not cripple REFINE since any missing functionality can be programmed in REFINE (as we have done in the abovementioned cases) using its powerful database programming language. However, as we hope to show in this paper, the lack of these features complicates the implementation of the SCA query processor and makes it harder to handle certain classes of program queries within the REFINE framework.

Section 2 provides a brief description of the SCA machinery required to understand this paper. Section 3 sketches the current features of REFINE and indicates the target environment we are trying to build. Section 5 elaborates on the features that are currently unavailable in REFINE, shows why they are important, and discusses what we have done to implement them. Finally, Section 6 provides a summary and conclusions.

2 SCA: Source Code Algebra

SCA models C source code as an *algebra*. Informally, algebras are mathematical structures that consist of data types (*sorts*) and operations defined on the data types (*operators*). Our objective in designing a Source Code Algebra (SCA) was to model the data types in the source code domain as sorts, and to design source code query primitives as operators. A clear analogy can be found in the relational data model, where the *relational algebra* serves as the underlying mathematical model. By modeling source code as an algebra, we can employ a non-procedural query language to query it. There is one major distinction though: SCA is a *generalized order sorted* algebra while relational algebra is a *one sorted* algebra. In other words, SCA handles a wide variety of data types and supports a *type hierarchy*. In contrast, relational algebra supports only one type, namely *relations*.

2.1 SCA Data Model

2.1.1 Many Data Types

The data types that arise in C source code modeling can be classified into three groups:

- **Atomic data types**: These are the basic data types such as INTEGER, FLOAT, BOOLEAN, CHAR, STRING, etc.
- **Composite data types (Objects)**: Examples of composite data types in C are syntactic ele-

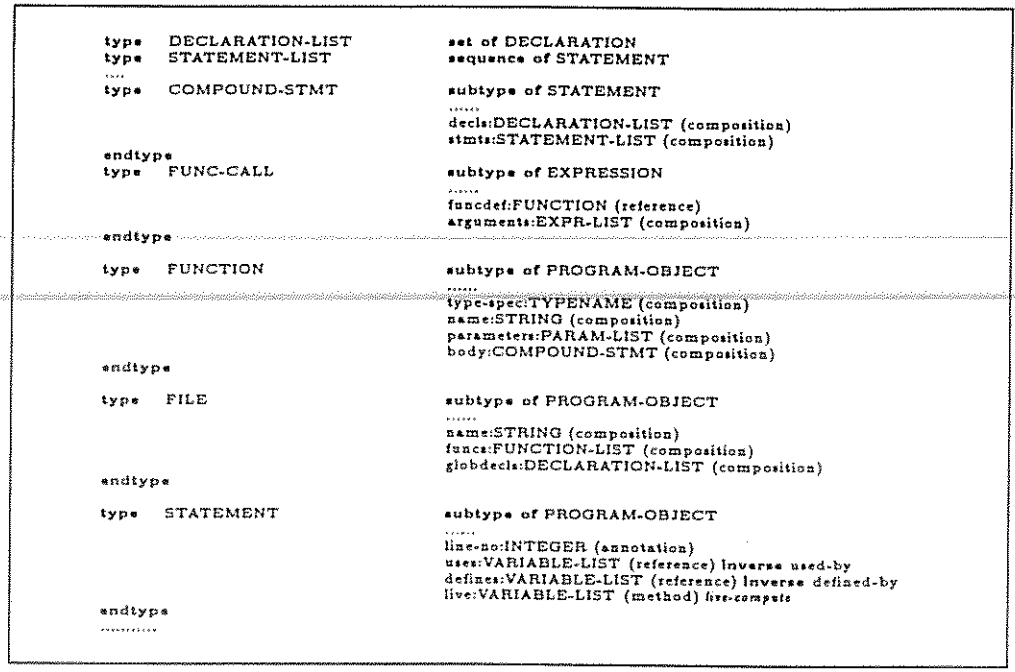


Figure 1: A part of the SCA Domain Model

ments of the language such as `while statement`, `relational expression`, `identifier`, etc.

- **Collection data types**: These are collections of other data types. For example, the type `statement-list` represents a *sequence* of `statement` objects. Similarly, the type `declaration-list` represents a *set* of `declaration` objects.

2.1.2 Type Hierarchy

An interesting feature that characterizes source code data types is the presence of a type hierarchy or *class hierarchy*. For example, `while-statements` are a subtype of the type `statements` (by specialization of *behavior*).

SCA incorporates the source code type hierarchy as an integral part of the algebraic framework. The algebra handles the notion of subtyping and inheritance, and supports *substitutability*, an important feature which lets an instance of a subtype be used in place of a supertype.

2.1.3 Object Attributes

There are four different kinds of *attributes* that may be associated with a source code object, namely, *components*, *references*, *annotations*, and *methods*.

Components model syntactic or structural information. In the case of a `while-statement` object, the components are its *condition* and *body*.

References model the associations between objects. In addition to simple cross-referencing information, they offer a way of modeling resource flow relationships that occur between objects. One set of important data flow relationships in the source code domain model are the "uses" and "defines" relationships.

Annotations are used to store all other relevant information about source code objects. Typical annotations to a source code object are line numbers, metrics, etc.

An attribute of an object can also be a *method* or a function that is computed on-the-fly. Methods are usually computed to obtain reference or annotation information, *during query execution*. Methods are a standard feature of object-oriented data models and can be used to introduce complex and specialized algorithms into the data model.

2.2 Source Code Algebra Operators

Given the source code data model in SCA, the next task is to define the algebra operators that are relevant to the task of querying source code. SCA is an algebra of atomic types, objects, and collections. We have used and extended operators from pre-existing object algebras for set operations, generalizing them

Operators	Description
< attribute >	*signature:* COMP ⟶ ANY *syntax:* < attribute > (< object >) *semantics:* Returns the value of the specified attribute
select	*signature:* COLLECTION(ANY1) ⟶ COLLECTION(ANY1) *syntax:* select$_{<boolean\ expression>}$(< objectcollection >) *semantics:* Chooses a subcollection based on a condition
project	*signature:* COLLECTION(COMP1) ⟶ COLLECTION(COMP2) *syntax:* project$_{<attributelist>}$(< objectcollection >) *semantics:* Retains only the specified attributes
extend	*signature:* COLLECTION(COMP1) ⟶ COLLECTION(COMP2) *syntax:* extend$_{<attribute:=algebraic\ expression>}$(< objectcollection >) *semantics:* Adds a new attribute to each object
retrieve	*signature:* COLLECTION(COMP) ⟶ COLLECTION(ANY) *syntax:* retrieve$_{<attribute>}$(< objectcollection >) *semantics:* Retrieves a specified attribute
closure	*signature:* COMP ⟶ SET(COMP) *syntax:* closure$_{<attributelist>}$(< object >) *semantics:* Finds all objects reachable using listed attributes
apply	*signature:* COLLECTION(ANY1) ⟶ COLLECTION(ANY2) *syntax:* apply$_{<operator>}$(< objectcollection >) *semantics:* Applies a unary operator to each element
product	*signature:* COLLECTION(COMP1) x COLLECTION(COMP2) ⟶ SET(COMP3) *syntax:* product(< objectcollection1 >, < objectcollection2 >) *semantics:* Cartesian product of two collections
flatten	*signature:* COLLECTION(COLLECTION(COMP1)) ⟶ COLLECTION(COMP1) *syntax:* flatten(< objectcollection >) *semantics:* Removes a level of nesting
pick	*signature:* COLLECTION(ANY1) ⟶ ANY1 *syntax:* pick$_{<boolean\ expression>}$(< objectcollection >) *semantics:* Picks an element out of a singleton collection
size_of	*signature:* COLLECTION(ANY) ⟶ INT *syntax:* size_of(< objectcollection >) *semantics:* Returns the size
reduce	*signature:* COLLECTION(ANY) ⟶ ANY *syntax:* reduce$_{<operator>}$(< objectcollection >) *semantics:* Applies a binary operator recursively to the collection
forall	*signature:* COLLECTION(ANY) ⟶ BOOL *syntax:* forall$_{<boolean\ expression>}$(< objectcollection >) *semantics:* True if all objects satisfy the condition
exists	*signature:* COLLECTION(ANY) ⟶ BOOL *syntax:* exists$_{<boolean\ expression>}$(< objectcollection >) *semantics:* True if even one object satisfies the condition
member_of	*signature:* COLLECTION(ANY1) x ANY1 ⟶ BOOL *syntax:* member_of(< collection1 >, element) *semantics:* True if element is a member

Table 1: SCA Operators for Objects and Collections

to operate on sequences wherever possible, and proposed appropriate operators for sequences. We have introduced seq_extract, a powerful new operator for sequences which uses regular expressions as the basis for extracting subsequences. SCA offers a unified approach to querying *collections*, whether they be sets or sequences. This is a departure from earlier approaches where the data model is either essentially set-oriented or sequence-oriented. Using the SCA operators, source code queries can be expressed as algebraic expressions. An evaluation of an algebraic expression on the source code representation yields the result of the query. A detailed description of SCA operators can be found in [13].

Table 1 shows SCA operators defined on objects and object collections. Operators specific to sets and sequences are shown in Tables 2 and 3 respectively.

Using these operators, users can express a wide variety of queries. For example, consider the SCA expression:

$$head_1(order_{no\_of\_func,>}(set\_to\_seq(\\ extend_{no\_of\_func:=size\_of(funcs)}(FILE))))$$

This expression evaluates to the file that has the maximum number of functions. First, the file objects are extended with a new field, namely no_of_func. The set of file objects is then converted into a sequence and arranged in decreasing order of no_of_func. The head of this sequence is the file with maximum functions.

Operator	Description
union	*signature:* SET(ANY1) x SET(ANY1) ⟶ SET(ANY1) *syntax:* union(< set >, < set >) *semantics:* Union of two sets
intersection	*signature:* SET(ANY1) x SET(ANY1) ⟶ SET(ANY1) *syntax:* intersection(< set >, < set >) *semantics:* Intersection of two sets
difference	*signature:* SET(ANY1) x SET(ANY1) ⟶ SET(ANY1) *syntax:* difference(< set >, < set >) *semantics:* Difference of two sets
subset_of	*signature:* SET(ANY1) x SET(ANY1) ⟶ BOOL *syntax:* subset_of(< set >, < set >) *semantics:* True if one set is a subset of another
set_to_seq	*signature:* SET(ANY1) ⟶ SEQ(ANY1) *syntax:* set_to_seq(< set >) *semantics:* Produces a random sequence from the set

Table 2: SCA Operators specific to Sets

Operator	Description
head	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* head$_{<n>}$(< *objectseq* >) *semantics:* Returns a sequence consisting of the first n elements.
tail	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* tail$_{<n>}$(< *objectseq* >) *semantics:* Returns a sequence consisting of the last n elements.
concat	*signature:* SEQ(ANY1) x SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* concat(< *objectseq* >, < *objectseq* >) *semantics:* Returns a concatenation of two sequences.
order	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* order$_{<attribute>,<ord>}$(< *objectseq* >) *semantics:* Returns a sequence ordered by the attribute values
seq_extract	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* seq_extract$_{<pattern>,<condition>}$(< *objectseq* >) *semantics:* Extracts a subsequence that fits the pattern
seq_element	*signature:* SEQ(ANY1) ⟶ ANY1 *syntax:* seq_element$_{<index>}$(< *objectseq* >) *semantics:* Returns the indexed element of the sequence.
subseq_of	*signature:* SEQ(ANY1) x SEQ(ANY1) ⟶ BOOL *syntax:* subseq_of(< *objectseq* >, < *objectseq* >) *semantics:* True if it is a subsequence
seq_to_set	*signature:* SEQ(ANY1) ⟶ SET(ANY1) *syntax:* seq_to_set(< *objectseq* >) *semantics:* Returns a set consisting of the sequence elements

Table 3: SCA Operators specific to Sequences

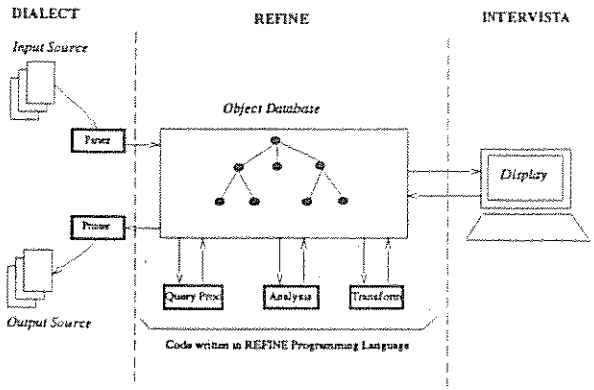


Figure 2: Software Refinery Architecture

3 Software Refinery

The Software Refinery toolkit [14] is a powerful reengineering system. It has gained considerable popularity amongst reengineering practitioners and academics as a convenient vehicle for both experimentation and industrial strength reverse engineering and reengineering. The toolkit contains an object database (called REFINE), a database programming language (also called REFINE), a parser-cum-printer generator (DIALECT), and a GUI development environment (INTERVISTA). The schematic architecture (adapted from [3]) is shown in Figure 2. We cover the object database and the programming language in this section.

3.1 REFINE Object Base

The REFINE object base is an object-oriented database that models the objects in an application domain and the relationships between the objects. Users can define their own classes and attributes and create schemas (also known as a domain model specification).

Typically, a program is represented in REFINE as an abstract syntax tree whose nodes are objects. The relationships between these objects are modeled as attributes.

The object base also supports a class hierarchy. For example, the most general class may be `program-object`. Other classes such as `program`, `function`, etc. are subclasses of `program-object`. Similarly, `while-statement` is a subclass of `statement`. A class hierarchy is semi-lattice structure that supports the inheritance relationship between the different classes.

3.2 REFINE database programming language

A powerful database programming language, also known as REFINE, is at the heart of Software Refinery. The language is multi-flavored, i.e., it supports a wide variety of programming styles and features including procedural, functional, rule-based, and pattern-based constructs. The syntax of REFINE is lisp-like. The procedural and functional constructs are useful in writing analysis routines. The rule-based and pattern-based constructs make it easy to write transformation routines for code reengineering and restructuring. Some of the other highlights of the REFINE language are:

- Support for high-level data types such as sets, sequences, trees and tuples, and their operations.
- Support for syntactic pattern matching.
- Flexible escape to Lisp, so users may write code in Lisp as and when necessary.

4 Implementing the SCA-based Query Processor

Our goal is to implement an SCA-based query processor within the REFINE environment. Figure 2 shows that currently, all processing tasks such as query processing, code analysis, and transformation must be programmed in REFINE using the database language. Our objective is to create the scenario shown in Figure 3, so that query processing and some analyses on code can be performed by an applicative query language. To reiterate, such an approach facilitates

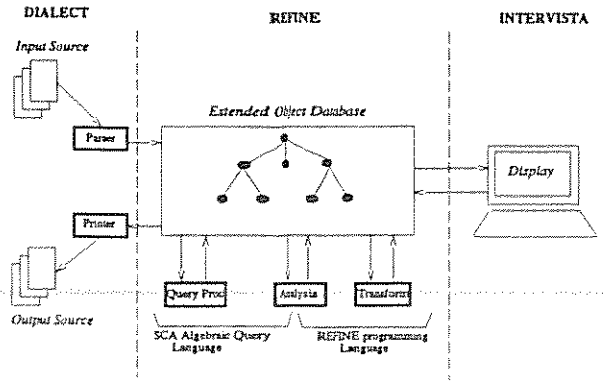


Figure 3: Modified Architecture

interactive program querying and promotes data independence, well-defined query semantics, and query optimization.

The implementation of our query processor contains two parts. First, a parser for SCA expressions must be constructed. The SCA parser converts SCA expressions written by a user into equivalent SCA expression trees. Finally, an evaluator of SCA expression trees must be implemented using the REFINE database programming language.

At this point, the SCA parser is complete and evaluator is nearing completion. The evaluator has already been used to process many SCA queries. Details of the queries that have been processed is beyond the scope of this paper.

5 Some Issues in Object-oriented Modeling

We present three important object-oriented modeling features that are unavailable in REFINE. We contend that REFINE stands to benefit from the inclusion of these features, both in terms of its modeling as well as query capabilities. We also contend that such features significantly simplify the implementation of the SCA query processor.

5.1 Collections as Classes

The REFINE data model does not support collections as *first class citizens*. REFINE classes are definable on objects, but not on collections of objects. For example, REFINE/C provides a class DECLARATION (actually DECLARATION-OBJECT but we use DECLARATION as a shorthand), the extent of which is the set of all declaration objects present in a given program. However, REFINE/C does not provide a class DECLARATION-LIST, the extent of which would be the set of all declaration *lists* in the program. On the other hand, the SCA data model treats collection data types at par with objects.

In the case of source code, collections are extremely important. Source code contains syntactic entities such as statement lists and declaration lists with the semantics of sequences and sets respectively. Queries on these collections must be supported. For example, consider the simple queries:

1. *Find all the declarations in the program.*
2. *Find all the declaration lists in the program.*

While these queries may appear contrived in isolation, such simple queries frequently occur in the context of more complex queries. For example, if the user wished to find all declaration lists with more than n declaration instances, or all declaration lists that contain m or more declarations of type T, then query 2 would need to be evaluated as the first step. Such queries apply equally well to parameter lists, statement lists, etc.

Declaration lists can occur as attribute values of objects such as FILE objects, COMPOUND STATEMENT objects, etc. In the SCA framework, since both `declaration` as well as `declaration-list` are supported as first class data types, the equivalent (simple) expressions for these queries are:

1. $select_{TRUE}(DECLARATION)$
2. $select_{TRUE}(DECLARATION - LIST)$

The REFINE implementation for the first query is simple: `instances-of`(DECLARATION). For the second query however, mapping the SCA collection data types to REFINE classes becomes the first issue. The query implementation follows from the mapping.

There are two ways to deal with the mapping problem. REFINE permits collection-valued attributes. For example, the type of an attribute may be SET(DECLARATION). One way is to create dummy classes for collections. For example, we can define a new class in REFINE/C called DECLARATION-LIST with exactly one attribute, say *list-attr*, which points to a declaration list. In this way, all declaration lists in the system can be associated with a dummy object of the class DECLARATION-LIST (through one level of indirection imposed by the attribute). To handle queries such as query 2, a query processor needs to identify collections as special classes, and transparently access the value of *list-attr* for each member of the dummy class DECLARATION-LIST.

The second way to deal with the mapping problem is to retain the attribute types as SEQ(STATEMENT), SET(DECLARATION), etc. For example, FILE has an attribute `globdecls` with type SET(DECLARATION) and COMPOUND-STATEMENT has an attribute `decls` with type SET(DECLARATION). Similarly, there may be many other attributes in the complete schema definition with the same type. To find all instances of `declaration-list` using REFINE, one has to write procedural code to navigate through objects, locate attributes with type SET(DECLARATION), and find their values.

The same query would have been much simpler if REFINE treated collections as *first class citizens*. In other words, collections ought to be treated as classes by themselves, at par with classes for single objects (STATEMENT, DECLARATION, etc.). The new classes would be STATEMENTLIST, DECLARATIONLIST, and so on. Finding all instances of `declaration-list` would then be equivalent to `instances-of`(DECLARATIONLIST), i.e., finding the members of the class, which is a simple class operation. The SCA expression would be:

$$select_{TRUE}(DECLARATIONLIST)$$

The result of the query would be a set of declaration lists.

5.2 Attribute Hierarchy or Grouping

The REFINE data model, with one exception, does not support attribute groups. On the other hand, the SCA data model supports four different kinds of attributes. These are *composition*, *reference*, *annotations*, and *methods*. This grouping of attributes can have important pay-offs in terms of query processing. Consider the query: *Find all function calls within the body of function* `foo`. Clearly, starting from object `foo`, the idea is to recursively explore its syntax subtree and filter out objects belonging to class FUNC-CALL. This can be efficiently done by pursuing only the composition attributes, as opposed to all possible attributes. Consider a different query: *Find all statements with which statement s shares a data-flow relationship*. Here, starting from s, we wish to traverse only the data-flow attributes (a subgroup of reference attributes) to locate target statements. Clearly, the ability to traverse attributes selectively is an important one and can be achieved using attribute groups. For example, the SCA expression for finding the function calls within `foo` is:

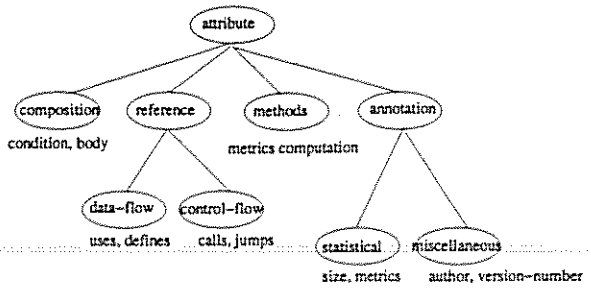


Figure 4: Attribute Grouping

$$select_{objectclass=FUNC-CALL}(closure_{composition}(pick_{name="foo"}(FUNCTION)))$$

Currently, REFINE supports composition attributes as a group. This is done using the notion of a *tree attribute*, effectively allowing the schema designer to register certain attributes as syntax tree attributes. A bunch of tree traversal operations are provided to navigate the syntax trees. There is no equivalent support for reference attributes. Nor can the schema designer decide a hierarchy of attribute groups of his or her choice. To implement the SCA attribute groups, explicit book-keeping is required to keep track of attributes.

This issue is resolved in object-oriented databases which support *attribute hierarchies*, and other object-oriented systems such as Telos [11], a development environment for information systems, which treats attributes as first class citizens. The ideal scenario is depicted in Figure 4. In addition to the four attribute groups, a schema designer may define other attribute groups as subgroups of previously-defined groups. An attribute group in the hierarchy represents a union of all attributes defined below it.

5.3 Extension of Class Schemas

REFINE does not permit the dynamic extension of class schemas. All class schemas are defined at compile time. This affects *view generation*, which is a standard feature in relational databases [7]. Views offer a mechanism to *compute* information not explicitly stored in the database. For example, the **join** operation in relational algebra computes new relations (or tables) from existing ones. In object-oriented databases, new views can be created by performing computations on existing attribute values of objects and storing the results as new attributes. Consider the query: *Show the five largest functions in terms of size (in lines)*. First, we

must compute the size of each function, then order them in decreasing order of their sizes, and finally choose the top five. To be able to order the functions, we need to explicitly *store* the size associated with each function. This should be done by extending the class schema associated with FUNCTION with a new attribute, say, *func_size*. For each function, the value of this new attribute is given by *end_line* minus *start_line*.

Currently, the only way to add a new attribute to a class schema is by explicitly compiling a new attribute. During query execution, the need to generate new attributes may arise naturally and each time, the query processor must suspend execution, re-compile the class definition with a new attribute, compute the value of the new attribute for each class member, and then resume query processing.

A simple class extension feature can solve this problem. In SCA, we have defined an extend operator, whose very purpose is to add new attributes to objects and compute them on-the-fly. The syntax of the operator is:

$$\text{extend}_{<attribute:=expression>}(< objectcollection >)$$

6 Summary

In this paper, we have discussed the idea of providing an algebraic query language interface for REFINE object bases. The idea is to provide a powerful, interactive query language for source code analysis and program comprehension, which also lends itself well to optimization. SCA is an object algebra suited to this purpose.

REFINE is a powerful and widely-used object-oriented database for programs. We have attempted to identify object-oriented modeling features that are important for querying on source code, but are currently lacking in REFINE. The three features we identify are support for collections as classes, attribute hierarchies, and support for dynamic class extension. Some general-purpose object-oriented databases such as GEMSTONE [4] currently do support some of these features, but they lack REFINE's specialized support for tasks such as parsing, program transformation, code pattern matching, etc., making them difficult to use for analyzing programs. We have shown that the inclusion of the above features is desirable in object-oriented database systems used for representing and analyzing programs in order to efficiently implement powerful languages for making queries on source code.

References

[1] R. Al-Zoubi and A. Prakash. Software Change Analysis via Attributed Dependency Graphs. Technical Report CSE-TR-95-91, Dept. of EECS, University of Michigan, May 1991.

[2] R. Brooks. Towards a Theory of Comprehension of Computer Programs. *International Journal of Man Machine Studies*, 18:543–554, 1983.

[3] E. Buss and J. Henshaw. A Software Reverse Engineering Experience. In *Proc. of the CAS Conference*. IBM Canada Ltd. Laboratory, Centre for Advanced Studies, 1991.

[4] P. Butterworth, A. Otis, and J. Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10):50–77, October 1991.

[5] Y. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[6] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[7] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1986.

[8] J.E. Grass. Object-Oriented Design Archaeology with CIA++. *Computing Systems: The Journal of the USENIX Association*, 5(1):5–67, Winter 1992.

[9] W. Kozaczynsky, J. Ning, and A. Engberts. Program Concept Recognition and Transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[10] H.A. Muller, M.A Orgun, S.R. Tilley, and J.S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[11] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[12] S. Paul and A. Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, pages 463–475, June 1994.

[13] S. Paul and A. Prakash. Supporting Queries on Source Code: A Formal Framework. *International Journal of Software Engineering and Knowledge Engineering*, September 1994. Special Issue on Reverse Engineering.

[14] Reasoning Systems, Palo Alto, CA. *REFINE User's Guide*, 1989.

[15] L.M. Wills. *Automated Program Recognition by graph parsing*. PhD thesis, MIT, 1992.

Session G:
Exploratory Trends and Tools

Determining the Usefulness of Colour and Fonts in a Programming Task

Riston Tapp

Department of Computer Science
University of Waterloo
Waterloo, ON, Canada N2L 3G1
rjtapp@watcgl.uwaterloo.ca

Rick Kazman

Department of Computer Science
University of Waterloo
Waterloo, ON, Canada N2L 3G1
rnkazman@watcgl.uwaterloo.ca

Abstract

This paper reports on the results of an experiment that was run in order to help determine if colour or font size was more useful for displaying code in a programming task, and if so, which was more useful. The null hypothesis of the experiment was that neither colour nor font size were of any benefit to users in programming tasks. The null hypothesis was refuted. It was determined that the colour display mechanism both lessened the time taken to perform a code optimization task, and was preferred by subjects. The use of the font size display mechanism showed no significant benefits.

1 Introduction

Throughout the history of computer programming there has been the desire to make programs as easy to understand as possible. High level languages, structured programming and even the proper naming of variables have all been developed for this purpose. With the development of laser printing and terminals capable of displaying code and graphics in various sizes and colours, new media through which to display computer programs are available. No longer is program display limited to line-oriented textual displays or line printer output. Of course, with these new media come new problems: with all of these new sizes, shapes and colours with which to play, how can these abilities be used to maximum effect?

There have been two techniques which have utilized the powers of graphical display. One technique is to present information to the user which is additional to, or different from the code being displayed. For instance, graphical overviews of a program can be displayed in addition to the program code itself. The other technique is to present additional information about the code artifact to the user as *part* of the displayed code. A simple forerunner of this technique is code formatting (as exemplified by the *vgrind* and *lgrind* systems in Unix).

This paper is concerned with the second technique of program presentation. Specifically, we are interested in the use of colour and font size for the display of computer program code. Both colour and font size are tools that could not be widely used until improved display mechanisms became common. Both colour and font seem to be potentially useful devices with which to improve code display. But are they really?

The subject of this paper is an experiment that was developed in order to help answer just this question. A 3x2 factorial experiment was run, with 3 different methods of displaying code and 2 different programming tasks used. One of the display methods used colour, another used font size, and the last used no special display tool (in order to obtain the effects due to colour, font size, and as a control case, respectively). The programming tasks included a code coverage task (a task ensuring that every line of given code was executed) and a code optimization task (a task to make given code run as fast as possible).

There have been other papers in which code display using multiple colours and font sizes has been discussed. However, whenever differing font sizes or colours have been used in code presentation, either the testing performed was not done in a manner by which the individual influences of colour and/or font size could be obtained, or no experimentation was even performed to begin with. This paper attempts to remedy these oversights. Indeed, the null hypothesis of the experiment discussed in this paper is that colour and font size in code display make no difference in programming tasks. Through the exploration of this

0-8186-5647-6/94 $04.00

hypothesis, this paper will help to determine which of colour or font size is more important for code display, if either of these display methods are of benefit at all.

2 Background

Before discussing this paper's experiment, some of the previous papers on the subject of giving additional information to the user via changes to the program code shall be reviewed. To begin, the idea of using changes in the code itself to signify meaning has three basic stages of growth. These stages include: code formatting, using colour, and visual coding.

One of the first methods of improving upon the look of code was the formatting of code. Many studies have been done on this subject. For instance, Love [5] determined that paragraphing of source code had no effect on subjects. Miara, Musselman, Navarro and Shneiderman [6], meanwhile, showed that certain levels of program indentation aided in program comprehension and user satisfaction. Even the use of capitalization was deemed useful, as Payne, Sime and Green [10] experimented by making the operation codes of a simple command language be in upper case, thereby gaining a decrease in operational errors. Perhaps the efforts of code formatting can best be summed up by citing the work of Oman and Cook [8], however. In their study they included the aspect of commenting code with certain elements of source code formatting in order to derive a book format paradigm of code formatting. Through their use of this paradigm in four separate experiments, they were able to show significant improvement to both program comprehension and maintenance efforts.

As soon as the capability of using colour arose, there were those who rushed to use it. Examples of the early use of colour as an aid in coding occur in the field of error reporting. Both the work of Reynolds [12] and of Oberg and Notkin [7] used colour to report errors to users in a method akin to prettyprinting. In addition to using sound and regular error messages, when users entered syntactically correct but semantically incorrect input into the Reynolds system, they were able to see this error by the colour of the input being different to what it should have been (since certain types of input would be mapped to certain colours). The Oberg and Notkin system, however, used a type of lazy evaluation of code as it was typed into the editor, allowing errors to gradually change in colour according to the age of the error. The user could then fix the errors according to personal preference, without forcing the user to correct the mistakes in the middle of a task. Unfortunately, however, neither of the Reynolds or Oberg and Notkin studies produced statistical evidence of the benefits of their methods.

Visual coding deals with the act of adjusting the look and feel of code. It involves the use of colour and font styles and sizes along with code formatting in order to present code in a meaningful way. Practitioners of this method of using the code itself to impart the meaning of a program include Gellenbeck and Cook [4] and Baecker [1] and Baecker and Marcus [2], [3]. Gellenbeck and Cook investigated the effectiveness of including module header comments and mnemonic module names, along with the use of a larger font size for the module headers, in code. The presence of all three factors aided program comprehension, but the addition of larger font sizes for the headers was only a very modest part. Baecker and Marcus developed a methodology for the use of visual coding. Indeed, they utilized every aspect of visual coding; however, they did not provide any statistical evidence of the effects of using their system. Furthermore, determining what detail of the visual coding causes which result in their system would be difficult. Both the Gellenbeck and Cook and the Baecker and Marcus studies are good beginnings, however, into the realm of visual coding.

3 Method

A between-subject, 3x2 factorial experiment was run. 39 subject had to perform two separate programming tasks on a computer and then fill out a questionnaire about the experiment. One task was to ensure the execution of every line (i.e., ensure code coverage) of a 259-line program via a test suite. The other task was to optimize a 549-line program with respect to speed. The questionnaire's function was to get the subjects to rate their experiences of the experiment. The null hypothesis of the experiment was that neither colour nor font size would be of any significant value for displaying code in a programming task.

The source code for both tasks was to be presented to each subject in one of three ways. One way was to display the code as it would normally be displayed, in the text editor of choice for the specific subject. Another way was to use varying font sizes for the code when it was to be displayed. For instance, if the task being performed was the coverage task, the subject would get to see the code displayed in two different font sizes—a large font size for lines of code that had been covered by their test suite, and a small font size for code that had not. If the task being performed was instead the optimization task, the code would be

displayed to the subject in font sizes ranging from a very small font size for lines of code that had been executed very few times to a very large font size for lines of code that had been executed many times. The third way in which code was to be displayed to a subject was to use multiple colours. Just as for the font size display method, if the subject was performing the coverage task, code would be displayed to the subject in two different colours—blue for lines of code that had been covered and gold for lines of code that had not. For the optimization task, the colours of the different lines of code would range from light blue for code that had been executed only a few times, through purple and on to red for lines of code that had been executed frequently.

Thus the experiment was set up to compare and contrast the use of colour versus font size versus the normal display method for code. Both the use of colour and the use of font size imparted extra information to a subject than without their use. Whether or not this extra information was successfully conveyed remained to be seen. To aid in this decision, the following measurements were taken. For the code coverage task measurements where taken to record: (a) the time taken to complete the task, (b) the accuracy with which coverage was obtained, and (c) the number of times the code was displayed. For the optimization task the same measurements were also taken, except that measurement (b) was instead a measurement of the speed of the optimized code. Through analysis of the results of these measurements, some determination of the usefulness of the display tools was possible.

3.1 Subjects

Thirty nine experienced C programmers at the University of Waterloo served as subjects. Most were graduate students, and all were at least in the third year of an undergraduate university program. Subjects were paid for their participation.

3.2 Materials

Preparation of the materials used in the experiment involved four main duties. They were: (a) preparation of the source code for the coverage task, (b) preparation of the source code for the optimization task, (c) preparation of the code display mechanism, and (d) preparation of the questionnaire.

The code for which the subjects were to provide code coverage was a simple triangle classification program. The user of the program would enter three numbers which were interpreted as the three sides of a triangle. The program would classify the triangle as equilateral, isosceles or scalene. The program was written in C and consisted of 259 lines of code and comments in five routines.

The code which the subjects were to optimize was an implementation of a KWIC Index Production System [9]. It was also written in C and consisted of 549 lines of code and comments in eleven routines. A separate input file was also created (unlike the input file for the coverage task, input for this task was static).

Because the code was to be optimized with respect to speed, it was deliberately written so as to perform poorly with respect to runtime efficiency. Indeed, the program was intentionally written with many inefficiencies. The reason for this design was that it was desired for the code optimization task to be task intensive rather than search or compilation intensive. In other words, it was desired that the subjects spend their time actually improving the code rather than searching for subtle improvements to the code or trying to get difficult improvements to compile.

To this end, a small palette of inefficiencies was included in the optimization task code. These inefficiences included such things as poor orderings of if/else constructs, placing code within loops which should precede the loops, choosing the wrong sorting algorithm and using inefficient data structures.

The preparation of the code display mechanism varied slightly depending on whether the code to be displayed was for the coverage task or for the optimization task, and on whether the code was to be displayed as usual, with varying font sizes, or with varying colours. No matter what the code was or how it was to be displayed, however, it still basically involved a two step process.

The first step of the process was to obtain *profile* information of the code in question: the number of times each line of code had been executed during the previous invocation of the compiled code. The specialized display mechanisms needed this information for both tasks, and it is readily available using any code profiler. In our case we used the Unix *tcov* command.

The next step of the display process differed slightly depending on the purpose of the code and how it was to be displayed. The step itself, however, involved only a simple conversion of the profile information (along with the original code) obtained from the first step, into a tagged text file. This tagged text file was tagged specifically for use by the text viewer lector (lector [11] is an X11 application which displays tagged text, allowing text to be viewed in multiple font sizes and colours, among other things), so that lines of code

would be displayed in the proper font size or colour corresponding to their profile information. Once this conversion was performed, all that remained to do in order to display the text was to run lector.

The only question that remained at this point was the question of what font sizes and colours were desired. For the code coverage task it was decided that the small font size (uncovered) would be Helvetica-12 and the large font size (covered) would be Helvetica-20. The corresponding colours were to be gold and blue. The reason for these choices was that the disparity between the two font sizes and the two colours needed to be distinct and the choices made seemed to meet this goal. For the code optimization task, the font sizes were chosen to range from Helvetica-8 to Helvetica-34 and the corresponding colours ranged from light blue through purple on to red. The main goal for deciding these ranges were to get a large enough range. For the font sizes, this was easy (there were plenty from which to chose). It wasn't as easy for the colours. Originally it was desired to only use one colour and to range from light to dark. However, the choices of shadings for one colour were not distinct enough, so the multicolour range was chosen, with attention paid to RGB values and visual experience when making the choice.

Thus the process of converting the profiled code into lector tagged text was a simple one. This profiling, lector conversion and display process was automated for user testing, and so was unseen by users.

For the purposes of our experiments, we also included, as a control group, the use of a standard display mechanism (the user's favourite text editor). Neither the profile information nor the lector conversion process was necessary for this display method.

The questionnaire was prepared in order to determine the subjects' likes and dislikes of the the display methods used, and of the experiment in general. It consisted of 10 questions asking the subject to rate (on an integer scale of 1 to 5, 1 being poor, 5 good) their experiences with the experiment, and their abilities with the tasks. The first eight questions dealt with rating the two new display methods introduced in the experiment, four similar questions for each display method. The remaining two questions asked the subjects to rate their abilities, one question per task.

3.3 Procedure

The procedure of the experiment consisted of four parts. Theses parts were: (a) orienting the subject, (b) the subject performing the first task, (c) the subject performing the second task, and (d) the subject completing the questionnaire and being debriefed. An experimenter was present during the experiments.

Once a subject arrived, he or she read a cover letter explaining in general terms what the experiment was about. Then the subject was randomly assigned a display method (either the normal display, font size display, or colour display) for each task. The one stipulation was that no one subject used the same display method for both the coverage and optimization tasks. As well, approximately half of the 39 subjects performed the code coverage task first, and then the code optimization task. The remaining subjects performed the tasks in the reverse order.

As soon as a subject was ready to perform the first task, he or she was given sheets of paper describing the task in greater detail. (For the remainder of this section, the first task will refer to the code coverage task and the second task to the code optimization task. Of course, this ordering was not always the case.) After reading these papers, the subject was allowed to ask questions about any troubling points and was given further oral advice. Then the first task was begun.

The time limit for the code coverage task was nominally set at 25 minutes. Subjects were allowed to finish within this time limit, but not to go over it. The actual course of the task went as follows. The subject was allowed two computer windows in which to operate. One window was to be used to run the display mechanism (or to view the code in a text editor of the subject's choice, if the display method randomly picked was not the font size or colour method). The other window was to be used to create the input text file containing the test suite with which the subject was to obtain code coverage.

The actions of the subject for the first task were: to (a) create a trial test suite, (b) run the display mechanism (or simply run the compiled code if there was no display mechanism to be used for this task), (c) view the newly displayed code in the lector text viewer (or view the output of the code coverage program if no display mechanism was being used in the trial), and (d) repeat this process if complete coverage was not obtained. As soon as a subject felt that he or she was finished the task, he or she notified the experimenter and the appropriate measurements were taken. Throughout the task, the subjects were allowed to ask questions of the experimenter, as long as these questions were of a procedural nature.

At this point the subject was ready to perform the second task. As with the first task, the subject was given sheets of paper explaining the task in greater detail than the cover sheet previously read. Questions

were allowed, and further oral advice was given.

The time limit for the code optimization task was 50 minutes (the whole experiment had a time limit of an hour and a half). If subjects were not finished at the end of this time, they were allowed extra time to make sure that their code was able to be compiled (without compilation there would be no way in which to determine how fast the optimized code ran).

Once the subject was ready to begin, the following procedure was followed. Again there were two windows allowed the subject, one window in which to optimize the code, the other window in which to run the display mechanism (or run the compiled code if no special display was allowed). The subject would then: (a) run the display mechanism (or run the code), (b) view the newly displayed code in the lector text viewer (or view the output of the KWIC Index program), (c) make changes to the code in order to optimize it with respect to speed, and (d) repeat this process if the subject felt that the code could be further optimized. As soon as the subject felt that the task was finished, he or she notified the experimenter and the appropriate measurements were taken. As with the first task, the subjects were allowed to ask questions of the experimenter throughout the second task, this time of both a procedural and a code comprehensive nature. (It was desired that the subjects understand as quickly as possible the workings of the code, to make sure that the user spent his or her time on the optimization task, rather than on program comprehension.)

Once both tasks were finished, subjects filled out the questionnaire and were debriefed. Once the questionnaire was completed, the subjects were paid and any remaining questions about the experiment were answered. Subjects were promised a synopsis of the experiment if they so desired.

4 Results

This section is divided into three parts. First, results of the data collected from the code coverage task experiment are given. Next, results of the data collected from the code optimization task are reported. Last, results of the questionnaire are presented. All of the results will be discussed in Section 5.

4.1 Code coverage task

Three dependent variables were measured for each of the subjects during the code coverage experiment. These variables were the time taken to complete the task (in minutes), the accuracy of the subject in completing the task (determined by the number of lines of code not covered), and the number of times that the code was viewed (or run) in completing the task. The result means as grouped by viewing method can be seen in Table 1.

	Colour	Font Size	No Format
Time taken	14.167	13.917	16
Lines missed	0.833	0.583	0.769
Iterations	5.5	5	2.769

Table 1: Result means for the code coverage experiment

All of the subjects finished the code coverage task within the allotted time of 25 minutes. Two results (one each from of the colour display method and the font size method) were discarded due to failure of the subjects involved to understand the task correctly. For the remaining results, analysis of variance tests were performed across the display method types.

There were no statistically significant differences between the display methods for the first two measurements. Comparing the average time taken to finish the task gave a result of $F(2,34) = 0.67$, $p > 0.05$. Indeed, the average times taken to finish the task were very similar. Comparing the accuracy of the various display methods, meanwhile, gave a result of $F(2,34) = 0.24$, $p > 0.05$. The average accuracy of the subjects using the colour display method could be improved (to 0.3 lines missed) by removing two outlier results, but even so, this improvement still did not give significant differences. Variance between subjects within a display method was just too high.

For the measurement of the number of iterations of viewing code, however, the availability of a new display methods did result in an increase of their use. The result of this analysis of variance was $F(2,34) = 3.48$, $p < 0.05$. The iteration process (create data, run, display) for subjects using the colour or font size display methods was performed significantly more often than the corresponding process (create data, run) for subjects using no specialized code viewer (almost twice as often).

4.2 Code optimization task

As with the code coverage task experiment, three dependent variables were measured for each subject in the code optimization task experiment. The only difference in variables measured from the previous task

	Colour	Font Size	No Format
Time taken	41.083	49.615	54.077
Speed increase	609.3	495.538	543.333
Iterations	6.333	9.077	11

Table 2: **Result means for the code optimization experiment**

	Colour	Font Size
Like using	4.23	3.76
Aid in task	4.08	3.85
Ease of use	4.36	4.24
Use again	4.36	3.71

Table 3: **Result means for the questionnaire display method questions**

to this task was that, instead of accuracy being measured, the speed increase of the optimized code (in μsec) was measured in its place. The result means can be seen in Table 2.

For this task, 22 out of the 39 subjects took longer than the allotted 50 minutes. All subjects were warned when their time was nearing its end, and, if they were not finished at the end of 50 minutes extra time was given to allow the subjects to obtain compilable code. Even so, 4 results were discarded due to code not compiling when speed testing was performed, three for the colour display method and one for the text editor display method. Analysis of variance tests were performed on the remaining results.

Unlike the code coverage task, the code optimization task showed significant differences when the average times taken via display method to complete the task were compared. Analysis of variance for this measurement gave a result of $F(2,35) = 4.07$, $p < 0.05$. As can be seen from Table 2, those subjects that used the colour display method finished the task noticeably faster than subjects using either of the other two display methods (17 percent faster than those using the font size method, 25 percent faster than those using the test editor method).

Comparing the average speed gained by the optimized code, however, showed no significant difference between display methods, $F(2,32) = 0.77$, $p > 0.05$. The variance in the speeds obtained by subjects for each display method were very high, thus hampering any possible significance.

As for the previous task, the comparison of the number of code viewing iterations between subjects utilizing the different viewing methods showed significant differences. This time the results were $F(2,35) = 2.97$, $p < 0.10$, and those subjects who used the specialized display methods (especially the colour display method) performed the iteration process less often than their text editor counterparts.

4.3 Questionnaire

The four display method questions presented in the questionnaire asked the subjects to rate the following areas: (a) how much they liked using a particular display method; (b) how much the display method aided in the task; (c) how easy it was to use and understand the display method; and (d) whether or not the subject would use the display method again if possible. The mean results of the questionnaire experience rating questions can be seen in Table 3.

In all cases, the average rating for the colour display method is higher than for the font size method. However, the difference in ratings is statistically significant only for the first and fourth questions. Analysis of variance for the first question differences gives $F(1,49) = 3.60$, $p < 0.10$, while for the fourth question $F(1,47) = 6.07$, $p < 0.025$. Subjects enjoyed using the colour method and would like to use it again, even though there were no statistical differences in terms of the perceived benefits of the method, as determined by the other two questions.

As far as the subjects' skill ratings are concerned, not much information was obtained. Scatterplot diagrams relating the subjects' skill ratings to actual performance showed no discernable patterns. In fact it was often the case that a subject with a poor rating would outperform more highly rated subjects. The mean skill rating for the code coverage task was 3.64, $\sigma = 0.81$, while the mean skill rating for the code optimization task was 3.09, $\sigma = 1.05$.

5 Discussion

The results were not exactly as expected for the code coverage experiment. Although no difference between the specialized display methods was expected, it was thought that the use of either of the colour or font size display would improve on the time taken to complete the task. This result did not occur. Perhaps the task itself was too easy. Indeed, some of the

subjects obtained code coverage through an element of chance, happening to use the correct input without realizing why it was correct. (For example, there was one line of code for which it was necessary to use a zero as input, rather than a negative number, even though the code appeared to accept either. Many subjects first tried negative input, only to have to backtrack through the code to determine why a zero had to be used. The fortunate subjects were those that happened to use zero first.) Even so, there was evidence of the specialized display methods being used to advantage. Indeed, both the colour display and the font size display resulted in approximately twice as many iterations (create data, run, display) as compared with the ordinary text editor display method (create data, run). This result, coupled with the statistically equal results of accuracy and time taken between display methods, seems to show that the use of the display tools replaced, to some extent, thinking on the subjects' part. The subjects using the enhanced displays obtained the same results as their text editor counterparts, but with greater reliance on the computer.

Much as for the code coverage results, the results for the code optimization task were also opposite to what was expected. For this task it was felt that the font size display method would be the most useful, since the differences in code execution speed translated very naturally into code font size differences (large lines were frequently executed, tiny lines were seldom executed). However, it appears that the colour range chosen, from blue to red, also translated quite readily to the optimization problem. In fact, it seemed to translate more readily than the font size method, given its superior results in the time taken to finish the task. (Perhaps the unusual use of screen real-estate by the font size display explains this difference. Many subjects stated that they found the varying line sizes to cause a peculiar (and distracting) arrangement of the code in the display window. Whether or not this unordinary sizing scheme is something to which it simply takes time to get accustomed is not known.)

As far as the speed optimizations are concerned, it seems that all subjects performed the same optimizations no matter what the display method used. This observation would explain the similarities in the speed increases obtained. Perhaps this result is due to the commonality of the artificially induced inefficiencies in the code. All subjects were able to remove these inefficiencies, but those who used the colour display method did so faster.

The results for the use of the visualization tools showed the special display tools (in particular the colour display tool) to have been used less often than code was run when using no special display method for the optimization task. This result is opposite to the corresponding result for the code coverage task. The reason for the decreased number of iterations of viewing the specialized displays when compared to the number of iterations of running the code for this task, however, could be the same as for their increased numbers in the code coverage task. As opposed to having to scan code sequentially and improve upon code as one sees inefficiencies, subjects who used the colour display method could gravitate immediately to the inefficiency 'hot spots' and ignore other, possibly less important ones. Using this methodology, subjects would indeed need to view the code less often than they would normally. Therefore, just as for the code coverage task, the specialized display tools again replaced human effort.

The purpose of the questionnaire was to try to determine subject response to the different display methods introduced in the experiment. As such, the analysis of the questionnaire results showed that the subjects felt that there was no difference between the colour or font size display methods as far as the ease of their use or in the aid they provided to the subject in completing their tasks. Interestingly, however, subjects did feel that they liked using the colour method more than the font size method, and they said that they would enjoy using it again. Even without any performance reasons to use colour, it would seem that there is a reason to do so in terms of user preferences.

6 Conclusions

The purpose of the experiment was to determine whether or not using colour or font sizes for code display was of any benefit to a programmer, and if so, which one was better. As such, the findings of the code display experiment refute the null hypothesis that colour and font size make no difference in code display. Indeed, the results seem to show that the use of colour is the more effective tool for promoting greater understanding of code to a user, and that indeed, using different font sizes is of uncertain advantage. This result must be understood within a context, however. It has not been shown that the use of varying font sizes is of no value—what has been shown is that the use of colour does make a significant difference in a real programming task. It is entirely possible that, under different assumptions, font size could also have been effective, or that colour could have been even more effective.

The statistical results garnered from the experiment show that the use of colour adds to both user performance and user satisfaction. Although subjects did not perform appreciably better for the code coverage task, they did spend less time in finishing the optimization task, and for both tasks subjects expressed their liking of the colour displays.

Subjects expressed their fondness for the font size displays as well. However, colour was distinctly preferred, and the performance results for subjects using the font size display were generally not any different than for those subjects using no specialized display. Subjects did not seem to adjust to the font size displays as quickly as to the colour ones, with many subject expressing a dislike of the spatial changes of the code caused by the varying font sizes. Perhaps the font size mechanism was more suited to the code coverage task than for the optimization task (due to the less size variation involved), but for this task too there was a problem with subjects missing uncovered code.

In the final analysis, it would seem that using font size to express extra information in code is of questionable value, an outcome with which the findings of Gellenbeck and Cook [4] agree.

It is recommended that further research be performed into using colour in the display of code. In particular, study needs to be done into using colour in tasks other than the ones used in this experiment, as well as into determining which colours work best for a specific task. The colour schemes used for the two tasks in this experiment seemed to work well, and perhaps they are as good as any other schemes. However, it needs to be known what colour schemes work best, and if different tasks require different colour schemes, it needs to be known which schemes fit which tasks.

The notion of using typographical aids to improve upon code appears to be a good one. The experimental results discussed in this paper would suggest that colour be one of these aids. More work needs to be done in each of the separate sections of this field, however, before all of the tools can be unified into one pervasive method of displaying code.

References

[1] R. Baecker. Enhancing Program Readability and Comprehensibility with Tools for Program Visualization. In *Proceedings of the 10th International Conference on Software Engineering (Raffles City, Singapore, April 11-15)*. ACM/SIGSOFT and IEEE/CS, New York, 1988, pp. 356–366.

[2] R. Baecker and A. Marcus. Design Principles for the Enhanced Presentation of Computer Program Source Text. In M. Mantei and P. Orbeton, editors, *Human Factors in Computing Systems–III*. North-Holland, Amsterdam, 1986, pp. 51–58.

[3] R. M. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs*. ACM Press, Reading, Massachusetts, 1990.

[4] E. M. Gellenbeck and C. R. Cook. Does Signaling Help Professional Programmers Read and Understand Computer Programs?. In J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*. Ablex Publishing Corporation, Norwood, New Jersey, 1991, pp. 82–98.

[5] T. Love. An Experimental Investigation of the Effect of Program Structure on Program Understanding. In *ACM Sigplan Notices*, 12(3), 105–113, 1977.

[6] R. J. Miara, J. A. Musselman, J. A. Navarro and B. Shneiderman. Program Indentation and Comprehensibility. In *Communications of the ACM*, November 1983, Volume 26, Number 11, pp. 861–867.

[7] B. Oberg and D. Notkin. Error Reporting with Graduated Color. In *IEEE Software*, 9(6), 33–38, 1992.

[8] P. W. Oman and C. R. Cook. Typographic Style is More than Cosmetic. In *Communications of the ACM*, May 1990, Volume 33, Number 5, pp. 506–520.

[9] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. In *Communications of the ACM*, December 1972, Volume 15, Number 12, pp. 98–103.

[10] S. J. Payne, M. E. Sime and T. R. G. Green. Perceptual structure cueing in a simple command language. In *Int. J. Man-Machine Studies* (1984), **21**, 19–29.

[11] D. R. Raymond. *lector—An Interactive Formatter for Tagged Text*. UW Centre for the New Oxford English Dictionary and Text Research, Waterloo, Ont., 1990.

[12] C. F. Reynolds. The Use of Colour in Language Syntax Analysis. In *Software—Practice and Experience*, Vol. 17(8), 513–519 (August 1987).

SFAC, a Tool for Program Comprehension by Specialization

Sandrine Blazy, Philippe Facon
CEDRIC IIE
18 allée Jean Rostand
91025 Evry Cedex, France
{blazy, facon}@iie.cnam.fr

Abstract

This paper describes a tool for facilitating the comprehension of general programs using automatic specialization. The goal of this approach was to assist in the maintenance of old programs, which have become very complex due to numerous extensions.

This paper explains why this approach was chosen, how the tool's architecture was set up, and how the correctness of the specialization has been proved. Then, we discuss the results obtained by using this tool, and the future evolutions.

1. Introduction

In [2], we have presented a new approach to assist in the comprehension of general programs, based on partial evaluation techniques to analyze the behaviour of source code. Here, after a brief recall of that work, we will first discuss the use of the tool, secondly we will show how it has been proved with respect to dynamic semantics, and thirdly we will present the evolutions we are working on.

It is well-known that program comprehension is a tedious and time consuming phase of software maintenance. This is particularly true when maintaining scientific application programs that have been written for decades, such as those developped at EDF. Electricité de France (EDF) is the national French company that produces, distributes and provides electricity to the whole country. As such, EDF has to deal with an extensible amount of computerized scientific applications. These application programs are mainly implemented in Fortran, which is an old-fashioned language, but which is still widely used in industry. The reduction of a program when some of its input variables are known was the main wish of EDF scientists and programmers. These people were faced with so general applicaton programs that existing market tools could not help them to understand their application programs. Thus, we have developped a technique for specializing programs by showing several views, a view representing a program functionality or a specific context.

This approach is complementary to classical techniques of representing programs at higher level of abstractions (reverse engineering techniques). Program specialization has been seldom used for program comprehension. Automatic program specialization has been used (as partial evaluation) in compilation to optimize programs or to generate compilers from interpreters (by partially evaluating the interpreter for a given program) [13].

In [2], we have described our technique for restricting the behaviour of a program to specific values of its input variables. We have shown an example of program reduction and we have formally specified the two main tasks of our specialization process by means of inference rules. The aim of this new paper is threefold: first to explain how our tool, SFAC (Specialization of Fortran programs as an Aid to their Comprehension) is used, and second to show how we have proved the completeness and soundness of the specialization rules we had previously presented. At least, we present the evolutions of our tool.

We focus on general purpose programs that are large and complex and whose application domain models encapsulate several models. This generality is implemented by Fortran input variables whose value does not vary in the context of the given application. We have distinguished two classes of such variables:

- *data about geometry*, which describe the modelled domain. They appear most frequently in assignment statements (equations that model the problem).
- *control data*. These are either *filters* necessary to switch the computation in terms of the context (modelled domain), or *tags* allowing to minimize risks due to precision error in the output values. Control data take a

0-8186-5647-6/94 $04.00

finite number of values; they appear in particular in conditions of alternatives or loops.

As an example illustrated in Figure 1, consider the modelization of a liquid flow along the surface of a nuclear power plant component. That volume is partitioned along the three axes, with a number of partitions of respectively IM, JM and KM. Moreover, the surface being porous, on a regular basis, IPOR is the relative side length of the solid part for each elementary cubic partition. Thus IM, JM, KM and IPOR are data about geometry. Now we have a filter, THERMODYNAMICAL_MODEL, which is the name of the law that characterizes the liquid. We have also a tag, PRESSURE, with integer values that correspond, by a table, to real pressure values, each one with a specific numerical precision.

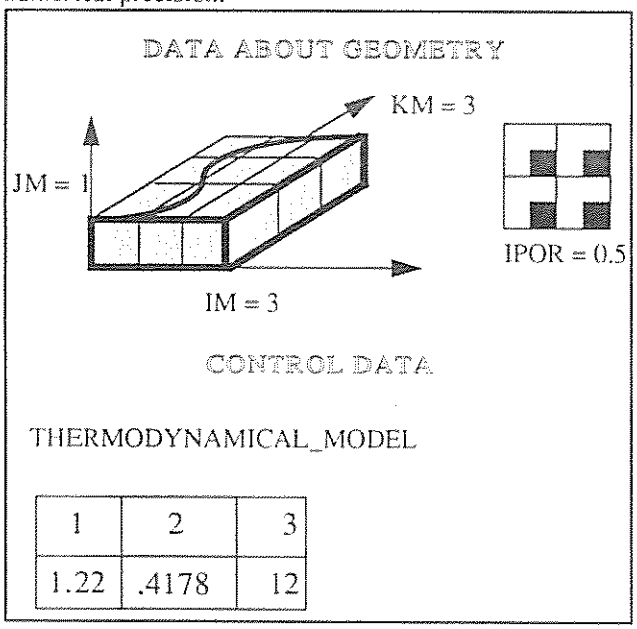


Fig. 1. Data about geometry and control data

Our approach aims at highlighting such input variables in old programs, that have become very complex due to extensive modifications. It led us to develop a technique for program specialization according to specific values of such variables (for instance, the specialization of a 3D-application into a 2D-one by fixing the value of a co-ordinate). This technique simplifies programs that are thus shorter and easier to understand.

What means to simplify a program in that context? We believe that to remove useless code is always beneficial to program understanding. In that case the objective is compatible with that of program optimization (dead code elimination), but this is certainly not the case in general. On the other hand, the replacement of (occurences of) variables by their values is not so obvious. The benefit depends on what these variables mean for the user: variables like PI, TAX_RATE, etc. are likely to be kept in the code; on the contrary, intermediate variables used only to decompose some computations may be not so meanigful for the user, and he may prefer to have them removed.

Replacing variables by their values may lead to dead code (by making the assignments to these variables useless) and thus gives more opportunities to remove code. However, this is certainly not a sufficient reason to do systematic replacement. Of course, even when there is no replacement, the known value of a variable is kept in the environment of our simplification rules, as it can give opportunities to remove useless code, for instance if the condition of an alternative may be evaluated thanks to that knowledge (and thus a branch may be removed).

The benefit of replacement depends not only on the kind of variable but also on the kind of user: a user who knows the application program well may prefer to keep the variables the meaning of which is already known to him; a user trying to understand an application program he does not know at all may prefer to see as few variables as possible. In fact, our experiments have shown that the tool must be very flexible in that respect. Thus, our tool works as follows. There are three options: no replacement, systematic replacement, and each replacement depending on the user.

The remainder of this paper is organized as follows. In section 2, we recall briefly how we have specified the specialization process. Then, we detail in section 3 how we have implemented our method for specializing Fortran programs. Section 4 explains how to prove the soundness and completeness of the specialization with respect to the dynamic semantics of Fortran. Examples of proofs for some Fortran statements are given in appendix. Section 5 presents conclusions and future work.

2. Specification of the Specialization

To specify the automatic program specialization, we have used inference rules operating on the Fortran abstract syntax and expressed in the natural semantics formalism, augmented with some VDM [9] operators. Natural semantics [11] has its origin in the work of G.Plotkin ([8], [14]). Under the name "structured operational semantics", he gives inference rules as a direct formalization of an intuitive operational semantics: his rules define inductively the transitions of an abstract interpreter. Natural semantics extends that work by applying the same idea (use of a formal system) to different kinds of semantic analysis (not only interpretation, but also typing, translation, etc.).

For every Fortran statement, we have written rules that define inductively the automatic program specialization [5]. Figure 2 gives two examples of these specialization

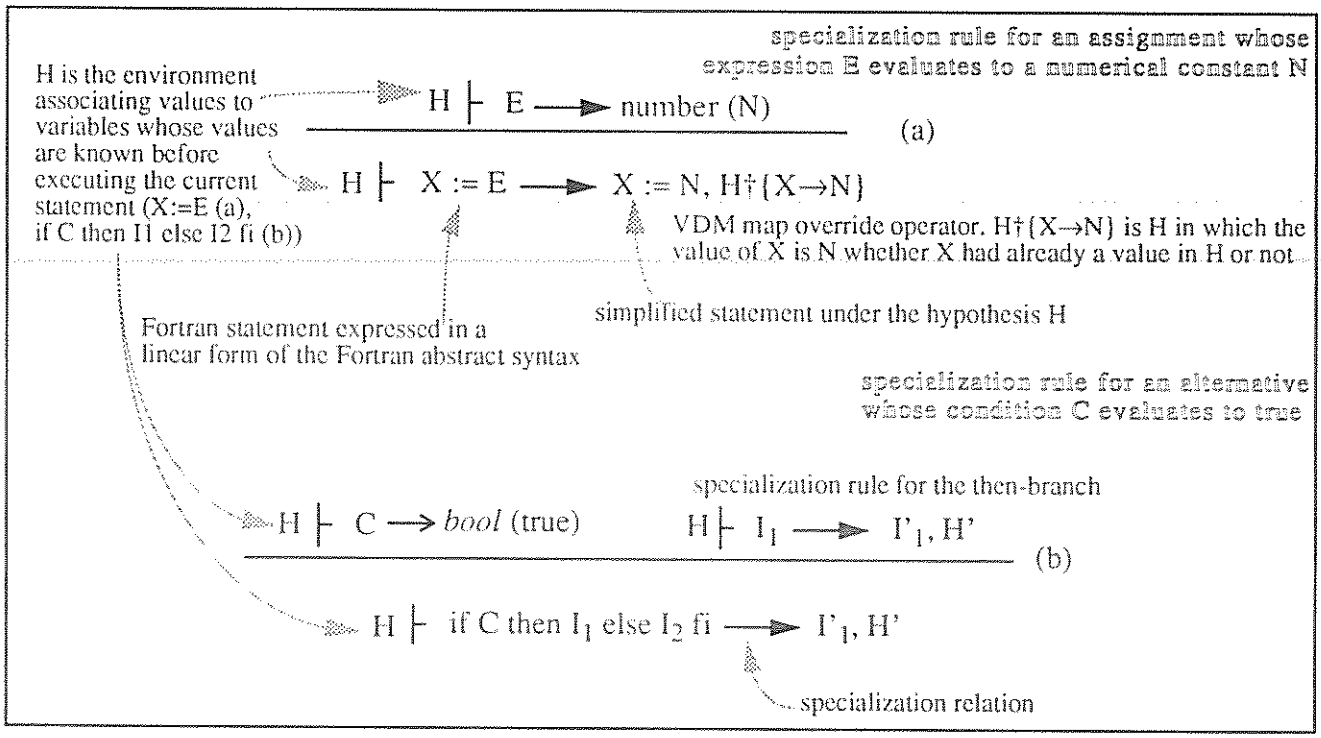


Fig. 2. Examples of specialization rules

rules. Rule (a) expresses the specialization of an assignment whose condition evaluates to a numerical constant and rule (b) expresses the specialization of an alternative whose condition evaluates to true.

3. The Specializer (SFAC)

We have implemented our method for automatic specialization of general programs by using the Centaur system [3], a generic programming environment parametrized by the syntax and semantics of programming languages. This section describes the overall architecture of our tool, SFAC. Then, it gives some quantitative results.

3.1. Architecture of SFAC

When provided with the description of a particular programming language, including its abstract and concrete syntaxes and semantics, Centaur produces an environment specific to that language. The resulting environment consists of a structured editor, an interpreter/debugger and other tools, together with an uniform graphical interface. In Centaur, program texts are represented by abstract syntax trees. The textual (or graphical) representation of abstract syntax trees nodes may be specified by pretty-printing rules. Centaur provides however a default representation.

We have used such a resulting environment, Centaur/Fortran, to build our automatic specializer. From Centaur/Fortran, we have implemented an environment for automatic specialization of general Fortran programs. Figure 3 shows the overall architecture of our system. In this Figure, Centaur/Fortran is represented by the grey part. It consists of :

- a Fortran parser and a tree builder, that have been generated by Centaur from a concrete syntax and an abstract syntax of Fortran
- a Fortran pretty-printer that displays Fortran abstract syntax trees as Fortran texts. Their layout may be customized.

A language for specifying the semantic aspects of languages called Typol is included in Centaur, so that the system is not restricted to manipulations that are based solely on syntax. Typol is an implementation of natural semantics. It can be used to specify and implement static semantics, dynamic semantics and translations. Typol specifications operate on abstract syntax trees; they are compiled into Prolog code. When executing these specifications, Prolog is used as the engine of the deductive system. A Typol program is roughly an unordered collection of axioms and inference rules. We have written Typol rules that implement the automatic specialization. These rules are very close to the ones we have formally specified.

Given a Fortran program or subroutine and some

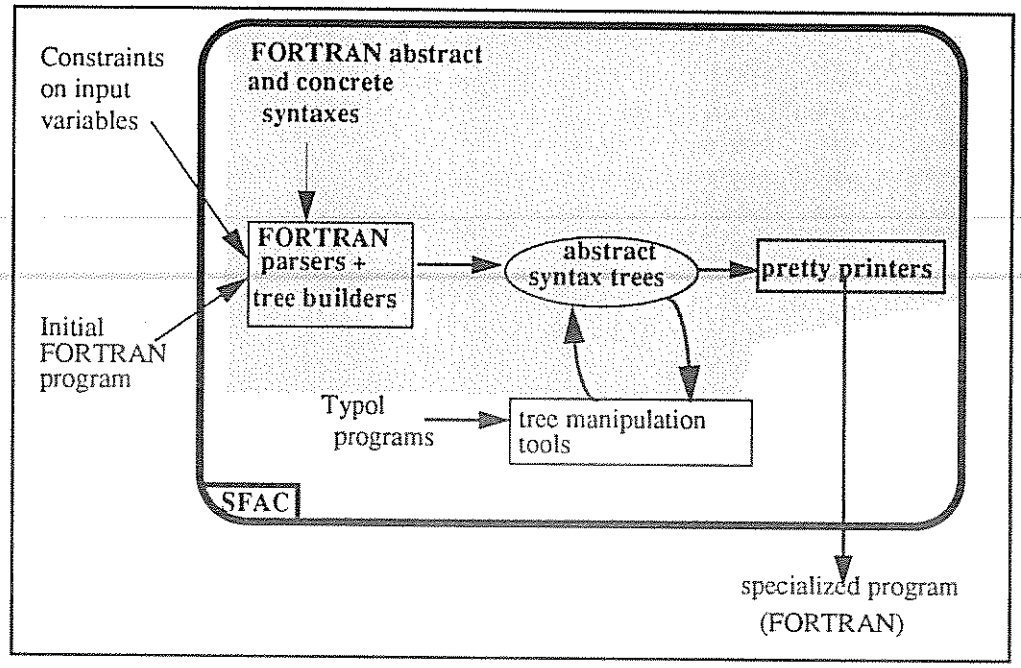


Fig. 3. The Centaur/FORTRAN environment

constraints on input variables (expressed as a list of equalities between variables and constants), the corresponding residual program is obtained by clicking in the window containing the Fortran code. The tool may be used in two ways: by visual display of the residual program as a part of the initial program (for documentation or for debugging) or by generating this residual program as an independent (compilable) program.

Note that some users found our tool interesting also for global program comprehension: in that special case they have not in advance known input variables, but they ask for interesting variables for specialization. Thus, we aim at providing such variables. A set of variables is interesting for specialization specially if their values at the program entry determines the value (true or false) of the conditions of somes alternatives. To detect such variables, we will perform backwards analysis, very close to those performed in program slicing [7]. We will also take into account the importance of the alternatives (number of statements in each branch).

3.2. Quantitative results

We have written about 200 Typol rules to implement our specializer. 10 rules express how to reach abstract syntax nodes representing simplifiable statements. 90 rules perform the normalization of expressions (this normalization allows to propagate constant values). Among the 100 rules for simplification, 60 rules implement the simplification of expressions. The 40 other rules implement the statements simplification. We have written about 25 Prolog predicates to implement the VDM operators we have used to specify the simplification. Thus, these operators are used in Typol rules as in the formal specification of the simplification.

The specializer may analyze any Fortran program, but it specializes only a subset of Fortran 77 (for large scientific applications at EDF, Fortran 77 is used exclusively to guarantee the portability of the applications programs between different machines). This subset is a recommended standard at EDF. For instance, it does not analyze any goto statement (they are not recommended at EDF), but only goto statements that implement specific control structures (e.g. a while-loop).

The average initial length of programs or subroutines we have analyzed is 100 lines of FORTRAN code, which is lengthier than the recommended length at EDF (60-70 lines). The reduction rate amounts from 25% to 80% of lines of code. This reduction is specially important when there is a large number of assignments and conditionals. This is the case for most subroutines implementing mathematical algorithms. For subroutines whose main purpose is editing results or calling other subroutines, the reduction is generally not so important.

4. Proof of Soundness and Completeness of the Specialization

For a tool like SFAC, it is essential to be sure about the correctnees of the specialization, specially if it is used for generating independent (compilable) programs. Our aim in this section is to show how to prove that the specialization we have specified is correct, with respect to the dynamic semantics of Fortran, given in the natural semantics formalism. Recall that partial evaluation of a program P with respect to input variables $x_1, ..., x_m, y_1, ..., y_n$ for the values $x_1 = c_1, ..., x_m = c_m$ must give a residual program P', whose input variables are $y_1, ..., y_n$ and the executions of $P(c_1, ..., c_m, y_1, ..., y_n)$ and $P'(y_1, ..., y_n)$ produce exactly the same results.

We will show that this is expressed by two inference rules, one expressing soundness (each result of P' is correct with respect to P) and one expressing completeness (each correct result is computed by P' too). As P and P' are deterministic, we could have only one rule using equality, but the demonstration of our two rules is not more complicated and is more general (being also applicable for non-deterministic programs).

To prove the specialization, we need a formal dynamic semantics of Fortran and we must prove the soundness and completeness of the specialization rules with respect to that dynamic semantics. To express the dynamic semantics of Fortran, we use the same formalism (natural semantics) as for specialization. Thus, the semantic rules we give have to generate theorems of the form

$$H \overset{sem}{\vdash} I : H'$$

meaning that in environment H, the execution of statement I leads to the environment H' (or the evaluation of expression I gives value H'). These rules are themselves not proved: they are supposed to define ex nihilo the semantics of Fortran, as G.Plotkin [14] and G.Kahn [11] did for other languages like ML.

To prove these rules would mean to have an other formal semantics (e.g. a denotational one) and prove that the rules are sound and complete with respect to it. But there is no such official semantics for Fortran. Thus that proof would rather be a proof of consistency between two dynamic semantics we give. That is outside the scope of our work: we want to prove consistency between simplification and dynamic semantics, not between two dynamic semantics.

Now how can we prove that the specialization system is sound and complete with respect to the dynamic semantics system? Instead of the usual situation, that is a formal system and an intended model, we have two formal systems: the specialization system and the dynamic semantics system (noted *sem*). A program P is specialized into P' under hypothesis H_0 on some input variables if and only if

$$Ho \vdash P \rightarrow P'$$

is a theorem of the specialization system.

Let us call H the environment containing the values of the remaining input variables. Thus, $H_0 \cup H$ is the environment containing the values of all input variables. With that initial environment, P' (respectively P) evaluates to H' if and if

$$Ho \cup H \overset{sem}{\vdash} P' : H' \quad \text{(respectively } Ho \cup H \overset{sem}{\vdash} P : H')$$

is a theorem of the dynamic semantics (*sem*) system.

Now, soundness of specialization with respect to dynamic semantics means that each result computed by the residual program is computed by the initial program. That is, for each P, P', H_0, H, H': if P is specialized into P' under hypothesis H_0 and P' executes to H' under hypothesis $H_0 \cup H$, then P executes to H'under hypothesis $H_0 \cup H$. Thus soundness of specialization with respect to dynamic semantics is formally expressed by the first rule of Figure 4.

Completeness of specialization with respect to dynamic semantics means that each result computed by the initial program P is computed by the residual program P'. Thus, it is expressed by the second inference rule of Figure 4. In fact, our approach to prove specialization is very close to the approach of [5] to prove the correctness of translators: in that paper, dynamic semantics and translation are both given by formal systems and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules (that are proved by induction on the length of the proof; here we will use rule induction instead).

Note that both rules are not the most restricting rules (for instance their initial environment is $Ho \cup H$ and not only H, to allow partial simplification).

$$\frac{Ho \vdash P \rightarrow P' \qquad Ho \cup H \overset{sem}{\vdash} P' : H'}{Ho \cup H \overset{sem}{\vdash} P : H'} \; \begin{matrix} (soundness) \\ (completeness) \end{matrix}$$

$$\frac{Ho \vdash P \rightarrow P' \qquad Ho \cup H \overset{sem}{\vdash} P : H'}{Ho \cup H \overset{sem}{\vdash} P' : H'}$$

Fig. 4. Soundness and completeness of the program specialization

The proof that both rules hold is given in [1].

5. Conclusion

Specialization has given satisfactory results for facilitating program comprehension. Our tool, SFAC, has performed important reductions by specialization of Fortran programs and subroutines. That tool has been proved by rule induction given the dynamic semantics of Fortran.

An industrialization of our tool is planned in the framework of a cooperation between EDF, CEDRIC IIE and Simulog, a company that provided us with some basic tools including Centaur/Fortran. To obtain an insdustrial tool from SFAC, we have to extend the latter in four main ways:

- to perform interprocedural analysis,
- to take into account some new operators of Fortran 90,
- to accept as specialization criteria more general constraints than only equailities between variables and constant values. Thus, the tool will propagate relational expressions, such as y = z and a> 7*b+5.
- to suggest pertinent variables for specialization, as explained previously (page 4).

For performance reasons, the industrial tool will not be based on Prolog operating on anstract syntax trees, but in Lisp operating on graphs representing control and data flow information. Our specialization rules are taken as specification by the team that will do the industrial implementation.

References

[1] S.Blazy La spécialisation de programmes pour l'aide à la maintenance du logiciel, Ph.D. Thesis, CNAM, Paris, December 1993.

[2] S.Blazy, P.Facon, *Partial evaluation as an aid to the comprehension of Fortran programs* IEEE Workshop on Program Comprehension, Capri, July 1993, 46-54.

[3] *Centaur 1.1 documentation*. INRIA, January 1990.

[4] P.D.Coward, *Symbolic execution systems - a review*. Software Engineering Journal, November 1988, pp.229-239.

[5] J.Despeyroux, *Proof of translation in natural semantics*. Symposium on Logic in Computer Science, Cambridge USA, June 86.

[6] C.Consel, S.C.Khoo, *Parametrized partial evaluation*. ACM TOPLAS, 15(3), July 1993, pp.463-493.

[7] K.B.Gallagher, J.R.Lyle, *Using program slicing in software maintenance*. IEEE TOSE, 17(8), August 1991, pp.751-761.

[8] M.Hennessy, *The semantics of programming languages*. Wiley eds., 1990.

[9] C.B.Jones, *Systematic software development using VDM*. Prentice-Hall, 2nd eds., 1990.

[10] N.D.Jones, P.Sestoft, H.Sondergaard, *MIX: a self-applicable partial evaluator for experiments in compiler generation*. Lisp and Symbolic Computation 2, 1989, pp.9-50.

[11] G.Kahn, *Natural semantics*. Proceedings of STACS'87, Lecture Notes in Computer Science, vol.247, March 1987.

[12] R.Kemmerer, S.Eckmann, *UNISEX: a UNIX-based Symbolic Executor for Pascal*. Software Practice and Experience, 15(5), 1985, pp.439-457.

[13] U.Meyer, *Techniques for evaluation of imperative languages*. ACM SIGSOFT, March 1991, pp.94-105.

[14] G. Plotkin, *A structural approach to operational semantics*. Report DAIMI FN-19, University of Aarhus, 1981.

Theory and Practice of Middle-Out Programming to support Program Understanding

K.H. Bennett
Department of Computer Science
University of Durham
Durham DH1 3LE, UK

M.P. Ward
Department of Computer Science
University of Durham
Durham DH1 3LE, UK

Abstract

Theories of top-down and bottom-up program comprehension have existed for several years, but it has been recognised that understanding rarely happens in practice in such a well-ordered way. The paper describes recent work and results at Durham on what is termed middle-out programming. The objective is to avoid the problems of top-down and bottom-up approaches, by designing a very high level language specific to the application domain. Domain knowledge is captured in the design of this language, which retains a strong formal basis. This paper takes the view that software engineering will become strongly application domain based, and that knowledge representation of the domain will be a crucial factor in supporting program comprehension.

An example of using this approach in the design of a large software system is presented. The main gain achieved in comprehension is through a large reduction in system size, and through a domain specific language. This separates the comprehension problem into two: understanding the requirements definition in a very high level domain language; and understanding the implementation of the language. The main novel features of our approach are: the use of domain specific very high level representations to avoid top-down and bottom-up problems; program comprehension based on a formal language approach; the ability to provide a theoretical basis for user-enhanceable systems; the simplification of the capture of domain knowledge; and the concurrent engineering of the development and requirements stages.

1 Introduction

Most of the problems concerned with program comprehension derive from understanding large systems (see [1] for a useful review). Small scale or toy problems are generally uninteresting and any general solution to program comprehension must address the issue of scale. Brooks has observed [2] that there are four key properties of large software systems which distinguish them from small systems. First of all, large systems are complex, and this is an essential property in that it cannot be abstracted away from. This leads to communication problems amongst developers, to enormously complicated state explosions, and to the lack of a coherent overview of the system, so that maintaining conceptual integrity becomes increasingly difficult. Large scale systems must also conform to complex human institutions as for example in the automation of the tax system.

Any useful large scale system will also be subject to constant change as it is used, this being generally referred to as software maintenance. Almost all software maintenance is currently carried out at the source code level. Finally, unlike many other engineering artifacts, there is no geometric or other representation of the abstract artifact of software. There are several distinct but interacting graphs of links between parts of the system to be considered including, for example, control flow, data flow dependency, time sequence etc. One way to simplify these in an attempt to control the complexity and make the program understandable is to cut links until the graphs become hierarchical structures [13].

One of the major problems in developing large scale systems is that there is often a thin spread of domain knowledge among software developers in many projects [4]. Often, customer requirements are extremely volatile too, so that encapsulating the exact requirements is extremely difficult. There is some evidence that the generic approach which has characterised much of software engineering to the present date is being recognised as infeasible, and software engineering will increasingly take a domain orientated point of view. This paper supports that view. The key issue is then the capture of domain knowledge, and

0-8186-5647-6/94 $04.00 © 1994 IEEE

the approach presented in this paper takes the starting point of a formally specified, domain orientated, very high level programming language. Our thesis is that a suitable language is a good way to make domain knowledge available, and the effect of developing in such a language is dramatically to reduce the development effort required while increasing maintainability, enabling reuse, and more generally enhancing program understanding to those who are expert in the domain.

2 Middle-Out Programming

In the history of Computer Science, the greatest single gain in software productivity has been achieved through the development of high level languages with suitable compilers and interpreters. The use of a high level language often allows a program to be implemented with an order of magnitude fewer lines of code than if everything were written in assembler. In addition, these lines of code will typically be easier to read, analyse, understand and modify, because they are "nearer" to the application, and further from the machine. In addition to the benefits of smaller code size and increased reliability, a major benefit of high level languages is that they encapsulate a great deal of programming knowledge in an easily usable form. For example, the compiler deals with the complexity of procedure calls and return mechanisms, recursion, optimisations etc.

In middle out programming, the first stage is to design a very high level, domain based language. This language includes operations, objects and representations characteristic of the application, and aims to capture knowledge of the application in the design of these language aspects. It is different to the provision of a library of useful operations as an addition to a conventional language, and to a standard package where application differences are coded via parameterisation.

The second step is to write the application in this language (using rapid prototyping if needed). In parallel, the language will be implemented.

The aim of our middle out programming approach is twofold. Firstly, by using a suitable very high level domain orientated language, the total code size should be considerably reduced, and its comprehensibility increased. Secondly, the design of the language itself forms a repository of domain knowledge in a form which is readily understandable and reusable by programmers working in that domain. This approach may be contrasted to the IKBS mechanism of representing domain knowledge in the form of e.g. rule based system. With rule based systems, the first problem is that the knowledge must be elicited, transferring knowledge from domain experts to a set of rules. Secondly, programmers must be able to extract and make use of information in the repository. This has worked well in some application domains, but seems difficult to generalise.

In contrast, the basic approach in our middle out programming is a language design providing a formal syntax and formal semantics for the language. Any language consists of a set of primitive operations together with language constructs and specialised abstract data types. Language design is a highly skilled task, which requires an expert in both the application domain and in Computer Science. This approach has been used in Chief Programmer teams. However, once completed, the expertise may be reused by subsequent programmers in the language. Hence, in middle out programming, subsequent development of the system may take place concurrently. This involves two steps, of implementing the software system in a new language, and then implementing the language in some existing computer language, for example by writing a compiler or interpreter. It is crucial to our approach that the middle level language should be formally specified. The benefits of precision, economy and clarity of a formal approach is considered very important, and contrasts with "line and bubble diagram" design methods in CASE tools, which have very weak semantics.

3 Comparison with Other Methods of Understanding

3.1 Top Down Development

Top down program development methods have been advocated by many people over a period of 30 years as a way of mastering the complexity of large programs by constructing them according to rigid hierarchical structures. The method typically starts with a high level definition of the systems to be developed, which may be an informal or a formal specification. The specification is refined into an architectural design, and this in turn is refined until an executable form is reached. Central to this approach is the use of divide and conquer, so that only a small part of the overall system is worked on at each stage, and the comprehension problem focuses on the current component being addressed. Many approaches to the formal refinement of mathematical specifications are based on this approach. It has certainly been shown to work well for small programs and well understood problems (see for example, Morgan's refinement Calculus [11,12,12].

The major difficulty is that the method itself provides no clue as to what the top level structure

should look like in a particular case. This is not a problem for toy programs, or for very well understood domains such a compilers. However, for large programs, choosing the wrong structures in the initial stages can have serious repercussions which can only be uncovered much later in the development. If a serious error is then realised, much of the development will have to be discarded and repeated. Often, successful top down design relies on using very experienced system designers, who have a good feel for simultaneous low level details. Even experienced designers may have problem in deciding how to decompose complex high level requirements. The system may be so large, that it is very difficult for a simple coherent top level specification to be produced. These problems also apply to object oriented design, which explains why it is difficult to achieve good object oriented design structures.

3.2 Bottom Up Development

In bottom up development, the programmer starts by implementing the lowest level general purpose utility routines. These are used as operations in a slightly higher level language to implement higher level routines, abstract data types and so on. Eventually, the top level structure of the program can be implemented. Compared to top down approaches, this allows early unit testing and integration testing. Unfortunately, it is also very difficult to decide what constitutes good low level design, and what the next level in the design should be in order to constitute progress. Design approaches may end in fruitless blind alleys, and much development work may have to be retraced. This is particularly a problem if the application is novel, and there is little previous experience of what high level routines would be useful.

3.3 Outside In development

Both of the above approaches have deficiencies which have long been recognised, and in practice, systems have been developed using a combination of the two. This reflects the practical difficulties in understanding how to build large systems in a strict top down or bottom up way. When the system is passed to maintenance, the only reliable source of information is typically the source code. Thus higher level abstractions must be constructed. Again this is often achieved (see Gilmore[6]) by establishing hypotheses, and then examining the system to confirm or refute them. This process too can work in top down, bottom up, or more typically a combination of the two.

4 Rapid Prototyping

A further major difficulty in large scale systems design is the inability to determine accurately the system requirements. As Lehman and others have pointed out, the users may not fully understand what they require anyhow, and the availability of the complete software system will probably modify what nebulous ideas they had of the original requirements. Therefore the rapid prototyping approach to requirements elicitation and to software development provides an attractive approach. It is typically much easier to design a larger complex system if one has already built a similar system in the past, by reusing the designer's previous experience. More fundamentally, users often have difficulties in articulating their precise requirements, but if they are presented with a system which does not do what they want, they will quickly find this out. Therefore, we argue that any reasonable approach to system understanding must include the understanding of the requirements of the user that can be provided by rapid prototyping. In our approach, a requirement can be expressed quickly in the very high level domain-oriented language; for sophisticated users this may be sufficient, but for others, interpretation can provide animation. Requirements are typically expressed in terms of domain concepts, rather than general programming concepts, so a domain-oriented language is a great help in assisting users formulate requirements.

5 Understanding Middle Out Programming

Understanding engineered software involves the understanding of both the process and the life cycle products. Understanding is important during the initial development phase, particularly for the large scale software developed by a team. It is crucial during subsequent software maintenance, when understanding of the software is a prerequisite to making alterations to it. This involves studying both life cycle products such as documentation, designs and the software itself, and also having understanding of the processes by which this is produced. We have argued that both top down and bottom up programming have apparently clear but actually infeasible processes. The life cycle products are often only clear in retrospect, when the design has been completed.

Unfortunately, both the above approaches do not readily incorporate rapid prototyping, and typically do not permit a useful representation of domain knowledge. They are good for generic software engineering

approaches, but less satisfactory for domain specific or user specific approaches. Middle out programming is a process and we shall argue that it provides life cycle products which are more appropriate for program understanding. It allows earlier representation of domain knowledge, encouraging concurrent engineering during initial development, and has a better representation for subsequent maintenance (in particular, at a higher level of abstraction than code). It also is compatible with a formal methods approach without the disadvantage of the strictly top down view which is often required for formal refinement of mathematical specifications.

Of course, when the design has been completed, and the trace from a specification to executable code has been completed, it does not matter how this has actually been generated. We argue that such a design history in not well suited to system comprehension, and has difficulties in keeping the various representations consistent. With top down approaches, the designers do not understand the software engineering issues and detailed implementation aspects. In bottom up analysis the designers do not have the understanding of the application domain which results in good design.

Our objective therefore is to study a middle out approach, which attempts to overcome the objections of these above approaches. Top down decomposition does not work well until the analyst has an almost complete concept of the system which may be very hard in a very big system. Bottom up development is unlikely to be successful if it is not possible to see how tactical developments fit into the big picture.

6 Middle Out Languages

The basis of middle out programming is the design of the formal, application domain based language. In our experience, the design of the language needs to be undertaken by a highly skilled application domain expert, who is also a highly able software engineer well versed in formal language theory. The design of any language is challenging, and cannot be undertaken by other than top grade staff. The language will contain application domain objects and operations which simplify and clarify the capture of domain knowledge. The use of a formally defined language permits precision and lack of ambiguity which simplifies its implementation. Once a language has been designed, its use and its implementation can proceed in parallel. The use of very high level languages has been advocated previously in software engineering. We claim that the novel results of the present research are as follows:

1. The language is highly application domain specific, and supports our view that software engineering will increasingly be less generic and more applications specific.
2. The approach is designed specifically to assist system understanding, particularly by encouraging rapid prototyping, and the production of highly maintainable systems which are portable.
3. The approach specifically addresses the problem of scale.
4. A design framework for user enhanceable systems is established (see below).
5. During initial development, concurrent engineering can be introduced to the process.

A language based approach is a step away from current trends in software engineering, which has favoured CASE tools in conjunction with methods based on "boxes and lines". Of course, key techniques such as configuration management continue to be necessary. A conventional waterfall type life cycle model involves many different representations and a key problem during maintenance is maintaining the consistency of these. Middle out programming is designed to encourage maintenance at the highest level of abstraction, and support automation in the generation of executable code. It is essential that a middle level language should be formally specified as it is the availability of the formal specification which will allow the system development and language implementation to be carried out independently. It is also important the language should be conceptually simple, easy to parse by both humans and computers, and should benefit from the latest development in programming language design and implementation.

7 A Case Study—FermaT

FermaT is a program transformation system based on the theory of program refinement and equivalence developed in [15,17] and applied to software development in [14,16] and to reverse engineering in [18,19]. The transformation system is intended as a practical tool for software maintenance, program comprehension, reverse engineering and program development. The system in based on semantic preserving transformations which are expressed in a wide spectrum language called WSL. The language includes both low level programming constructs and high level non-executable specifications. This means that refinement from a specification to an implementation, and reverse engineering to determine the behaviour of an existing

program can both be carried out by means of semantic preserving transformations within a single language.

In this paper we shall concentrate on the construction of the FermaT tool using the middle out programming approach. The original version of FermaT was written in LISP and C, and comprised of over 60,000 lines of source code. It exhibited the classic problems of a large, incomprehensible and difficult to maintain system. Much of the code is concerned with performing operations on programs and in particular writing program transformations. Therefore, in broad terms, the middle level language had to address the writing of program transformations. This included the need for abstract data types for representing programs as tree structures and constructs for pattern matching, pattern filling and iterating over components of the program structure. These are very typical high level operations in a program transformation system. Full details of WSL and $\mathcal{META}$WSL can be found in [17].

It was realised that the wide spectrum language WSL itself formed a very high level language in which transformations could be expressed. It was also language which had been very carefully formally defined both syntactically and semantically, in order to allow the expression of program transformations. Our approach was therefore to extend the WSL to a language termed $\mathcal{META}$WSL.

As an example, one of the simplest transformations in our system permits the merging of two contiguous assignments into a single assignment. Such a transformation is expressed in WSL. To express the implementation of the transformation, additional work in necessary such as checking for side effects, and also manipulating a program in internal tree form. WSL was extended to $\mathcal{META}$WSL to incorporate the latter type of feature. Thus $\mathcal{META}$WSL is a language which is directly designed for the application domain of writing transformation systems. It is actually implemented by writing a translator from meta WSL to LISP, adding a small LISP run time library, and executing the LISP. Although the two versions of the system are not directly comparable, the prototype version required some 42,000 lines of LISP in the key transformation engine. In the middle out program approach, this was reduced to approximately 5,000 lines of $\mathcal{META}$WSL. In addition, the $\mathcal{META}$WSL involves much smaller procedures and data types, suggesting that it is easier to understand too. $\mathcal{META}$WSL encapsulates much of the expertise developed over the past 10 years research in Durham in program transformation theory and transformation systems. As a result, this expertise is readily available to programmers, some of whom have only recently joined the project. Working in meta WSL it only takes a small amount of training before new programmers become effective at implementing transformations, and enhancing the functionality of existing transformations.

8 Examples of Middle Out Programming

Although we have described middle out programming as a novel approach, we can find examples of several successful large scale software development projects which use a middle level language layer. These were not necessarily developed in middle out order by design and language first, and do not all provide formal specifications of the middle language.

One of the first areas where the value of domain specific languages, especially languages with domain specific constructs, was Monte Carlo simulations. A system called MONTECODE [7] is an interpreted language written for writing Monte Carlo simulations. Its constructs include random sampling from distributions, management of queues, building histograms and event to event scanning.

CSL (Control Simulation Language [3]) was designed for use in the field of complex logical problems. It uses set operations and specialised constructs, including iteration of the elements of the set and finding an element in a set which meets various criterion. CSL is translated into FORTRAN, with a ratio of CSL to FORTRAN statements of the order of 1:5. It was reported that ratio of time writing similar programs in CSL and in FORTRAN was also of the same order.

Knuth's TEX typesetting program [8] was written in the WEB literate programming language [9] whose aim was to remove some of the deficiencies of Pascal and allows source code and documentation to be intertwined in the same source file. TEX itself is implemented as small set of primitive type setting operations together with a macro processor. The plain set of text macros were designed by Knuth to form a basic type setting package. A more extensive set of macros forms the basis of the LATEX [10] and these allow the user to concentrate on the structure of the text rather than on formatting commands. In effect LATEX implements a structured type setting language which the author uses to implement a document. The LATEX3 project, which is currently developing the new version of LATEX is planning yet another language level built on top of the TEX macro language.

The "MAKE" system [5] represents another domain based language for configuration and version management.

Nygaard's SIMULA 67 was designed after it was found that the control and data structures of then-current high level languages could not support the modelling of simulations, in particular the way that simulated objects interacted. This demonstrates that a high level language cannot be constructed out of a low level one by adding library functions.

9 User Enhancable Systems

The term user enhanceable system refers to the development of programming systems in which the user takes responsibility for writing and (especially) maintaining applications in a very high level system. Often, as in word processors this does not take the form of a conventional language at all. This approach has been advocated as a solution to increasing software productivity without suffering the drawbacks of rigid packaged software. An excellent example of such a user enhanceable system is the conventional spreadsheet. The spreadsheet product provides a very high level of application domain language to allow the non specialist user to write financial applications. Ease of use is enhanced by extensive graphics presentational devices. Thus the end user does not have to program in a conventional sense but simply enter figures into columns and apply simple functions to them, typically using the mouse to assist.

Spreadsheets do however readily exemplify the danger of inappropriate very high level languages. Spreadsheets are all too often used for applications far removed from simple financial computations. In typical modern spreadsheet systems, multiple spreadsheets may be linked, complex formula involving macros and procedures may be defined etc. The net result is a system which is totally incomprehensible, and all to often contains many errors. Modifications to the spreadsheet have to be made at the cell level by imposing a new structure and trying to remember the original structure. All the original structure, which indicates which cell belonged together and how groups of cells were related to each other has been lost. The effect is as if the user had entered a program in a high level language which the system then immediately translates into the low level language of cells and contents, throwing away the source code. All changes have to be made by patching the object code rather than by updating the source code and recompiling.

There is thus a need for one or more domain specific very high level languages which are higher than the original typical spreadsheet. For example a language is required for defining financial models. A typical program in this language would take the raw accounts data and produce cash flow forecasts, balance sheet rejections etc.

Many other examples of user enhanceable systems exist e.g. in CAD systems, electronic instrumentation etc.

Middle out programming provides a way of addressing these systems, by stating the problem as one of language design at the appropriate level of abstraction.

10 Advantages of Middle Out Programming

10.1 Separation of Concerns

In our experiences with the FermaT system, the middle out programming method using a formally defined middle language provides the complete separation of concerns of design issues, which are addressed in a domain specific language, and implementation issues, which are addressed in the implementation of the language. This is a very important result for program comprehension.

10.2 High Development Productivity

We have found that the approach reduces the size of the system, and a few lines of code are often sufficient to implement highly complex functions. The implementation of the language is often kept small since only those features are relevant to a particular problem domain need to implemented. The major success of this approach, compared (for example) with 4 GLs is that the language is restricted to highly specialised domains.

10.3 Highly Maintainable Design

The reduced size of the software alone indicates that maintaining it will require considerably less effort. Typically, we have found that major functions of the system are implemented very compactly in a few lines of code. We have found that bug fixing and making enhancements is relatively easy and there is a reduced chance of unexpected side effects. Some of the advantages have much in common with modular design and object orientated systems in trying to localise and hide implementation decisions. Some fundamental designs decisions can not always be captured in a module. In our very high level language approach this is facilitated by permitting constructs to be used anywhere in the code.

10.4 Portability

Porting the very high level language is a concern only of the implementation; the application level design is unaffected. In FermaT, porting the system from LISP to C only required rewriting the bottom level translator , and this was a comparatively small task occupying a small number of days.

10.5 Opportunities for Reuse

The major benefit we have found in the approach is the ability to encapsulate a great deal of domain knowledge, including knowledge of which data types, operations and execution methods are important in the domain, and what are the best ways to implement them. This kind of knowledge is extremely useful for requirements elicitation for new systems in the same application domain, and program comprehension of the existing system. One of the main advantages of a well designed domain specific language is the new programming constructs which can be combined, approximately orthogonally in various ways. This is much more powerful than for example trying to write a C program in assembler, where the C compiler has been replaced by a large library of assembler routines. In the Draco project reuse of design information was encouraged by the use of specific domain languages together with the recorded results of domain analysis. The system under development is written in a number of many different domain languages and these programs are refined into the languages of other domains and ultimately into executable code. Our approach uses a single domain for several related development projects rather than several small domains for each project. Our contention is that the best representation of domain knowledge is the design and implementation of a domain specific programming language. Since our domain languages are implemented programming languages, there is no need for refinement to an existing programming language.

11 Conclusions

A combination of rapid prototyping with middle out development of each prototype would appear to be a good development approach for many large scale software development projects. The middle out approach provides a way of addressing the problems of scale. This is achieved by reducing the total size of a large system by expressing it in a very high level language. In our experience, program comprehension is considerably improved by using the language oriented to the need of the application specialist. In our approach, the language is designed to be based on domain concepts and objects, so the structure of the code and each operation closely matches the behaviour and effect of that operation. This contrasts with other approaches, where the user is obliged to understand concepts expressed in programming or software engineering terms.

According to our original criteria, we find that the complexity of systems is much reduced since the system is divided into two independent sections; the application domain using a very high level domain orientated language, and a translator or interpreter for the language. This reduction in complexity and separation of issues improves understanding of the system by the language implementor, and the understanding of the requirements by the user.

We have also found that the improved comprehension assists in including integrity checks and consistency tests, and perhaps most encouraging, we have found at that the system is much more understandable under software maintenance. For the future research, we plan to extend our investigations to this approach for formal methods. In FermaT, meta WSL is a formal language defined on a foundation of denotational semantics. Much of our research is focused on the semantics of this language and its suitability for expressing program transformations. Thus meta WSL appears to be a suitable language for mathematically rigorously defined specifications. This would seem to have some attractions over the conventional top down refinement method advocated by many formal methods.

Acknowledgements

This work was supported by the Science and Engineering Research Council (now the EPSRC) project "A Proof Theory for Program Refinement and Equivalence; Extensions".

References

[1] K. H. Bennett, B. J. Cornelius, M. Munro & D. J. Robson, "Approaches to Program Comprehension," *J. Syst. and Soft.* 14 (Feb., 1991), 79–84.

[2] F. P. Brooks, "No Silver Bullet," *IEEE Computer* (Apr., 1987).

[3] J. N. Buxton & J. G. Laski, "Control and Simulation Language," *Comput. J.* 5 (1962), 194–199.

[4] B. Curtis, H. Krasner & N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Comm. ACM* 31 (Nov., 1988), 1268–1287.

[5] S. I. Feldman, "MAKE—A Program for Maintaining Computer Programs," *Software - Practice and Experience* 9 (1979), 255–265.

[6] D. J. Gilmore, "Models of Debugging," *Proceedings of Fifth European Conference on Cognitive Ergonomics, Urbino, Italy* (Sept., 1990).

[7] D. H. Kelly & J. N. Buxton, "Montecode—an Interpretive Program for Monte Carlo," *Comput. J.* 5 (1962), 88–93.

[8] D. E. Knuth, *The TEXBook*, Addison Wesley, Reading, MA, 1984.

[9] D. E. Knuth, "Literate Programming," *The Computer Journal* 27 (1984), 97–111.

[10] L. Lamport, *LATEX A Document Preparation System*, Addison Wesley, Reading, MA, 1986.

[11] C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[12] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[13] D. L. Parnas, "Designing Software," *IEEE Trans. Software Eng.* 5 (Mar., 1979).

[14] H. A. Priestley & M. Ward, "A Multipurpose Backtracking Algorithm," *J. Symb. Comput.* (1993), to appear.

[15] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

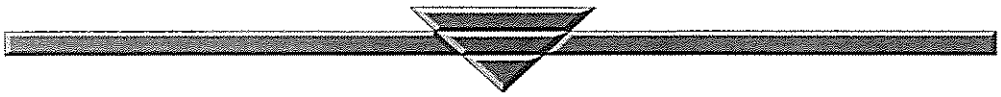
[16] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," Submitted to IEEE Trans. Software Eng., 1992.

[17] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," *Submitted to Formal Aspects of Computing*, New York–Heidelberg–Berlin (1993).

[18] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122.

[19] M. Ward & K. H. Bennett, "A Practical Program Transformation System For Reverse Engineering," *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (1993).

Author Index



Notes

IEEE Computer Society Press Titles

AUTHORED BOOKS

Advances in ISDN and Broadband ISDN
Edited by William Stallings
(ISBN 0-8186-2797-2); 272 pages

Advances in Local and Metropolitan Area Networks
Edited by William Stallings
(ISBN 0-8186-5042-7); 448 pages

Advances in Real-Time Systems
Edited by John A. Stankovic and Krithi Ramamritham
(ISBN 0-8186-3792-7); 792 pages

Architectural Alternatives for Exploiting Parallelism
Edited by David J. Lilja
(ISBN 0-8186-2642-9); 464 pages

Artificial Neural Networks — Concepts and Control Applications
Edited by V. Rao Vemuri
(ISBN 0-8186-9069-0); 520 pages

Artificial Neural Networks — Concepts and Theory
Edited by Pankaj Mehra and Banjamin Wah
(ISBN 0-8186-8997-8); 680 pages

Artificial Neural Networks— Forecasting Time Series
Edited by V. Rao Vemuri and Robert D. Rogers
(ISBN 0-8186-5120-2); 220 pages

Artificial Neural Networks— Oscillations, Chaos, and Sequence Processing
Edited by Lipo Wang and Daniel L. Alkon
(ISBN 0-8186-4470-2); 136 pages

Autonomous Mobile Robots: Perception, Mapping and Navigation — Volume 1
Edited by S. S. Iyengar and A. Elfes
(ISBN 0-8186-9018-6); 425 pages

Autonomous Mobile Robots: Control, Planning, and Architecture — Volume 2
Edited by S. S. Iyengar and A. Elfes
(ISBN 0-8186-9116-6); 425 pages

Branch Strategy Taxonomy and Performance Models
Written by Harvey G. Cragon
(ISBN 0-8186-9111-5); 150 pages

Bridging Faults and IDDQ Testing
Edited by Yashwant K. Malaiya and Rochit Rajsuman
(ISBN 0-8186-3215-1); 128 pages

Broadband Switching: Architectures, Protocols, Design, and Analysis
Edited by C. Dhas, V. K. Konangi, and M. Sreetharan
(ISBN 0-8186-8926-9); 528 pages

Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions
Edited by Milo Tomasevic and Veljko Milutinovic
(ISBN 0-8186-4092-8); 448 pages

Codes for Detecting and Correcting Unidirectional Errors
Edited by Mario Blaum
(ISBN 0-8186-4182-7); 224 pages

Communication and Computer Networks: Modelling with Discrete Time Queues
Written by Michael E. Woodward
(ISBN 0-7273-0410-0); 280 pages

Computer-Aided Software Engineering (CASE) (Second Edition)
Edited by Elliot Chikofsky
(ISBN 0-8186-3590-8); 184 pages

***Readings in* Computer-Generated Music**
Edited by Denis Baggi
(ISBN 0-8186-2747-6); 232 pages

Computer Algorithms: Key Search Strategies
Edited by Jun-ichi Aoe
(ISBN 0-8186-2123-0); 154 pages

Computer Arithmetic II
Edited by Earl E. Swartzlander, Jr.
(ISBN 0-8186-8945-5); 412 pages

Computer Communications: Architectures, Protocols, and Standards (Third Edition)
Edited by William Stallings
(ISBN 0-8186-2712-3); 360 pages

Computer Graphics Hardware: Image Generation and Display
Edited by H. K. Reghbati and A. Y. C. Lee
(ISBN 0-8186-0753-X); 384 pages

Computer Graphics: Image Synthesis
Edited by Kenneth Joy, Nelson Max, Charles Grant, and Lansing Hatfield
(ISBN 0-8186-8854-8); 380 pages

Computer Vision: Principles
Edited by Rangachar Kasturi and Ramesh Jain
(ISBN 0-8186-9102-6); 700 pages

Computer Vision: Advances and Applications
Edited by Rangachar Kasturi and Ramesh Jain
(ISBN 0-8186-9103-4); 720 pages

Current Research in Decision Support Technology (IS Series)
Edited by Robert W. Blanning and David R. King
(ISBN 0-8186-2807-3); 256 pages

Decision Fusion
Written by Belur V. Dasarathy
(ISBN 0-8186-4452-4); 300 pages

Digital Image Warping
Written by George Wolberg
(ISBN 0-8186-8944-7); 340 pages

***Readings in* Distributed Computing Systems**
Edited by Thomas Casavant and Mukesh Singhal
(ISBN 0-8186-3032-9); 632 pages

Distributed Computing Systems: Concepts and Structures
Edited by A. L. Ananda and B. Srinivasan
(ISBN 0-8186-8975-0); 416 pages

Distributed Mutual Exclusion Algorithms
Edited by Pradip K. Srimani and Sunil R. Das
(ISBN 0-8186-3380-8); 168 pages

Distributed Open Systems
Edited by Frances Brazier and Dag Johansen,
(ISBN 0-8186-4292-0); 192 pages

Digital Image Processing (Second Edition)
Edited by Rama Chellappa
(ISBN 0-8186-2362-4); 816 pages

For further information call toll-free 1-800-CS-BOOKS or write:

IEEE Computer Society Press, 10662 Los Vaqueros Circle, PO Box 3014,
Los Alamitos, California 90720-1264, USA

IEEE Computer Society, 13, avenue de l'Aquilon,
B-1200 Brussels, BELGIUM

IEEE Computer Society, Ooshima Building, 2-19-1 Minami-Aoyama,
Minato-ku, Tokyo 107, JAPAN

Domain Analysis and Software Systems Modeling
Edited by Ruben-Prieto Diaz and Guillermo Arango
(ISBN 0-8186-8996-X); 312 pages

Expert Systems:
A Software Methodology for Modern Applications
Edited by Peter G. Raeth
(ISBN 0-8186-8904-8); 476 pages

Fault-Tolerant Software Systems:
Techniques and Applications
Edited by Hoang Pham
(ISBN 0-8186-3210-0); 128 pages

Formal Verification of Hardware Design
Edited by Michael Yoeli
(ISBN 0-8186-9017-8); 340 pages

Genetic Algorithms
Edited by Bill P. Buckles and Frederick E. Petry
(ISBN 0-81862935-5); 120 pages

Global States and Time in Distributed Systems
Edited by Zhonghua Yang and T. Anthony Marsland
(ISBN 0-8186-5300-0); 168 pages

Groupware: Software for Computer-Supported Cooperative Work
Edited by David Marca and Geoffrey Bock
(ISBN 0-8186-2637-2); 600 pages

Implementing Configuration Management:
Hardware, Software, and Firmware
Written by Fletcher J. Buckley
(ISBN 0-7803-0435-7); 256 pages

Information Systems and Decision Processes (IS Series)
Written by Edward A. Stohr and Benn R. Konsynski
(ISBN 0-8186-2802-2); 368 pages

Interconnection Networks for Multiprocessors and Multicomputers: Theory and Practice
Edited by Anujan Varma and C.S. Raghavendra
(ISBN 0-8186-4972-2); 584 pages

Knowledge-Based Systems: Fundamentals and Tools
Edited by Oscar N. Garcia and Yi-Tzuu Chien
(ISBN 0-8186-1924-4); 512 pages

Multidatabase Systems:
Advanced Solution for Global Information Sharing
Edited by Ali R. Hurson, M. W. Bright, and S. H. Pakzad
(ISBN 0-8186-4422-2); 400 pages

Network Management
Edited by William Stallings
(ISBN 0-8186-4142-8); 368 pages

Nearest Neighbor Pattern Classification Techniques
Edited by Belur V. Dasarathy
(ISBN 0-8186-8930-7); 464 pages

Object-Oriented Databases
Edited by Ez Nahouraii and Fred Petry
(ISBN 0-8186-8929-3); 256 pages

Optic Flow Computation: A Unified Perspective
Written by Ajit Singh
(ISBN 0-8186-2602-X); 256 pages

Readings in
Real-Time Systems
Edited by Yang-Hann Lee and C. M. Krishna
(ISBN 0-8186-2997-5); 256 pages

Real-Time Systems Abstractions, Languages, and Design Methodologies
Edited by Krishna M. Kavi
(ISBN 0-8186-3152-X); 550 pages

Real-Time Systems Design and Analysis
Written by Phillip A. Laplante
(ISBN 0-7803-0402-0); 360 pages

Reduced Instruction Set Computers (RISC) (Second Edition)
Edited by William Stallings
(ISBN 0-8186-8943-9); 448 pages

Simulation Validation:
A Confidence Assessment Methodology
Written by Peter L. Knepnell and Deborah C. Arangno
(ISBN 0-8186-3512-6); 168 pages

Software Engineering: A European Perspective
Edited by Richard H. Thayer and Andrew D. McGettrick
(ISBN 0-8186-9117-4); 696 pages

Software Engineering Project Management
Edited by Richard H. Thayer
(ISBN 0-8186-0751-3); 512 pages

Software Management (Fourth Edition)
Edited by Donald J. Reifer
(ISBN 0-8186-3342-5); 656 pages

Software Metrics:
A Practitioner's Guide to Improved Product Development
Written by Daniel J. Paulish and Karl-Heinrich Möller
(ISBN 0-7803-0444-6); 272 pages

Software Reengineering
Edited by Robert S. Arnold
(ISBN 0-8186-3272-0); 688 pages

Software Reuse — Emerging Technology
Edited by Will Tracz
(ISBN 0-8186-0846-3); 400 pages

Software Risk Management
Edited by Barry W. Boehm
(ISBN 0-8186-8906-4); 508 pages

Standards, Guidelines and Examples on System and Software Requirements Engineering
Edited by Merlin Dorfman and Richard H. Thayer
(ISBN 0-8186-8922-6); 626 pages

System and Software Requirements Engineering
Edited by Richard H. Thayer and Merlin Dorfman
(ISBN 0-8186-8921-8); 740 pages

Systems Network Architecture
Edited by Edwin R. Coover
(ISBN 0-8186-9131-X); 464 pages

Test Access Port and Boundary-Scan Architecture
Edited by Colin M. Maunder and Rodham E. Tulloss
(ISBN 0-8186-9070-4); 400 pages

Validating and Verifying Knowledge-Based Systems
Edited by Uma G. Gupta
(ISBN 0-8186-8995-1); 400 pages

Visual Cues: Practical Data Visualization
Written by Peter R. Keller and Mary M. Keller
(ISBN 0-8186-3102-3); 350 pages

Visual Programming Environments: Paradigms and Systems
Edited by Ephraim Glinert
(ISBN 0-8186-8973-0); 680 pages

Visual Programming Environments: Applications and Issues
Edited by Ephraim Glinert
(ISBN 0-8186-8974-9); 704 pages

Visualization in Scientific Computing
Edited by G. M. Nielson, B. Shriver, and L. Rosenblum
(ISBN 0-8186-8979-X); 304 pages

VLSI Algorithms and Architecture: Advanced Concepts
Edited by N. Ranganathan
(ISBN 0-8186-4402-8); 320 pages

VLSI Algorithms and Architecture: Fundamentals
Edited by N. Ranganathan
(ISBN 0-8186-4392-7); 320 pages

Volume Visualization
Edited by Arie Kaufman
(ISBN 0-8186-9020-8); 494 pages

X.25 and Related Protocols
Written by Uyless Black
(ISBN 0-8186-8976-5); 304 pages

IEEE Computer Society Press

Offices of the IEEE Computer Society

Headquarters Office
1730 Massachusetts Avenue, N.W.
Washington, DC 20036-1903
Phone: (202) 371-0101 — Fax: (202) 728-9614

Publications Office
P.O. Box 3014
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1264
Membership and General Information: (714) 821-8380
Publication Orders: (800) 272-6657 — Fax: (714) 821-4010

European Office
13, avenue de l'Aquilon
B-1200 Brussels, BELGIUM
Phone: 32-2-770-21-98 — Fax: 32-2-770-85-05

Asian Office
Ooshima Building
2-19-1 Minami-Aoyama, Minato-ku
Tokyo 107, JAPAN
Phone: 81-3-408-3118 — Fax: 81-3-408-3553

IEEE Computer Society

IEEE Computer Society Press Publications

Monographs: A monograph is an authored book consisting of 100-percent original material.
Tutorials: A tutorial is a collection of original materials prepared by the editors and reprints of the best articles published in a subject area. Tutorials must contain at least five percent of original material (although we recommend 15 to 20 percent of original material).
Reprint collections: A reprint collection contains reprints (divided into sections) with a preface, table of contents, and section introductions discussing the reprints and why they were selected. Collections contain less than five percent of original material.
Technology series: Each technology series is a brief reprint collection — approximately 126-136 pages and containing 12 to 13 papers, each paper focusing on a subset of a specific discipline, such as networks, architecture, software, or robotics.
Submission of proposals: For guidelines on preparing CS Press books, write the Managing Editor, IEEE Computer Society Press, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1264, or telephone (714) 821-8380.

Purpose

The IEEE Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 100,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

All members receive the acclaimed monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

***Computer* magazine:** An authoritative, easy-to-read magazine containing tutorials and in-depth articles on topics across the computer field, plus news, conference reports, book reviews, calendars, calls for papers, interviews, and new products.
Periodicals: The society publishes six magazines and five research transactions. For more details, refer to our membership application or request information as noted above.
Conference proceedings, tutorial texts, and standards documents: The IEEE Computer Society Press publishes more than 100 titles every year.
Standards working groups: Over 100 of these groups produce IEEE standards used throughout the industrial world.
Technical committees: Over 30 TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.
Conferences/Education: The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.
Chapters: Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.