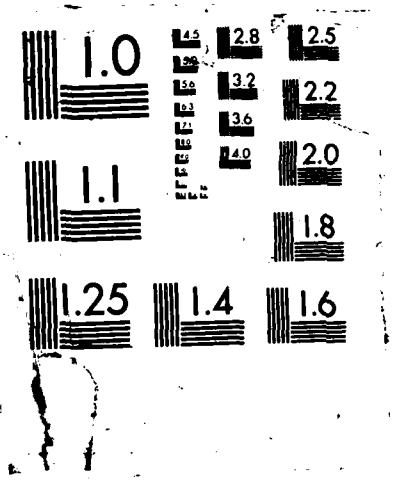
AD-A187 472 DERIVING ABSTRACTIONS FROM A SOFTWARE OBJECT NETWORK (U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA Y CHEN ET AL. AUG 87 NPS52-87-037 1/1

UNCLASSIFIED F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0

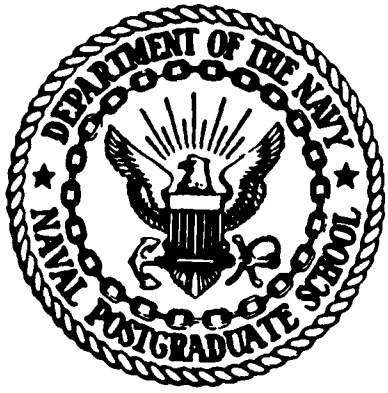
IC FILE COPY

AD-A187 472

NPS52-87-037

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
NOV 1 0 1987
S D
CD

DERIVING ABSTRACTIONS FROM A SOFTWARE

OBJECT NETWORK

Yih-Farn Chen

C. V. RAMAMOORTHY

August 1987

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

87

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-87-037	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Chief of Naval Operations (OP-094)
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943	7b. ADDRESS (*City, State, and ZIP Code*)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Space and Warfare Systems Cmd 8b OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N0003987WREF312
8c. ADDRESS (*City, State, and ZIP Code*) Washington, DC 20363	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Deriving Abstractions from a Software Object Network

12. PERSONAL AUTHOR(S)
Yih-Farn Chen and C. V. Ramamoorthy

13a. TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (*Year, Month, Day*)	15 PAGE COUNT
Technical	FROM _____ TO _____	August 1987	20

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Program database, software restructuring, modules, layers, ripple effects, abstraction

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

High level abstractions from programs can be obtained by (1) extracting relational information from programs to form a software object network, and (2) deriving high level abstractions from that network. We show how to obtain several interesting abstractions such as subsystems, ripple effects, logical layers and modules from a software object network represented by a C program database. These abstractions assist programmers in understanding the program structure and point out potential areas for improvement. We then demonstrate how rule-based software restructuring can be performed by accessing the relational information stored in the program database.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL C. V. Ramamoorthy	22b TELEPHONE (*Include Area Code*) 408/646-3390 / 22c OFFICE SYMBOL

☆ U.S. Government Printing Office 1986—606-24

Deriving Abstractions from a Software Object Network

Yih-Farn Chen†
C. V. Ramamoorthy

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

High level abstractions from programs can be obtained by (1) extracting relational information from programs to form a software object network, and (2) deriving high level abstractions from that network. We show how to obtain several interesting abstractions such as subsystems, ripple effects, logical layers and modules from a software object network represented by a C program database. These abstractions assist programmers in understanding the program structure and point out potential areas for improvement. We then demonstrate how rule-based software restructuring can be performed by accessing the relational information stored in the program database.

1. Introduction

Traditionally, a programmer utilizes certain editing commands to understand and modify software. Such an approach has several severe drawbacks:

(1) In order to trace the logical structure of a program efficiently, a programmer has to memorize the location of many objects and relations among them.

(2) A programmer is forced to constantly view a program as a set of files, lines, and characters. A conceptually simple software modification task may have to be transformed into numerous error-prone editing actions. It is difficult to guarantee that the integrity of a program is maintained after several modifications.

(3) Programmers are reluctant to do any major software restructuring, even if the potential saving of future maintenance cost can be great. Programs become increasingly difficult to understand as new features are added in.

In view of these problems, we have developed the C Information Abstraction (CIA) System[1,2]. The goal of CIA is to provide an entity-relationship view to C programmers by constructing a C program database from the source code. Based on this relational

† This work was supported in part by the Chief of Naval Operations (OP-094) and in part by the California MICRO program under contract No. 532434-10900.

DTIC COPY INSPECTED 6

A-1

view, a set of tools are constructed to facilitate software understanding and to automate the software manipulation process. This paper deals with the problem of deriving high level abstractions from the C program database and using these abstractions to guide the software restructuring process.

Section 2 gives an outline of the CIA system; Section 3 introduces the concept of software object network; Section 4 deals with the extraction of subsystems from C programs for reusability; Section 5 shows how to compute ripple effects; Section 6 presents an algorithm for deriving logical layering from C programs; Section 7 discusses the modularization of C programs; Section 8 introduces a set of rules for software restructuring; finally, Section 9 gives the conclusion.

2. The C Information Abstraction System

The CIA system consists of three major components (Figure 1):

(1) The C Abstractor, which constructs a program database by extracting relational information from a set of C programs.

(2) The Information Viewer, which provides relational views to users and allows a set of library calls to programmers.

(3) The Investigator, which constructs high level software views from the relational views.

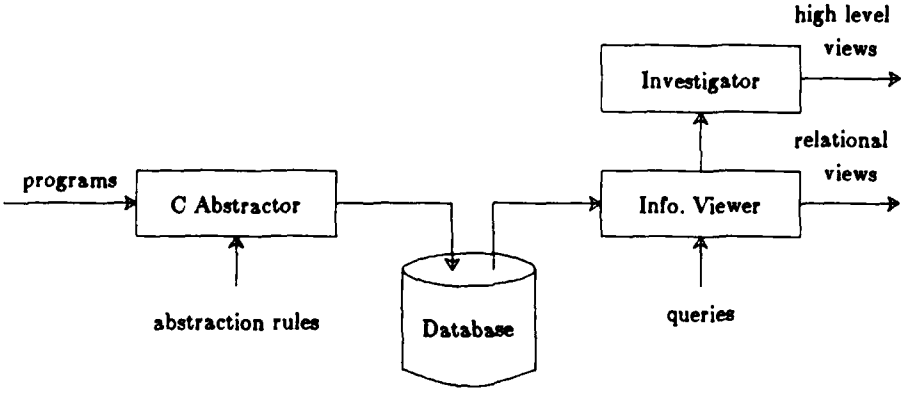


Figure 1: The outline of the C Information Abstraction System

Information to be extracted by the C Abstractor is determined by the conceptual view of the C programs. We decided to simplify this view by concentrating on global objects, i.e. objects that can be referenced across function or file boundaries. However, this simple conceptual view is enough to support most important abstractions as we shall see later. Figure 2 shows the conceptual view of the C program database. Relations shown in solid lines can been extracted by the current CIA system; relations shown in dotted lines will be available in the next version of CIA. Definitions of all the relations are listed in Table 1.

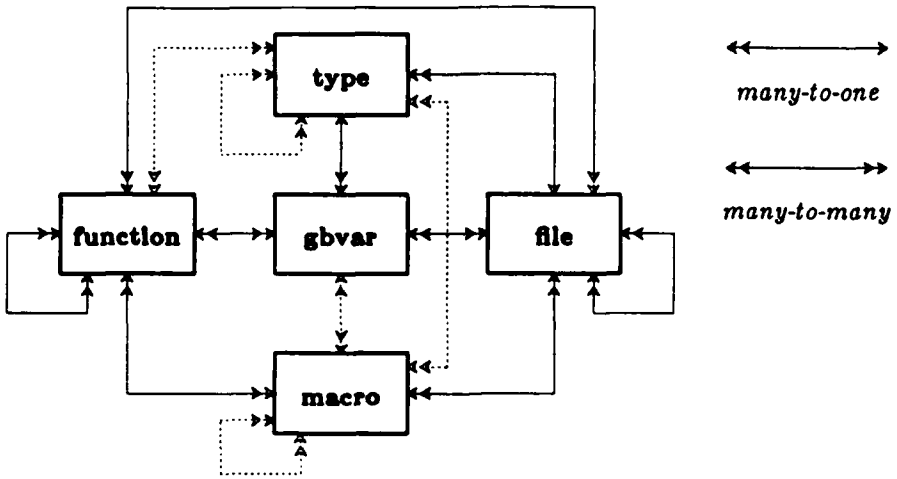


Figure 2: The Conceptual View of the C Program Database

Table 1. Definitions of the Relationships

num	*obj_type1*	*obj_type2*	*rel_type*	*definition*
1	file	file	m-to-m	file1 includes file2
2	function	function	m-to-m	function1 calls function2
3	gbvar	function	m-to-m	gbvar1 referenced in function2
4	macro	function	m-to-m	macro1 referenced in function2
5	function	file	m-to-m	function1 referenced or defined in file2
6	macro	file	m-to-m	macro1 referenced or defined in file2
7	gbvar	file	m-to-m	gbvar1 referenced or defined in file2
8	type	file	m-to-m	type1 referenced or defined in file2
9	gbvar	type	m-to-1	gbvar1 defined as type2
10*	type	function	m-to-m	type1 referenced or defined in function2
11*	type	type	m-to-m	type1 referenced in type2
12*	macro	type	m-m	macro1 referenced in type2
13*	macro	macro	m-m	macro1 referenced in macro2
14*	macro	gbvar	m-m	macro1 referenced in gbvar2

* future extensions

To illustrate some of these relations, Example 1 shows a portion of a simple C program. Comments are associated with each line wherever there are relations established due to references.

Example 1:

```
#define HEADER 12
#define PACKET 128
#define CONTENT PACKET-HEADER     /* macro-macro relation */
char buffer[CONTENT];             /* gbvar-macro relation */
struct message {
  char header[HEADER];            /* type-macro relation */
  char content[PACKET];           /* type-macro relation */
};
struct packet {
  struct message m;               /* type-type relation */
  int timestamp;
};
struct packet p1;                 /* type-gbvar relation */
receive()
{
  p1=read(buffer, PACKET);        /* function-function relation */
  /* 2 function-gbvar relations */ /* function-macro relation */
}
```

In Example 1, the single statement

```
p1=read(buffer, PACKET)
```

establishes four relations because of the references to two global variables *p1* and *buffer*, the reference to the function *read*, and the reference to the macro *PACKET*.

Besides the relational information, the program database also keeps information about the location, size, static scope, data type and other attributes of each object. Therefore, the Infoview system is capable of providing three important functions (among others):

(1) Retrieval of information about the attributes of a software object.

(2) Access to relations among software objects.

(3) Retrieval of the definition (contents) of an object.

With the program database, the Investigator can easily derive high level software abstractions. The view of programs as *Software Object Networks* will first be discussed. Then we willl show how various abstractions can be computed from this network, and how these abstractions can guide rule-based software restructuring.

3. Software Object Network

In general, we can build a *software object network* from the program database by assigning a node for each object and an arc for each relationship between two global objects. For example, Figure 3 shows the object network constructed from Example 1. In the rest of this paper, we shall use a slightly more complex software object network shown in Figure 4 to illustrate several interesting concepts. Big squares denote function objects and small squares denote global objects of other types. Arcs represent reference relationships. The subnetwork constructed from the function objects and their relationships is

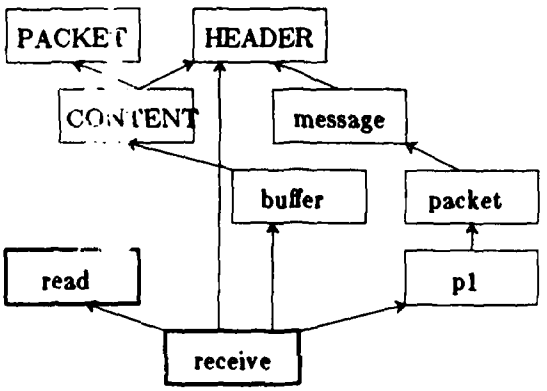


Figure 3: The Object Network Constructed from Example 1

considered the backbone of the program structure. The other global objects and references are considered as the flesh.

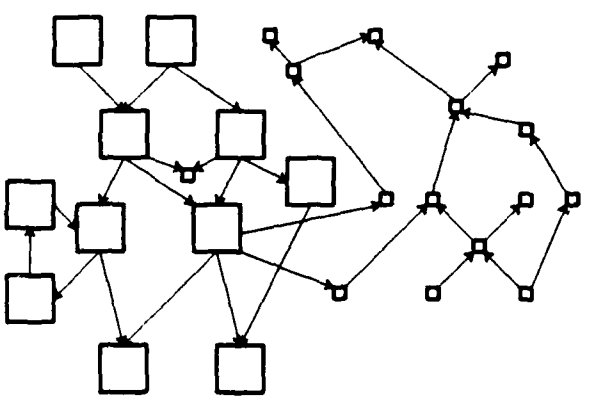


Figure 4: A Typical Software Object Network

An object network for a reasonably large program would be too complex to understand because a human can only handle a few objects and relations at a time. To simplify the understanding of program structure, we need to derive some abstractions from the object network. We shall examine the following abstractions and discuss their roles in program understanding and automation:

(1) Reflexive, transitive closure of the reference relation.

(2) Topological sorting of the reference relation.

(3) Clustering based on common references.

The first abstraction is crucial for reusing subsystems in programs and for calculating ripple effects. The second abstraction helps reveal logical layers in a program. The third abstraction gives the logical modularization of a program. The object network shown in Figure 4 would be used repeatedly to illustrate these abstractions in the

following sections.

Based on these abstractions, a user can usually identify the weaknesses in the program structure, and a possible software restructuring may be called for. We shall see how a high level software restructuring operation can be broken down into a series of primitive operations. The rules governing each primitive operation for the purpose of maintaining program integrity will be given. Exercising these rules requires accessing the program database for detailed examination of the relations among software objects.

4. Extraction of Subsystems for Reusing Software

The effort in developing a new software system can be substantially reduced if certain software objects in other systems can be reused in the new system[3,4]. Reusing a software object involves several tasks:

(1) *Identification:* Identify a reusable object that satisfys the needs.

(2) *Extraction:* Extract that object and its associated objects.

(3) *Integration:* Integrate the set of extracted objects with the new system.

The first task can be accomplished, to some extent, with the help of structured comments described in[1,2] or the attributed nodes in the French MENTOR project[5] by associating machine-processable comments with each reusable software object. Luqi suggests the use of normalization transformations to solve the identification problem for software adopting different specifications.[6].

We shall concentrate on the second and third tasks. We begin by defining the term *subsystem*. A subsystem associated with an object X is a set of objects that can be reached by following the reference links in the object network starting from the object X. In other words, the mapping from an object to its subsystem is defined by the reflexive, transitive closure of the reference relations in the object network. We shall call this mapping *Sub*. For example, black boxes in Figure 5 show the subsystem of the function F, i.e. Sub(F). This type of calculation is similar to the *reachability* analysis in the state transition diagrams for finite state machines.

Reusing an object requires extracting its whole subsystem so that there will be no missing references in the new system. And if we would like to reuse a set of objects $O_1, O_2, ..., O_n$, then the objects to be extracted are $Sub(O_1) \bigcup Sub(O_2) \bigcup \cdots \bigcup Sub(O_n)$.

Integrating objects of a subsystem into the new system may introduce name conflicts. We shall come to this problem later when we deal with the problem of software restructuring.

As an example, if we want to reuse the data structure "struct packet" in Example 1, then the following should be extracted:

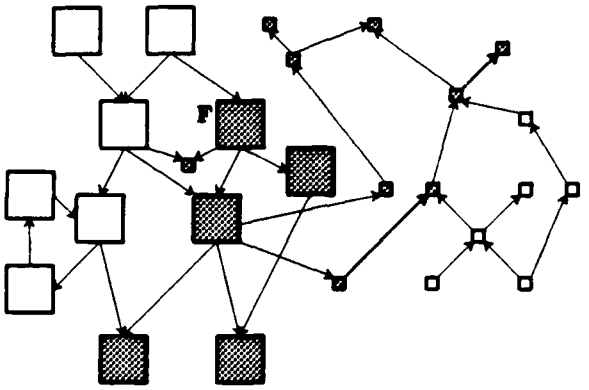


Figure 5: Extraction of a Subsystem from the Object Network

```
#define HEADER 12
#define PACKET 128
struct message {
  char header[HEADER];              /* macro-type relation */
  char content[PACKET];
};
struct packet {
  struct message m;              /* type-type relation */
  int timestamp;
};
```

This portion of code can be compiled without missing references. On the other hand, if we want to extract the function *receive*, then the whole program portion shown in Example 1 and the function *read* and its associated subsystem (defined in other files) should all be extracted. The whole extraction process can then be automated using the information stored in the program database.

The concept of subsystem leads us to define four types of *weight* for each global object:

(1) The *basic weight* of each global object is 1.

(2) The *actual weight* of a global object is the number of lines used in the source code to define that object.

(3) The *basic association weight* of a global object X is the number of global objects in the subsystem of X. For example, the basic association weight of function F in Figure 5 is 14.

(4) The *actual association weight* of a global object X is the sum of the actual weight of all global objects in the subsystem of X. In other words, it is the number of lines that a programmer has to go through to fully understand X.

Intuitively, a *heavy* object (an object with a large association weight) is more difficult to understand than a *light* object (an object with a small association weight). However, other factors such as cross-module referencing, the degree of sharing of referenced objects,

and control complexity may have to be taken into account. In any case, the *actual association weight* appears to be a reasonably good measure to indicate the amount of effort required to understand and reuse an object.

5. Ripple Effects

Occasionally, a programmer would like to change the definition of a particular object or even remove the object. Such a modification may affect the correct operation of many other objects. This is termed the *ripple effects*. To guarantee the program integrity, the identification of those affected objects is necessary. We shall call the set of objects involved in the ripple effects the *ripple set*. The mapping from an object to its ripple set is simply defined by the reflexive, transitive closure of the *referenced-by* relation, which is the *reverse* of the reference relation. For example, Figure 6 shows the ripple set of the global object *g*. In Example 1, the ripple set of the data type "struct packet" consists of the global variable "p1" and the function "receive".

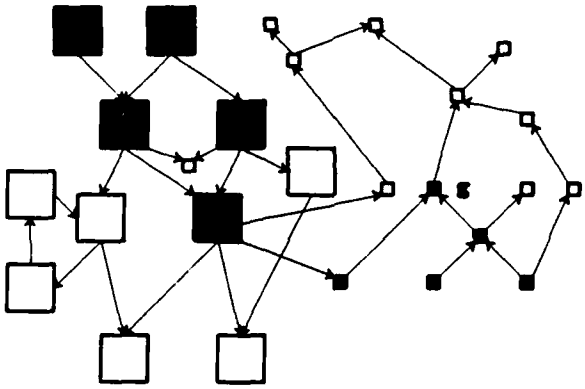


Figure 6: Calculation of the Ripple Set

Frequently, only a part of a large system under development need be tested. In this case, the ripple set of an object X indicates the set of objects that cannot be tested if X is specified but not fully implemented. For example, in Figure 6, if the object g is not fully implemented, then the function F cannot be fully tested.

Our definition here is different from the one defined by Yau, et al.[7]. In their approach, detailed ripple effect analysis is provided and it is possible to determine that a particular statement is affected by a change in the definition of a global variable. Their approach requires complex lexical analysis and error flow analysis. Our approach sets the granularity at the level of global objects; required changes inside a global object need not be identified. However, once the ripple set is determined from the program database, a programmer can easily identify the impact on each affected object. Our approach simplifies the program analysis and substantially reduces maintenance cost.

The *stability* of a program can be defined in terms of the resistance to the potential ripple effect. Interested readers should refer to [8] and[9].

6. Layering

One way to understand a complex system is to view it in several layers. Each layer presents an abstraction which reduces the complexity visible to upper layers. Parnas and Siewiorek refer to each layer as a *virtual machine*, which provides a set of new *instructions* or *operators* to the upper layers[10].

Ideally, each software object in a program belongs to a particular layer. With a layering structure, a user can elect to examine a program only to a certain level of details, i.e., only objects that belong to levels about that one need to be extracted and viewed. Unfortunately, most existing programs do not present this layering explicitly. What we propose to do is assign logical layering to objects in C programs. This logical layering information would be useful for generating other high level software views such as the association weight of all objects.

The construction of a layering of all global objects is possible; however, we shall initially concentrate on the layering of functions because it gives us a backbone view of the program without other global objects cluttering the view.

The function layering problem is defined as follows:

> Given a set of functions $f_1, \ldots, f_n$, and
> a set of function call relations of the form $f_i \rightarrow f_j$,
> derive an integer *labeling* L such that if $f_i \rightarrow f_j$,
> then $L(f_i) \leq L(f_j)$.

A *topological sorting* algorithm [11] can be applied here to solve the function layering problem. The basic algorithm is the following:

```
l = 1;
F = the set of functions;
while (F nonempty) do
    Find all functions that are not called by any other functions;
    Assign label l to all these functions;
    Remove these functions from F;
    l = l + 1;
done
```

However, the above algorithm does not handle recursive functions properly, i.e. closed paths in the function call graph. To solve this problem, we can collapse all *strongly connected components* into single nodes using the algorithm described in [12] before we apply the topological sorting algorithm.

Figure 7 shows the labels obtained for each function by applying the above layering algorithm. If there is not a direct link between node G and J, then the system represents a perfect set of virtual machines because the functions at each layer do not reference any functions at more than one level below it. This is termed *loss of transparency* in [10], which is a desirable property in a layered system. However, if there is a link between node G and J, then layer 3 does not construct an ideal virtual machine. Note that the result of the layering using the topological sorting is not affected by such a link.

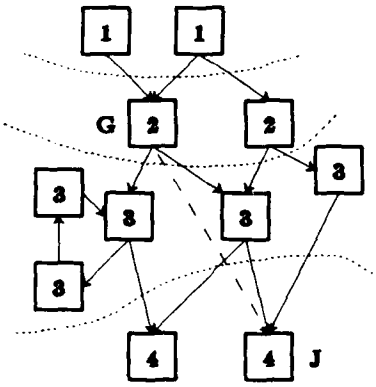


Figure 7: Layering of Functions Using Topological Sorting

The function layering obtained may not correspond exactly to the view of human programmers. However, the labeling gives us some ideas about the *depth* of each function in the program structure. Moreover, the difference in the human view and the logical view could provide hints for potential structure improvements.

If we apply the layering algorithm to all global objects, we can discover unreferenced objects because they become roots in the layering process and thus would be assigned level 1. If an unreferenced object is not a function or is a function but not one of the legal entry points of the system, then it should be removed.

7. Modularization

For the purpose of *information hiding* [13] and ease of compilation, a program is usually partitioned into a set of modules. Each module is a set of logically-related functions and private data. Unfortunately, many existing programs were not designed with modularization in mind. For poorly structured systems, we would like to obtain the logical modularization by reducing the *coupling* between modules and increase the *cohesion* among objects inside a module[14]. Two modules are strongly coupled if there exist many cross references among them. Objects in a module are cohesive if they are logically-related. When a software system becomes large and complex, programmers tend to place newly written functions in wrong modules or group unrelated functions under the same module.

Two problems are associated with incorrect modularization:

(1) *Low Traceability*: Because of unnecessary coupling, many additional modules must be understood to fully understand one module; thus the maintenance cost is higher than necessary.

(2) *Longer compilation time*: Whenever a change must be made, unnecessary modules are likely to be affected. Therefore, to implement a change might require the recompilation of modules which, otherwise, would be unaffected.

To obtain the logical modularization, we start with the idea of cohesion, i.e. trying to identify functions that are logically related. For a small system, a human can easily perform the job. However, for a large software system, software tools are necessary. One method to detect the logical relations between functions is to examine the *degree of sharing* of each function pair. The idea is to count the number of common references between two functions. If this number is large, then chances are good that the two are logically related since they operate on similar objects.

Hutchens and Basili performed a modularity study based on the concept of *used data binding*[15]. A used data binding is defined as an ordered triple (p,x,q) where p and q are functions and x is a variable referenced in both functions p and q. A hierarchy of clusters can be created by calculating dissimilarity matrixes iteratively from used data bindings.

Hutchens and Basili's modularization study on several projects shows a significant degree of correspondence between the automatically generated module structures and those defined by the program developers. However, their study concentrates on the sharing of variables between functions. Our C program database contains all the global object references for each function. As a result, an algorithm can be developed to calculate the sharing of global variables, data types, macros, and functions between each pair of functions and to cluster functions accordingly.

Note that there are as many ways to measure the degree of sharing as there are in measuring the weight of a global object:

(1) *basic sharing*: only the number of shared global objects is counted.

(2) *weighted sharing*: the sum of the actual weight (see Section 4) of shared objects is counted.

(3) *basic association sharing*: the number of objects in the union of the subsystems of the shared references is counted. Note that this is different from simply summing up the basic association weight of all shared references, which would cause some objects to be counted more than once.

(4) *actual association sharing*: the sum of the actual weight of all global objects in the union of the subsystems of the shared references is counted.

For example, Figure 8 shows the basic association weight of each node. Using type (3) calculation, the basic association sharing between function F and function G is 12. We also calculated the basic sharing for all pairs of functions in a subset of an airline

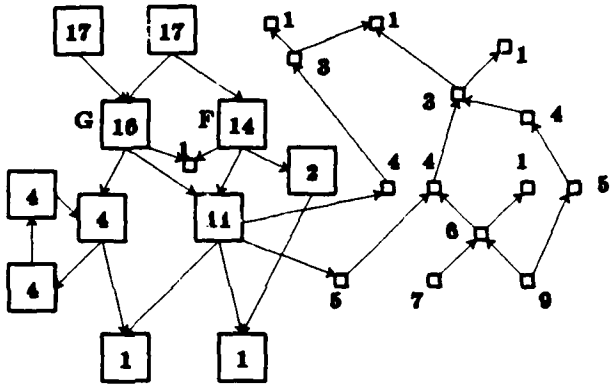


Figure 8: The Basic Association Weight of Each Object

reservation program. This subset includes six major functions: *add_flight*, *subtract_flight*, *empty_flight*, *cancel*, *reserve*, and *manifest*. The result is shown in Table 2. As the table shows, the function *reserve* and the function *cancel* share three function references, nine macro references, and one global variable reference; the total basic sharing is 13. We can immediately guess that these two functions are strongly related due to this strong *binding*.

In the original program, the six functions are partitioned into two modules, with the former three grouped in one module, and the latter three in the other one. Our goal was to compare how the physical modularization differs from the logical modularization using the *basic sharing* measure.

We applied our own clustering algorithm (details are beyond the scope of this paper) based on the basic sharing measure. The final normalized binding strength among the six functions are shown in Figure 9, with 0.60 as the cutoff point. The result suggests that it is more reasonable to assign the two functions *add_flight* and *subtract_flight* in one module and the other four functions in another one. After we examined the airline program in detail, we agreed with the suggestion. The *empty_flight* function is designed to remove all reservations in a particular flight and so it actually operates at the "reservation level". Therefore, it shares a lot of data structures with the three functions *reserve*, *cancel*, and *manifest*. On the other hand, the *add_flight* and *subtract_flight* functions operate at the "flight level"; therefore, they should be grouped together.

It is easy to understand why the author of the airline reservation program created the wrong modularization. People tend to group the three functions ended with "flight" together. However, because of the wrong modularization, whenever we make a change in the data structure associated with reservations, both original modules need to be recompiled. This experiment shows how the difference between logical modularization and the physical modularization can provide hints for possible structure improvements.

REPRODUCED AT GOVERNMENT EXPENSE

Table 2. The Basic Sharing Among Six Functions					
func_name1	*func_name2*	*function*	*macro*	*gbvar*	*total*
add_flight	add_flight	1	8	3	12
cancel	add_flight	0	7	1	8
cancel	cancel	3	1	12	16
empty_flight	add_flight	0	7	1	8
empty_flight	cancel	1	9	2	12
empty_flight	empty_flight	3	10	2	15
manifest	add_flight	0	4	1	5
manifest	cancel	2	7	1	10
manifest	empty_flight	1	7	1	9
manifest	manifest	4	8	1	13
reserve	add_flight	0	6	1	7
reserve	cancel	3	9	1	13
reserve	empty_flight	1	8	1	10
reserve	manifest	2	8	1	11
reserve	reserve	5	10	1	16
subtract_flight	add_flight	0	7	1	8
subtract_flight	cancel	0	7	1	8
subtract_flight	empty_flight	1	7	1	9
subtract_flight	manifest	0	5	1	6
subtract_flight	reserve	0	7	1	8
subtract_flight	subtract_flight	2	8	1	11

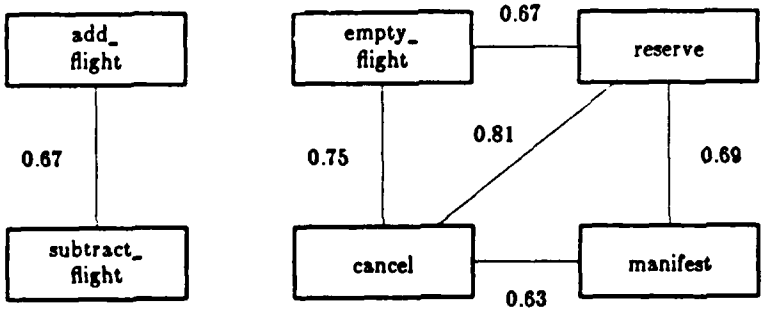


Figure 9: The Binding Strength Between Pairs of Functions

8. Software Restructuring

After the analysis of layering and modularization proposed in the previous two sections, a programmer may wish to restructure his program in order to reduce future maintenance costs. However, software restructuring must be performed with caution. Any change in a module may affect the correct operation of modules in other places as explained before in the section on ripple effects. To tackle this problem, we break down most software restructuring operations into a set of primitive operations, which can be specified precisely. Higher level restructuring operations such as automatic

modularization can be composed from a series of these primitive operations with certain condition checkings using the program database.

If we view a software system as a set of modules, then there are several primitive restructuring operations:

R1. Renaming an object in a module
R2. Inserting a new object into a module
R3. Deleting an object from a module
R4. Moving an object from a module to another module

The rules that govern each of the above operations are shown in the following. Exercising these rules requires accessing information stored in the program database.

R1: Rename(ObjType, ObjName, NewObjName)

(1) Check to see if there is an object of the type *ObjType* with the name *NewObjName*; if true, resolve the conflicts (this requires interactive input from the programmer).

(2) Check to see if there are references to *ObjName*; if true, change all these references to refer to *NewObjName*. Note that the static scope of the object specified by *ObjName* must be considered in determining the references.

R2: Insert(ObjType, ObjName, Scope, Module)

(1) Check to see if there is already an object of the type *ObjType* with the name *ObjName*; if true, rename *ObjName* to a name that does not exist in the module (if *Scope* is static), or to a name that has not been used in any modules of the system (if *Scope* is non-static).

(2) Place object *ObjName* in the specified module.

(3) Insert all objects referenced by *ObjName* by recursively applying rule R2. This is necessary to bring the whole subsystem associated with *ObjName* into the new system if the object is obtained from another existing system.

R3: Delete(ObjType, ObjName, Module)

(1) Check to see if there are any references to *ObjName*; if true, issue warnings and the deletion operation is rejected.

(2) Perform a topological sorting on the objects in the subsystem of the object specified by *ObjName*, with strongly connected components collapsed to single objects.

(3) Construct a *delete set* as follows: Put the object *ObjName* in the delete set. Starting from layer 1 of the subsystem, for each object, check whether it is referred to by only objects in the delete set. If true, add that object to the delete set.

(4) Remove all objects in the delete set.

R4: Move(ObjType, ObjName, OldModule, NewModule)

(1) Delete the object *ObjName* from the original module using R3; however, objects in its delete set is placed in a set B, and its subsystem is specified by the set S.

(2) Insert object *ObjName* and the objects in the set B to the new module using R2. Declare all objects in the set $S-B$ as external references in the new module.

Reasons for the algorithms used in R3 and R4 are left as an exercise to the reader. As we can see, even a simple object movement may require detailed consistency verifications in many functions and modules. We would like to automate these verifications using information available in the program database. High level restructuring can then be accomplished by composing these primitives. An automatic modularization program like the one discussed in the previous section would invoke the required restructuring primitives.

9. Conclusion

We have shown that high level abstractions from programs can be obtained by (1) extracting out relational information from programs to form a software object network, and (2) deriving high level abstractions from the software object network. We also demonstrated how to obtain several interesting abstractions such as subsystems, ripple effects, logical layers and modules from a software object network represented by a C program database. Computation of these abstractions can be speeded up by storing and operating the software object network in the connection memory[16]. These abstractions assist programmers in undersanding the program structure and point out potential areas for improvement. We then demonstrated how rule-based software restructuring can be performed with the relational information stored in the program database. Based on our two-year experience in using the C program database, we believe that a program database is indispensable for programmers who want to have a better view of their programs and who need to automate their software development and maintenance tasks.

Acknowledgement

Many people contributed to the implementation of the C Information Abstraction System. Michael Nishimoto implemented the current C Abstractor; Wen-Ling Chen implemented most of the INFOVIEW commands; Benjamin Chang and Scott Nishimoto are responsible for most commands in the Software Investigator; Lenora Eng and Joo-Seok Song also contributed their work in the early days of the CIA system. We would also like to thank our colleagues in the Berkeley GENESIS group, in particular Atul Prakash and Vijay Garg, and our friends at Columbus Bell Labs, in particular Charlie Fritsch, Bruce Wachlin, Ivan Brohard, Doyt Perry, and Michael Buckley for valuable comments and discussion throughout the development of the CIA system. Benjamin Chang provided many comments on an early draft of this paper. Finally, we would like to thank Professor Vincent Lum, Chairman of the Computer Science Department, and all the faculty and staff in this department for making our stay at Naval Postgraduate School a delightful experience.

References

1. Yih-Farn Chen and C.V. Ramamoorthy, "The C Information Abstractor," *The Tenth International Computer Software and Applications Conference (COMPSAC)*, Chicago, October 1986.

2. Michael Nishimoto and Yih-Farn Chen, "Tutorial on the C Information Abstraction System," *Tech. Report No. UCB/CSD 327*, Computer Science Division, University of California, Berkeley, Spring 1987.

3. Ellis Horowitz and John B. Munson, "An Expansive View of Reusable Software," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, pp. 477-487, September 1984.

4. C.V. Ramamoorthy, Vijay K. Garg, and Atul Prakash, "Reusability Support in GENESIS," *The Tenth International Computer Software and Applications Conference (COMPSAC)*, October 1986.

5. Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, ed. Erik Sandewall, pp. 128-140, McGraw-Hill, Inc., 1984.

6. Luqi, "Normalized Specifications for Identifying Reusable Software," *Tech. Report NPS52-87-007*, Computer Science Department, Naval Postgraduate School, March 1987.

7. S. Yau, "Ripple Effect Analysis of Software Maintenance," *Proc. of 2nd International Computer Software and Applications Conference*, 1978.

8. Norman Loongsung Soong, "A Program Stability Measure," *Proc. Annu. ACM Conf.*, pp. 163-173, 1977.

9. Stephen S. Yau and James S. Collofello, "Some Stability Measures for Software Maintenance," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 6, pp. 545-552, November 1980.

10. D.L. Parnas and D.P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Comm. ACM*, vol. 18, no. 7, pp. 401-408, July 1975.

11. Donald E. Knuth, *The Art of Computer Programming (2nd Ed.)*, 1, pp. 258-265, Addison-Wesley Publishing Company, 1973.

12. Sara Baase, *Computer Algorithms*, pp. 157-163, San Diego State University, 1978.

13. D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.

14. Edward Yourdon and Larry Constantine, *Structured Design*, pp. 84-141, Prentice-Hall, Inc., 1979.

15. D.H. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8, pp. 749-757, August 1985.

16. W. Daniel Hillis, "The Connection Machine," *A.I. Memo #646*, MIT Artificial Intelligence Laboratory, September 1981.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943-5100	2
Office of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943-5100	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	10
David K. Hsiao Code 52Hq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
Y. F. Chen Computer Science Division (EECS) University of California Berkeley, CA 94720	20
Chief of Naval Research 800 N. Quincy St. Arlington, VA 22217	1

END

FEB.

1988

DTIC