Information flow metrics for the evaluation of operating systems' structure

Sallie Marie Henry

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation

Henry, Sallie Marie, "Information flow metrics for the evaluation of operating systems' structure " (1979). Retrospective Theses and Dissertations. Paper 7213.

This Dissertation is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
HENRY, SALLIE MARIE
INFORMATION FLOW METRICS FOR THE EVALUATION
OF OPERATING SYSTEMS' STRUCTURE.

IOWA STATE UNIVERSITY, PH.D., 1979
Information flow metrics for the evaluation
of operating systems' structure

by

Sallie Marie Henry

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College Iowa State University
Ames, Iowa

1979
TABLE OF CONTENTS

CHAPTER I. INTRODUCTION

System Design and Structure 3
Software Metrics and Information Flow 11
Statement of the Problem 14
Plan of the Thesis 15

CHAPTER II. COMPARISON OF ORDERING RELATIONS

Deficiencies of Other Ordering Relations 18
Information Flow Definitions 22
Informal Example of Information Flow 23
Analysis of Calls Detected in Information Flow 25
Comparison of Calls, Dependency, and Information Flow 29

CHAPTER III. MECHANISMS FOR INFORMATION FLOW ANALYSIS

Generation of Relations 37

General format of a relation 37
Examples for relation generation 41
Actual relations for selected UNIX routines 49

Algorithm for Generation of the Information Flow Structure 55

Algorithms for Information Flow Structures 59

Derived calls 60
Local flows 61
Global flows 62

Memoryless Procedures 63

CHAPTER IV. MEASUREMENTS OF THE UNIX OPERATING SYSTEM

Measurement Definitions 65
### Measurement of UNIX Procedures

- Page 67

### Measurements of UNIX Modules

- Page 72

### Interface Measurements

- Page 80
  - Protocol interface complexities
  - Binding
  - Coupling

### Level Complexities

- Page 88

### CHAPTER V. EMPIRICAL JUSTIFICATION OF INFORMATION FLOW

- Correlation of Errors to Measurements
  - Page 92
- Correlation of Design Flaws to Measurements
  - Page 94
    - Module violation
    - Redesign of the BUF module
    - Redesign of the INODE module

### CHAPTER VI. CONCLUSIONS AND SUGGESTIONS FOR ADDITIONAL INVESTIGATION

- Page 108

### BIBLIOGRAPHY

- Page 112

### ACKNOWLEDGMENTS

- Page 117

### APPENDIX A. "C" CODE FOR BREADA AND INCORE

- Page 118

### APPENDIX B. UNIX MODULE DESCRIPTIONS

- Page 119

### APPENDIX C. GLOBAL FLOWS FOR UNIX MODULES

- Page 120

### APPENDIX D. UNIX MODULE COMPLEXITIES

- Page 121

### APPENDIX E. REDESIGN OF THE ITRUNC PROCEDURE

- Page 127

### APPENDIX F. REDESIGN OF THE NAMEI PROCEDURE

- Page 130
# LIST OF FIGURES

| Figure 2.1. | An example to illustrate the USES relation. | Page 19 |
| Figure 2.2. | The USES relations generated from the example in Figure 2.1 | Page 20 |
| Figure 2.3. | An example of the dependency relation | Page 21 |
| Figure 2.4. | An example of information flow | Page 24 |
| Figure 2.5. | Information flow relations generated from the example in Figure 2.4 | Page 25 |
| Figure 2.6. | Dependency relations generated for the example in Figure 2.4 | Page 25 |
| Figure 2.7. | Call from A to B with parameters | Page 26 |
| Figure 2.8. | Example of a missed call | Page 27 |
| Figure 2.9. | A calling structure for three modules | Page 30 |
| Figure 2.10. | Possible flow structures corresponding to Figure 2.9 | Page 30 |
| Figure 2.11. | Example 1 of the flow structure | Page 31 |
| Figure 2.12. | Example 2 of the flow structure | Page 31 |
| Figure 2.13. | Example 3 of the flow structure | Page 32 |
| Figure 2.14. | Comparison of calls and flows in UNIX | Page 33 |
| Figure 2.15. | Composition of functions for information transfer | Page 34 |
| Figure 2.16. | Relationship among subsets of the connections within a system | Page 35 |
| Figure 3.1. | Example 1 of relation generation | Page 43 |
| Figure 3.2. | Example 2 of relation generation | Page 44 |
| Figure 3.3. | Example 3 of relation generation | Page 46 |
| Figure 3.4. | Example 4 of relation generation | Page 48 |
Figure 3.5. Pseudo code for UNIX routines .................. 50
Figure 3.6. Algorithm for generation of the information flow structure .................. 56
Figure 3.7. An information flow structure for Figure 3.4 .................. 58
Figure 3.8. Analysis of calls .................. 61
Figure 3.9. Analysis of local flows .................. 62
Figure 4.1. Procedure complexities .................. 68
Figure 4.2. Distribution of procedure complexities .................. 71
Figure 4.3. UNIX module descriptions .................. 73
Figure 4.4. Global flows for UNIX modules .................. 74
Figure 4.5. Module complexities .................. 76
Figure 4.6. Percent of module complexity for largest procedures .................. 80
Figure 4.7. Description of the interface between two modules .................. 82
Figure 4.8. Protocol interface complexities .................. 84
Figure 4.9. Binding through protocol interface .................. 84
Figure 4.10. Binding through direct flows .................. 87
Figure 4.11. Coupling between UNIX modules .................. 87
Figure 4.12. Partial hierarchy for UNIX .................. 88
Figure 4.13. Graph of accumulated complexities for the UNIX hierarchy .................. 90
Figure 4.14. Summary of the measurements and corresponding features .................. 91
Figure 5.1. Correlation of module violations to errors .................. 93
Figure 5.2. Correlation of procedure complexity to errors .................. 94
| Figure 5.3. | Distribution of fan-in and fan-out for GETBLK and BREAD. | 97 |
| Figure 5.4. | Fan-in and fan-out for GETBLK and BREAD with additional level of abstraction. | 99 |
| Figure 5.5. | Graph of accumulated complexities for the revised hierarchy for UNIX with redesign of BUF. | 101 |
| Figure 5.6. | Complexities for the revised NAMEI procedure. | 102 |
| Figure 5.7. | Graph of accumulated complexities for the hierarchy for UNIX with implementation change to NAMEI. | 103 |
| Figure 5.8. | Complexities for the new level of abstraction in INODE. | 104 |
| Figure 5.9. | Accumulated complexities for UNIX hierarchies. | 105 |
| Figure 5.10. | Complexities for design and implementation changes to NAMEI. | 107 |
CHAPTER I. INTRODUCTION

One of the most pressing problems facing computer scientists today is the high cost of developing and maintaining software. As hardware costs rapidly decline the expanding community of computer users will continue to generate an increasing demand for low cost, high quality software. One of the most important aspects of software quality is its reliability. Software reliability, in turn, focuses attention both on the rate of failure of the software and also on the associated maintenance costs. Fortunately, cost control and quality assurance are compatible objectives which are both achieved when software complexity is controlled. An essential tool for establishing this control is the development of a complexity measurement mechanism which will identify weaknesses in the design and implementation of a system.

This research will concentrate on the development of a measurement tool for operating systems. Operating systems were chosen since they form the interface between hardware and application programs and, as such, are important objects of study in software design. In addition, operating systems in general consist of more lines of code than most application programs and tend to be more complex both in design and in the interrelationships of the parts. Thus, the mechanism developed to evaluate operating systems' structure might also have value when applied to application programs.

Past and present research in software reliability has concentrated on two areas: The development of a structured design methodology, and
development of a practical set of software metrics. A design methodology aims at comprehensible, high quality designs which, when properly implemented, lead to quality software. If usable metrics are also available, the quality of that software can be monitored and quality standards can be enforced throughout the system design and implementation.

A design methodology implicitly uses an ordering relation which establishes a certain structure on the components of the system. The designers and implementors understand the system in terms of this structure. Thus, the ordering relation provides part of the basis necessary for measuring system complexity. This ability to measure system complexity implies the capability to locate weaknesses in the system design and to determine the level of software quality. However, the ordering relation used in the design process is not a sufficient basis for measuring complexity. Contained within the structure of a system, and integral in the development of complexity metrics, are the more detailed connections among system components. The comparison between various ordering relations is examined in Chapter II.

This research is an investigation into the quantitative evaluation of software design and implementation. It provides mechanisms for observing the information flow between operating system components and develops a set of design metrics to measure software quality. The background for this research is surveyed in the remainder of this chapter emphasizing: 1) previous work in system design and structure, and 2) current work in software metrics and information flow.
In order to understand a system, a structure must be placed on that system. As discussed by Parnas (1974), the word structure refers to a partial description of a system showing it as a collection of parts and some relations between those parts. Such a structure is called hierarchical if a relation on pairs of the parts $(R(\alpha, \beta))$ allows a definition of levels by saying that:

1. level 0 is the set of parts $\alpha$ such that there does not exist a $\beta$ where $R(\alpha, \beta)$, and
2. level i is the set of parts $\alpha$ such that
   a. there exists a $\beta$ on level i-1 such that $R(\alpha, \beta)$, and
   b. if $R(\alpha, \gamma)$ then $\gamma$ is on level i-1 or lower.

Several operating systems have illustrated various hierarchical structures. Two hierarchies are presented below the "THE" operating system: the calling or program hierarchy and the Habermann hierarchy. In addition, hierarchies associated with the Multics operating system, the RC4000 operating system, and the Venus operating system are also discussed.

A layered design is a well-known hierarchical design structure. In this case the relation, $R$, is simply the calling structure. The calling program is able to ignore how the called program works and the called program can make no assumptions about the calling program. With a program hierarchy each level serves as a virtual machine for the next higher level. This calling hierarchy or program hierarchy was demonstrated by Dijkstra with the implementation of the "THE" multiprogramming system.
Another relation used to give a hierarchical structure to a system is the Habermann hierarchy (Parnas, 1974). This relation is based upon the philosophy that A "gives work to" B. Each level gives work to the level below it. When this hierarchical structure was applied to the "THE" system, the hierarchical structure was the same as a program hierarchy. However, this would not necessarily be the case if the Habermann hierarchy was applied to another system.

Another example of a hierarchical system is the protection hierarchy illustrated by the Multics operating system. There are two levels in Multics; the first level is called the supervisor, and the second consists of the users. The hierarchy exists at the supervisor level, and within the supervisor are levels called rings. The innermost ring (the lowest level) consists of those procedures which can access certain data. The relation, R, represented in Multics is a "can be accessed by" relation. This protection hierarchy is not a program hierarchy, in that calls can take place in both directions across the rings (Daley, 1972).

The RC4000 multiprogramming system has a hierarchy based on ownership of memory. In this case the relation, R, is A "allocates a memory region to" B. This hierarchy allocates specific areas of memory to processes primarily due to the lack of virtual memory (Brinch Hansen, 1970; Lauesen and Brown, 1975).

The design of the Venus operating system was based on a hierarchy of levels of abstraction, or a calling hierarchy. A level is defined by the abstraction it supports and the resources it requires for that
abstraction. Lower levels are not aware of the resources used at higher levels. Levels of abstraction are used in the resource management to provide users with virtual devices (Liskov, 1972b).

In addition to the work described above in implementing specific hierarchies, research has been active in exploring various concepts used in the design of hierarchically-structured systems. These concepts include the notions of a module, a program family, kernels, system structuring, specification techniques, and language features. Each are discussed, in turn, below.

The term module has been used extensively in the literature with various definitions. Even without a specific definition for module, the concept of modularization has been generally accepted as a necessary ingredient of good structure. Modularization implies a system which: 1) allows internal module design decisions to be made independently of other modules; 2) increases comprehensibility since a module may be understood in terms of its effects and not its implementation; the system may be understood one module at a time; 3) creates system stability since each module can be tested and debugged independently of other modules; and 4) increases system flexibility because the implementation of one module can be drastically changed without affecting other modules. These benefits imply greater ease in detailed design, and an increase in speed of implementation, more adequate testing and debugging, and less costly maintenance. These all, in turn, are used to enhance reliability.

Parnas (1972a) pursues three issues with regard to module design structures. First, the criteria used in decomposing systems into modules
can greatly influence the maintainability and comprehensibility of a system. It is very important in the original design of a system to choose the "best", not always the most conventional, decomposition. The second aspect of modularization discussed by Parnas (1972b) is the connection between modules. The connections between modules are defined by the assumptions the modules make about each other. The connections are the distribution of information within the system and consist of the flow of control, passed parameters, and shared data structures. A third concept discussed by Parnas (1975) is the use of transparency with respect to the design of hierarchical systems. In designing a system with a bottom-up approach, each layer in the system provides a virtual machine below it. A layer is transparent if any sequence of states which could be obtained by programming the base machine can also be obtained by programming the virtual machine. In the cases of synchronization and interrupts, the loss of transparency is a goal of operating system designers. In other instances, the existence of transparency is the goal.

A popular concept presented by Parnas et al. (1976) and by Habermann et al. (1976) is that of program families. This concept is discussed primarily with respect to operating systems, and is similar to the concept of a family of upward compatible hardware systems. Not all machines require a general operating system. Some have special purpose functions requiring only a special purpose operating system. The idea of families implies that the members of a family will share as much of the software as possible. This shared software comprises the lowest layers in a hierarchical system, particularly those layers which deal with
synchronization, process management, and address space management. The hardware, together with the shared software, forms a virtual machine for the members of a family. The low layers of an operating system are at times referred to as the kernel or nucleus of an operating system.

There are several examples of kernel operating systems. The kernel of the Hydra multiprocessor operating system (Wulf et al., 1974; Wulf and Pierson, 1975) was designed to contain the set of mechanisms from which an arbitrary set of operating system facilities and policies can be conveniently constructed. Hydra also implemented as part of the kernel, a sophisticated set of protection mechanisms (Levin et al., 1975). The RC4000 system was previously described as a hierarchical system based on ownership of memory; however, it also implements the concept of a nucleus with a family of special purpose operating system functions. The purpose of this system nucleus is to implement the fundamental concepts of processes, communication among processes, creation, control, and removal of processes. Another nucleus operating system is described by Shaw (1975) where the nucleus has three components; a set of primitive operations and data structures for management of processes and resources, a library of service routines for implementing a variety of resource management strategies, and facilities for cleanly handling all input-output and interrupts.

Significant research has addressed the concept of system structuring. The resulting design methodologies are based on ordering relations, some of which differ from the intuitive idea of hierarchy by recognizing that there exists connections within systems that are not discovered by a program (calling) hierarchy. These ordering relations are "uses", 
presented by Parnas (1976b) and the "dependency relation" described in a report by Janson (1977) discussing work done by Feiertag.

The uses relation is defined such that A uses B if correct execution of B may be necessary for A to complete its work. Alternately, A uses B if there exists situations in which correct functioning of A depends upon the availability of a correct implementation of B. Uses differs from calls in that there are situations where a call does not imply a uses relationship. An example of this is where A calls B, but B's function does not affect A (B might simply print an error message). An example where A uses B but does not call B, occurs in the interrupt handling process, where A is a device driver and B is an interrupt service routine.

The uses relation does not consider the affect of shared data structures. If A and B do not use each other but share a data structure, one cannot verify the correctness of A unless the management of the shared data structure is understood. This requires verifying the correctness of B.

The dependency relation is more precise than uses and also identifies more connections between the components of a system. Module A depends on module B if the correctness of A cannot be verified without verifying the correctness of B, i.e., if A makes any assumption about the operation of B. More precisely, there are three cases of dependency:

1. if A transfers control to B and expects B to return control and potential results after it has completed its computation,
2. if A sends a message to B and expects to receive a reply message
with potential results, or

3. if A shares a data structure with B and expects B not to effect the integrity of that data structure.

The dependency relation recognized that there were some implicit connections in systems that were not discovered by previous structuring techniques. Dependency attempts to find these connections by precisely defining a relation. The uses and dependency relations are discussed in more detail in Chapter II.

The development of formal specification techniques is a critical component of quality design. It is believed that formal specification techniques will come to play a fundamental role in the development of reliable software. The goals of formal specification techniques are (Parnas, 1972b):

1. the specification must provide the user with all the information he needs and nothing more,

2. the specification must provide the implementor with all the information he needs and nothing more, and

3. the specification should view the program in terms of the user and the implementor alike.

Liskov (1975) presented a survey of various formal specification techniques and showed that these specification techniques are necessary for proving the correctness of a system and for better comprehensibility of the system. To prove the correctness of an entire operating system is too complex to be practical, however, Walter et al. (1975) and Saxena and Bredt (1975) define the formal specifications necessary for an operating
system kernel to be provably secure. Saxena used an extended version of Pascal to define the specifications for his system. With the use of this language, the modules and module interfaces could be explicitly defined to reveal the connections between the system components. Parnas (1976a; 1977) has researched the development of formal specifications for an operating system family. The lowest levels of an operating system, when precisely specified, represent a reliable virtual machine for the support of the operating system family. The formal specification techniques pertinent to this research are discussed in Chapter VI.

The use of a specific programming language, such as an extended version of Pascal, together with the use of formal specification techniques, is necessary to prove the correctness of programs. Structured programming insists on developing software in a hierarchical structure. Infante and Mantanare (1975) developed axioms to prove programs correct one level at a time. These axioms allow any level to be proven correct without looking at any other level. This effort gives definite relationship to the interfaces at the different levels in a structured program. Robinson et al. (1975) presents a methodology for the design and development of a secure operating system. The methodology is based on the hierarchical decomposition of the system which allows a proof of correctness for each level. It is the intent of Robinson's et al. research to provide feasible tools for proving correctness, thus developing reliable and secure software.
Software Metrics and Information Flow

The design methodology and concepts outlined above play a major role in the software reliability. The second ingredient of reliability is the development of a set of measurements to evaluate the quality of the software design. The quantitative aspects of system structure have been explored under the title of software quality assurance. At issue is the development of quantitative measures of software quality. Applying these metrics to the design structure can identify areas of probable weaknesses in the design, and give an unbiased evaluation of design quality. To evaluate the structure of the software it is necessary to understand the connections among the components of the system.

Myers (1976) and Stevens, Myers and Constantine (1974) have stated that one of the requirements for reliable software is a "good" design. Goodness is based on measures of module cohesion, module strength, and module coupling. Cohesiveness deals with the binding of the elements within the module or the connections between the elements within a module. The connections between the modules must be known in order to understand the relationships and dependencies within the system. By minimizing the connections between the modules, the system will be easier to understand and will decrease the number of paths through which errors might propagate through the system. Module coupling is a measure of the strength of the associated connections from one module to another. Module strength involves analyzing the function or functions of the module; it describes what the module does. The goal is to design the system so that each module has one function. Using these measures a structured
design methodology has been developed to maximize the "goodness" of the system.

Yin and Winchester (1978) have developed several metrics used to evaluate design structure. These metrics consider the complexities of the module with respect to the connections between the modules and the data structures used by the modules. These metrics were applied to two systems, one large system with 1000 modules and a smaller system. There was a very high correlation (0.98 for the first system and 0.99 for the second system) between the module complexity and the number of errors detected. These metrics provide specific guidelines for enhancing software design quality. Some additional metrics proposed by Yin and Winchester involve the calls to and from a given module and the data structures that interface with the module. No attempt has been made to validate these additional metrics since the necessary calculations are very difficult to derive without automated aids.

Substantial investigation has been done in the area of measuring program complexity for programmer performance in the program implementation and subsequent program modification. Halstead developed a set of metrics to measure program complexity based on the number of conceptually unique operands, the number of operators, and the total frequency of the operators and operands (Halstead, 1977). These measurements have been successful in predicting programmer effort corresponding to problem complexity (Woodfield, 1979).

McCabe (1976) developed a definition of complexity measurement based on the decision structure of a program. These measurements count the
number of basic control path segments which, when combined will generate every possible path through a program. McCabe's metrics relate to the assumed mental difficulty of a programming task, since additional control paths make a program more difficult to understand.

In an experiment on programmer performance Halstead's measurements, McCabe's measurements, and program length as measured by the number of statements were compared (Curtis et al., 1979). The results of this experiment showed that the number of statements in a program proved to be as strongly related to performance on the experimental task as the Halstead and McCabe metrics.

In summary, there has been substantial research in the area of reliable software. Both the techniques for design methodologies and software quality metrics are dedicated to increasing the comprehensibility and maintainability of software. Complexity measurements revealing weaknesses in system design and system implementation are presented in Chapter IV. A critical element in the structure of a system is the connection between the modules and layers. The research for this thesis uses the flow of information within a system to reveal the connections between modules. Therefore, this chapter must also include the previous work done in information flow.

Denning (1976) and Denning and Denning (1977) investigated information flow control to be used in proving security in computer systems. An information flow policy specifies a flow relation representing all permissible flows between security classes. Denning presents a compile-time mechanism that detects the flow of information in violation of the flow
policy. By verifying a secure flow of information at compile-time, there is a corresponding decrease in the need for run-time security checking. Information flow has also been investigated for both sequential and concurrent software by Shaw (1978). This investigation deals with the development of a language called flow expressions, which describes the flow of information through the system. The flow expressions provide another tool for software design, analysis and understanding.

Statement of the Problem

The thesis of this research is that a set of measurements based on the flow of information connecting system components can be used to evaluate software design and implementation.

The information flow measurements permit evaluation of the complexities of the procedures and modules within the system, and the complexities of the interfaces between the various components of the system. Design and implementation features evident from the information flow measurements reveal difficulties in the following areas: poor functional decomposition of procedures and modules, improper modularization, poorly designed data structures, and modifiability.

This investigation is motivated by the author's conviction that an automated set of metrics is a necessary tool for the designers and implementors of software for use in the evaluation of the system structure. Reliability of software is an important goal of all software development. In order to have reliability in software, it is essential that
all of the connections, both explicit and implicit, between components in a system be known. By analyzing the information flow through a system the implicit connections, not uncovered by other ordering relations, are discovered. The analysis of information flow is explained, and with the connections unveiled in this analysis, a set of measurements is developed to discover flaws in the system structure. The information flow analysis and corresponding measurements are presented for the UNIX operating system (Lions, 1977).

The UNIX operating system was chosen as the vehicle for the information flow analysis for several reasons. First, UNIX is written in a high level language which makes the generation of the relations much easier. Second, UNIX is a large enough operating system upon which to base a meaningful experiment and yet is small enough to be manageable for this project. The third reason for selecting UNIX is that it is a viable software system designed for use and is not a toy or experimental system. Fourth, UNIX is universal in that it is installed in many environments and on several different machines. The fifth reason is the functionality of UNIX. UNIX contains a powerful I/O system, a minimal amount of memory management, and some protection features.

Plan of the Thesis

This research is presented in the following framework. In the next chapter, a comparison of various ordering relations is given. Attention is focused on the different connections which are observed in a system and the types of connections observed with a calling structure, the
dependency relation, and information flow are compared.

Chapter III defines the mechanisms needed for analyzing the flow of information through a system. The analysis consists of three phases: 1) the generation of relations, 2) developing the algorithm necessary for the generation of the information flow structure, and 3) presentation of the algorithms for analyzing the information flow structure.

Chapter IV discusses the metrics developed to measure software quality and displays the complexity measurements obtained by performing an information flow analysis on the UNIX operating system. The measurements are presented for procedures, modules, and module interfaces.

Chapter V presents an empirical justification of the measures presented in Chapter IV. This justification focuses attention on both the correlation of error detection to the complexity measurements and the effect of the measurements to evaluate the system design and implementation.

Chapter VI summarizes the conclusions of this research, poses some additional questions, and speculates on future directions of research into the use of information flow with respect to additional metrics needed to measure software design structures.
CHAPTER II. COMPARISON OF ORDERING RELATIONS

The term "structure" refers to the atomic elements, modules, comprising the system and the ordering relation, connections, defined between these elements. Thus, two systems have different structure if they are constructed from different components or if the same components are connected in different ways. There are, of course, many different ways of defining the "connections" between modules. For example, the calling structure shows the control flow connections within a system but leaves many other possible connections, e.g., through shared data items, undetected. In a more general sense, Parnas defined the connections between modules to be "the assumptions the modules make about each other" (Parnas, 1971).

Recall that the connections between modules are fundamental components of system complexity. Extensive connections between modules make the interactions among the modules more difficult to understand or control and correspondingly, more complex and less reliable. On the other hand, reducing the connections between modules narrows the possible ways modules may affect each other. Thus, the ordering relation which defines the visible connections between modules is a central part of metrics for measuring software complexity. This ordering relation must possess two characteristics. First, the relation must be automatable, implying that the definition of the relation must be, at least, precise. Second, the relation must be robust, revealing as many connections between modules as possible.
In this chapter, the uses (Parnas, 1976b) and the dependency relations (Janson, 1977) are discussed in some detail with examples emphasizing the observed connections between the modules. The information flow relation will be informally described; and the connections it observes between modules will be presented. The mechanisms for generating these information flow connections are presented in the next chapter. A comparison of the connections observed for calls, dependency, and information flow will be presented in relation to all the possible connections within a system.

Deficiencies of Other Ordering Relations

Recall that the uses relation is defined as: A uses B if A requires a correct version of B. As noted in Chapter I, uses differs considerably from a calling relation and, by definition alone, uses reveals more of the total connections among modules.

Uses is a very interpretive relation as seen in the following example. Figure 2.1 shows two modules, A and B, and their relationship to each other. The example is concerned only with the general functions performed by A and B. Module A deposits some information in a data structure, D, then calls module B passing to it some parameters. Module B completes its computation using the information it received from A and information it retrieved from D.

The definition of uses is open to various interpretations, and these interpretations imply the existence of different uses relations between modules A and B. A possible interpretation for module A is whether or
Figure 2.1. An example to illustrate the USES relation

not to assume global or local specifications for the assumptions A makes about B. If A has global specifications for B, this implies that A has knowledge of the function of B and that A cannot expect B to perform correctly. Local specifications means that A knows only how to call B and implies that A assumes that B functions correctly. Module B is concerned with whether or not the input it receives from A and from D is correct.

Figure 2.2 displays the various uses relations generated from the interpretations of global or local specifications for A and correct or incorrect input for B. The $\rightarrow$ designates the existence of a uses relation and $\rightarrow\leftarrow$ denotes no uses relation. The assumption of global specification implies that A uses B but the assumption of local specification implies that A does not use B. Assuming correct input for B indicates that B does not use A; however, incorrect input shows that B uses A. All possible combinations of these two interpretations result in all four possible uses relations. Uses appears to reveal more connections within a system than a calling relation but, as illustrated by the above example, the definition lacks precision. To the extent that the uses relation is
made more precise, it is drawn toward the dependency and information
flow relations presented below. Therefore, the uses relation will be
abandoned in favor of them.

\[
\begin{array}{c|c|c}
\text{Global} & \text{Local} \\
\text{specifications} & \text{specifications} \\
\hline
\text{correct} & A \rightarrow B & A \leftarrow \rightarrow B \\
\text{input} & B \leftarrow \rightarrow A & B \leftarrow \rightarrow A \\
\hline
\text{incorrect} & A \rightarrow B & A \leftarrow \rightarrow B \\
\text{input} & B \rightarrow A & B \rightarrow A \\
\end{array}
\]

Figure 2.2. The USES relations generated from the example in Figure 2.1

Unlike the uses relation, the dependency relation, A depends on B, is precise and defined specifically if at least one of the following three conditions hold:

1. if A transfers control to B and expects B to return control and potential results after it has completed its computation,
2. if A sends a message to B and expects to receive a reply message with potential results, or
3. if A shares a data structure with B and expects B not to effect the integrity of that data structure.

If the language used to describe a system contains special constructs for message passing, the second of these conditions would not be difficult to automate. However, if message passing is achieved using shared buffers with semaphores, for example, it would be extremely difficult to recognize instances of message passing. With this exception, the definition of the dependency relation is quite precise and dissolves any of
the interpretation problems encountered with the uses relation. In both the uses and dependency relations, the general philosophy is to discover the connections between modules. Since dependency is much more precise than uses, the dependency relation will be the primary target of the comparisons presented later in this chapter.

Although the dependency relation reveals many connections in a system, it does not expose all of the connections. For example, three modules, A, B, and C, and their connections are shown in Figure 2.3. A calls C passing some parameters. C performs some computation using information from a local data structure, D, and returns a value to A. Subsequently, A calls B passing the value that C returned. Clearly, dependency will show the connection between A and C and the connection between A and B. However, dependency will not show any connection between C and B. Even though the connection between C and B is indirect, a connection does exist since B must make some assumption about its input which comes indirectly from C.

![Figure 2.3. An example of the dependency relation](image-url)
Information Flow Definitions

In order to understand the concept of information flow, it is necessary to establish some terminology to distinguish between the various types of information flow. There are basically two types of flows, global flows and local flows. The set of local flows is partitioned into two disjoint classes of direct local flows and indirect local flows. The following definitions describe these terms.

Definition 2.1: There is a global flow of information from module A to module B through a global data structure, D, if A deposits information into D and B retrieves information from D.

The term "effective" parameter is used in the following definitions. To define "effective" at this point will cloud the effort being made to distinguish the types of information flow. It is suggested that the reader substitute the intuitive concept of parameter for "effective" parameter. Also, the idea of a module "utilizing" an output value is mentioned in the definitions, and again, the reader is asked to substitute the intuitive notion of utilize. The exact definitions are presented later in this chapter.

Definition 2.2: There is a local flow of information from module A to module B if one or more of the following conditions hold:

1. if A calls B passing an effective parameter,
2. if B calls A and A returns a value to B which B subsequently utilizes, or
3. if C calls both A and B passing an output value from A to B.
Definition 2.3: There is a direct local flow of information from module A to module B if condition 1 of definition 2.2 is used for a local flow.

Definition 2.4: There is an indirect local flow of information from module A to module B if condition 2 or condition 3 of definition 2.2 is used for a local flow.

These definitions are illustrated by the example presented in the following section.

Informal Example of Information Flow

In this section, the concepts of local flows and global flows are informally presented by example. The mechanisms of deriving the local and global flows are bypassed here but are presented in the next chapter.

Figure 2.4 shows six modules, A, B, C, D, E, F; a data structure, DS; and the connections among these modules and the data structure. Module A retrieves information from DS and then calls B passing an effective parameter; module B then updates DS. C calls D passing an effective parameter. Module D calls E with an effective parameter and E returns a value to D which D then utilizes and passes to F. The function of F is to update DS.

The direct flows generated for this example are:

\[ A \rightarrow B, \]
\[ C \rightarrow D, \]
\[ D \rightarrow E, \]
\[ D \rightarrow F. \]

These flows are simply the ones observed in a calling structure.

The indirect local flows in this example are:
Figure 2.4. An example of information flow

\[ E \rightarrow D, \]
\[ E \rightarrow F. \]

The first flow results when \( E \) returns a value and \( D \) utilizes the value (condition 2 of definition 2.2) and the second flow results from condition 3 of definition 2.2. The global flows are:

\[ B \rightarrow A, \]
\[ F \rightarrow A. \]

Both \( B \) and \( F \) update DS and \( A \) retrieves information from DS. The information flow relations generated from this example are displayed in Figure 2.5.

For the same example the dependency relations generated are displayed in Figure 2.6. The flow of control relations are the same as the calling relations; the data structure relations are generated since \( A \) and \( B \), and \( A \) and \( F \), share a data structure. Note that dependency
does not reveal the indirect connections between E and D, and E and F.

Global flows
B \rightarrow A
F \rightarrow A

Local flows
Direct
A \rightarrow B
C \rightarrow D
D \rightarrow E
D \rightarrow F

Indirect
E \rightarrow D
E \rightarrow F

Figure 2.5. Information flow relations generated from the example in Figure 2.4

Flow of control relations
A \rightarrow B
C \rightarrow D
D \rightarrow E
D \rightarrow F

Data structure relations
A \rightarrow B
A \rightarrow F

Figure 2.6. Dependency relations generated for the example in Figure 2.4

Analysis of Calls Detected in Information Flow

Many software development and analysis techniques are based on a calling structure which, barring recursion, define a hierarchical layering of system components. The calling structure is also an element in the definition of the dependency relation. In this section, necessary and sufficient conditions will be established which determine when a call relation is visible from an information flow viewpoint.

A call from module A to module B is examined under two conditions,
first, module A passes parameters to module B, and second, module A does not pass parameters to module B. Within each case there is an additional assumption, that is, whether B is a function returning a value or not.

Figure 2.7 displays skeleton code for a call from module A to module B. Assume that B uses at least one of its parameters to update a data structure. This means that either B directly updates a data structure using information received from its parameters or, by a sequence of calls, a data structure is updated based on information passed from B's parameters. The sequence of calls may involve any number of modules.

\[
A() \\
. \\
. \\
B(k_1, k_2, \ldots, k_n) \\
. \\
. \\
\text{where } n \geq 1.
\]

Figure 2.7. Call from A to B with parameters

In the condition where module A calls module B passing parameters and where B is not a function returning a value, the call from A to B is not missed if for some \( i, 1 \leq i \leq u \), \( k_i \) receives information from one of the following:

1. one of A's parameters,
2. a data structure,
3. a constant, or
4. a returned value from a third module which is modified in A prior to the call.
The call is missed if for all $i$, $1 \leq i \leq n$, $k_i$ is a return value from a function and $k_i$ is not modified by $A$. Skeleton code illustrating an example where a call will be missed is given in Figure 2.8.

```
A()
    .
    .
    ki = C()
    .
    .
B(..., ki, ...)
    .
    .
```

Figure 2.8. Example of a missed call

If $B$ is a function returning a value, the call from $A$ to $B$ is not missed if any $k_i$ receives information from $A$ following one of the conditions specified above. Furthermore, the call is not missed if the value returned from $B$ is used for one of the following:

1. to update a data structure in $A$,
2. to update one of $A$'s output parameters, or
3. to update a returned value from $A$.

The call is missed if all of $B$'s parameters follow the format specified in Figure 2.8 and if the returned value is not used for any of the purposes stated above.

The second condition of a call from module $A$ to module $B$ is examined if module $B$ has no parameters. Under the assumption that $B$ is not a function returning a value, the call is always missed. If $B$ does return
a value and if that value is used as specified above, the call will never be missed.

With the preceding explanation of when a call is detected within the information flow analysis, the terms "effective" parameter and "utilizing" an output value which were used earlier in this chapter can now be defined.

Definition 2.5: Module A calls module B passing an effective parameter if that parameter received information from one of the following:

1. one of A’s parameters,
2. a data structure, or
3. a constant,
4. a returned value from a third module which is modified in A prior to the call.

Definition 2.6: If module A calls module B and B returns a value to A, A utilizes the returned value if A uses the returned value for one of the following:

1. to update a data structure,
2. to update an output parameter, or
3. to update a return value.

To summarize, the call from A to B is detected if one of the parameters to B is an effective parameter, or if A utilizes the returned value from B.
Comparison of Calls, Dependency, and Information Flow

The previous section demonstrated that an information flow relation does not completely subsume a calling relation and, by extension, does not subsume the dependency relation. However, recall that the example presented in Figure 2.4 showed that information flow was not subsumed by the dependency relation or by a calling relation. Since these relations are not totally ordered, this section will attempt to describe, both theoretically and empirically, the similarities and differences among calls, dependency, and information flow.

As a first point of comparison between calls and information flow it can be easily demonstrated that information flow is a much more sensitive relation. For the given calling structure with three modules in Figure 2.9, module C calls module A and module B. Figure 2.10 shows that there are 64 different flow structures for this single calling structure. This example demonstrates that information flow exhibits the more detailed and specific connections within a system.

Three examples are presented in order to explain the various flow structures in Figure 2.10. The flow structure together with pseudo code necessary for module C to achieve the flow structure is presented.

There is only one flow relation generated as a result of the pseudo code shown in Figure 2.11. The pseudo code reveals a call from C to A but since there are no parameters passed and since C does not utilize A's output there is no flow relation from C to A. The pseudo code also shows C calling B and passing the returned value from A. Since C does not modify x, it is not an effective parameter and there is no flow
Figure 2.9. A calling structure for three modules

Figure 2.10. Possible flow structures corresponding to Figure 2.9
from C to B. This example does however illustrate condition 3 of definition 2.2 and contains an indirect local flow from A to B.

C: 

A → B

x = A()
B(x)

Figure 2.11. Example 1 of the flow structure

A second example of a flow structure from Figure 2.10 is given in Figure 2.12. The pseudo code shows a call from C to A and again since A requires no parameters and since C does not utilize A's return value, there is no flow between C and A. Since C updates x before passing it to B, x is now an effective parameter creating a direct local flow from C to B. As in example 1, the indirect local flow between A and B still exists.

Figure 2.12. Example 2 of the flow structure
Figure 2.13 shows a direct local flow from C to A and from C to B since both calls require that an effective parameter be passed. There is an indirect local flow from A to B because the output value from A contributes to B's parameter. Applying condition 2 of definition 2.2 gives an indirect local flow from B to C since C utilizes B's returned value to update the data structure, DS.

\[
\begin{align*}
  C(y) \\
  A(y) \\
  x &= A(y) \\
  x &= x + y \\
  z &= B(x) \\
  DS &= z \\
  \ldots
\end{align*}
\]

Figure 2.13. Example 3 of the flow structure

The comparison of calls and information flow showed that information flow reveals many more implicit connections than calls. A further discussion of the comparison of calls and information flow involves measures found in the UNIX operating system. Figure 2.14 displays the number of exact calls in UNIX, the number of detected calls, local flows, and global flows found by analysis of information flow. There were 58 calls that were missed by the information flow analysis. Of these missed calls, 37 calls were missed because there was not an effective parameter and 21 calls were missed because an output value was not utilized.
Clearly, information flow reveals many more connections between the components of a system than a calling relation. It was previously shown that the dependency relation does not observe the indirect local flows. The following comparison shows all the connections possible in a system and which of these connections are observed by calls, dependency, and information flow. The term connection must be defined in order to understand this comparison.

Definition 2.7: There is a connection between module A and module B if an action in A effects the behavior of B.

When this definition of connection is applied to computer software, there are only two areas of concern; the area of flow of control and the area of information transfer. Flow of control implies 1) creation and deletion of processes and 2) the call-return mechanism. Creation and deletion of processes are dynamic and will therefore not be considered in the discussion since the connections of concern are time invariant. The call-return mechanism is actually a two-way flow of control, however, it is more conventional to assume a call-return as a single entity with an assumed return function.

Information transfer encompasses three classes: 1) effective
parameter passing, 2) information storage and retrieval through data structures, and 3) composition of functions. Synchronization is also a part of information transfer; it is assumed that synchronization is not embedded in the hardware, but contains primitives in the low level software. With this assumption, synchronization can be viewed as a data structure transfer. Composition of functions with respect to information transfer is examined in light of an example. Figure 2.15 shows a composition of B and C. C is evaluated and the result of C is passed to B as a parameter. Even though the call to B is made from A there is a transfer of information from C to B.

A:

\[
x = B(C(y))
\]

Figure 2.15. Composition of functions for information transfer

Figure 2.16 shows the subset relationships among the connections observed by calls, dependency, and information flow. The flow of control connections are exhibited by exact calls while dependency observes all of the flow of control connections, information flow observes only those calls that contain a direct local flow. The class of direct local flows is equivalent to all information transfers involving effective parameters. The global flows are the information transfers through data structures and are observed by both information flow and dependency. Composition
of functions is encompassed by the indirect local flows. The indirect local flow is observed by information flow but not dependency. Figure 2.16 shows which of the possible connections in a system are observed by information flow and dependency.

![Diagram of connections]

Figure 2.16. Relationship among subsets of the connections within a system

Ignoring the global flows since both dependency and information flow observe these connections, the comparison of dependency and information flow consists of comparing the missed calls and the indirect local flows. There were 58 missed calls for the UNIX system, and 118 indirect local flows. This observation shows that information flow reveals more connections than dependency. It is not obvious how the dependency relation would have to be revised to observe the indirect local flows. However, if the program counter were assumed to be information passed as an implicit parameter, the information flow relation would observe all of the connections within a system.

Several ordering relations have been defined as possible bases for
for software metrics. The characteristics of these metrics require a precise definition and the ability to reveal all of the connections between the components of a system. Information flow and dependency show more precise and detailed connections between system components than a calling relation. Furthermore, information flow observes the implicit connections which are missed by the dependency relation.
CHAPTER III. MECHANISMS FOR INFORMATION FLOW ANALYSIS

An information flow analysis of a system takes place in three phases. The first phase involves generating a set of relations indicating the flow of information through input parameters, output parameters, returned values from functions, and data structures. The second phase generates an information flow structure from these relations. The third phase analyzes the information flow structure to determine the derived calls, the local flows, and the global flows. The mechanisms necessary for each phase are presented in this chapter. At the end of the chapter is a discussion of memoryless procedures and the motivation for excluding them from the information flow analysis.

Generation of Relations

The first phase of an information flow analysis is the generation of relations for the system. This section is divided into three subsections presenting the general format of the relations, a detailed example for generating relations, and the specific relations generated for selected UNIX routines.

General format of a relation

The general format of a relation is:

\[ L \leftarrow R_1, R_2, \ldots, R_{\text{count}}; \]

where \( L \) may be in one of the following forms:

1. \( P.DS \) where \( P \) is a procedure name and \( DS \) is the name of a data structure.
2. \texttt{P.0} where \texttt{P} is a procedure name and \texttt{0} is the symbol used to indicate a return value from \texttt{P}.

3. \texttt{P.j.O} where \texttt{P} is a procedure name, \texttt{j} is an integer indicating a parameter position in \texttt{P}, and \texttt{O} is the symbol representing an output parameter.

4. \texttt{P.j.I} where \texttt{P} is a procedure name, \texttt{j} is an integer indicating a parameter position in \texttt{P}, and \texttt{I} is the symbol representing an input parameter.

\texttt{R_i} may be in one of the following forms:

1. \texttt{S.DS} where \texttt{S} is a procedure name and \texttt{DS} is the name of a data structure.

2. \texttt{S.0} where \texttt{S} is a procedure name and \texttt{0} is used to indicate a return value from \texttt{S}.

3. \texttt{S.j.O} where \texttt{S} is a procedure name, \texttt{j} is an integer indicating a parameter position in \texttt{S}, and \texttt{O} is the symbol representing an output parameter.

4. \texttt{S.j.I} where \texttt{S} is a procedure name, \texttt{j} is an integer indicating a parameter position in \texttt{S}, and \texttt{I} is the symbol representing an input parameter.

5. \texttt{S.null} where \texttt{S} is a procedure name and \texttt{null} represents no flow of information from \texttt{S}.

6. \texttt{S.constant} where \texttt{S} is a procedure name and \texttt{constant} represents a value used in \texttt{S}.

7. \texttt{S.error} where \texttt{S} is a procedure name and \texttt{error} represents an invalid flow of information through this path.
The various forms of $L$ and $R_i$ shown above must be used in accordance with the following rules:

1. $L$ is of the form $P.DS$
   This is to be used only when generating the relations from procedure $P$ to indicate that $P$ updates $DS$ with $R_i$.

2. $L$ is of the form $P.O$
   This to be used only when generating the relations from procedure $P$ to indicate that $P$ is a function and that the value $P$ returns is based on information from $R_i$.

3. $L$ is of the form $P.j.O$
   This is to be used when generating the relations for procedure $P$ to indicate that $P$ updates its $j$th parameter with information from $R_i$. There must be a unique relation for each of $P$'s parameters.

4. $L$ is of the form $P.j.I$
   This is to be used when generating the relations for procedure $T$ only if $T$ calls $P$ to indicate that the $j$th input parameter to $P$ receives information from $R_i$. Procedure $T$ must generate a unique relation for each parameter passed to $P$.

5. $R_i$ is of the form $S.DS$
   This is to be used only when generating the relations from procedure $S$ to indicate that $S$ reads information from $DS$ and passes that information to $L$.

6. $R_i$ is of the form $S.O$
   This is to be used when generating the relations from procedure
T only if T calls S and S returns a value to T. S.0 indicates that the returned value from S flows into L.

7. Ri is of the form S.j.0

This is to be used when generating the relations for procedure T only if T calls S passing some parameters and later T uses that parameter. S.j.0 indicates that S's jth output parameter flows into L.

8. Ri is of the form S.j.I

This is to be used only when generating the relations for procedure S to indicate that S's jth input parameter passes information to L.

9. Ri is of the form S.null

This is to be used only when generating the relations for procedure S. S.null will only be used if S does not update a parameter (i.e., the parameter was an input only parameter). The relation would be S.j.0 ← S.null;

10. Ri is of the form S.constant

This is to be used only when generating the relations for procedure S to indicate that S uses a constant to flow into L.

11. Ri is of the form S.error

This is to be used when generating the relations for procedure S only if S calls T and a parameter to T is an output only parameter. If T tries to obtain information from that parameter an error would result. The relation would be T.j.I ← S.error;
Examples for relation generation

The following four examples demonstrate the technique for generating information flow relations. Each example introduces another possible type of relation. The examples are divided into a flow graph representing the flow of information and control, skeleton code which corresponds to the flow graph, and finally the relations which are generated from the code.

The examples are simplified. However, the purpose of the examples is to explain how the information flow relations are generated. The details are thus omitted since they do not contribute to an understanding of the generation of these relations. Prior to describing the examples in detail, the notation used in the flow graphs and code is given.

Notation for figures:

- represents a complex data structure
- represents a subroutine procedure
- represents a function procedure which returns a value

- represents information flow to and from data structures

- represents flow of control (calls)

- represents flow of control with some value returned

Notation for code: + is some complex operation or group of operations which results in a value
* is a complex operation like +
= is a complex process updating a data structure
The general format of a relation is:

\[ L \leftarrow R_1, R_2, \ldots, R_{\text{count}} \text{ where } \text{count} \geq 1, \text{ and} \]

\[ \leftarrow \text{ indicates the flow of information from } R_1, R_2, \ldots, R_{\text{count}} \text{ to } L \]

L is a parameter, a data structure, or a returned value.

R_i is a parameter, a data structure, constant null, or error.

A closer look at the examples will clarify how relations are generated.

The initial value of a subroutine's ith parameter is referred to as the subroutine's ith input parameter and the final value of a subroutine's ith parameter is referred to as the subroutine's ith output parameter.

The flow graph in Figure 3.1 shows that information flows from D_1 and D_2 into A, A performs some operation on this information and updates D_3. Recall that D_1, D_2, and D_3 are complex data structures and that A is a complex subroutine. The code for the flow graph indicates that A retrieves information from D_1 and D_2 via a complex operation (+) and this information together with a constant (1) are used in some complex process (=) to update D_3. The relation which is generated shows that A updates data structure D_3 (A.D_3), (i.e., information flows into D_3 via subroutine A) from data structure D_1 (A.D_1) and D_2 (A.D_2) and from a constant (A.constant).

Expanding the example, in Figure 3.2 subroutine A retrieves information from D_1 and D_2 and calls subroutine B to update D_3. The code for the flow graph illustrates that subroutine A retrieves information from D_1 and D_2 and passes that information to subroutine B via parameters x and y. Note that only x and y are passed as parameters to B and not the entire data structures D_1 and D_2. Subroutine B then updates data
Flow graph

\[ D3 = D1 + D2 + 1 \]

Code

\[
\begin{align*}
A() & \quad . \\
\quad . & \\
\quad D3 & = D1 + D2 + 1 \\
\quad . & \\
\quad . & \\
\end{align*}
\]

Relation set

Figure 3.1. Example 1 of relation generation

structure D3 using the parameter information it received from A.

The relation set in Figure 3.2 shows that relations A1 and A2 are generated from subroutine A and relations B1, B2, and B3 are generated from subroutine B. Relation A1 indicates that information flows into
Flow graph

A() B(p, q)
• 
•
•
\[ x = D1 + 1 \]
\[ y = D2 \]
B(x, y)
•
•

D3 = p + q

Code

A1 \[ B.1.I \leftarrow A.D1, A \text{ constant; } \]
A2 \[ B.2.I \leftarrow A.D2; \]
B1 \[ B.1.O \leftarrow B.\text{null}; \]
B2 \[ B.2.O \leftarrow B.\text{null}; \]
B3 \[ B.D3 \leftarrow B.1.I, B.2.I; \]

Relation set

Figure 3.2. Example 2 of relation generation
B's first input parameter, x, (B.1.I) from information retrieved from D1 by A (A.D1) and from a constant (A.constant). B's second input parameter, y, (B.2.I) receives information retrieved by A from D2 (A.D2). Relation B1 shows that B's first output parameter, p, (B.1.0) is not affected (B.null) because p never appears on the left hand side of an = sign and p is never passed as a parameter to another subroutine. The same is true of B's second output parameter, q, as illustrated by relation B2. B updates D3 (B.D3) based on B's first input parameter (B.1.I) and B's second input parameter (B.2.I).

The example has been expanded one step further in Figure 3.3 to include subroutine C which is called by B and which returns values to B via its parameters. The code for subroutine A is unaffected by this change. Subroutine B utilized local variables r and s as parameters to subroutine C. C returns a value in its output parameter, s, which B utilizes to update data structure D3. Subroutine C performs a computation which results in the value to update D3. C also alters the value of j which is really a local variable in subroutine B, namely r.

Since the code for subroutine A did not change neither do A's relations. As shown in relation B1, B's first output parameter (B.1.0) is determined by C's first output parameter (C.1.0). B's second parameter is still not affected (B.2.0<—B.null). C's first parameter (C.1.I) is the same as B's first input parameter (B.1.I). C's second input parameter (C.2.I) is, via local variable r, the same as B's second input parameter (B.2.I). Note that local variables do not participate in the information flow analysis. The third input parameter to C (relation B5) is
46

Flow graph

A() B(p,q) C(i,j,k)
- - -
- - -
- - -
x=D1+1 r=q k=i+j
y=D2 C(p,r,s) j=j+1
B(x,y) D3=s
- - -
- - -
- - -

Code

A1 B.1.I ← A.D1,A.constant;
A2 B.2.I ← A.D2;
B1 B.1.0 ← C.1.0;
B2 B.2.0 ← B.null;
B3 C.1.I ← B.1.I;
B4 C.2.I ← B.2.I;
B5 C.3.I ← B.error;
B6 B.D3 ← C.3.0;
C1 C.1.0 ← C.null;
C2 C.2.0 ← C.2.I,C.constant;
C3 C.3.0 ← C.1.I,C.2.I;

Relation set

Figure 3.3. Example 3 of relation generation
undefined at the time of the call since it is strictly an output parameter. Thus, this relation indicates that C's third input parameter (C.3.I) will be in error (B.error) if subroutine C attempts to utilize it as an input parameter. Relation B6 illustrates that data structure D3 is updated by subroutine B (B.D3) based upon C's third output parameter (C.3.O).

Relations C1, C2, and C3 are generated from subroutine C and show that the final value of C's first parameter (C.1.O) is not affected by the code in subroutine C (see relation C1). The first parameter is a pure input parameter. C's second parameter is an input/output parameter and its final value (C.2.O) is changed based on C's second input parameter (C.2.I) and a constant (C.constant). C's third parameter (C.3.O) is only an output parameter and C's first and second input parameters flow into it (C.1.I and C.2.I).

The fourth example in Figure 3.4 includes a function F which is invoked by subroutine A. The result of F is one of the parameters which is passed to subroutine B. Function F retrieves information from D2 and returns it to A. Subroutines B and G are not changed.

The code for subroutine A now reveals that A retrieves information from D1 and uses that information to pass parameter x to F. The value returned from F is then used as the second parameter to B. The code for routines B and C is not affected. Function F utilizes the information received from C1 to selectively retrieve information from D2 via operation * and returns this value.

The relation set in Figure 3.4 now includes those relations generated from function F and the new relations generated from A. The
Flow graph

A() \quad B(p,q) \quad C(i,j,k) \quad F(m)
\cdot \quad \cdot \quad \cdot \quad \cdot
\cdot \quad \cdot \quad \cdot \quad \cdot
\cdot \quad \cdot \quad \cdot \quad \cdot
x=D1+1 \quad r=q \quad k=i+j \quad n=D2*m
y=F(x) \quad C(p,r,s) \quad j=j+1 \quad return(n)
B(x,y) \quad \cdot \quad \cdot \quad \cdot
\cdot \quad \cdot \quad \cdot
\cdot \quad \cdot \quad \cdot

Code

A1 \quad F.1.I \leftarrow A.D1,A.constant;
A2 \quad B.1.I \leftarrow A.D1,A.constant,F.1.O;
A3 \quad B.2.I \leftarrow F.0;
B1 \quad B.1.0 \leftarrow C.1.0;
B2 \quad B.2.0 \leftarrow B.null;
B3 \quad C.1.I \leftarrow B.1.I;
B4 \quad C.2.I \leftarrow B.2.I;
B5 \quad C.3.I \leftarrow B.error;
B6 \quad B.D3 \leftarrow C.3.0;
C1 \quad C.1.0 \leftarrow C.null;
C2 \quad C.2.0 \leftarrow C.2.I,C.constant;
C3 \quad C.3.0 \leftarrow C.1.I,C.2.I;
F1 \quad F.1.0 \leftarrow F.null;
F2 \quad F.0 \leftarrow F.D2,F.1.I;

Relation set

Figure 3.4. Example 4 of relation generation
relations for B and C remain the same. Examining relation A1, F's first input parameter (F.I.I) is affected by information retrieved from D1 by A (A.D1) and by a constant (A.constant). Relation A2 shows that B's first input parameter (B.I.I) is also affected by information retrieved from D1 by A (A.D1), by a constant (A.constant), and by F's first output parameter (F.1.O). Relation A3 reveals a new type of relation and indicates that B's second input parameter depends upon the value returned by function F (F.0). The relations for function F reveal that the value of F's first parameter is not changed (F.null) and that the value returned by function F (F.0) depends upon the information retrieved from D2 by F (F.D2) and on the initial value of F's first parameter (F.1.I).

The relations for a given subroutine are generated by examining the code for that subroutine only. By analyzing F's relations it is revealed that F does not affect its first output parameter; therefore in relation A2, F.1.O does not affect B's first input parameter. When generating the relations for A, only the code for subroutine A is seen, so it is not known that F does not affect its first output parameter. The analysis of the information flow structure reveals this fact.

Actual relations for selected UNIX routines

The following example illustrates the actual relations which are generated from two UNIX routines, BREADA and INCORE. These routines are used to manipulate the BUF data structure. UNIX routines are written in the "C" language, but for purposes of readability these routines have been coded in a pseudo high level language (see Figure 3.5). The "C"
1. PROCEDURE breada (adev, blkno, rablkno) RETURNS (pointer);
2. CONSTANT  
   b_done = 02;
3.  b_read = 01;
4.  b_async = 0400;
5. POINTER  
   rbp, rabp; /* pointers to a buf data structure */
6. INTEGER  
   dev, x;
7. dev = adev;
8. rbp = 0;
9. x = CALL incore (dev, blkno);
10. IF (x = 0)
11.    THEN DO; rbp = CALL getblk (dev, blkno);
12.     IF (rbp.b_flags & b_done = 0)
13.       THEN DO; rbp.b_flags = rbp.b_flags | b_read;
14.       rbp.b_wcount = -256;
15.       CALL rkstrategy (rbp);
16.       END;
17.     END:
18. x = CALL incore (dev, rablkno);
19. IF (rablkno > 0 AND x = 0)
20.    THEN DO; rabp = CALL getblk (dev, rablkno);
21.     IF (rabp.b_flags & b_done = 1)
22.       THEN CALL brelse (rabp);
23.     ELSE DO; rabp.b_flags = rabp.b_flags | b_read | b_async;
24.     END:
25.       CALL rkstrategy (rabp);
26.     END;
27.     END;
28. IF (rbp = 0)
29.    THEN DO; x = CALL bread (dev, blkno);
30.   RETURN (x);
31. END;
32. CALL iowait (rbp);
33. RETURN (rbp);
34. END breada;

Figure 3.5. Pseudo code for UNIX routines
B1. breada.1.o ← breada.null
B2. breada.2.o ← incore.2.o, getblk.2.o, breada.2.o
B3. breada.3.o ← incore.2.o, getblk.2.o
B4. incore.1.i ← breada.1.i
B5. incore.2.i ← breada.2.i, breada.3.i
B6. getblk.1.i ← breada.1.i, incore.1.o, incore.0, breada.3.i
B7. getblk.2.i ← breada.2.i, breada.3.i, incore.0, incore.2.o
B8. rkstrategy.1.i ← getblk.0, incore.0, breada, buf, breada.constant
B9. breIse.1.i ← getblk.0, incore.0, breada.3.i, breada.buf, breada.constant
B10. bread.1.i ← breada.1.i, getblk.0, getblk.1.o, incore.1.o
B11. bread.2.i ← breada.2.i, getblk.0, getblk.2.o, incore.2.o
B12. iowait.1.i ← getblk.0
B13. breada.0 ← bread.0, getblk.0, incore.0
B14. breada.buf ← breada.constant, getblk.0, incore.0, breada.3.i

Relations for breada

1. PROCEDURE incore (adev, blkno) RETURNS (pointer);
2. POINTER bp; /* pointer to a buf data structure */
3. POINTER dp; /* pointer to a rktab data structure */
4. INTEGER dev;
5. dev = adev;
6. dp = rktab (dev);
7. bp = dp.bforw;
8. DO WHILE (bp ∼= dp
9.     IF (bp.b_blkno = blkno AND bp.b_dev = dev)
10.     THEN RETURN (bp);
11.     bp = bp.b_forw;
12. END;
13. RETURN (0);
14. END incore;

I1. incore.1.o ← incore.null;
I2. incore.2.o ← incore.null;
I3. incore.o ← incore.buf, incore.rktab, incore.1.i, incore.2.i, incore.constant;

Relations for incore

Figure 3.5 (Continued)
language UNIX routines are presented in Appendix A.

Figure 3.5 contains the code for BREADA and the relations that are generated from the code. Relation B1 states that BREADA does not update its first parameter. BREAD's second parameter (BREADA.2.0) is updated based on INCORE's second output parameter, GETBLK's second output parameter, and on BREAD's second output parameter. In statement 9 BREADA's second parameter, BLKNO, is passed to INCORE and it is possible for INCORE to update its second parameter. Therefore, information may flow from INCORE's second output parameter (INCORE.2.0) to BREADA's second output parameter (BREADA.2.0). The same flow is possible from GETBLK (statement 11) and BREAD (statement 29). Relation B3 is similar to B2 in that INCORE's second output parameter may flow into BREADA's third output parameter (statement 18) and GETBLK's second output parameter may flow into BREADA's third output parameter (statement 20).

INCORE is called from BREADA in lines 9 and 18. Since INCORE has two parameters, two relations must be generated, namely B4 and B5. Even though the parameters to INCORE are different only one set of relations needs to be generated for the calls to INCORE. Relation B4 shows that information flows from BREADA's first input parameter to INCORE's first input parameter. INCORE's first parameter is the same in both calls, therefore there is only one element on the right side of relation B4. Note that the local variable, DEV, is disregarded in generating the relations. INCORE's second input parameter (INCORE.2.1) receives information from BREADA's second input parameter (line 9) and from BREADA's third input parameter (line 18). The two calls are combined in one
relation.

Like INCORE, GETBLK is also called from two places in BREADA (lines 11 and 20). The calls are combined to form one set of relations for GETBLK (relations B6 and B7). GETBLK's first input parameter (GETBLK.1.I) receives information directly from BREADA's first input parameter (BREADA.1.I) and may receive information if INCORE updates its first output parameter (INCORE.1.0). Since the call to GETBLK in line 11 will only be performed if the output from INCORE is zero (lines 9 -10), the parameters to GETBLK are dependent on the output from INCORE (lines 18 - 19) and on RABLKN0 which is BREADA's third input parameter. Therefore BREADA.3.1 also flows into GETBLK.1.I. GETBLK's second input parameter receives information directly from BREADA's second input parameter (line 11) and from BREADA's third input parameter (line 20). INCORE's second output parameter may flow into GETBLK's second input parameter, lines 9 and 20, therefore INCORE.2.0 flows into GETBLK.2.I. As above from lines 10 and 19 INCORE.0 and BREADA.3.I also flow into GETBLK's second input parameter.

Relation B8 states that RKSTRATEGY's first input parameter receives information from the output from GETBLK, GETBLK.0 (line 11). RKSTRATEGY.1.I also depends on INCORE.0 (lines 9 - 10) and on the BUF structure (BREADA.BUF) and a constant, BREADA.CONSTANT, from line 12. Note that the variables used in the conditionals flow into the parameters of a subroutine call.

The first input parameter to BRELSE, line 22 (relation B9) receives information from GETBLK.0 (line 20), INCORE.0 (lines 18 - 19), BREADA.3.I (line 19), BREADA.BUF and BREADA.CONSTANT (line 21).
The call to \texttt{BREAD} in line 29 generates relations B10 and B11. The first input parameter to \texttt{BREAD} is dependent on \texttt{BREADA}'s first input parameter (\texttt{BREADA.1.1}). \texttt{GETBLK} and \texttt{INCORE} may modify their first output parameter (lines 9, 11, 18, 20) therefore \texttt{GETBLK.1.0} and \texttt{INCORE.1.0} flow into \texttt{BREAD.1.I}. The output from \texttt{GETBLK} (\texttt{GETBLK.0}) in line 11 also flows into \texttt{BREAD.1.I}. The right side of relation B11 for \texttt{BREAD}'s second input parameter follows directly from the above discussion.

Relation B12 demonstrates that the first input parameter to \texttt{IOWAIT} comes from the output from \texttt{GETBLK}. Clearly line 11 shows this flow of information. A test is made on the value of \texttt{RBP} in line 28, if this test is true a return from \texttt{BREADA} is executed (line 30) and \texttt{IOWAIT} will not be called. This is therefore a second flow of information from \texttt{GETBLK.0}.

There are two return statements from \texttt{BREADA}, lines 30, 33, and again they can be combined in one relation, namely relation B13. By examining the return statements, it is found that the output from \texttt{BREAD} in line 29 (\texttt{BREAD.0}), or the output from \texttt{GETBLK} (line 11) may be returned. The output from \texttt{IOWAIT}'s first output parameter (line 32) also may flow into \texttt{BREADA.0}.

\texttt{BREADA} also updates the \texttt{BUF} data structure (relation B14). The information needed to update a data structure is a pointer to the data structure, the values on the right of the assignment statement and any conditional values. The \texttt{BUF} structure is updated by \texttt{BREADA} in lines 13, 14, 23 and 24. Constants are always used on the right side of the assignment statements. The output from \texttt{GETBLK}, \texttt{RBP}, and \texttt{RABP}, (lines 11, 20) are the pointers used to update \texttt{BUF}, therefore \texttt{GETBLK.0} flows into \texttt{BUF}.  

INCORE.0 and BREADA.3.I are needed for the conditionals (lines 10, 19) necessary to update BUF.

Three relations are generated for INCORE. The first two relations (II and I2) state that INCORE does not modify its output parameters.

There are two return statements from INCORE (lines 10, 13). The information flowing into the output from INCORE (relation I3) comes from a CONSTANT, line 13, from the RKTAB data structure, line 6, from INCORE.1.I, the subscript to RKTAB in line 6, from INCORE.2.I and the BUF data structure used in the conditional in line 9.

Algorithm for Generation of the Information Flow Structure

The entire information flow structure is generated from the relations by the recursive algorithm in Figure 3.6. The structure is then analyzed by another algorithm to determine derived calls, local flows, and global flows.

Recall that the general format of a relation is:

\[ L \leftarrow R_1, R_2, \ldots, R_{\text{count}}; \quad \text{where count} \geq 1. \]

\( L \) consists of a procedure name and a data structure name, an input parameter, an output parameter, or a return value from a function.

\( R_i \) consists of a procedure name and a data structure name, constant, null, error, and input parameter, an output parameter, or a return value from a function.

The relations are sorted alphabetically on \( L \) and are stored internally in an array, RELATION. Associated with each relation is \text{COUNT}, which denotes the number of elements on the right of each relation.
INFOFLOW (.ds,top)
  FOR all relations i
    IF (L of relation (i) = .ds)
      THEN attach (L of relation(i),desc(top))
           expand (relation(i),desc(top))
  END

EXPAND (relation(i),p)
  FOR all j = 1,...,relation(i).count
    attach(r(j) of relation (i) to I.F.S. at p)
    IF ((r(j) is an input parameter OR an output parameter, OR a return value) AND r(j) is not already on this path in the I.F.S.)
      THEN FOR all relations k
          IF (L of relation (k) = r(j))
            THEN expand (relation(k), desc(p))
  END

Figure 3.6. Algorithm for generation of the information flow structure

The two routines INFOFLOW and EXPAND, in Figure 3.6, are used to generate the information flow structure. INFOFLOW (.DS,TOP) is the initial call where .DS is the name of the data structure for which the information flow structure will be generated. TOP is a pointer to the top of the information flow structure.

INFOFLOW will search all of the relations and will call EXPAND for each relation where L is of the form _.DS. EXPAND accepts as input a relation and a pointer, P, into the information flow structure. Each element on the right of the relation, Rj, will be recursively expanded and attached at the descendant of F in the information flow structure (I.F.S.).

The information flow relations are used to generate an information flow structure. This structure shows the various paths through which
information flows into a given data structure. Figure 3.7 represents the information flow structure generated by analyzing the relation set in Figure 3.4.

By examining the relation set it is evident that only one routine, namely B, updates D3 since there is only one relation where the left side has the form \texttt{__D3} (relation B6). Information flows into D3 from B based on C's third output parameter (C.3.0). Relation C3 indicates that C's third output parameter receives information from C's first input parameter (C.1.I) and C's second input parameter (C.2.I). By applying this procedure to C.1.I and C.2.I, relations B3 and B4 reveal that information flows into C.1.I and C.2.I from B's first input parameter (B.1.I) and B's second input parameter (B.2.I) respectively. Information flows into B.1.I (relation A2) from information A retrieved from D1 (A.D1), from a constant in A (A.constant) and from F's first output parameter (F.1.0). Relation A3 indicates that information flows from F's returned value (F.0) into B.2.I. This procedure halts when a path terminates with a data structure, a constant, or a null, therefore the paths ending in A.D1 and A.constant are complete. No information flows into F's first output parameter as specified by relation F1, hence the path terminates with F.null. Information flows into F's output (F.0) from information F retrieves from D2 (F.D2) and from F.1.I (see relation F2). Again, since D2 is a data structure another path, namely the one ending in F.D2, in the information flow structure is complete. Relation A1 indicates that information flows into F.1.I from information A retrieved from D1 (A.D1) and from a constant (A.constant). The information flow structure
Figure 3.7. An information flow structure for Figure 3.4
now complete since all the paths terminate with a data structure, a constant or a null.

Algorithms for Information Flow Structures

Analysis of the information flow structure will reveal the derived calling structures between procedures, the local flow of information between procedures and a global flow of information for a given data structure.

An algorithm is applied to each path in the information flow structure to determine the calls and local flows. The global flow is generated by analyzing the entire information structure.

Analysis of each path in the information flow structure will determine the calls and local flow of information. Each element in the structure will exist in one of the following forms:

- $X.DS$ (null, constant, error and all data structure names are in this category)
- $X.0$ (returned value from a function)
- $X.k.I$ (X's kth input parameter, where $k \geq 1$)
- $X.k.O$ (X's kth output parameter, where $k \geq 1$)

Each consecutive pair of elements on each path in the information flow structure is inspected to determine the calls and local flows. The pair $(X._, Y._)$ indicates that $X._$ is immediately above $Y._$ on some path in the structure. If $(X._, X._)$ is a consecutive pair in the structure there does not exist any calls or local flows for that pair and is therefore ignored.
Derived calls

The pairs in Figure 3.8 labeled "not possible" (entries 1, 3, 5, 7, 13, 15) will never appear consecutively on a path in the information flow structure. When coding the relations for a procedure, X, only the code for X is inspected, therefore it is not possible for either an input parameter from Y or a data structure from Y to flow into a data structure in X, or a returned value from X, or an output parameter from X.

Entries 2 and 4 indicate that X calls Y and Y returns a value either through a function or an output parameter. X then utilizes the information returned from Y to update a data structure.

Entries 6, 8, 14 and 16 specify that X calls Y. In order for these pairs to appear on a path in the information flow structure, X calls Y and again Y returns a value either as a function or through an output parameter. X utilizes the information from Y for its returned value (entries 6, 8) or its output parameter (entries 14, 16). Y calls X (entries 9, 11) when Y passes information to X through an effective parameter (X.k.I) and that information was retrieved from a data structure by Y or from one of Y's input parameters.

In some cases a third procedure, Z, calls Y and Y returns a value either by a function call or an output parameter. Z then calls X passing the information that Z received from Y. In this circumstance there is no call between X and Y (entries 10, 12).
Figure 3.8. Analysis of calls

<table>
<thead>
<tr>
<th>Y.DS</th>
<th>X.DS</th>
<th>X.O</th>
<th>X.k.I</th>
<th>X.k.O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>not possible</td>
<td>not possible</td>
<td>Y calls X</td>
<td>not possible</td>
</tr>
<tr>
<td>Y.O</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>X calls Y</td>
<td>X calls Y</td>
<td>no calls</td>
<td>X calls Y</td>
</tr>
<tr>
<td>Y.k.I</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>not possible</td>
<td>not possible</td>
<td>Y calls X</td>
<td>not possible</td>
</tr>
<tr>
<td>Y.k.O</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>X calls Y</td>
<td>X calls Y</td>
<td>no calls</td>
<td>X calls Y</td>
</tr>
</tbody>
</table>

Local flows

As previously stated the pairs represented by entries 1, 3, 5, 7, 13, 15 cannot appear in a path in the information flow structure.

Entries 2 and 4 indicate that X calls Y receiving information from Y and uses that information to update a data structure. Information flows from Y to X. (See Figure 3.9.)

Information flows from Y into X as revealed in entries 6, 8, 14, and 16 when X calls Y and Y returns a value either by a function or through an output parameter. X then utilizes this value in its returned value (entries 6, 8) or to update an output parameter (entries 14, 16).

Y calls X passing information to an input parameter of X from a data structure or from one of Y's input parameters as shown in entries 9 and 11. Therefore information flows from Y into X.

Entries 10 and 12 indicate that a third procedure, Z, calls Y and passes the returned value from Y to X. Even though there is no call
### Figure 3.9. Analysis of local flows

<table>
<thead>
<tr>
<th></th>
<th>X.DS</th>
<th>X.O</th>
<th>X.k.I</th>
<th>X.k.O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y.DS</td>
<td>not possible</td>
<td>not possible</td>
<td>Y flows X</td>
<td>not possible</td>
</tr>
<tr>
<td>Y.O</td>
<td>Y flows X</td>
<td>Y flows X</td>
<td>Y flows X</td>
<td>Y flows X</td>
</tr>
<tr>
<td>Y.k.I</td>
<td>not possible</td>
<td>not possible</td>
<td>Y flows X</td>
<td>not possible</td>
</tr>
<tr>
<td>Y.k.O</td>
<td>Y flows X</td>
<td>Y flows X</td>
<td>Y flows X</td>
<td>Y flows X</td>
</tr>
</tbody>
</table>

between X and Y, there is an indirect local flow of information from Y to X through Z.

**Global flows**

For a given data structure, DS, all procedures subordinate to the initial level (top) of the information flow structure deposit information into the data structure. Inspection of the lowest level (bottom) of the information flow structure will reveal all procedures of the form A.DS, i.e., those procedures which retrieve information from data structure, DS, and through some path cause another procedure to update that data structure. The procedures at the top of the information flow structure and those procedures of the form A.DS at the bottom of the information flow structure constitute the global flow for DS. Information flows from all of the procedures at the top into all procedures (A.DS) at the bottom of the information flow structure.
Memoryless Procedures

Some of the procedures in the UNIX operating system were not considered in the information flow analysis because they are memoryless procedures. A memoryless procedure is one "which is guaranteed to have kept no record of data supplied after it has completed its task" (Fenton, 1974). Memoryless procedures are incapable of communicating information about data and hence do not serve a purpose on an information flow path. Successive invocations of these procedures are unaffected by prior calls. If the memoryless procedures were considered in the information flow analysis, connections between procedures would be generated that do not functionally exist.

Sixteen of the 223 UNIX routines were found to be memoryless. Seven of the procedures were eliminated because they performed functions that are present in the hardware of some machines, for example, double precision operations, shifts, and procedures used to return the maximum or minimum of two parameters. The other set of memoryless procedures consisted of those procedures which dealt with physical operations concerning memory and memory management. These routines deal with clearing or copying a segment of memory, and the concept of physical memory is not included in the information flow analysis.
CHAPTER IV. MEASUREMENTS OF THE UNIX OPERATING SYSTEM

An information flow analysis was performed on the UNIX operating system and the global flows and local flows derived from the information flow analysis were used to obtain measurements for the UNIX system. Recall that the motivation for using UNIX was presented in Chapter I. These measurements, including the complexity measurements for each procedure, are presented in this chapter. The local flows and global flows are used to define the modules in the system, and to reveal weaknesses in these modules. Also, the interfaces between the modules are examined placing emphasis on the protocol interface, the binding, and the coupling between the modules. Finally, a graph is presented indicating the levels in the UNIX operating system and their accumulated complexities. Prior to revealing these measurements, it is necessary to define several terms used in the explanation of the measurements. These terms are presented in the next section.

It is important to mention that the information flow analysis may be performed immediately after the design phase when the external specifications have been completed but before the implementation has begun. The measurements taken at this point must rely on estimates of the code length in order to compute the complexity measures. Based on these estimates the design can be evaluated for possible flaws before the investment in implementation has begun. This permits a design-measure-redesign cycle which is considerably shorter and less expensive than the more common design-implement-test-redesign cycle. The measurements may also be taken after the implementation phase using the exact code length for each
procedure. The measurements taken at either point will show possible areas where redesign or reimplementation is needed, and where maintenance of the system might be difficult. These measurements allow the design and implementation decisions to be evaluated for potential reliability problems.

The metrics presented below define a "complexity value" which attempts to measure the inherent complexity (difficulty of understanding) of a given problem solution expressed as a collection of software. It should be kept in mind that the complexity of a given problem solution is not necessarily the same as the unmeasurable "complexity" of the problem being solved. In the following text the terms "complexity value", "complexity of the problem solution", and "complexity" are all used interchangeably.

Measurement Definitions

The terms fan-in, fan-out, complexity, and module are specifically defined for information flow in order to present the measurements in this chapter. Fan-in and fan-out are described with respect to individual procedures.

Definition 4.1: The fan-in of procedure A is the number of local flows into procedure A.

Definition 4.2: The fan-out of procedure A is the number of local flows from procedure A.
The complexity of a procedure encompasses two factors: the complexity of the procedure code and the complexity of the procedure's connections to its environment. The code complexity used in these measurements is defined as the number of lines of code, or length. Bowen (1978) points out that, according to the literature, the number of statements in a routine is the only single factor that has significant positive correlation with the number of errors. The connections of a procedure to its environment are determined by the fan-in and the fan-out. The formula used for the complexity of a procedure is:

\[ \text{length} \times (\text{fan-in} \times \text{fan-out})^2 \]

The term fan-in * fan-out represents the total possible number of combinations of an input source to an output source. This term is squared to reflect the author's experience that the difficulty in understanding operating system components stemmed, not from the action of the component itself, but from the interrelationships of this component with others through their common environment. The interrelationships are very complicated, particularly for operating systems, but this may not be true to the same degree for software in general. For example, a compiler may have straightforward connections and very complex code. In this case, a different formula for complexity should be applied. The validity of the approach used in this thesis for operating systems, emphasizing the connections to the environment, is supported by the material in this chapter and the next showing a correlation between this measurement and reliability characteristics.
Measurement of UNIX Procedures

The UNIX procedures and their associated complexities are listed in Figure 4.1. Procedures written in assembly language and memoryless procedures were eliminated from the information flow analysis. The total local flows from all of the data structures were used to compute the complexities, and as a result the procedure complexities displayed later in this chapter will differ slightly from these numbers. When computing other complexities only the local flows for a given data structure are used.

The procedure complexities shown in Figure 4.1 represent a broad spectrum of complexities. Readers unfamiliar with the UNIX system should find this wide range believable - UNIX contains some very simple and easy to understand procedures and some extremely difficult to understand procedures. Readers familiar with UNIX should find that specific procedures complexities reflect their experience in attempting to understand or modify particular procedures. A distribution of the order of complexity of UNIX procedures is given in Figure 4.2.

Note that there is only one procedure with complexity $10^7$, NAMEI. This procedure was designed very early in the development of UNIX, was thoroughly debugged, and remained unchanged. NAMEI is a very large procedure and its connections to the rest of the system are very complex (Lions, 1977). Both the design and implementation of NAMEI are examined in Chapter V.

The purpose of measuring systems is to isolate design and implementation problems. The procedure complexities reveal three potential
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Complexity value</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>84035</td>
</tr>
<tr>
<td>alloc</td>
<td>186624</td>
</tr>
<tr>
<td>badblock</td>
<td>1215</td>
</tr>
<tr>
<td>bawrite</td>
<td>21608</td>
</tr>
<tr>
<td>bdwrite</td>
<td>3375</td>
</tr>
<tr>
<td>bflush</td>
<td>288</td>
</tr>
<tr>
<td>binit</td>
<td>480</td>
</tr>
<tr>
<td>bmap</td>
<td>167580</td>
</tr>
<tr>
<td>bread</td>
<td>1170000</td>
</tr>
<tr>
<td>breada</td>
<td>42768</td>
</tr>
<tr>
<td>brelse</td>
<td>48400</td>
</tr>
<tr>
<td>bwrite</td>
<td>14400</td>
</tr>
<tr>
<td>canon</td>
<td>4050</td>
</tr>
<tr>
<td>chdir</td>
<td>756</td>
</tr>
<tr>
<td>chmod</td>
<td>52</td>
</tr>
<tr>
<td>chown</td>
<td>44</td>
</tr>
<tr>
<td>cinit</td>
<td>135</td>
</tr>
<tr>
<td>clock</td>
<td>78003</td>
</tr>
<tr>
<td>close</td>
<td>10</td>
</tr>
<tr>
<td>closef</td>
<td>1088</td>
</tr>
<tr>
<td>closei</td>
<td>84</td>
</tr>
<tr>
<td>clrbuf</td>
<td>13</td>
</tr>
<tr>
<td>core</td>
<td>57600</td>
</tr>
<tr>
<td>cpass</td>
<td>76</td>
</tr>
<tr>
<td>creat</td>
<td>144</td>
</tr>
<tr>
<td>devstart</td>
<td>550</td>
</tr>
<tr>
<td>dup</td>
<td>432</td>
</tr>
<tr>
<td>estabur</td>
<td>250880</td>
</tr>
<tr>
<td>exec</td>
<td>544500</td>
</tr>
<tr>
<td>exit</td>
<td>52675</td>
</tr>
<tr>
<td>expand</td>
<td>31581</td>
</tr>
<tr>
<td>falloc</td>
<td>720</td>
</tr>
<tr>
<td>fluhshtty</td>
<td>576</td>
</tr>
<tr>
<td>fork</td>
<td>38</td>
</tr>
<tr>
<td>free</td>
<td>11200</td>
</tr>
<tr>
<td>ftent</td>
<td>144</td>
</tr>
<tr>
<td>getblk</td>
<td>5175296</td>
</tr>
<tr>
<td>getc</td>
<td>20160</td>
</tr>
<tr>
<td>geterror</td>
<td>360</td>
</tr>
<tr>
<td>getf</td>
<td>72576</td>
</tr>
<tr>
<td>getfs</td>
<td>536256</td>
</tr>
<tr>
<td>getmdev</td>
<td>784</td>
</tr>
<tr>
<td>getpid</td>
<td>4</td>
</tr>
<tr>
<td>getswit</td>
<td>4</td>
</tr>
<tr>
<td>getuid</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 4.1. Procedure complexities
<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>grow</td>
<td>1127</td>
</tr>
<tr>
<td>gtime</td>
<td>5</td>
</tr>
<tr>
<td>ialloc</td>
<td>83655</td>
</tr>
<tr>
<td>ifree</td>
<td>7500</td>
</tr>
<tr>
<td>iget</td>
<td>166212</td>
</tr>
<tr>
<td>init</td>
<td>1900</td>
</tr>
<tr>
<td>incore</td>
<td>5600</td>
</tr>
<tr>
<td>iodone</td>
<td>960</td>
</tr>
<tr>
<td>iomove</td>
<td>3300</td>
</tr>
<tr>
<td>iowait</td>
<td>4800</td>
</tr>
<tr>
<td>iput</td>
<td>1199900</td>
</tr>
<tr>
<td>issig</td>
<td>72</td>
</tr>
<tr>
<td>itrunc</td>
<td>224000</td>
</tr>
<tr>
<td>iupdat</td>
<td>118784</td>
</tr>
<tr>
<td>kill</td>
<td>100</td>
</tr>
<tr>
<td>k1close</td>
<td>20</td>
</tr>
<tr>
<td>k1open</td>
<td>480</td>
</tr>
<tr>
<td>k1read</td>
<td>12</td>
</tr>
<tr>
<td>k1rint</td>
<td>160</td>
</tr>
<tr>
<td>k1sgetty</td>
<td>6</td>
</tr>
<tr>
<td>k1write</td>
<td>12</td>
</tr>
<tr>
<td>k1xint</td>
<td>28</td>
</tr>
<tr>
<td>link</td>
<td>152</td>
</tr>
<tr>
<td>lpcannon</td>
<td>765</td>
</tr>
<tr>
<td>lpclose</td>
<td>5</td>
</tr>
<tr>
<td>lpint</td>
<td>7</td>
</tr>
<tr>
<td>lpopen</td>
<td>99</td>
</tr>
<tr>
<td>lpoutput</td>
<td>7</td>
</tr>
<tr>
<td>lpoutput</td>
<td>28</td>
</tr>
<tr>
<td>lpwrite</td>
<td>7</td>
</tr>
<tr>
<td>main</td>
<td>3828</td>
</tr>
<tr>
<td>maknode</td>
<td>18375</td>
</tr>
<tr>
<td>malloc</td>
<td>50421</td>
</tr>
<tr>
<td>mapalloc</td>
<td>88</td>
</tr>
<tr>
<td>mapfree</td>
<td>36</td>
</tr>
<tr>
<td>mfree</td>
<td>2592</td>
</tr>
<tr>
<td>mknod</td>
<td>1408</td>
</tr>
<tr>
<td>mmread</td>
<td>216</td>
</tr>
<tr>
<td>mmwrite</td>
<td>279</td>
</tr>
<tr>
<td>nmamei</td>
<td>27342000</td>
</tr>
<tr>
<td>newproc</td>
<td>9500</td>
</tr>
<tr>
<td>nice</td>
<td>11</td>
</tr>
<tr>
<td>notavail</td>
<td>135</td>
</tr>
<tr>
<td>open</td>
<td>99</td>
</tr>
<tr>
<td>openl</td>
<td>45325</td>
</tr>
<tr>
<td>openi</td>
<td>972</td>
</tr>
<tr>
<td>owner</td>
<td>4536</td>
</tr>
<tr>
<td>passc</td>
<td>176</td>
</tr>
</tbody>
</table>

Figure 4.1 (Continued)
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>physio</td>
<td>28000</td>
</tr>
<tr>
<td>pipe</td>
<td>1116</td>
</tr>
<tr>
<td>prdev</td>
<td>108</td>
</tr>
<tr>
<td>prele</td>
<td>22275</td>
</tr>
<tr>
<td>procxmt</td>
<td>3096</td>
</tr>
<tr>
<td>profil</td>
<td>9</td>
</tr>
<tr>
<td>psig</td>
<td>156</td>
</tr>
<tr>
<td>psignal</td>
<td>375</td>
</tr>
<tr>
<td>ptrace</td>
<td>2754</td>
</tr>
<tr>
<td>putc</td>
<td>39</td>
</tr>
<tr>
<td>rdwr</td>
<td>18125</td>
</tr>
<tr>
<td>read</td>
<td>4</td>
</tr>
<tr>
<td>readi</td>
<td>99072</td>
</tr>
<tr>
<td>readp</td>
<td>26250</td>
</tr>
<tr>
<td>rexit</td>
<td>5</td>
</tr>
<tr>
<td>rkaddr</td>
<td>18</td>
</tr>
<tr>
<td>rkintr</td>
<td>207</td>
</tr>
<tr>
<td>rkread</td>
<td>4</td>
</tr>
<tr>
<td>rkstart</td>
<td>40</td>
</tr>
<tr>
<td>rkstrategy</td>
<td>464</td>
</tr>
<tr>
<td>rkwrite</td>
<td>4</td>
</tr>
<tr>
<td>savu</td>
<td>28</td>
</tr>
<tr>
<td>sbreak</td>
<td>800</td>
</tr>
<tr>
<td>sched</td>
<td>102600</td>
</tr>
<tr>
<td>seek</td>
<td>172</td>
</tr>
<tr>
<td>setgid</td>
<td>9</td>
</tr>
<tr>
<td>setpri</td>
<td>192</td>
</tr>
<tr>
<td>setrun</td>
<td>6000</td>
</tr>
<tr>
<td>satuid</td>
<td>11</td>
</tr>
<tr>
<td>sgtty</td>
<td>261</td>
</tr>
<tr>
<td>signal</td>
<td>116</td>
</tr>
<tr>
<td>sleep</td>
<td>262440</td>
</tr>
<tr>
<td>smdate</td>
<td>72</td>
</tr>
<tr>
<td>smount</td>
<td>41552</td>
</tr>
<tr>
<td>ssig</td>
<td>14</td>
</tr>
<tr>
<td>sslep</td>
<td>80</td>
</tr>
<tr>
<td>stat</td>
<td>22</td>
</tr>
<tr>
<td>statl</td>
<td>6480</td>
</tr>
<tr>
<td>stime</td>
<td>72</td>
</tr>
<tr>
<td>stop</td>
<td>153</td>
</tr>
<tr>
<td>stty</td>
<td>10</td>
</tr>
<tr>
<td>sumount</td>
<td>1920</td>
</tr>
<tr>
<td>sureg</td>
<td>416</td>
</tr>
<tr>
<td>suser</td>
<td>488</td>
</tr>
<tr>
<td>swap</td>
<td>2600</td>
</tr>
<tr>
<td>swtch</td>
<td>71</td>
</tr>
<tr>
<td>timeout</td>
<td>120</td>
</tr>
<tr>
<td>times</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 4.1 (Continued)
trap 4752  
trapl 10  
tthread 44  
ttrstrt 32  
ttstart 7776  
ttwrite 720  
ttyinput 2112  
ttyoutput 106  
ttytty 80  
uchar 9  
ufalloc 44  
unlink 3744  
update 8192  
wait 31250  
wakeup 50400  
wdir 540  
wflushtty 432  
write 4  
writei 188416  
writep 46800  
xalloc 235008  
xcdec 360  
xfree 4275  
xswap 48600 

Figure 4.1 (Continued)

Order of Number of UNIX complexity procedures

| 0 | 17 |
| 10 | 38 |
| 2 | 41 |
| 3 | 27 |
| 4 | 26 |
| 5 | 12 |
| 6 | 3 |
| 7 | 1 |
problem areas in a given procedure. First, the measurements show procedures which possibly lack functionality. A high fan-in and fan-out reveals a large number of connections of that procedure to its environment indicating the procedure may perform more than one function. Second, a high complexity shows stress points in a system, a procedure with high information traffic through it. At such a stress point it is difficult to implement a change to the specific procedure because of the large number of potential effects on its environment and, indirectly, on other procedures. The third area indicated by these measurements is that of inadequate refinement. The inadequate refinement could be caused by either a problem in implementation or design. An implementation difficulty would be indicated by a large procedure, i.e., many lines of code. Perhaps the procedure should be divided into two or more separate procedures. The inadequate refinement could also appear as a missing level of abstraction in the design process. This would be indicated by a large fan-in or fan-out. Both of these difficulties are further investigated in Chapter V.

Measurements of UNIX Modules

The procedure complexities are used to establish module complexities. Parnas' (1977) definition of the term module was adopted for use in this thesis.

Definition 4.3.: A module with respect to a data structure, D, consists of those procedures which either update D or retrieve information from D.

The information flow analysis was performed for each data structure
in the UNIX operating system. The modules in UNIX simply consist of those procedures which read from or write to a data structure. The module names are simply the data structure names. Only a subset of the UNIX modules is presented in this chapter, since the other modules are relatively trivial. Figure 4.3 displays the modules discussed in this chapter and their corresponding descriptions. Appendix B contains the other UNIX modules and their descriptions.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>buffer information for the block I/O system</td>
</tr>
<tr>
<td>file</td>
<td>file information for each open file</td>
</tr>
<tr>
<td>filesys</td>
<td>super block information for resource allocation</td>
</tr>
<tr>
<td>inode</td>
<td>active file information for the general disposition of the file</td>
</tr>
<tr>
<td>kill</td>
<td>character device information for each terminal</td>
</tr>
<tr>
<td>lpp1</td>
<td>device information for the line printer</td>
</tr>
<tr>
<td>mount</td>
<td>super block information for mounted files</td>
</tr>
<tr>
<td>proc</td>
<td>process information for each active process</td>
</tr>
<tr>
<td>text</td>
<td>text segment information for unaltered code and data</td>
</tr>
<tr>
<td>u</td>
<td>all process information not needed for swapping</td>
</tr>
</tbody>
</table>

Figure 4.3. UNIX module descriptions

Examination of the global flows in each module reveals the size of the module and all possible interconnections between the module procedures and the data structure. Figure 4.4 displays the number of read only procedures, write only procedures, and read_write procedures for each module. The number of global flows involves all paths of information possible among these procedures through the data structure. The formula used to calculate the number of global flows is:

\[
\text{write} \times (\text{read} + \text{read_write}) + \text{read_write} \times (\text{read} + \text{read_write} - 1).
\]
Module | Global flows | Procedures
---|---|---
buf | 226 | read,write,read_write
file | 36 | 5,10,8
filesystem | 46 | 3,4,3
inode | 448 | 1,6,4
kill | 106 | 1,16,11
lpl1 | 8 | 7,6,4
mount | 11 | 2,2,1
proc | 425 | 3,2,1
text | 11 | 7,11,12
u | 3303 | 13,37,34

Figure 4.4. Global flows for UNIX modules

This shows all possible flows of information from those procedures which can update the data structure (write and read_write) to all procedures which retrieve information from the data structure (read and read_write).

The global flows measurement will indicate overloaded data structures. Figure 4.4 shows that the U data structure has 3303 global flows with 84 procedures. Clearly, the U data structure, as compared to all other data structures, is extremely overloaded. The primary reason for this is that one function of this structure is to pass error codes to other levels in the system. Since over half of the procedures have access to the U data structure, the module is so complex that it distorts other measurements. Accordingly, the U structure will not be given further consideration in this analysis. UNIX is not the only operating system with a complex, overloaded data structure. IBM OS/360 has a large global data structure called the communications vector table. The inability to control access to this table has led to a number of reliability and adaptability problems (Myers, 1976).
Appendix C displays the global flows, read only procedures, write only procedures, and read_write procedures for the other UNIX modules.

The complexity of a module is taken to equal the sum of the complexities of the procedures within the module. Figure 4.5 gives the procedures within each module, the complexity of each procedure, and the total complexity of the module. Appendix D gives the complexities for the other UNIX modules. (The complexities for individual procedures differ from those in Figure 4.2. Recall that the complexities in Figure 4.2 were calculated with the total local flows from all data structures, and the complexities in Figure 4.5 were calculated using only the local flows for the given module.)

It is interesting to note that the majority of a module's complexity is due to a few very complex procedures. Figure 4.6 reveals the modules, their complexity, the sum of the three largest procedures' complexities, and the percentage of that sum to the module complexity. In all but one case the three most complex procedures constitute more than 85% of the module complexity.

The global flows and the module complexities show four areas of potential design or implementation difficulties for the module. First, as with the U structure the global flows indicates a poorly refined (i.e., overloaded) data structure. Redesign of the data structure to segment it into several pieces may be a solution to this overloading. Second, the module complexities indicate improper modularization. It is desirable that a procedure be in one and only one module, and the measurements illustrate violations of this property. High global flows and a low or
Module complexity

**buf**
- bawrite: 1296
- bdwrite: 3375
- flush: 648
- binit: 270
- bread: 919828
- breada: 29700
- brelse: 13225
- bwrite: 12544
- clrbuf: 13
- devstart: 22
- getblk: 2446136
- geterror: 10
- incore: 5600
- iodone: 1215
- iowait: 1200
- itrunc: 102060
- mapalloc: 22
- mapfree: 9
- notavail: 240
- physio: 2520
- rkintr: 92
- rkstrategy: 1044
- wakeup: 14

3541083

**file**
- alloc: 8100
- closef: 612
- falloc: 720
- openl: 14800
- pipe: 124
- rdwr: 6525
- readp: 1512
- seek: 172
- sgtty: 29
- writep: 468

33062

Figure 4.5. Module complexities
filesys
alloc   32400
dup     48
free    7168
fstat   36
getfs   190000
ialloc  31680
ifree   4800
iinit   684
newproc 855
smount  848
update  268

268807

inode
access  15435
bmap    85500
chmod   13
chown   44
closef  272
core    1764
exec    11520
getmdev 256
ialloc  31680
iget    69825
iput    1293975
itrunc  138915
iupdat  90944
link    342
main    1392
maknode 9375
mknod   198
namei   11551995
pipe    31
plock   48
prele   17600
readi   52675
readp   2688
smdate  288
smount  848
statl   6480
sumount 120
unlink  3744
update  512
writei  56350
writep  3328
xalloc  14688

Figure 4.5 (Continued)
Figure 4.5 (Continued)
proc
  clock  3852
  exit   8428
  expand 21141
  getpid  4
  grow   828
  issig  18
  kill   25
  klopen 30
  main   1392
  newproc 855
  nice   1
  physio 4480
  procxmt 775
  psig   155
  psignal 960
  ptrace  544
  sched  2850
  setpri  48
  setrun  2160
  setuid  11
  signal 116
  sleep  338560
  ssig   14
  stop   272
  sureg  26
  wait  1250
  wakeup 13454
  xalloc 19992
  xfree   76
  xswap  13824
  436151

text
  newproc  855
  sched  1026
  sureg   26
  xalloc  22950
  xccdec  10
  xfree   19
  24886

Figure 4.5 (Continued)
average module complexity indicate a third area of difficulty, namely poor internal module construction. Numerous procedures have direct access to the data structure but there is little communication among the procedures. Fourth, a low global flows and high module complexity may reveal either a poor functional decomposition within the module or a complicated interface with other modules.

Interface Measurements

The interface between modules is important for two reasons. First, the interface allows the system components to be distinguished. Second, the interface serves to connect the components of the system together. The interface between two modules consists of three factors: 1) the protocol interface, 2) the binding of the modules, and 3) the coupling of the modules. These three factors will be defined and examined in the following sections.

<table>
<thead>
<tr>
<th>Module</th>
<th>Complexity</th>
<th>Complexity of three largest procedures</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>3541083</td>
<td>3468024</td>
<td>98</td>
</tr>
<tr>
<td>file</td>
<td>33062</td>
<td>29425</td>
<td>89</td>
</tr>
<tr>
<td>filesystem</td>
<td>268807</td>
<td>254080</td>
<td>95</td>
</tr>
<tr>
<td>inode</td>
<td>13462921</td>
<td>12984995</td>
<td>96</td>
</tr>
<tr>
<td>kill</td>
<td>3262</td>
<td>2120</td>
<td>65</td>
</tr>
<tr>
<td>lpl1</td>
<td>855</td>
<td>829</td>
<td>97</td>
</tr>
<tr>
<td>mount</td>
<td>135503</td>
<td>135084</td>
<td>99</td>
</tr>
<tr>
<td>proc</td>
<td>436151</td>
<td>379693</td>
<td>87</td>
</tr>
<tr>
<td>text</td>
<td>24886</td>
<td>24831</td>
<td>99</td>
</tr>
</tbody>
</table>

Figure 4.6. Percent of module complexity for largest procedures
The protocol interface includes those procedures which translate information between two modules. Using the module and interface complexity measurements, the accumulated complexity of a (sub)system can be calculated since the complexity of a (sub)system is equal to the sum of the complexities of its components. For example, the complexity of two related modules A and B is the sum of the complexity of module A, the complexity of module B, and the complexity of the protocol interface from A to B. The protocol interface is defined as follows:

Definition 4.4: The protocol interface from module A to module B consists of those procedures which are not in any other module and which receive information from module A and send information to module B.

The complexity of the protocol interfaces is discussed in this section.

Intuitively, the binding between two modules is a measure of how sensitive the first module is to the external functions provided by the second module. The more sensitive the relationship the tighter the binding. This concept is referred to as binding because tightly bound modules cannot be easily understood or altered independently - they are bound together. Notice that the binding relationship is not symmetric - module A may be tightly bound to module B without the reverse being true. The binding between two modules includes the binding through the protocol interface and the binding through a direct flow of information.

A related concept, coupling, refers to the strength of the connections between modules. The coupling of two modules is based on their binding to each other. The scale of the numbers used for the binding
and coupling measurements is not the same as the scale used for the complexity measurements. The reader is cautioned not to directly compare these measurements.

Figure 4.7 depicts two modules A and B, the protocol interface between them, and the connections through the protocol interface and direct flows. The labels on Figure 4.7 are discussed later in this section.

Figure 4.7. Description of the interface between two modules
Protocol interface complexities

The procedures included in a protocol interface are not in any other module. Figure 4.8 shows the complexities of the protocol interface for the UNIX modules. The protocol interfaces for UNIX do not include many procedures and those procedures are not very complex, but this may not be true for other operating systems or other software in general.

Figure 4.6 showed that a few procedures dominated the complexities for the modules. This is also true for the protocol interface complexities. In each case, there is one procedure that contributes most of the protocol interface complexity. Note that the protocol interface complexity from INODE to BUF, FILE, and FILESYS is the same. INODE uses the same set of procedures as an interface to all three modules.

Binding

The connections between two modules is a function of the number of procedures involved in exporting and importing information and the number of paths used to transmit the information. The binding from module A to module B through a protocol interface involves five factors (refer to Figure 4.7):

1. the number of procedures sending information from module A (NSP),
2. the number of procedures receiving information from module B (NRP),
3. the number of procedures in the protocol interface (NPI),
4. the number of paths to the protocol interface from module A (SPI),
5. the number of paths from the protocol interface to module B (PIR).
### Figure 4.8. Protocol interface complexities

<table>
<thead>
<tr>
<th></th>
<th>buf</th>
<th>file</th>
<th>filesys</th>
<th>inode</th>
<th>kl1l</th>
<th>lp1l</th>
<th>mount</th>
<th>proc</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>108</td>
<td>108</td>
<td>0</td>
<td>432</td>
<td>0</td>
</tr>
<tr>
<td>file</td>
<td>18165</td>
<td>--</td>
<td>24717</td>
<td>18165</td>
<td>18336</td>
<td>192</td>
<td>0</td>
<td>15228</td>
<td>0</td>
</tr>
<tr>
<td>filesys</td>
<td>18165</td>
<td>24696</td>
<td>--</td>
<td>18144</td>
<td>18336</td>
<td>192</td>
<td>0</td>
<td>18576</td>
<td>0</td>
</tr>
<tr>
<td>inode</td>
<td>35784</td>
<td>35784</td>
<td>35784</td>
<td>--</td>
<td>16292</td>
<td>108</td>
<td>0</td>
<td>36216</td>
<td>3920</td>
</tr>
<tr>
<td>kl1l</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lp1l</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mount</td>
<td>0</td>
<td>0</td>
<td>1215</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>432</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>proc</td>
<td>35301</td>
<td>35280</td>
<td>35301</td>
<td>32580</td>
<td>15872</td>
<td>192</td>
<td>35301</td>
<td>--</td>
<td>3920</td>
</tr>
<tr>
<td>text</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>108</td>
<td>108</td>
<td>0</td>
<td>0</td>
<td>--</td>
</tr>
</tbody>
</table>

### Figure 4.9. Binding through protocol interface

<table>
<thead>
<tr>
<th></th>
<th>buf</th>
<th>file</th>
<th>filesys</th>
<th>inode</th>
<th>kl1l</th>
<th>lp1l</th>
<th>mount</th>
<th>proc</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>file</td>
<td>15</td>
<td>--</td>
<td>18</td>
<td>63</td>
<td>28</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>filesys</td>
<td>3</td>
<td>14</td>
<td>--</td>
<td>32</td>
<td>14</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>inode</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>--</td>
<td>39</td>
<td>10</td>
<td>0</td>
<td>45</td>
<td>12</td>
</tr>
<tr>
<td>kl1l</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lp1l</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mount</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>proc</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>14</td>
<td>8</td>
<td>3</td>
<td>--</td>
<td>2</td>
</tr>
<tr>
<td>text</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>--</td>
</tr>
</tbody>
</table>
The formula for the binding of two modules through a protocol interface is defined to be the following:

\[(\text{NSP} + \text{NPI}) \times \text{SPI} + (\text{NPI} + \text{NRP}) \times \text{PIR}.\]

The term \((\text{NSP} + \text{NPI}) \times \text{SPI}\) represents the strength of the connection between module A and the protocol interface by including the number of paths transmitting information (SPI) and the number of procedures involved in the information transfer (NSP + NPI). Likewise, the term \((\text{NPI} + \text{NRP}) \times \text{PIR}\) represents the strength of the connection between the protocol interface and module B. The measurements for the binding through a protocol interface for the UNIX modules are given in Figure 4.9.

The binding of two modules is also concerned with the direct flow of information between the modules. Like the binding through a protocol interface, the communication between two modules through a direct flow of information involves the number of procedures importing and exporting information and the number of paths involved in the transfer of information. The binding through direct flow of information from module A to module B consists of three factors (refer to Figure 4.7):

1. the number of procedures sending information to module A (NSP),
2. the number of procedures receiving information from module B (NRP),
3. the number of direct flows from A to B (DF).

The formula for the binding of two modules through direct flow of information is defined to be the following:

\[(\text{NSP} + \text{NRP}) \times \text{DF}.\]

The term \((\text{NSP} + \text{NRP})\) reflects the number of procedures involved in the
communication and DF represents the number of information paths from module A to module B. The measurements for the binding through direct flow for the UNIX modules are given in Figure 4.10.

Note that the binding is not symmetrical. INODE is tightly bound to BUF, but BUF is loosely bound to INODE. Figures 4.9 and 4.10 show that there is little communication through a protocol interface and most of the communication is through a direct flow of information.

Coupling

A primary design goal is to minimize the connections among the modules. Module coupling is a measure of the data relationships among the modules. Myers (1976) explores six categories of coupling which involved parameter passing and data structures. The information flow metrics can recognize some of these categories, namely those modules that are content coupled and those that are common coupled. Content coupling refers to a direct reference between the modules, this is observed by the binding through direct flows. Common coupling refers to the sharing of a global data structure and this is observed with the module measurement for module violation.

The coupling between two procedures A and B consists of the sum of four factors:

1. the binding through the protocol interface from A to B,
2. the binding through the protocol interface from B to A,
3. the binding through direct flow of information from A to B,
4. the binding through direct flow of information from B to A.

The measurements of coupling for the UNIX modules are given in Figure 4.11.
### Figure 4.10. Binding through direct flows

<table>
<thead>
<tr>
<th></th>
<th>buf</th>
<th>file</th>
<th>filesys</th>
<th>inode</th>
<th>kill</th>
<th>lkill</th>
<th>mount</th>
<th>proc</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>--</td>
<td>2</td>
<td>24</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>file</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>filesys</td>
<td>49</td>
<td>2</td>
<td>--</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>inode</td>
<td>323</td>
<td>18</td>
<td>88</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>56</td>
<td>6</td>
</tr>
<tr>
<td>kill</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>lkill</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mount</td>
<td>42</td>
<td>0</td>
<td>42</td>
<td>48</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>proc</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>text</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
</tr>
</tbody>
</table>

### Figure 4.11. Coupling between UNIX modules

<table>
<thead>
<tr>
<th></th>
<th>buf</th>
<th>file</th>
<th>filesys</th>
<th>inode</th>
<th>kill</th>
<th>lkill</th>
<th>mount</th>
<th>proc</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>--</td>
<td>17</td>
<td>--</td>
<td>34</td>
<td>191</td>
<td>1.75</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>file</td>
<td>0</td>
<td>76</td>
<td>34</td>
<td>--</td>
<td>--</td>
<td>39</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>filesys</td>
<td>355</td>
<td>8</td>
<td>50</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>inode</td>
<td>4</td>
<td>28</td>
<td>14</td>
<td>39</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>kill</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lkill</td>
<td>46</td>
<td>52</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>mount</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>proc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>--</td>
</tr>
</tbody>
</table>
The coupling measurements show the strength of the connections between two modules. An observation of the coupling measurements for the UNIX modules reveals very strong connections between INODE and BUF, FILE, FILESYS, and PROC. This strong coupling of INODE to the rest of the system indicates that a substantial change to the INODE module would strongly affect other system components. Coupling also indicates a measure of modifiability. If modifications are made to a particular module, the coupling indicates which other modules are affected and how strongly the other modules are connected. These measurements are useful during the design phase of a system to indicate which modules communicate with which other modules and the strength of that communication. During implementation or maintenance, the coupling measurement is a tool to indicate what effect modifying a module will have on the other components of a system.

Level Complexities

Given a hierarchy for a system, the accumulated complexity for the levels may be computed using the module complexities. A partial hierarchy for the UNIX operating system is given in Figure 4.12. The modules displayed represent the character devices, block devices, the file system and the process management routines.

<table>
<thead>
<tr>
<th>Level</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>proc,text</td>
</tr>
<tr>
<td>4</td>
<td>mount</td>
</tr>
<tr>
<td>3</td>
<td>file,filesys</td>
</tr>
<tr>
<td>2</td>
<td>inode</td>
</tr>
<tr>
<td>1</td>
<td>buf</td>
</tr>
<tr>
<td>0</td>
<td>k111,lpl1</td>
</tr>
</tbody>
</table>

Figure 4.12. Partial hierarchy for UNIX
Figure 4.13 displays a graph of the levels in the UNIX hierarchy and the accumulated complexity for the system. Level 0 which has a complexity of 4,117 has been omitted from the graph. The increase in complexity between levels 0 and 1, and levels 1 and 2 indicates a missing level of abstraction. When the slope of the accumulated complexities indicates a large increase, the procedures should be analyzed for a missing level of refinement. The design of these levels should be reviewed for possible redesign. The addition of this missing level of abstraction is further discussed in the next chapter. A comparison of various design alternatives is possible by reviewing the total accumulated complexities prior to implementation. The system with the lowest total complexity would be selected for implementation.

The procedure, module and interface measurements presented in this chapter reveal potential design and implementation difficulties. Figure 4.14 summarizes the measurements together with the particular design and implementation features associated with the measurements.
Figure 4.13. Graph of accumulated complexities for the UNIX hierarchy
<table>
<thead>
<tr>
<th>Measurement</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure measurements</td>
<td>1. lack of functionality</td>
</tr>
<tr>
<td></td>
<td>2. stress points in the system</td>
</tr>
<tr>
<td></td>
<td>3. inadequate refinement</td>
</tr>
<tr>
<td>Module measurements</td>
<td>1. poorly designed data structures</td>
</tr>
<tr>
<td></td>
<td>2. improper modularization</td>
</tr>
<tr>
<td></td>
<td>3. poor module design</td>
</tr>
<tr>
<td></td>
<td>4. poor functional decomposition</td>
</tr>
<tr>
<td>Interface measurements</td>
<td>1. strength of the binding between modules</td>
</tr>
<tr>
<td></td>
<td>2. measure of modifiability</td>
</tr>
<tr>
<td>Level measurements</td>
<td>1. missing level of abstraction</td>
</tr>
<tr>
<td></td>
<td>2. comparison of design alternatives</td>
</tr>
</tbody>
</table>

Figure 4.14. Summary of the measurements and corresponding features
The correlation between the complexity measurements and a collection of changes suggested for the UNIX system is investigated in order to further justify the set of metrics presented in Chapter IV. The correlation between certain design and implementation flaws within the system and the measurements is also investigated.

Correlation of Errors to Measurements

Although UNIX is installed at many computer facilities, there are errors in the code. This is a typical condition for production operating systems. For example, IBM OS/360 contains approximately 1000 errors in every release (Yourdon, 1975). Ferentz, from Rockefeller University, has collected a list of suggested changes for UNIX. In private correspondence with Ferentz (1979), the author obtained a list of these recommended changes to UNIX. Eighty of the changes involved procedures used in the information flow analysis. These corrections and enhancements will be used to determine the ability of the complexity measurement to predict procedures which, with a high probability, will contain errors. The suggested changes to UNIX consist of both actual errors and some necessary performance enhancements. These changes correspond to any natural update to an existing operating system.

One of the design features observed by the module measurements was that of improper modularization; those procedures located in more than one module. It is the goal of modularization to have each procedure in
one and only one module (Parnas, 1977). Procedures which violate this principle are more prone to error due to the large number of connections involving more than one module. In the UNIX operating system there are 53 such procedures and 38 were involved in the list of UNIX errors received from Ferentz. Figure 5.1 displays the distribution of UNIX procedures in one module and those contained in more than one module, the number of changes associated with these procedures, and the percentage of procedures to be changed. The conclusion from this is simply that procedures which violate modularity are more prone to error. This conclusion is not at all surprising and documents the validity of the principles advanced under the title of abstract data typing.

<table>
<thead>
<tr>
<th>Procedures in more procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of procedures</td>
</tr>
<tr>
<td>53</td>
</tr>
<tr>
<td>Number of changes</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>Percent</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>Procedures in one procedures</td>
</tr>
<tr>
<td>Number of procedures</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>Number of changes</td>
</tr>
<tr>
<td>42</td>
</tr>
<tr>
<td>Percent</td>
</tr>
<tr>
<td>38</td>
</tr>
</tbody>
</table>

Figure 5.1. Correlation of module violations to errors

The correlation of errors to procedure complexity as well as modularization was investigated. There is a high correspondence between the recommended changes to UNIX and high procedure complexity. Eleven out of the twelve procedures with complexity $10^5$, and two out of the three procedures with complexity $10^6$ required changes. The procedure with complexity $10^7$ did not require a change. However, as indicated by
Lions (1977), NAMEI was designed early, debugged thoroughly, and since its connections to the system are very complex NAMEI was not changed. Figure 5.2 reveals the number of procedures for each order of complexity, the number of procedures with recommended changes, and the percentage of those procedures containing errors.

<table>
<thead>
<tr>
<th>Order of complexity</th>
<th>Number of procedures</th>
<th>Number of procedures with errors</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>38</td>
<td>12</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
<td>19</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>19</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>15</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>11</td>
<td>92</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.2. Correlation of procedure complexity to errors

The percentages in Figure 5.2 reflect a high correlation between the order of complexity of the UNIX procedures and the recommended changes. A simple procedure is less prone to error than a more complex one. In Figure 5.2 compare level 0 to level 5 or 6.

Correlation of Design Flaws to Measurements

The measurements in Chapter IV reveal certain design flaws in the system. The graph of accumulated complexity presented in Figure 4.12 shows a large increase in complexity between level 0 and level 1. This large increase in complexity is because level 0 has an extremely low complexity and because there is a missing level of abstraction in the
BUF data structure. If the BUF module were redesigned to include this level of abstraction, the complexity of level 1 would be greatly reduced. The results of this redesign are presented in this section. The three procedures contributing the most to the complexity of the BUF module are: GETBLK, BREAD, and ITRUNC. GETBLK and BREAD are investigated in order to redesign the BUF module, however, prior to the redesign the ITRUNC procedure is examined.

Module violation

As was noted earlier, the module measurements revealed that ITRUNC is contained in two modules namely, BUF and INODE. Examination of the code for ITRUNC reveals that this violation of modularity can be easily remedied. The function of ITRUNC is to free all the disk blocks associated with a specified INODE structure. If ITRUNC is allowed to access only the INODE data structure and instead of accessing the BUF structure directly calls another procedure ITRUNC' to perform this function, the complexity of ITRUNC with respect to the BUF module is removed. The complexity of INODE will not be affected by this change. The complexity of ITRUNC' (216) must be included in the BUF module. The complexity of the BUF module is therefore reduced by 101,844. Although the complexity of BUF is still 3,439,239, the third largest procedure has been removed leaving only GETBLK and BREAD to redesign. The "C" code for ITRUNC, the revised code for ITRUNC, and the code for ITRUNC' is displayed in Appendix E.
Redesign of the BUF module

Since the complexity of ITRUNC has been virtually eliminated from the BUF module, GETBLK and BREAD will be investigated in order to evaluate the causes of complexity in the BUF module. The BUF data structure contains buffer information for the block I/O system. Examination of the BUF module reveals that the BUF data structure is used for several functions. Buffers are allocated for five separate functions:

1. for swapping space,
2. for user file space,
3. for the super block (this contains information used in allocating resources),
4. for the i-list (used by the INODE module),
5. for temporary storage.

The first four of these functions correspond to distinct areas of the physical disk while the last function does not correspond to any use of disk space. These separate spaces are manipulated in distinct ways. The BUF structure operations fail to distinguish among the physical and logical distinctions between these five functions. This failure is the root cause of the complexity of the BUF module because it leads to an exaggerated fan-in, fan-out for some BUF procedures. To remedy this situation an additional level of abstraction, distinguishing the five functions noted above, will be added. It will be shown that the addition of this structural abstraction, desirable from an esthetic standpoint, also leads to a reduction in the complexity measurements for the BUF module.

To add the level of abstraction to the BUF module, it is necessary to divide the local flow for GETBLK and BREAD into the functions
mentioned before. The procedures outside of the BUF module which now call GETBLK or BREAD directly, will now call a procedure for a specific function. The new procedure will in turn call GETBLK or BREAD. In adding these five procedures the logical structure of BUF is changed. These procedures represent a level of abstraction which logically changes the interface to BUF from the other modules. The addition of the new level will reduce the fan-in and fan-out for GETBLK and BREAD thus reducing their complexities. Figure 5.3 displays the fan-in and fan-out for GETBLK and BREAD. Nineteen procedures contribute to the fan-in for GETBLK, and eleven contribute to the fan-out. BREAD's fan-in is nineteen and the fan-out is fourteen.

Figure 5.3. Distribution of fan-in and fan-out for GETBLK and BREAD
Figure 5.4 displays the fan-in and fan-out for GETBLK and BREAD with the addition of the procedures A, B, C, D, and E. These new procedures correspond to the five functions of the BUF structure listed above. Procedures A - E constitute the new level of abstraction. The numbers at the bottom of the figure indicate the fan-in and fan-out for GETBLK and BREAD from procedures within the BUF module, i.e., five procedures inside the BUF structure contribute to the fan-in for GETBLK. The new fan-in for GETBLK is nine and the fan-out is eleven. However, since GETBLK is a read-write procedure an additional fan-in and fan-out is added. The new complexity for GETBLK is 806,400 as compared to the previous 2,446,136. This represents a 67% reduction in complexity. The new fan-in for BREAD is seven and the fan-out is nine plus one for each since BREAD is a read-write procedure. The new complexity is 83,200 as compared to the previous 919,828 representing a reduction of 91%.

Since each of the new procedures is relatively simple it will be assumed that each of the procedures A, B, C, D, and E has 10 lines of code. The complexities of these procedures are:

A - 9000
B - 39690
C - 23040
D - 64000
E - 23040

The complexity of this new level is 158,770, and the new complexity for the BUF module is 962,335. The previous complexity for BUF was 3,541,083 resulting in a 73% reduction.

The graph of accumulated complexities in Figure 4.12 can be derived for the new BUF module and the additional level of abstraction. The
Figure 5.4. Fan-in and fan-out for GETBLK and BREAD with additional level of abstraction
levels are:

- level 6 - proc, text
- level 5 - mount
- level 4 - file, filesys
- level 3 - inode
- level 2 - procedures A - E
- level 1 - buf
- level 0 - klll, lpl1.

Figure 5.5 represents the graph of accumulated complexities for the new levels. The addition of the level of abstraction smooths out the slope of the graph. However, there is still a large rise from level 0 to level 1. One reason for this is the small complexity for Klll and Lpl1, and the high complexity of GETBLK is another. Even though the complexity of GETBLK was substantially reduced, the fan-in and fan-out remained very large. The procedures within the BUF module are the cause of this factor. The next iteration of the redesign process would be to investigate the code for those procedures which interact with GETBLK in the BUF module. It may be necessary to add another level of refinement to the BUF module. This possible redesign is not pursued farther in this thesis.

Redesign of the INODE module

There is still a large increase in the complexity between level 2 and level 3 in Figure 5.5. The high INODE module complexity is due primarily to NAMEI which contributes 86% to the complexity. There are two reasons for this high complexity. First, NAMEI is a very long procedure, 155 lines of code, indicating a possible implementation problem. Second, NAMEI has a fan-in and fan-out of 13 and 21 respectively. This indicates
Figure 5.5. Graph of accumulated complexities for the revised hierarchy for UNIX with redesign of BUF.
a possible design problem. These two possible problems of NAMEI are investigated further.

NAMEI could have been further refined as three procedures. Partitioning NAMEI into three procedures will greatly reduce its complexity. The three procedures are called NAMEI, NAMEI', and NAMEI''. The "C" code for the original NAMEI and the code for these three procedures is in Appendix F. Examination of these coding changes reveals that NAMEI contains a large loop with a nested inner loop. The changes to NAMEI removed the large loop to NAMEI' and the inner loop to NAMEI''. This implementation change reflects accepted standards of good programming practice. The fan-in, fan-out, length, and complexity of NAMEI, NAMEI', and NAMEI'' are given in Figure 5.6. The complexity of the new NAMEI is now 1,531,747. This change in the implementation of NAMEI represents an 87% reduction in complexity and a 74% reduction for the INODE module. The complexity of the INODE module is now 3,442,673. Figure 5.7 displays a graph of the accumulated complexities for the UNIX hierarchy with this implementation change to NAMEI.

<table>
<thead>
<tr>
<th></th>
<th>Fan-in</th>
<th>Fan-out</th>
<th>Length</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>namei</td>
<td>17</td>
<td>13</td>
<td>31</td>
<td>1,514,071</td>
</tr>
<tr>
<td>namei'</td>
<td>2</td>
<td>6</td>
<td>79</td>
<td>11,376</td>
</tr>
<tr>
<td>namei''</td>
<td>2</td>
<td>5</td>
<td>63</td>
<td>6,300</td>
</tr>
</tbody>
</table>

Figure 5.6. Complexities for the revised NAMEI procedure
Figure 5.7. Graph of accumulated complexities for the hierarchy for UNIX with implementation change to NAMEI.
The function of \texttt{NAMEI} is to search a directory and return a pointer to an \texttt{INODE} structure. \texttt{NAMEI} is called for three distinct functions:

1. seek
2. create
3. delete.

Just as in the \texttt{BUF} structure there appears to be a missing level of abstraction. By adding three procedures to interface with \texttt{NAMEI}, the logical structure of \texttt{INODE} is changed and another level of abstraction is added. Figure 5.8 shows the fan-in, fan-out, and complexity for these three procedures and for \texttt{NAMEI}. Again, it is assumed that each of these procedures is 10 lines of code in length. Note that the fan-in and fan-out for \texttt{NAMEI}' and \texttt{NAMEI}'' do not change. By adding this level of abstraction, the complexity of \texttt{NAMEI} is reduced by 98\% to 20,776. The complexity of the new level is 65,490, and the new complexity of the \texttt{INODE} module is 1,931,702, an additional 44\% reduction.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Fan-in</th>
<th>Fan-out</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>seek</td>
<td>8</td>
<td>10</td>
<td>64,000</td>
</tr>
<tr>
<td>create</td>
<td>4</td>
<td>3</td>
<td>1,440</td>
</tr>
<tr>
<td>delete</td>
<td>1</td>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td>namei</td>
<td>4</td>
<td>6</td>
<td>3,100</td>
</tr>
<tr>
<td>namei'</td>
<td>2</td>
<td>6</td>
<td>11,376</td>
</tr>
<tr>
<td>namei''</td>
<td>2</td>
<td>5</td>
<td>6,300</td>
</tr>
</tbody>
</table>

Figure 5.8. Complexities for the new level of abstraction in \texttt{INODE}

With both the implementation and design enhancements to \texttt{NAMEI}, the \texttt{INODE} module complexity has been reduced by 86\%. Figure 5.9 displays a graph of accumulated complexities for:
Figure 5.9. Accumulated complexities for UNIX hierarchies
1. the original UNIX hierarchy
2. UNIX hierarchy with redesign of a level of abstraction for BUF
3. UNIX hierarchy with implementation change to NAMEI
4. UNIX hierarchy with redesign of a level of abstraction for INODE.

The levels in the newest UNIX hierarchy with the level of abstraction to INODE are:

- level 0: kill, lp1l
- level 1: buf
- level 2: procedures A-E
- level 3: inode
- level 4: procedures seek, create, delete
- level 5: file, filesys
- level 6: mount
- level 7: proc, text

It was recognized that there were both design and implementation difficulties with NAMEI. Originally, the complexity of NAMEI was 11,551,995. After the implementation change, the complexity was reduced to 1,531,747. The addition of a level of abstraction to NAMEI reduced the complexity to 20,776. An alternative to this analysis of NAMEI is to add the level of abstraction first and the implementation change second. With this alternative, the level of refinement is used with the original code for NAMEI, resulting in a complexity of 357,120. Figure 5.10 displays these complexities. Information flow allows a designer to compute these complexities prior to any actual changes to the code. The designer may then compare the difficulties in accomplishing a design or implementation change, or both, against the respective reduction in complexity.
In summary, the complexity measurements presented in Chapter IV reveal a high correlation to detected errors and necessary performance changes in the UNIX operating system. The measurement also identifies appropriate areas for redesign and reimplementation as demonstrated in the reworked BUF and INODE modules.
CHAPTER VI. CONCLUSIONS AND SUGGESTIONS
FOR ADDITIONAL INVESTIGATION

This research has investigated the development of an integrated set of metrics to assist system designers and implementors in developing higher quality, more reliability software. Reliability implies that all of the connections between the components of a system must be known. Considerable research has been reported on designing ordering relations which determine the connections in a system. The most popular ordering relation is that of a flow of control, or a calling hierarchy. Later ordering relations such as uses and dependency recognized that many connections are not observed by a calling hierarchy. This research showed that uses is not a precise concept and that dependency attempts to be more defined than uses but still does not observe all of the connections between system components. Information flow reveals most of the flow of control structure (direct local flow), the connections through data structures (global flows), and also the implicit connections represented by the indirect local flows.

The information flow analysis is valuable for all software systems. In this dissertation an operating system is used for the information flow analysis since it is the author's belief that operating systems are among the most complex software systems. The measurements available from the information flow analysis may be used at three stages of software development. The first stage, after the design specifications have been written, reveal such design problems as:

1. lack of functional decomposition in procedures and modules,
2. improper modularization,
3. poorly designed modules,
4. poorly designed data structures,
5. stress points in a system,
6. inadequate refinement,
7. strength of binding between modules,
8. modifiability measures,
9. missing level of abstraction,
10. comparison of design alternatives.

A second stage of system development where information flow measurements are used is during implementation. Information flow allows the implementor to know exactly which procedures and modules flow into a given procedure. Also, if a change is required to a procedure or a data structure, all other procedures affected by these changes are observed. The third stage is after implementation to evaluate the system's structure and to project the affects of any enhancements to the system.

An information flow analysis was performed on the UNIX operating system and complexity measurements were presented for the procedures, modules, and protocol interfaces present in UNIX. Binding and coupling measurements were also presented for the modules. A high correlation was shown between the procedure complexity measurements and the suggested changes to UNIX from the UNIX User's Group (Ferentz, 1979). The changes involved not only actual errors in UNIX, but also some necessary performance enhancements. The measurements also pointed to some areas of design and implementation difficulties. The analysis of these problem
areas led to improvement in both design and implementation by reducing the number of connections among components and thus, reducing the complexity.

An automated set of metrics is a necessary requirement for the designers and implementors of complicated software systems in order to allow the evaluation of the system's structure. Since the set of metrics presented in this dissertation is based on information flow, all information connections between the system components are observed.

The information flow analysis does have one drawback, namely the missed calls. It is the belief of the author that it would be a simple procedure to revise the generation of the relations and to modify the algorithm which generates the information flow structure in order to observe these missed flow of control relations. The modification to the relations would include the notation to treat the program counter as a special type of implicit parameter.

The use of information flow requires further investigation into other operating systems since the UNIX operating system does not involve any great degree of memory management or very sophisticated protection features. These areas may reveal additional benefits to the information flow analysis.

There was no attempt made in this research to separate parallelism or synchronization. Both of these areas are essential in operating systems and should be an interesting aspect to investigate. Shaw (1978) defined a means of describing software in terms of flow expressions. Flow expressions describe sequential and concurrent flows of entities,
such as control, messages, and resources through system software components. Perhaps Shaw's use of concurrency could be related to the information flow analysis.

Design flaws can be recognized by taking the complexity measurements after the design phase. In order to achieve this, it is necessary to develop a formal specification technique to allow the information flow analysis to be done at this point. An area of further study is the investigation into applying the research in specification techniques to the information flow analysis.

The information flow approach to software quality measurements is a valuable tool to software developers as an aid in design, implementation, and evaluation of the system. Further investigation into the areas mentioned above could reveal even more benefits of information flow metrics.
BIBLIOGRAPHY


Janson, P. A. 1977. Using Type-extension to Organize Virtual Memory Mechanisms. IBM Zurich Research Laboratory RZ 858. Ruschlikon, Switzerland.


In the production of this dissertation, the support received has ranged from technical to moral. All of it was important. The roles played during the research and writing of this dissertation can best be summarized by:

<table>
<thead>
<tr>
<th>Role</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Director</td>
<td>Dennis G. Kafura</td>
</tr>
<tr>
<td>Technical assistance coordinator</td>
<td>Dennis G. Kafura</td>
</tr>
<tr>
<td>Committee</td>
<td>Dennis G. Kafura, Robert M. Stewart, Denise M. Eckstein, Roy Zingg, Arthur V. Pohm</td>
</tr>
<tr>
<td>Typist</td>
<td>Gwen Ethington</td>
</tr>
<tr>
<td>Moral Support</td>
<td>Tom Fritz, David Henry, Dennis Kafura, Jim Wittneben</td>
</tr>
</tbody>
</table>

The author wishes to give a special acknowledgment to her husband, David, and her sons, Troy and Bryan, whose love, understanding, and encouragement were an essential ingredient during the research and writing of this dissertation.
APPENDIX A. "C" CODE FOR BREADA AND INCORE

breada(adev, blkno, rablkno)
{
    register struct buf *rbp, *rabp;
    register int dev;

    dev = adev;
    if (!incore(dev, blkno)) {
        rbp = getblk(dev, blkno);
        if (((rbp->b_flags & B_DONE) == 0) {
            rbp->b_flags = B_READ;
            rbp->b_wcount = -256;
            rkstrategy(rbp);
        }
    }
    if (rablkno && !incore(dev, rablkno)) {
        rabp = getblk(dev, rablkno);
        if (rabp->b_flags & B_DONE)
            brelse(rabp);
        else {
            rabp->b_flags = B_READ: B_ASYNC;
            rabp->b_wcount = -256;
            rkstrategy(rabp);
        }
    }
    if (rbp==0)
        return (bread(dev, blkno));
    iowait(rbp);
    return(rbp);
}

incore(adev, blkno)
{
    register int dev;
    register struct buf *bp;
    register struct devtab *dp;
    dev = adev;
    dp = bdevsw[adev.d_major].d_tab;
    for (bp=dp->b_forw; bp != dp; bp = bp->b_forw)
        if (bp->b_blkno==blkno && bp->b_dev==dev)
            return(bp);
    return(0);
}
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callout</td>
<td>argument information for timed calls</td>
</tr>
<tr>
<td>canonb</td>
<td>buffer for erase and kill</td>
</tr>
<tr>
<td>cfree</td>
<td>character lists</td>
</tr>
<tr>
<td>cfreelist</td>
<td>pointer to character lists</td>
</tr>
<tr>
<td>coremap</td>
<td>space for core allocation</td>
</tr>
<tr>
<td>curpri</td>
<td>scheduling information</td>
</tr>
<tr>
<td>execnt</td>
<td>number of processes in exec</td>
</tr>
<tr>
<td>ipc</td>
<td>tracing variables</td>
</tr>
<tr>
<td>kltbuf</td>
<td>transmit buffer for console</td>
</tr>
<tr>
<td>kltcsr</td>
<td>transmit control status register for console</td>
</tr>
<tr>
<td>lbolt</td>
<td>time of day in 60th not in time</td>
</tr>
<tr>
<td>lpbuf</td>
<td>line printer buffer</td>
</tr>
<tr>
<td>lpsr</td>
<td>line printer status register</td>
</tr>
<tr>
<td>maplock</td>
<td>allocates the unibus map</td>
</tr>
<tr>
<td>maxmem</td>
<td>actual maximum memory per process</td>
</tr>
<tr>
<td>rkba</td>
<td>RK buffer address register</td>
</tr>
<tr>
<td>rkcs</td>
<td>RK control status register</td>
</tr>
<tr>
<td>rkda</td>
<td>RK disk address register</td>
</tr>
<tr>
<td>rktab</td>
<td>private state information for each block device</td>
</tr>
<tr>
<td>rkwc</td>
<td>RK word count register</td>
</tr>
<tr>
<td>rootdev</td>
<td>device of root</td>
</tr>
<tr>
<td>rootdir</td>
<td>pointer to the inode of the root directory</td>
</tr>
<tr>
<td>rrkbuf</td>
<td>buffer of RK</td>
</tr>
<tr>
<td>runin</td>
<td>scheduling flag</td>
</tr>
<tr>
<td>runout</td>
<td>scheduling flag</td>
</tr>
<tr>
<td>runrun</td>
<td>scheduling flag</td>
</tr>
<tr>
<td>swapdev</td>
<td>swap device</td>
</tr>
<tr>
<td>swapmap</td>
<td>space for swap allocation</td>
</tr>
<tr>
<td>swbuf</td>
<td>swap buffer</td>
</tr>
<tr>
<td>time</td>
<td>time in seconds from 1970</td>
</tr>
<tr>
<td>tout</td>
<td>time of day of next sleep</td>
</tr>
<tr>
<td>tttbuf</td>
<td>transmit register for terminal</td>
</tr>
<tr>
<td>tttcsr</td>
<td>transmit control status register for terminal</td>
</tr>
<tr>
<td>uisa</td>
<td>user instruction space address register</td>
</tr>
<tr>
<td>uisd</td>
<td>user instruction space descriptor register</td>
</tr>
</tbody>
</table>
### APPENDIX C. GLOBAL FLOWS FOR UNIX MODULES

<table>
<thead>
<tr>
<th>Module</th>
<th>Global flows</th>
<th>Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>callout</td>
<td>1</td>
<td>0,1,1</td>
</tr>
<tr>
<td>canomb</td>
<td>0</td>
<td>0,0,1</td>
</tr>
<tr>
<td>cfree</td>
<td>0</td>
<td>0,0,1</td>
</tr>
<tr>
<td>cfreelist</td>
<td>0</td>
<td>0,0,1</td>
</tr>
<tr>
<td>coremap</td>
<td>15</td>
<td>7,1,1</td>
</tr>
<tr>
<td>curpri</td>
<td>2</td>
<td>2,1,0</td>
</tr>
<tr>
<td>execnt</td>
<td>0</td>
<td>0,0,1</td>
</tr>
<tr>
<td>ipc</td>
<td>2</td>
<td>0,0,2</td>
</tr>
<tr>
<td>kltbuf</td>
<td>0</td>
<td>0,1,0</td>
</tr>
<tr>
<td>kltcsr</td>
<td>0</td>
<td>0,2,0</td>
</tr>
<tr>
<td>lbolt</td>
<td>0</td>
<td>0,0,1</td>
</tr>
<tr>
<td>lpbuf</td>
<td>0</td>
<td>0,1,0</td>
</tr>
<tr>
<td>lpsr</td>
<td>1</td>
<td>1,0,1</td>
</tr>
<tr>
<td>maplock</td>
<td>1</td>
<td>0,1,1</td>
</tr>
<tr>
<td>maxmem</td>
<td>0</td>
<td>0,0,1</td>
</tr>
<tr>
<td>rkba</td>
<td>0</td>
<td>0,1,0</td>
</tr>
<tr>
<td>rkcs</td>
<td>1</td>
<td>0,1,1</td>
</tr>
<tr>
<td>rkda</td>
<td>1</td>
<td>1,1,0</td>
</tr>
<tr>
<td>rktab</td>
<td>6</td>
<td>0,2,0</td>
</tr>
<tr>
<td>rkwc</td>
<td>0</td>
<td>0,1,0</td>
</tr>
<tr>
<td>rootdev</td>
<td>1</td>
<td>1,0,1</td>
</tr>
<tr>
<td>rootdir</td>
<td>1</td>
<td>1,1,0</td>
</tr>
<tr>
<td>rrkbuf</td>
<td>0</td>
<td>2,0,0</td>
</tr>
<tr>
<td>runin</td>
<td>4</td>
<td>0,1,2</td>
</tr>
<tr>
<td>runout</td>
<td>6</td>
<td>0,0,3</td>
</tr>
<tr>
<td>runrun</td>
<td>0</td>
<td>0,4,0</td>
</tr>
<tr>
<td>swapdev</td>
<td>0</td>
<td>3,0,0</td>
</tr>
<tr>
<td>swapmap</td>
<td>13</td>
<td>6,1,1</td>
</tr>
<tr>
<td>swbuf</td>
<td>0</td>
<td>0,0,1</td>
</tr>
<tr>
<td>time</td>
<td>8</td>
<td>2,2,1</td>
</tr>
<tr>
<td>tout</td>
<td>2</td>
<td>2,1,0</td>
</tr>
<tr>
<td>tttbuf</td>
<td>0</td>
<td>0,1,0</td>
</tr>
<tr>
<td>tttcsr</td>
<td>0</td>
<td>1,0,0</td>
</tr>
<tr>
<td>uisa</td>
<td>5</td>
<td>1,2,1</td>
</tr>
<tr>
<td>uisd</td>
<td>5</td>
<td>1,2,1</td>
</tr>
</tbody>
</table>
### APPENDIX D. UNIX MODULE COMPLEXITIES

<table>
<thead>
<tr>
<th>Module</th>
<th>Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>callout</td>
<td>clock</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>timeout</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>548</td>
</tr>
<tr>
<td>canonb</td>
<td>canon</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>cfree</td>
<td>cinit</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>cfreelist</td>
<td>cinit</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>coremap</td>
<td>exit</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>expand</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>main</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>malloc</td>
<td>9261</td>
</tr>
<tr>
<td></td>
<td>mfree</td>
<td>2592</td>
</tr>
<tr>
<td></td>
<td>newproc</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>sched</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>xccdec</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>xswap</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12255</td>
</tr>
<tr>
<td>curpri</td>
<td>setpri</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>setrun</td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>swtch</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1619</td>
</tr>
<tr>
<td>execnt</td>
<td>exec</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td></td>
<td>180</td>
</tr>
<tr>
<td>ipc</td>
<td>procxmt</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>ptrace</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>
APPENDIX D (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Line</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>kltbuf</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>klrnt</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>kltcsr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>klopen</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>klrnt</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>lbolt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>clock</td>
<td>107</td>
<td>107</td>
</tr>
<tr>
<td>lpbuf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lpstart</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>lpsr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lpopen</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>lpstart</td>
<td>28</td>
<td>39</td>
</tr>
<tr>
<td>maplock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mapalloc</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>mapfree</td>
<td>9</td>
<td>31</td>
</tr>
<tr>
<td>maxmem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>rkba</td>
<td></td>
<td></td>
</tr>
<tr>
<td>devstart</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>rkcs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>devstart</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>rkintr</td>
<td>23</td>
<td>45</td>
</tr>
<tr>
<td>rkda</td>
<td></td>
<td></td>
</tr>
<tr>
<td>devstart</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>rkstart</td>
<td>10</td>
<td>32</td>
</tr>
</tbody>
</table>
APPENDIX D (Continued)

rktab
   binit  30
   rkintr  23
   rkstart  10
   rkstrategy  29

rkwc
   devstart  22

rootdev
   iinit  19
   main  348

rootdir
   main  87
   namei  155

rrkbuf
   rkread  4
   rkwrite  4

runin
   clock  107
   sched  114
   sleep  40

runrun
   clock  963
   setpri  48
   setrun  1500
   swtch  71

swapdev
   exit  43
   swap  650
   wait  50

   743
APPENDIX D (Continued)

swapmap
exit 43
malloc 9261
mfree 2592
sched 114
wait 50
xalloc 102
xfree 19
xswap 24
12205

swbuf
swap 650
650

time
clock 107
gtime 5
iinit 171
sslep 20
stime 8
311

tout
clock 107
sslep 20
stime 8
135

tttbuf

ttstart 2400
2400

tttcsr
ttstart 24
24

uisa
main 87
mmread 24
mmwrite 31
sureg 26
168
APPENDIX D (Continued)

uisd
main  87
mmread  24
mmwrite  31
sureg  26
      168

u
access  56000
alloc  15876
bmap  74780
chdir  1344
chmod  13
clock  1712
close  40
core  57600
cpass  19
dup  432
estabur  250880
exec  162000
exit  688
expand  7424
falloc  20
fork  38
fstat  36
geterror  90
gelf  98784
getmdev  256
getpid  4
getswit  4
getuid  5
grow  3312
gtime  5
ialloc  31680
iget  100548
iomove  4752
issig  18
kill  25
klopen  30
link  1368
lpopen  11
main  1392
maknode  960
mknod  88
mmread  24
mmwrite  124
APPENDIX D (Continued)

namei 13857155
newproc 855
nodev 4
nosys 4
open 176
openl 14800
openi 972
owner 1134
passc 99
physio 2520
pipe 31
procxmt 86
profil 9
psig 624
ptrace 136
rdwr 16704
readi 52675
readp 1512
rexit 5
savu 7
sbreak 1800
seek 688
setgid 9
sgtty 464
sleep 190440
smount 848
ssig 14
stop 17
stty 10
sumount 120
sureg 26
suser 200
times 9
trap 2112
trapl 10
ttystty 80
uchar 9
ufalloc 44
unlink 1664
wait 1800
wdir 1215
writei 110446
writep 4212
xalloc 33048
xfree 19

15170894
APPENDIX E. REDESIGN OF THE ITRUNC PROCEDURE

Previous Code for ITRUNC

itrunc(ip)
int *ip;
{
    register *rp, *bp, cp;
    int *dp, ep;

    rp = ip;
    if((rp->i_mode&(IFCHR&IFBLK)) != 0)
        return;
    for(ip = &rp->i_addr[7]; ip >= &rp->i_addr[0]; ip--)
    if(*ip {
        if((rp->i_mode&ILARG) != 0) {
            bp = bread(rp->i_dev, *ip);
            for(cp = bp->b_addr+512; cp >= bp->b_addr; cp--)
            if(*cp) {
                if(ip == &rp->i_addr[7]) {
                    dp = bread(rp->i_dev, *cp);
                    for(ep = dp->b_addr+512;
                        ep >= dp->b_addr; ep--)
                        if(*ep)
                            free(rp->i_dev, *ep);
                    brelse(dp);
                }
                free(rp->i_dev, *cp);
            }
            brelse(bp);
        }
        free(rp->i_dev, *ip);
        *ip = 0;
    }

    rp->i_mode &= ~ILARG;
    rp->i_size0 = 0;
    rp->i_size1 = 0;
    rp->i_flag |= IUFD;
APPENDIX E (Continued)

Revised Code for ITRUNC

```c
itrunc(ip)
int *ip;
{
    register *rp, *bp, *cp;
    int *dp, *ep;
    int bufl[256], buf2[256], bp1, bp2;
    /* arrays added to hold contents of blocks from itrunc */
    bp1 = &bufl;
    bp2 = &buf2;
    rp = ip;
    if((rp->i_mode&(IFCHR&IFBLK)) != 0)
        return;
    for(ip = &rp->i_addr[7]; ip >= &rp->i_addr[0]; ip--)
        if(*ip) {
            if ((rp->i_inode*ILARG) != 0)
                bp = itrunc (rp->i_dev,*ip,*bpl);
                for (cp = 255; cp >=0; cp--)
                    if (buf1[cp]) {
                        if (ip == &rp->i_addr[7]) {
                            dp = itrunc (rp->i_dev,*bufl[sp],*bp2);
                            for (ep = 255; ep >=0; ep--);
                            if (buf2[ep])
                                free(rp->i_dev,buf2[ep]);
                            brelse(dp);
                        }
                        free(rp->i_dev, buf[cp]);
                    }
                free(rp->i_dev, buf[cp]);
            brelse(bp);
        }
    free(rp->i_mode, *ip);
    *ip = 0;
}
rp->i_mode = &~ILARG;
rp->i_size0 = 0;
rp->i_size1 = 0;
rp->i_flag |= IUPD;
```
APPENDIX E (Continued)

Code for ITRUNC

itrunc (dev,blk,p)
int dev,blk,*p;
{
    int *bp,i,*cp;
    bp = bread (dev,blk);
    cp = bp->b_addr:
    for (i = 0; i < 256; i++)
        *p++ = *cp++;
    return (bp);
}
APPENDIX F. REDESIGN OF THE NAMEI PROCEDURE

Original "C" Code for NAMEI

```c
namei(func, flag)
int (*func)();
{
    register struct inode *dp;
    register c;
    register char *cp;
    int eo, *bp;

    /*
    * If name starts with '/ ' start from
    * root; otherwise start from current dir.
    */
    dp = u.u_cdir;
    if((c=(*func)()) == '/ ')
        dp = rootdir;
    iget(dp->i_dev, dp->i_number);
    while(c == '/ ')
        c = (*func)();
    if(c == '0' && flag != 0) {
        u.u_error = ENOENT;
        goto out;
    }

cloop:
    /*
    * Here dp contains pointer
    * to last component matched.
    */
    if(u.u_error)
        goto out;
    if(c == '0 ')
        return(dp);
    /*
    * If there is another component.
    * dp must be a directory and
    * must have x permission.
    */
    if((dp->i_mode&IFMT) ! = IFDIR) {
        u.u_error = ENOTDIR;
        goto out;
    }
}```
if (access(dp, IEXEC))
    goto out;

/*
 * Gather up name into
 * users' dir buffer.
 */

cp = &u.u_dbuf[0];
while (c != '/' && c != '0' && u.u_error == 0) {
    if (cp < &u.u_dbuf[DIRSZ])
        *cp++ = c;
    c = (*func)();
}
while (cp < &u.u_dbuf[DIRSZ])
    *cp++ = '0';
while (c == '/')
    c = (*func)();
if (u.u_error)
    goto out;

/*
 * Set up to search a directory.
 */

u.u_offset[1] = 0;
u.u_offset[0] = 0;
u.u_segflg = 1;
eo = 0;
u.u_count = ldiv(dp-i_sizel, DIRSZ+2);
bp = NULL;
eoloop:

/*
 * If at the end of the directory,
 * the search failed. Report what
 * is appropriate as per flag.
 */

if (u.u_count == 0) {
    if (bp != NULL)
        brelse(bp);
    if (flag == 1 && c == '0') {
        if (access(dp, IWRITE))
            goto out;
        u.u_pdir = dp;
    }
}
APPENDIX F (Continued)

if(eo)
  u.u_offset[1] = eo-DIRSZ-2; else
  dp->i_flag = IUPD;
return(NULL);
}

u.u_error = ENOENT;
goto out;

/*
 * If offset is on a block boundary,
 * read the next directory block.
 * Release previous if it exists.
 */

if((u.u_offset[1]&0777) == 0) {
  if(bp != NULL)
    brelse(bp);
  bp = bread(dp->i_dev,
             bmap(dp, ldiv(u.u_offset[1], 512)));
}

/*
 * Note first empty directory slot
 * in eo for possible creat.
 * String compare the directory entry
 * and the current component.
 * If they do not match, go back to eloop.
 */
bcopy(bp->b_addr+(u.u_offset[1]&0777), &u.u_dent, (DIRSZ+2)/2);
  u.u_offset[1] += DIRSZ+2;
  u.u_count--;  
if(u.u_dent.u_ino == 0) {
  if(eo == 0)  
    eo = u.u_offset[1];
  goto eloop;
}
for(cp = &u.u_dbuf[0]; cp < &u.u_dbuf[DIRSZ]; cp++)
  if(*cp != cp[u.u_dent.u_name - u.u_dbuf])
    goto eloop;

/*
 * Here a component matched in a directory.
 * If there is more pathname, go back to
 * eloop, otherwise return.
 */
APPENDIX F (Continued)

    if(bp != NULL)
        breise(bp);
    if(flag==2 && c=='o') {
        if(access(dp, IWRITE))
            goto out;
        return(dp);
    }
    bp = dp->i_dev;
    iput(dp);
    dp = iget(bp, u.u_dent.u_ino);
    if(dp == NULL)
        return(NULL);
    goto cloop;

out:
    iput(dp);
    return(NULL);

Revised Code for NAMEI

namei(func, flag)
int (*func)();
{
    register struct inode *dp;
    register *ip;
    int x;
    register c;
    register char *cp;
    int eo, *bp;

    /*
     * If name starts with '/' start from
     * root; otherwise start from current dir.
     */
    dp = u.u_cdir;
    if((c=(*func)()) == '/')
        dp = rootdir;
    iget(dp->i_dev, dp->i_number);
    while (c == '/')
        c = (*func)();
    if(c == '0' && flag != 0) {
        u.u_error = ENOENT;
        goto out;
    }
APPENDIX F (Continued)

    ip = namei' (func, flag, &x, c, dp);
    if (x)
        goto out;
    return (ip);

out:
    iput(dp);
    return(NULL);
}

New Code for NAMEI'

namei' (func, flag, x, c, dp)
int (*func) ();
int *x;
register *ip;
register struct inode *dp;
register c;
{
    register char *cp;
    int eo, *bp,y;
    *x = 0;

cloop:
    /*
     * Here dp contains pointer
     * to last component matched.
     */

    if(u.u_error) {
        *x = 1;
        return;
    }
    if(c == '0')
        return(dp);

    /*
     * If there is another component,
     * dp must be a directory and
     * must have x permission.
     */

    if((dp->i_mode&IFMT) != IFDIR) {
        u.u_error = ENOTDIR;
        *x = 1;
        return;
    }
APPENDIX F (Continued)

if (access(dp, IEXEC)) {
    *x = 1;
    return;
}

/*
 * Gather up name into
 * users' dir buffer.
 */

cp = &u.u_dbuf[0];
while (c != '/' && c != '0' && u.u_error == 0) {
    if (cp < &u.u_dbuf[DIRSIZ])
        *cp++ = c;
    c = (*func)();
}
while (cp < &u.u_dbuf[DIRSIZ])
    *cp++ = '0';
while (c == '/')
    c = (*func)();
if (u.u_error)
    *x = 1;
    return;
}

/*
 * Set up to search a directory.
 */

u.u_offset[1] = 0;
u.u_offset[0] = 0;
u.u_segflg = 1;
eo = 0;
u.u_count = ldiv(dp->i_sizl, DIRSIZ+2);
bp = NULL;

ip = namei"(bp, flag, dp, eo, &y);
if (y == 1) {
    *x = 1;
    return;
}
if (y == 2)
    return (ip);

/*
 * Here a component matched in a directory.
 * If there is more pathname, go back to
 * clloop, otherwise return.
 */
APPENDIX F (Continued)

if (bp != NULL)
    brelse(bp);
if (flag==2 && c=='0') {
    if(access(dp, IWRITE)) {
        *x = 1;
        return;
    }
    return(dp);
}
bp = dp->i_dev;
iput(dp);
dp = iget(bp, u.u_dent.u_ino);
if(dp == NULL)
    return(NULL);
goto cloop;

New Code for NAME''

dnamei'' (bp, flag, dp, eo, y)
register struct inode *dp;
int *bp, eo;
{    *y = 0;
    eloop:

    /*
     * If at the end of the directory, 
     * the search failed. Report what 
     * is appropriate as per flag. 
     */
    if(u.u_count == 0) {
        if(bp != NULL)
            brelse(bp);
        if(flag==1 && c=='0') {
            if(access(dp, IWRITE)) {
                *y = 1;
                return;
            }
        }
    u.u_pdir = dp;
    if(eo)
        u.u_offset[1] = eo-DIRSIZ-2; else 
    dp->i_flag = IUPD;
    *y = 2;
    return(NULL);
APPENDIX F (Continued)

    }
    u.u_error = ENOENT;
    *y = 1;
    return;
}

/*
 * If offset is on a block boundary,
 * read the next directory block.
 * Release previous if it exists.
 */
if((u.u_offset[0] & 0777) == 0) {
    if(bp != NULL)
        brelse(bp);
    bp = bread(dp->i_dev,
        bmap(dp, ldiv(u.u_offset[0], 512)));
}

/*
 * Note first empty directory slot
 * in eo for possible creat.
 * String compare the directory entry
 * and the current component.
 * If they do not match, go back to eloop.
 */

bcopy(bp->b_addr+(u.u_offset[0] & 0777), &u.u_dent, (DIRSIZ+2)/2);
u.u_offset[0] += DIRSIZ+2;
u.u_count--;
if(u.u_dent.u_ino == 0) {
    if(eo == 0)
        eo = u.u_offset[0];
    goto eloop;
}
for(cp = &u.u_dbuf[0]; cp < &u.u_dbuf[DIRSIZ]; cp++)
    if(*cp != cp[u.u_dent.u_name - u.u_dbuf])
        goto eloop;