INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9207120

An application of debugging theory to program modification for the partial automation of computational perfective maintenance

Salb, David, Ph.D.

City University of New York, 1991

Copyright ©1991 by Salb, David. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

AN APPLICATION OF DEBUGGING THEORY TO PROGRAM MODIFICATION FOR THE PARTIAL AUTOMATION OF COMPUTATIONAL PERFECTIVE MAINTENANCE

by

DAVID SALB

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1991

© 1991

DAVID SALB

All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

June 19, 1991
Date

Chair of Examining Committee

June 19, 1991
Date

Executive Officer

Dr. Michael Anshel (CCNY)

Dr. Linda Friedman (Baruch)

Dr. Howard Wasserman (Queens)

Supervisory Committee

The City University of New York

Abstract

AN APPLICATION OF DEBUGGING THEORY TO PROGRAM MODIFICATION FOR THE PARTIAL AUTOMATION OF COMPUTATIONAL PERFECTIVE MAINTENANCE

by

David Salb

Adviser: Professor Howard Rubin

Program maintenance is done to correct errors in program design and coding and is called corrective maintenance. When it is also done to improve a program's function as requirements change within an environment it is called perfective maintenance.

This paper addresses Weiser's work on program slicing (decomposing a program into the smallest <u>executable</u> portion which contains the variable that is to be changed). It also considers the work of Korel in program debugging using execution traces to assist a programmer in correcting an erroneous program. It discusses some of the work done in the maintenance area concerning program change and applies some of those concepts to perfective maintenance. We propose, here, some of the foundations necessary for the design of a tool for simple computational perfective maintenance. We suggest some heuristics that may be applied and show how, in general, a semi-automated system would endeavor to effect a change

in an existing program. We introduce the concept *shadow of influence* and show how the automated system should minimize its impact when a modification is done. This work should be of interest to researchers in the area of program enhancement, extension and transformation. Expansion of this work could develop a useful tool to assist in the modification of existing programs without direct programmer intervention.

Acknowledgements

Many people have assisted me in various stages of my graduate studies. First and foremost I would like to thank my mentor, Professor Howard Rubin, for suggesting the problem for my thesis and introducing me to both academicians and business professionals in the field. He also directed my attention to some very useful sources. Despite his many activities he always responded quickly to my needs and gave generously of his precious time to meet with me to review my progress and direct me in my research. I am very grateful to him for this.

Professor Michael Anshel deserves my gratitude for his encouragement and helpful intervention during a very trying time in my graduate career. His assistance in some aspects of the research of this thesis is very much appreciated.

Professor Linda Friedman assisted in my research by directing me to some very useful research tools. She provided me with highly worthwhile source material, and perhaps even more important, she gave of her valuable time to review my work and guide me in my research and writing.

Professor Stewart Weiss also directed me to some useful sources.

Table of Contents

1.0 Introduction 1

1.1 The Maintenance Problem 8

2.0 Background 22

2.1 The User-System Interface 37

2.2 The Systems Functions 40

2.3 Locating the proper line of code to be changed 42

2.4 Implementing the change 45

3.0 An Algorithm for Interface and Change . . . 51

4.0 Future Directions 60

Appendix 63

A1 Sample Session 63

A2 Screen Displays for Sample Session 67

A3 Program Slice 74

A4 Shadow of Influence 78

A5 Program Sample Flow Trace 82

A6 System Configuration for Program Modification 83

A7 Icons 84

A8 COBOL BNF Grammar Production Rules 85

A9 Bibliography 89

Figures Directory

Fig 1. Ashcroft & Manna's unstructured program 11

Fig 2. Ashcroft & Manna's resulting program . 11

Fig 3. An interchange routine 17

Fig 4. A formal proof for the interchange routine 17

Fig 5. Original program to be sliced 26

Fig 6. Slice on nw: Word Counter 26

Fig 7. Sample program 32

Fig 8. Program Dependence Sub-network for above program 32

Fig 9. Screen 1 67

Fig 10. Screen 1A 67

Fig 11. Screen 2 68

Fig 12. Screen 2A 68

Fig 13. Screen 2B 69

Fig 14. Screen 2C 69

Fig 15. Screen 3 70

Fig 16. Screen 3A 70

Fig 17. Screen 4 71

Fig 18. Screen 5 71

Fig 19. Screen 5A 72

Fig 20. Screen 5B 72

Fig 21. Screen 5C 72

Fig 22. Screen 6 73

Icons

Icon 1. Perform Modification as Specified . . 84

Icon 2. Map Program Flow-graph 84

Icon 3. Display Program Code 84

Icon 4. Locate Change Point 84

Icon 5. Display the Shadow of Influence of the Variable 84

1.0 Introduction

"Software Engineering is the means by which we attempt to produce all of this software in a way that is both cost-effective and reliable enough to deserve our trust...[It is] the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them" [14]. Unfortunately, these principles have not been applied to many software systems that are in actual use today. As a result, many programs written at that time and still used today are highly unstructured and difficult to understand. As a business's conditions change, a portion of its software needs to be modified (maintained) to meet its new requirements (called <u>perfective maintenance</u> [60]). The options a business manager has in this situation are few. He can have his programming team redesign the application program from the very beginning, this time, (hopefully) using a more responsible approach to program (and system) design and structure (and avoid a repetition of the problem in the future). Or, he can assign one or more individuals to study the current maze of programs and interfaces that constitute the application system, attempt to make some sense out of it, locate the place

within the program (or programs) to make the change, and hope that no other portion of code within the tangled web of instructions is impacted by the adjustment. With either alternative a long and tedious process can be expected before a successful modification is implemented.

Ever since Dijkstra's celebrated letter [33] about the evils of unstructured programming there has been a flurry of activity in trying to design and code programs in a structured way. Research has been done to define, measure and correct program complexity and unstructuredness. Early studies have considered structured programming to be those programs that contain no GOTO statements in them. It was felt that "the quality of ... programmers was inversely proportional to the density of GOTO statements in their programs". [34] Others supported the cause against the use of GOTO statements by arguing that they should be eliminated from programming languages entirely. Although not all GOTOs are bad, the legitimate uses for such a construct are rare indeed. Two arguments for the use of GOTOs are convenience and efficiency. As far as convenience is concerned, Wulf [106] argues that GOTO-less programming is something that needs adapting to and, once mastered, is equally as convenient as standard programming. With regard to efficiency, "More computing sins are committed

in the name of efficiency (without necessarily achieving it) than any other single reason - including blind stupidity". [106] The dangers of and damage caused by GOTOs are far greater than the benefits of their use. There are ample alternate constructs to replace the loss of GOTO. In fact, at least one language (BLISS) has been developed which has no GOTO verb at all[106]!

The academic community responded to Wulf's attack of the GOTO with two papers, one by Hopkins [47] the other by Knuth [54], arguing that there is some merit to the GOTO statement. Its application, however, must be done carefully to avoid the "rat's nest of control flow problems" [106]. Both Hopkins and Knuth argue that efficiency is an important consideration in program construction albeit in a small percentage of a program's code. Structured code may be twenty to thirty percent less efficient than code in which GOTOs were used. Kernighan concludes that "good programming is not synonymous with GOTO-less programming and it certainly does not have to be wasteful of time and space" [51].

Parnas [78] recognized that structured programs are developed from a structured design. Program style is enhanced by modularizing its components into logical chunks allowing each module to contain only that code that would be necessary to accomplish the particular

"function." He introduced the concept of "information hiding" and explained how modules should be designed to avoid the "excessive coupling between modules"[98]. Stevens [98] expressed several methods to improve module construction. The first and foremost requirement is that a module be designed as simply as possible. Each module should be as separate as possible from the next so as to avoid the excessive coupling mentioned earlier. Second, a module should be observable - one should be capable of easily perceiving why and how actions occur. In addition, the modules should be constructed so that a minimal amount of information is passed from one to the other.

The ultimate goal is to design and code programs that are simple to understand and easy to maintain so that when changes are required they can be done quickly and efficiently. To maximize these goals we need a way to quantify and measure program understandability and program maintainability. In this way we can provide some assistance to a business manager in his decision to redesign a system or modify the existing one. The more difficult a program is to understand the more costly a modification solution may be.

The field of software measurement has helped determine which programs are best suited for modification as opposed to complete rewrites. Software complexity

measures have taken several forms. Various metrics have been proposed to quantify program understandability. Shneiderman [91] for example, takes a general approach and considers a "well written" program to be one that a competent programmer (akin to the reasonable man in economics) would be able to reconstruct functionally 90% of the program after 10 minutes of study. McCabe [67] describes a graph-theoretic complexity measure. He assumes that complexity is not based on size but rather on the decision structure of the program. The cyclomatic number, V(G), measures the number of linear independent paths in a program. For structured programs,

$$V(G) = \text{\# of compares} + 1.$$

An automated tool, FLOW, was developed to analyze FORTRAN programs and to compute the cyclomatic complexity. If a module exceeded the upper bound assigned for cyclomatic complexity (about 10), the module was sent back to the programmer for re-coding.

McClure [69] advanced another metric. It is similar to McCabe in that it defines program complexity on the basis of compares, but it also includes a count of the number of variables used in the compares. Thus the

complexity of a module,

$C(m) = C + V$

(ie the number of compares in the module plus the number of control variables referenced in that module).

Halstead [38], chose to analyze program complexity using concepts from thermodynamics. He used four basic measures:

n_1 = the number of distinct operators in a program

n_2 = the number of distinct operands in a program

N_1 = the total number of occurrences of the operators

N_2 = the total number of occurrences of operands

$n = n_1 + n_2$ is the size of the vocabulary,

$N = N_1 + N_2$ is the length of the program

All these numbers are easily obtainable from a compilation. The estimated program length (assuming a well structured program) is:

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

Naturally, the longer the program the more difficult it is to understand. Also, a large disparity between N and $\hat{N}$ indicates a highly unstructured program. The volume of an implementation, i.e. the number of bits necessary to specify a program, is defined as:

$$V = N \log_2 n$$

Halstead hypothesized a conservation law between the level of abstraction, L, and the volume,

V: - LV = constant.

A program translated from a higher level language to a lower level one would have its L decrease while its V would increase since more operators are required for the implementation.

Thus L is defined as:

V^*/V where V^* is the most compact representation possible for the given algorithm.

Programming difficulty increases as volume increases and decreases as level increases. The effort, E, necessary to create a program is V/L. Finally, programming time, T, is E/S where S is a constant representing the speed of the programmer - i.e. "the number of mental discriminations per second of which he is capable" [38].

Other measures of program complexity include: lines of code - the longer the program the more complex it is considered, and Yau & Collofello's [108] Logical Stability Metric which measures a module's (and by extension, a program's) resistance to the potential ripple effect - an undesired change in a program output caused by modifying a portion of code.

1.1 The Maintenance Problem.

Software maintenance is a difficult issue. Martin & McClure [65] define maintenance as the changes in software after it has been delivered to the user. They explain that maintenance is done to correct errors, improve design, adapt to different hardware-software requirements, interface a program to others, adapt to a change in files, or enhance the current application.

Most maintenance requirements are a result of changing requirements rather than reliability problems [61]. It has been found that 42% of maintenance is in the area of perfective maintenance [60]. Nearly 70% of this is for new reports and adding data to existing reports, 20% for reformatting and consolidation or condensing of reports, and 10% for "other." Of the person-hours spent on maintenance, 10% is for reformatting a report and up to 27% is to add data to an existing report [60]. It accounts for approximately 75% of the cost of the life of the software. This percentage is expected to increase due to increased time that the software is used [65]. A Department of Defense study determined that the cost of some Air Force avionics software amounted to $75/instruction and the maintenance cost at one point went as high as $4000/instruction [31].

Many maintenance tasks are on the Junior Programmer level (eg. expanding or modifying reports, changing a computation, etc.). A basic problem is that inexperienced programmers are doing the maintenance and thus errors are compounded. In addition many programmers are reluctant to do this type of work since it is held in such low regard [19].

Approximately 75 to 80 percent of existing software was coded before structured programming methods were put into use [88]. These programs are complex and their intent is difficult to grasp. They were not designed to be efficiently maintained and thus any changes to the code can result in disastrous "side effects." Many of these programs still perform a useful function for us. They were created with a great deal of time, effort, and money. It has been estimated that "the amount of money spent on software in the US grows approximately 12 percent each year, and the demand for added software functions grows even faster." [48] The software development budget for the US Air Force's F-16 jet-fighter is $85-million with an additional $250-million expected to be spent on maintenance costs [22]. The value of software at this point must be very high. To discard this software and create new structured programs that accomplishes the same function is a very

expensive proposition. These programs must first be "understood" before one can attempt to duplicate their behavior. Many existing programs are selected as maintenance candidates because they are highly successful in satisfying user needs and requirements. These programs are chosen to be modified to adapt to new criteria and specifications in the changing "business world." The users view these change requests as mere extensions of the program function [17].

A piece of software is considered maintainable if it is testable, understandable, and modifiable [15]. Testable refers to the ease in which one can demonstrate that the changes in the software result in the desired outcome. Understandability is determined by the amount of difficulty required in comprehending the program. Written specifications, flow charts, and other documentation increase the understandability of a program. Modifiable refers to the ease in which the actual code can be modified. Some issues connected with modifiability are the modulariztion of instructions and the extent to which modules are coupled.

The task at hand is to somehow transform otherwise usable unstructured or weakly structured programs that have a low maintainability factor into ones that are structured and thus can be more easily understood and

maintained. Bohm & Jacopini [16] provided the theoretical basis for *transforming* an unstructured program into a structured one. They show that most unstructured program segments can be rewritten into portions that are structured in nature. Their proof assumes use of flow diagrams and Turing Machines with only two formation rules alone. Thus any language that could be converted into that form could use the same technique. Ashcroft and Manna [5] extended Bohm & Jacopini's results to show that transformations to structured programs is not only theoretically possible but practically possible. They provide an algorithm for accomplishing the translation of "unstructured" flowchart programs into structured while programs. To accomplish this they required the introduction of boolean variables [5]. For example, see figure 1 for their unstructured program (as adapted from the flow chart presented in the paper). After the application of their algorithm their structured program (as appears in their paper) was as in figure 2. Note that they required the addition of a boolean variable to control the looping process.

Software restructuring is a technique which transforms an existing program into one that is more readable, understandable, and modifiable. Examples of

```
    start  (x);
    x := a(x);
L1: if p(x) then  x := e(x);
          else  goto L2;
    goto L1;
L2: if q(x) then  x := b(x);
            else  goto L4;
L3: if r(x) then  x := d(x);
            else  goto L5;
    goto L3;
L4: x := g(x);
    goto L7;
L5: if s(x) then  x := c(x);
            else  goto L6;
    goto L1;
L6: x := f(x)
L7: halt  (x);
```

Fig 1. Ashcroft & Manna's unstructured program

```
    start (x̄);
    x̄ := a(x̄);
    t := true;
    while t do
        [while p(x̄) do x̄ := e(x̄);
        if q(x̄) then [x̄ := b(x̄);
             while r(x̄) do x̄ := d(x̄);
             if s(x̄) then x̄:= c(x̄)
                  else [x̄ := f(x̄); t := false]]
             else [x̄ := g(x̄); t := false]];
    halt (x̄);
```

Fig 2. Ashcroft & Manna's resulting program

restructuring techniques applied to programs include:

a. pretty printing (to space and align lines of code in a program so that the logical relationships

and control flow among the program statements are more easily discernable and comprehensible).

b. manual restructuring to adjust lines of code to conform to a pre-established standard.

c. editing documentation for readability and comprehension.

d. creating indices for both program variables and software documentation topics as they relate to the program specifically as well as the system as a whole.

e. reusable code insertion in select modules of the program.

f. adjustment of the physical order of the program instructions by automated means (restructuring engines). Many of these engines rewrite the code using structured coding techniques and constructs including the removal of all GOTO statements within the program, using the CASE structure as opposed to a series of IF-THEN-ELSE statements, and DO and WHILE loop controls.

g. translate the code (using any of the many code translators on the market) into another language and then retranslate it back (again using a commercial translator) to its original language. The expectation is that the retranslation will

avoid much of the unstructured coding problem in favor of structured code.

The need for restructuring programs becomes most apparent when modification requests are presented. A user is asked for a change to an existing system so that changes in his environment or "way of doing business" could be accommodated. This request requires that the program that handled the particular task be adjusted. To accomplish the change, a programmer would then have to study the program, understand it and then make the change in such a way as to avoid introducing undesired changes to the function of the program. As more of the changes or "fixes" are applied to the code, the program would become less "structured" and more difficult to understand [4].

To automate the restructuring process Baker [9] and DeBalbine [29], using [5],[16] built restructuring engines for FORTRAN. Applying similar techniques others have built restructuring engines for COBOL [18],[4], [42], [72] to transform a program into a form that is easier to read and thus easier to understand. For the COBOL engine some additional transformation work was necessary since it was discovered that some of the unstructured programs took advantage of "gaps" in the

compiler code and did not conform to the syntax rules established for COBOL [72]. The resultant program from these engines simplified the maintainer's modification strategy. An additional consequence of the structuring engine is the standardization of style. Many restructurers pretty print the output by indenting the code to emphasize the control structure and organization of the program. Any new variables that may be added to enhance the structure of the program would of course be standardized. McClure, in a survey of programmers, reports that standardizing programs has made programs more understandable [65].

> "IBM reported an average an average of 40% productivity savings in real-time, business application, and system application software projects employing software structured techniques. ... Other organizations reported that maintenance costs for software developed with structured techniques are reduced by a ratio of 3:1 compared to maintenance costs for "unstructured" software. Error rates in tested unstructured software average one error per 200 lines of source code. But in many structured software systems, production error rates are averaging less than one error per 1000 lines of source code [65].

Tom Gilb, who in one of the skeptics of the reported gains in using structured programming, nevertheless reports that one advantage of structured code is that it reduces the logical test paths in a program. He states that "in a 21 statement subroutine the difference can be 144 test cases to 90,000,000,000" [40].

Reverse engineering involves the process of developing a set of specifications for a system by examining its components in a methodical manner without the benefit of the designers plans and drawings. The intention of the "engineer" is to be able to understand the system sufficiently so that, if necessary, he can reconstruct it. For software system modification the comprehension component is most vital since without it no changes could be *properly* installed [21]. Reverse engineering is most useful when applied to unstructured programs requiring change. Reverse engineering an unstructured program produces a specification for the program as output. The resultant specification can then be more easily analyzed and thus allow the program to be more easily modified.

Program verification has been one area investigated to assist in the development (and maintenance - see [32]) of programs. Programmers prove the correctness of a program using sound mathematical logic. The expectation

is that since we understand the foundations of the mathematical proof so well (it has been around for centuries) we can readily apply it to the verification of program correctness with much success. Unfortunately, proving a program correct requires a lot of work. For example, *see figure 3 for a program segment to perform a simple swap and figure 4 (both taken from [7]) for a formal proof of correctness of the interchange routine.*

```
{x = x0 and y = y0}
t := x;
x := y;
y := t
{x = y0 and y = x0}
```

Fig 3. An interchange routine

```
1.{t = x0 and x = y0} y:=t {y=x0 and x = y0}

2.{t = x0 and y = y0} x:=y {t = x0 and x = y0}

3.{t = x0 and y = y0}
    x:=y; y:=t
    {y = x0 and x = y0}

4.{x = x0 and y = y0} t:=x {t=x0 and y = y0}

5.{x = x0 and y = y0}
    t:=x; x:=y; y:=t
    {y = x0 and x = y0}
```

Fig 4. A formal proof for the interchange routine

Also, just as in a mathematical proof, our proof may be faulty [17]. The richness of mathematical theorems and proofs is only so because they have withstood the test of time. The proofs that were presented were reviewed by hundreds of people throughout history and verified again and again (some proofs in fact were found to be fallacious[1]). We do not have this luxury with programs and software systems which are generally beyond their deadline date and over budget.

The solution to the maintenance problem appears to be the writing of all program systems in code that can be easily understood and maintained. In addition, clear and accurate documentation of the design, code, and history of the system should be carefully preserved to provide an audit trail of the work done and the rationale behind it. Some current programs are being written in fourth-generation languages (4GL) which facilitate coding and incorporate structured programming standards. For some applications these higher level language programs can be developed faster than their third-generation level counterpart thus saving both time and money in the

[1] A well-known example is the four-color problem (can any map be colored with only four colors?). A. B. Kempe "proved" it to be true and published his result in 1879. Years later, P.J. Heawood pointed out that this proof was *not correct*. [10]

program development cycle. Because of the ease of use and the simple command structure, many non-computer users are designing their own programs utilizing any of the 4GLs on the market. Some systems professionals have developed prototype systems with these high-level languages. Thus the user can get the "feel" of the way the new system will "behave" and modify his initial system request as he obtains a more profound understanding of what his needs are. It is interesting to note that, according to Chapin [88], while these 4GLs are beneficial for the development of new programs, they are more expensive and difficult to maintain [88]. This suggests that, at the current time, 3rd generation languages such as COBOL should be the language of choice for software development project that are expected to be used for an extended period. Naturally, the code should be properly structured and documented with appropriate models to assist the maintainer with future modifications.

Another advance in programming tools is the program generator. The less sophisticated generators allow a programmer to write mnemonics which are then expanded into the full language command, while the more elaborate versions can write nearly an entire program based on detailed specifications [25]. The outcome is a program product that is uniform in both form and style. The

tedium of writing the code is minimized and the program is produced in a more timely manner.

Computer-aided software engineering (CASE) tools have been developed to assist in the full spectrum of the software development cycle. Those tools that address the early part of the cycle (concentrating on the analysis and high-level design of a system) are referred to as upper-CASE while those tools that address the lower end of the development cycle are called lower-CASE products. Case tools allow for the use of formal system development techniques in a practical and economic way [22]. The CASE tool user can exploit the graphics capabilities to design flow charts, entity relationship diagrams and data flow diagrams with ease. Moreover many systems are capable of redrawing and sometimes rearranging these diagrams when changes are made to the original design. Some tools have the ability to check for consistency and completeness of design and notify the user if any deficiencies are detected. A key feature for some tools is the integration of the documentation component in all parts of the design. They are linked to data dictionaries and the like to provide a comprehensive software development management. Thus the tool can serve as a repository for all information needed about the particular system being developed.

While the advancement of these innovations are ideal for new program development, they do not adequately address the large amount of software developed and produced prior to these mechanisms. Management would, in general, be reluctant to discard the large investment in pre-CASE software and would delay redesign of existing software to comply with the standards now in vogue. There is a need to assist programmers in maintaining this software in an efficient and cost-effective manner.

2.0 Background

Existing software must be maintained to adapt it to the changing requirements of the environment. It is difficult, however, to modify (and debug) programs whose code is unstructured and generally complex. A number of techniques have been promoted to guide the programmer through the complicated code structure and assist him in analyzing the program and understanding it.

Program Slicing is a method suggested by Weiser [104], [102] to assist programmers in maintaining programs. The technique involves the decomposition of a program by extracting only those lines of code that would be affected by a change in a variable within a program. Thus, if one wishes to modify a program (e.g., change a variable), Weiser suggests that the programmer "slice" the program - decompose it by selecting those instructions that pertain to the variable to be modified and exclude all other lines of code. The result is an executable program without any unrelated program instructions. A program slice does not contain only those instructions referencing the selected variable alone, but rather the aggregate of the instructions whose variables would be affected due to a change in the selected variable.

The algorithm for slicing, presented in [39] applies to structured as well as unstructured code. Two definitions that are useful in the program slicing paradigm are *defs(n)*, the set of variables assigned a value at statement n (the left hand side of an assignment statement), and *refs(n)*, the set of variables referenced at statement n (the right hand side of an assignment statement). The values of variables in defs(n) after statement n is executed depend on the values of the variables in refs(n) before statement n is executed. In order to compute a slice one must first identify the *active set*, a set of variables associated with each program statement. The active set at statement s are those variables whose values just before execution of statement s might affect the value of a variable v just before the execution of statement n. Initially all active sets are set to null with the exception of the set related to statement n which is set to v. The active set is computed as follows (in this algorithm we are restricting ourselves only to straight-line code - branches, loops, etc. will be discussed shortly):

1. Initialize the active sets to null.
2. Starting at s = n - 1 compose the active set at statement s from the active set at statement

s + 1 until s = 1 (start with the variable being sliced and continue, following the logical flow of statements, toward the beginning of the program). The active set at statement s is the union of: a) the set of all variables active at statement s + 1 but not defined at s and, b)the set of all variables referenced at statement s if the active set at statement s + 1 has a non-empty intersection with defs(s). Thus, all variables that are in some way related to the sliced variable are included in the active set. The slice now includes all statements s such that:

$$\text{defs}(s) \cap (\text{active set at statement } s + 1) \neq \emptyset$$

A *control set*, the set of control statements, handles statements such as if-then and loop by defining a set of statements that are to be included in the slice when any such statement is contained within the slice. Compound statements such as if-then-else are managed by adding the enclosing part of the compound statement to the control set of each statement within the scope of the compound statement. When a statement is added to the slice (as a result of one of its variables being a member of the active set), its control set statements are added as well. The variables at a statement that caused the

control statement to be included and variables referenced at the control statement are annexed to the active set for that statement.

GOTO statements are included if they have non-empty active sets. The target labels of the GOTOs are inserted in the control set as well as the control set of the target labels and so on until the slice consists of all target labels and their associated control sets. Output commands are included if they are associated with any variable within the slice. *For an example of a program slice taken from [39] see figure 5 for the original program and figure 6 for a slice on nw.*

```
1  #define YES 1
2  #define NO 0
3  main()
4  {
5       int c, nl, nw, nc, inword ;
6       inword = NO ;
7       nl = 0;
8       nw = 0;
9       nc = 0;
10      c = getchar();
11      while ( c != EOF) {
12          nc = nc + 1;
13          if ( c == '\n')
14              nl = nl + 1;
15          if ( c== ' ' || c== '\n' || c== '\t')
16              inword = NO;
17          else if (inword == NO) {
18              inword = YES ;
19              nw = nw + 1;
20          }
21          c = getchar();
22      }
23      printf("%d \n",nl);
24      printf("%d \n",nw);
25      printf("%d \n",nc);
26 }
```

Fig 5. Original program to be sliced

Once a program is sliced, and only the pertinent code for a programmer's modification requirements remains, he is in a better position with regard to discovering the effect and consequences of a change to a program. A programmer selects a portion of code to be changed (to be discussed later) and by program slicing isolates the relevant code that would be affected by a modification. By using any of the available debugging tools a

```
1  #define YES 1
2  #define NO 0
3  main()
4  {
5       int c, nl, nw, nc, inword ;
6       inword = NO ;
8       nw = 0;
10      c = getchar();
11      while ( c != EOF) {
15          if ( c== ' ' || c== '\n' || c== '\t')
16              inword = NO;
17          else if (inword == NO) {
18              inword = YES ;
19              nw = nw + 1;
20          }
21          c = getchar();
22      }
24      printf("%d \n",nw);
26 }
```

Fig 6. Slice on nw: Word Counter

programmer can modify a value and, via the automated debugging system, determine the effect of such a change.

Conventional debugging tools allow a user to trace and follow a program execution while recording the various states that occur.
The user can establish "break points" where program execution pauses to allow the user to examine the states and the contents of select variables of interest. These debuggers require awareness of the program coding structure as well as basic knowledge of program modification techniques. Recent tools such as PROVIDE [75], permit one with a more high level understanding of

the program to interact with the code and understand it. PROVIDE is a dynamic debugging tool developed by Moher that incorporates a graphics strategy in presenting information about a program as opposed to a more textual approach. The user is guided through various portions of the code by selecting icons from a window format. Some windows have pictorial data displays with "hooks" that the user can "attach" to and manipulate. Although at this point it is only experimental, Moher reports success in presenting programs to users at the level of abstraction of their choice.

Given a modification request, locating the portion of code to be modified is a very difficult issue. In the most elementary sense, a programmer sufficiently familiar with the construction of the program can pinpoint the exact line of code to be changed. At the opposite extreme, a user who has little or no programming experience would have no familiarity with the program, how it works, what data elements are employed, etc. could at best make only an uneducated guess as to where the change should be done. What is needed is a automated system that can serve the entire spectrum of modification needs of a program.

Korel [56], using his PELAS system, devised a tool to locate bugs within a program addressing only corrective

maintenance. Traditionally, bugs are found by setting breakpoints within a program and, using a well-understood test data-base, examine the contents of some of the variables and thus locate the source of the programming problem. The PELAS system uses the control flow graph of a program to aid in pinpointing the source of the defect. A *control flow graph* of a program is a directed graph, $CG = (S, A, en, ex)$ having a unique entry node *en* and a unique exit node ex. S is the set of nodes (instructions). A single instruction corresponds to an assignment, input, or output statement; in addition, it corresponds to the <expression> part of *if* or *while* statement (called test instruction). A is the set of arcs which represent a possible transfer of control from one instruction to the other. A *path* in the control graph (called control path) is a sequence of $\langle a_1, a_2, \cdots, a_n \rangle$ of instructions such that (a_i, a_{i+1}) is in A for all a_i , $1 \leq i < n$. A path $\langle en, a_1, a_2, \cdots, a_n \rangle$ starting at en is *feasible* if there exist input data which cause the path to be traversed during program execution.

An *execution trace*, $T = \langle en, a_1, a_2, \cdots, a_n \rangle$, is a sequence of instructions which correspond to a feasible path starting at en. Thus, using a control flow graph, we monitor not only the flow of instructions from one line of code to the next but also the order and frequency

in which they occur. This allows us to distinguish between the same instruction being executed more than once within the flow graph. A use of variable v is an instruction X in which this variable is referenced. A use can be a test instruction, an assignment instruction, or an output instruction. A *definition* of variable v is an instruction X which assigns a value to that variable. A definition can be an assignment instruction or an input instruction.

In order to understand the impact of a change of a variable ("side-effects") as well as the localization of an instruction to be modified, one must study the interdependence of the instructions and variables of a program. There are three basic types of dependencies (influences), data influence, control influence, and potential influence.

<u>Data Influence</u> [56] refers to the data flow, where one instruction assigns a value to a variable and another instruction uses that variable. More formally, $X^{(q)}$ has data influence on $Y^{(p)}$ iff there exists a variable v such that:

1) X is a definition of v,
2) Y is a use of v, and
3) there is no definition of v between q and p.

We assume that all variables are simple variables - a variable having no components. Array variable, record variables. etc. are assumed to have simple variables as their components.

> <u>*Control Influence*</u> [56] applies to conditional statements and the target statement if the conditions are met or not. it does not refer to the successor of the conditional statement. Thus in if ... then z; a=b; , z is the target statement, a=b is the successor statement. The scope of influence for if and while statements is defined as follows:
>
> 1) **if** *X* **then** *B1* **else** *B2*: Instruction Y is in the *scope of influence* of *X* iff *Y* appears in *B1* or *B2*.
>
> 2)**while** *X* **do** *B*: instruction Y is in the scope of influence of *X* iff *Y* is in *B* or X = Y.
>
> Control influence is defined as follows: $X^{(q)}$ has control influence on $Y^{(p)}$ iff
>
> 1) *X* is in a test instruction (if ... then ... else ..., or **while** ...),

2) all instructions between q and p (including Y) are in the scope of influence of X.

If a test instruction $X^{(p)}$, by altering the flow of execution, can modify the value of input variable of instruction $Y^{(p)}$ then $X^{(q)}$ has potential influence on $Y^{(p)}$. More formally, $X^{(q)}$ has potential influence on $Y^{(p)}$ iff there exists a variable v such that:

1) X is a test instruction,

2) Y is a use of v,

3) there is no definition of v between q and p,

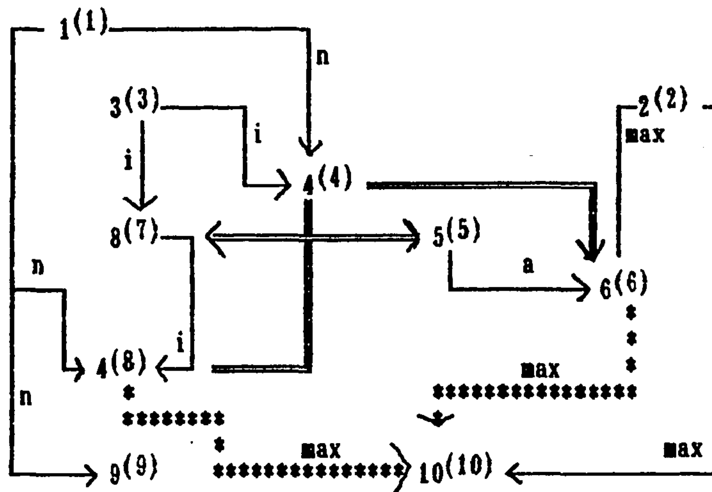
4) there is a control path from X to Y along which v is modified.

See figures 7 and 8 taken from [56] for an example of program flow tracing with emphasis on the various influences within a program.

In the traditional Software Development Life Cycle (SDLC) a series of diagrams - Data Flow Diagrams (DFD), entity-relationship diagrams, etc. may be drawn to communicate the analyst's ideas and requirements of the

```
1   read (n);
2   max :=0 ;
3   i := 1 ;
4   while i < n do
        begin
5           read (a) ;
6,7         if a < max then max := a ;
8           i := i + 1 ;
        end ;
9   write (n) ;
10  write (max) ;
```

Fig 7. Sample program



Legend:

———————— data influence

════════ control influence

************ potential influence (for values n=4, a=1)

Fig 8. Program Dependence Sub-network for above program

system to the programmer. Included in the communication are usually formal specifications that indicate what the function of each program is to be, what the inputs are, and what outputs are expected. The programmer then takes

these requirements and designs a program to satisfy them.

Some recent work has been done in approaching program modification from the design end of SDLC as opposed to the programming end. The ISDOS project at the University of Michigan developed a Problem Statement Analyzer (PSA) as a tool to analyze and document information system requirements and design. PSA is associated with the Problem Statement Language (PSL) that allows expression and manipulation of the user's requirements of an Information System. It is argued that if a program is developed based on a specification then modifying the specification should produce a modified program [23]. The advantage of this approach is that undesirable side affects may be more easily noticed and avoided. When making changes directly in a program, often the impact of such changes are not fully understood and unwanted results often occur.

In this thesis, we wish to present a framework for semi-automated program modification. Much work has been done in the area of program debugging ranging from assertion of specifications and proving their correctness [32] to tracing the program control flow and comparing the anticipated results verses the actual outcome [56]. Our approach will provide a user with a vehicle to maintain

existing programs. At this stage, we will restrict ourselves to well-defined, well-understood programs that are both structured and error-free. The types of modifications we will investigate will be limited to perfective computational changes within a program as seen (or required) by a non-programming professional. Such a user has limited needs in program maintenance. His motivation for the change requirements (which we will call *triggers*) is focused on the outputs he sees (reports, fields of records, etc.). For example, consider the sample programming session in the appendix (see page 63). The user wishes to modify the percentage used to calculate an individual's tax obligation. The trigger in this case would be the entry appearing beneath the column TAXES on the output report. This entry contains the resultant tax calculation based on some tax table within the program. It is the user's first link to locating the instructions that would allow him to accomplish his desired change.

Our aim is to establish a framework for a system that not only supplies a vehicle for program change, but also provides for user-friendly communication between the user and the program to facilitate such revisions. The general sketch of the structure of the system appears in the appendix (see page 83).

The user must communicate with the modification system and express his desired changes. Direct communication with the target program itself is undesirable since this requires the user to be considerably knowledgeable of the program's mechanism and structure in which case a tool to aid in the maintenance of the code is superfluous. The modification system and not the user must interface with the program to provide a smooth implementation of the user's desired changes. The following is an elucidation of some of the key components needed for such a modification system.

2.1 The User-System Interface

Arrangements must be made to assist the user in communicating easily with the expert system as well as with the target program environment. Recent advances in program development and user-friendly program interfaces using graphic tools (windows, hypertext, etc.) has received much attention currently and is thought to assist the user in his computer tasks (see for example [35] and [89]). A set of icons, displayed on the screen, symbolize the various functions the user needs to perform. These icons may be in context-sensitive form and they may further guide the user to more specialized sub-functions within the basic operation selected. The user positions a pointing device (mouse) on the desired icon and by "clicking" the mouse while positioned there, selects the desired function.

The user needs a handle for his trigger - the output item that motivates the need to change the outcome. He must be able to identify the program's output value that he desires to modify so that the system can apply the changes to the variable associated with that value. We envision a display screen in a window format which displays some of the icons that the user needs to interface with the system (see appendix page 84 for

sample icons). The user would respond to prompts as to which program and which output from that program he is interested in investigating. The user can then further study the effects and consequences of a change he wishes to implement by choosing the icon of interest and applying it to the information displayed on the screen. It is important that only actual data and calculations be employed so that the ability to ascertain the proper code within the program is maximized. It is possible, however, to allow the user to furnish a key field on the output detail description line with actual data that would appear in a file or data base. The expert system would then be responsible for retrieving the record for that key and supplying the balance of the detail line fields. Alternatively, the expert system can randomly select a record (or records) to be displayed on the screen. The display of data presented to the user, involves an execution of a portion(s) of the program to calculate the output values that are to be presented based on the supplied fields. The result is a realistic sketch of the outputs that are normally found on the report. The user can tailor the screen display to show any record (or category of record) he wishes by choosing the appropriate records that reflect his modification needs. The user, via the user-expert interface, and an

appropriate screen setup, can review the computational history of a variable and express how he would like the calculation to appear now [see appendix page 63 for a sample session]. Human factors need to be more thoroughly investigated before such a screen can be effectively designed.

2.2 The Systems Functions

Once the user has successfully indicated which variable is to be modified as well as how its computation should now be determined, the system (expert) must now execute the desired change. To implement a modification, a five-stage process is required:

1. locate the proper line of code that is to be modified and report any "ripple" problems.
2. decide on how the line is to be changed (eg. insert a new line changing the current line by a factor, or change the value of an initialized variable in main memory)
3. insert the code in the program
4. modify the size, position of variable(s) (in record descriptions) as necessary for the computation
5. examine the programming environment for the impact of such a change and adjust it accordingly

For documentation purposes the system should signify the expert's change and indicate (as a comment or in an area not used by the compiler) which code was modified.

Based on our constraint that the triggers are user-motivated (as opposed to a request by a data processing professional who might be more concerned with

enhancing the efficiency or esthetics of a program), one of the key rules to implement when locating the code that is to be modified is that the search for the variable to be changed begins with the output variable name found on the report (or output record). Then, through program flow, follow the logical path of the program backward to locate the statement containing the source variable.

2.3 Locating the proper line of code to be changed

Much research has been devoted to debugging programs so that they perform as intended. Some automated debuggers (eg. [56]) are able to assist the user in locating the specific portion of code that is in error. The user then makes the appropriate change in the code at that point to allow the program to perform as designed. We regard program modification as a special form of program debugging. The program is not "debugged" to remove an error - an unintended instruction, rather, we debug the program to correct an "error" - a <u>no longer</u> *intended* instruction. Thus many of the same techniques and procedures that are used to locate errors of disparity between program specification and program code can now be employed to locate our "errors" - the ones due to changes in a program's computation requirements. The user has already supplied (or has been supplied from the expert system) an "example" of what the results of a computation currently is (the system-user interface). The user now interacts with the system by providing an example of how the output should be. He must supply the actual equation or computation rather than the result of such a calculation. This is necessary since there may be many ways to arrive at the new result desired. We wish

to avoid having the system evaluate many different examples of data to arrive at the proper computation. An unsophisticated user, given the current program computation and meaningful variable names, should have little difficulty in understanding and following the format of the selected code to arrive at the new calculation desired.

The system must now locate the correct line of code to make the change. We start by first slicing the program on the output variable and then, for the slice generated, determine the program dependence network. We continue backward until such time as the variable under consideration (the output variable or its predecessor where predecessor refers to the statement that assign a value to the output variable v) is defined (typically the left hand side of an equation). This line is flagged as a potential change point. We continue backward and flag all such lines where variable v is affected. Next, we temporarily modify the first flagged line with the desired computation change and report back to the user the effect of such a change. By effect, we refer to any "ripple effect" consequences that may occur as a result of the modification. If the results are satisfactory to the user, i.e. the program runs correctly subject to his new computation, we run the program with some additional

test data to provide some further surety that the program works as desired.

2.4 Implementing the change

Computational changes fall into several categories:

- a: simple computations
- b: computations dependent on some condition
- c: computations as a result of some iteration
- d: combinations of the above (which we may assume, by composition, is reduced to the above three)

Variable to be modified can be found as:

- a: targets of a simple assignment (MOVE in COBOL)
- b: the left hand side of an equation (we include the ADD/SUBTRACT/MULTIPLY/DIVIDE ... GIVING ... in COBOL)
- c: an initialized variable residing in main memory (and never modified within the program)
- d: a variable modified in main memory (which may or may not be initialized)
- e: a variable that references a table or array in main memory (This is a special case of d - the value of the variable depends on a condition - the value of the index)

f: a variable residing in a record of a file (like e above, but the possible values are not readily available)

After the line of code has been identified for modification, an analysis must be done to ascertain the category of instruction we are dealing with. We must distinguish between computations that are part of a conditional statement, computations that are not dependent on any other instruction in the flow graph (straight line code) and a computation segment that is part of an iteration. For computations in straight line code (slc) form a simple replacement of the new computation for the old is done. Conditional computations are handled by interacting with the user and presenting the three available choices for modification.

a) modify the consequent portion of the condition (target of the if statement)
b) modify only the condition portion of the statement
c) modify both the condition and target portions of the statement.

Computations within iterations are a dilemma since any modification done to the calculation is repeated each time within the loop and the result may not emerge as

desired. Here the user must exert extreme caution as to what change is to be done. The expert system can assist the user in his decision by providing an execution trace for each computation change proposed. Service can also be provided to the user for circumventing side-effects. The system can not arbitrarily assume that a computational change causing a "ripple" elsewhere within a program is undesirable. What can be done is to grant the user the option of sustaining the change as is (with its ripple), or, to prevent the effect from occurring elsewhere within the program, afford the option of using a local variable to hold the new computation and keep the balance of the program as is. This is the function of the shadow of influence icon in conjunction with the show code icon.

Once the modifying code has been added (or existing code modified), dimension analysis must be performed to ensure that the (new) expected range of values to be used do not exceed the current size allocation of the variable. If a size adjustment to the variable is required then change version control must be implemented to insure that other programs that use the affected file are modified to reflect the modification done to their files. A data dictionary would be needed to govern which files reside in which programs as well as how different

files interrelate and possibly share variables. The use of a data base system for data retrieval as opposed to individual files within a program would drastically reduce this problem, however, many older programs (the kind that this system would maintain) still use file structures as opposed to data base technology.

The method of implementation and the choice of which modification technique is best is subject to heuristics, a sample of which appears below.

Types of Heuristics

1. Select the latest (in the program flow) definition of the variable to be changed as the one to attempt to modify first. Latest refers to the variable which is the closest predecessor to the output variable (including the output variable itself) that has been defined by a computation within the program. Ignore simple assignment statements and search backward (using the flow graph of the program) until first computational statement appears.
2. Attempt to change a variable by changing a constant in RHS of the variable's computation.
3. Consider changing the initialized value of a RHS variable at its definition point in cases where the code itself does not change its value.

4. When changing a value attempt to change it at the initialization area as opposed to the code area.
5. If modification is to be global in nature, select the code location that maximizes the shadow of influence (see page 55) of the change variable.
6. Display the constant value in addition to the constant variable name to assist the more sophisticated user.
7. Provide the user with the option of applying a local variable to the computation and thus contain the modification to this statement only.
8. When encountering conditional statements, inspect all forward paths from the condition to the logical end of the program that contain the variable to be changed. This will allow the system to handle multiple computation statements based on a condition.
9. Do not search for history of change variable (from beginning of the program to present) if LHS of computation is to be changed (eg. COMPUTE).
10. For variable change on RHS - there is no need to report past history of variable but all slices from change point and logically further <u>must</u> be reported to prevent "side effect" problems.
11. Slice program on the selected change variable before producing flow graph. This will minimize the entries

in the flow graph table and reduce the searching effort.

12. When user directs change to occur within a loop (or a performed routine within a looping structure) warn user that the results are dependent on the number of executions of the loop.

13. After identifying the location of code to change, examine the compiler syntax analysis output for use of changed variable in execution paths other than the current one and report the results to the user. Apply the same analysis for influenced variables that are within the shadow of influence of the changed variable.

3.0 An Algorithm for Interface and Change

1. Produce simulated display of output report with x's and 9's to indicate character and number strings respectively

2. Have the user identify the field he wishes to modify (computation changes) and provide a sample entry (modification by example)

3. Slice program backward on selected variable (related to output report variable)

4. Generate a flow graph for the slice

5. Based on the example presented locate the portion of the program to be modified

    5.1. Repeat until a non-trivial assignment statement with the target variable on RHS are encountered.

        5.1.1. Search the flowgraph backward beginning with the output variable (first target variable) of the report

5.1.2. For trivial assignments (computations) identify the LHS variable as the new target variable

5.1.3. Queue all target variables

5.2. For each target variable on the queue

5.2.1. Present user with the program instruction at each entry of the queue as a potential change point

5.2.2. Slice on the variable forward (for "example" first and then generally) and report any impact "ripple" if changed

6. Make the change - choose from change in:

6.1. In-line code

6.2. Subroutine parameter

6.3. Initialized variable - including tables (arrays)

6.4. Un-initialized variable

note: the choice of which of the above areas to effect the change involves some heuristics. Generally, the order of choice would be in the sequence listed. However, there are times, for example, when for the benefit of program understandability a modification may be done in an initialized variable (when this variable is

referenced only locally and nowhere else) as opposed to in-line code.

7. Report impact of other paths used (cross reference on target variable and indicate other possible paths - excluding selected example)

8. Run new program using "example" as test data

The user provides "examples" of sample inputs and inspects the results that the program would produce. Based on the "example" the user indicates which portion of the outcome should change. It is then the job of the "modifier" to locate the proper portion of code to effect the change as well as present the user with the COBOL code together with the values these variables would assume given the "example" provided. The goal of the "modifier" is to seek the portion of code that would have the least influence (ripple) on the balance of the program. Clearly, a change done at (logical) point x (within the flow graph of a program) will have no influence on variables within the logical flow of the program prior to point x. Our objective is to choose the "best" (location) x such that the desired change is accomplished yet the change is closer to the end of the

program flow than to the beginning. We shall call the potential influence of a change in a variable v's value to the future use of that variable by other variables the *shadow of influence for variable* v. We are in effect trying to force the change to happen in such a way that the shadow of v is minimal and still accomplish the desired result in our output. The modifier must recognize that minimizing the shadow is not the only criteria for "proper" program change. Issues such as program comprehension and modifiability must be considered. At this point however, the unsophisticated "junior programmer" ("modifier") at best presents the first place to affect the change as well as the method chosen, the user must accept or reject the recommended code change as the specific case warrants.

Statement 1: *The algorithm will locate the portion of code to be modified.*

Discussion: The flow graph that is generated for the sample data field consists of the sequence of instructions that were executed to arrive at the current output. By following the flow graph backward we must arrive at the variable that caused the output. Our

traversal of the flowgraph is constrained to the slice generated by the output variable. Thus all searching is limited to variables that affect the output variable. The candidate variable for change is that variable that is reached first in flow (backward) that affects a computation. The output will be changed as desired since we are changing the candidate variable that caused the output.

Definition: The *shadow of influence for a variable* v is the lines of code that would be affected from line k and forward (in the flow graph) by a change in v.

Statement 2: *The algorithm minimizes the shadow of influence when changing variable* v.

Discussion: By approaching the modification from the output generated, a significant portion of computations have already transpired (In general, report output statements occur toward the logical end of a program since we are reporting

the calculations based on the inputs). Thus, any change done close to the generated output will necessarily avoid influence on lines of code occurring logically before it. The algorithm selects the minimum affect of the modification by choosing the method of change that localizes the affect (in-line code as opposed to modification of initialized variables) and forces the change to impact only future (in the logical flow of the program) executable lines of code. The candidate variable selected for change was the one that was closest to generating the output (we followed the flow graph backward) thus any change done on the variable can have no influence on any (logical) code before it.

Statement 3: *When the shadow of influence is not localized, the algorithm reports the impact of the change.*

Discussion: Gerald Weinberg's "unexpected linkages" (i.e. a changed variable was used elsewhere in the program and by

modifying the variable in one place other code was affected as well) are one of the serious problems encountered in program modification. At this point, however, the "modifier" has no way of knowing whether the linkage is expected or not. The user must be informed of the consequences of his change in the very likely event that the modified variable is not localized. Thus the user can make an informed decision as to the type of change he wants and the best technique that will accomplish his requirements.

Needed changes are indicated by the user supplying real "examples" of what type of input data is to be affected. The output of that input is located to allow the desired modification to be implemented. Often the user wishes to modify a family of changes, i.e. a group or class of similar "examples". Here, the shadow of influence of v may increase in size to cover all of the members of the family. The system modifier must view the user requests as a single request that is the union of the set of "examples" supplied. In contrast, a user may wish to qualify the type of change to be done by

providing a series of "example" that are meant to contain rather than expand the range of data to be affected. The system modifier must then consider the intersection of the "examples" and not the union. This view may necessitate a larger shadow than would otherwise be required.

The algorithm will successfully locate and change the user supplied "example" to the desired effect. The user will be aware of the "ripple effect" of this change since the algorithm provides for rerunning the program with the new changes in place. The user will be cautioned if the location of change is such where the variable is used later in the program in some calculation. It is the responsibility of the system modifier to follow the entire path of the program and report on all consequences of the changed variable.

Empirical studies [93] have shown that programmers understand programs faster if they understand the underlying structure and design of the system. In general, a high-level understanding of a program structure is the primary step in full comprehension of the mechanics of a system. The system modifier provides this ability by removing unaffected variables (slicing) and by establishing the link among the modules (program flow). The maintainer is freed from the mental gymnastics

of organizing and categorizing the modules of the program in order to locate the lines of code to be changed.

The second part of the system modifier's job is to "correct" the "error" in the proper way. The approach here is to minimize the potential "side effects" of a change by isolating the results to the "examples" provided.

4.0 Future Directions

This research discusses some of the foundations needed for the development of a partially automated system for computational perfective maintenance. It applies some of the debugging techniques heretofore reserved exclusively for corrective maintenance and adapts them to pertain to perfective maintenance as well. Some heuristics are suggested which would generally aid the future modification of the program in terms of program understandability and structure. At this time, this research is restricted to modifications involving a single execution trace. It is useful in that it equips a non-computer professional with a tool for perfective maintenance in a direct and uncomplicated way.

Implementation of such a system must include several components. A compiler component is necessary to generate the control flow structures (data influence, control influence and potential influence), and to obtain the program slice. It is also useful in showing the shadow of influence of a variable change at a particular point within the program. A data dictionary is useful in that it would allow the user to identify variables in terminology he recognizes as opposed to the program's variable name. In addition, the data dictionary would

assist in the adjustment of the targeted variable in other programs within the system environment. A command analyzer would provide the user with the capability of more abstract communication with the program to be changed. More research must be done in this area to define the command structure and syntax required. A final element would be a documentation component to control the changes implemented. Ideally, it should identify when the change was made, why it was done and by whom it was effected.

Further analysis is needed to allow the user to make more general computational changes that relate to several different execution traces. This will allow the user to specify a *class of examples* which are to be modified. The final step is to allow program modification from a functional view. A program that had one function could be "modified" to perform an entirely different one. Program functions consist of a series of assignment, conditional, iterations, and computation statements designed to perform a particular task. Function abstraction is the process by which the precise program commands contained in a module are distilled into a form that describe the basic rules of the module. It requires the application of function-theoretic concepts combined

with data analysis, program slicing and pattern recognition to the program under investigation.

This research should be of use to Hausler et al. who are developing an automated system where the input is the program under examination - generally a structured program (which simplifies the design of the function abstractor) and the output is a set of "business rules" which describe the program's actions [43]. By identifying the "business rules" of a module and using a consistent 4GL-type language, an expert system can be developed to assist in locating and modifying a program function. Other researchers who are involved in program transformations (see [79]) may wish to implement some of the concepts discussed here in their systems as well.

Appendix

A1 Sample Session

Adjust Tax Rate

The user wishes to modify the percentage of tax for a particular income bracket. He selects a "sample individual" and "runs the program" for that data. What follows is a description of how the system would make the change to the program.

1. SCREEN 1 & 1A The opening menu is displayed and requests the program name and (internal) file name of the report.
2. Screen 2 - (alg step 1) displays the output report format with headers. The user "fills in the blanks" with a "sample" entry (Screen 2A). Note that only information that would come from the external file is entered by the user, the balance is calculated by the system. Fields to be entered by the user are indicated by pound-sign (#) after the field length. Fields that should be supplied by the external file and do not appear on the report are prompted by the system along with

an indication of their size and attributes (Screen 2B & 2C).

3. Screen 3 - 3A - (alg step 2) displays the result of the system executing this "snapshot" of data on the report. The balance of the entries are supplied. The user then identifies the TAXES field to indicate that this calculation is to be changed.

4. Screen 4 - beneath the TAXES entry the system responds with the variable name OUT_TAX <- FED-TAX and the line number (163) that produced this result. This means that the entry on the report came from a direct transfer of FED-TAX to OUT-TAX. This is an application of heuristic #1 - choosing the latest point within the program flow to make the change. The user is prompted with Continue? An affirmative response has the system slice on FED-TAX at line 163 (as per heuristic #11 -slicing the program before a flow graph is generated). Slice backward on FED-TAX (only, since FED-TAX is on LHS of equation; this will cut down on the nodes of the flow graph; heuristic #9) and delineate

the flow graph for the slice. (alg steps 3 & 4)

5. (alg step 5) Screen 5 - Follow the control path of the execution and choose the FED-TAX computation at 182. The system displays the variables of the computation where FED-TAX is computed in both variable form and value form for the given "sample" (heuristic #6). The user decides that the percentage, NDXD-PCT (NDX) should be changed and types a # by that field. Slice on NDXD-PCT (NDX) to the logical end of the program to determine if the change done at this point will impact other statements (heuristic #10). In this sample session it does, FED-TAX is used at line 172. The user is informed of the potential "problem" and opts to allow the program to affect this portion of code as well. The system then responds with a prompt for the replacement value (Screen 5A) and the user responds with 18 (Screen 5B).

6. (alg step 6) Since the value of NDXD-PCT (NDX) is stored in a table (as per heuristic #3) the system replaces line 100 ... VALUE 17 with ... VALUE 18. At this time

the system also generates the flow graph along with the data, control, and potential influence links for line 163. We note that there is a reference to FED-TAX (heuristic #8) at lines 180 & 186.

7. (alg step 7) report impact of other lines - Screen 5C

8. (alg step 8) run prog and report result to user (not shown) Screen 6 - Session complete.

A2 Screen Displays for Sample Session

```
PROGRAM MODIFIER

Enter Program Name: __________
Enter Report  Name: __________
```

Fig 9. Screen 1

```
PROGRAM MODIFIER

Enter Program Name: XXX
Enter Report  Name: OUTFILE
```

Fig 10. Screen 1A

NAME	GROSS.PAY	TOT.PAY	TAXES	(cont)
XXXXXXXXXXXX#	9999999.99	9999999.99	9999999.99	
SSN	FICA			
XXXXXXXXX#	9999999.99			

Fig 11. Screen 2

NAME	GROSS.PAY	TOT.PAY	TAXES	(cont)
SMITH	9999999.99	9999999.99	9999999.99	
SSN	FICA			
123456789	9999999.99			

Fig 12. Screen 2A

```
  NAME            GROSS.PAY    TOT.PAY       TAXES       (cont)
SMITH             9999999.99   9999999.99    9999999.99

     SSN          FICA
     123456789    9999999.99

Please supply IN-DEP   9 , IN-RATE 99.99 , IN-HOURS 99.9
```

Fig 13. Screen 2B

```
  NAME            GROSS.PAY    TOT.PAY       TAXES       (cont)
SMITH             9999999.99   9999999.99    9999999.99

     SSN          FICA
     123456789    9999999.99

Please supply IN-DEP   1 , IN-RATE 10.00,  IN-HOURS 40.0
```

Fig 14. Screen 2C

NAME	GROSS.PAY	TOT.PAY	TAXES	(cont)
SMITH	400.00	380.77	49.96	

SSN	FICA
123456789	26.80

Fig 15. Screen 3

NAME	GROSS.PAY	TOT.PAY	TAXES	(cont)
SMITH	400.00	380.77	49.96#	

SSN	FICA
123456789	26.80

Fig 16. Screen 3A

```
  NAME            GROSS.PAY    TOT.PAY       TAXES       (cont)
SMITH              400.00      380.77        49.96#
                                         OUT-TAX <- FED-TAX (163)

     SSN          FICA

     123456789    26.80                   Continue? _
```

Fig 17. Screen 4

```
  NAME            GROSS.PAY    TOT.PAY       TAXES       (cont)
SMITH              400.00      380.77        49.96

                                         OUT-TAX <- FED-TAX (163)

COMPUTE FED-TAX ROUNDED = NDXD-TAX (NDX) + (.01 * NDXD-PCT (NDX)# * (TAX-
INC - NDXD-INCOME (NDX - 1))).

COMPUTE FED-TAX ROUNDED = 16.68 + (.01 * 17 * (380 - 185.00)).

     SSN          FICA

     123456789    26.80

WARNING! Line 172 impacted by proposed modification
```

Fig 18. Screen 5

```
 NAME              GROSS.PAY    TOT.PAY        TAXES      (cont)
SMITH               400.00       380.77         49.96
                                         OUT-TAX <- FED-TAX (162)
COMPUTE FED-TAX ROUNDED = NDXD-TAX (NDX) + (.01 * NDXD-PCT (NDX)# * (TAX-
INC - NDXD-INCOME (NDX - 1))).
COMPUTE FED-TAX ROUNDED = 0 + (.01 * 17 * (380.77 - 185.00)).
     SSN          FICA
     123456789    26.80
Entry is from a table and has the value 17.
                                          Change value to ? 99
```

Fig 19. Screen 5A

```
 NAME              GROSS.PAY    TOT.PAY        TAXES      (cont)
SMITH               400.00       380.77         49.96
                                         OUT-TAX <- FED-TAX (162)
COMPUTE FED-TAX ROUNDED = NDXD-TAX (NDX) + (.01 * NDXD-PCT (NDX)#  * (TAX-
INC - NDXD-INCOME (NDX - 1))).
COMPUTE FED-TAX ROUNDED = 0 + (.01 * 17 * (380.77 - 185.00)).
     SSN          FICA
     123456789    26.80
Entry is from a table and has the value 17.
                                               Change value to 18
```

Fig 20. Screen 5B

```
  NAME            GROSS.PAY     TOT.PAY        TAXES      (cont)
SMITH              400.00        380.77          49.96
                                          OUT-TAX <- FED-TAX (163)

COMPUTE FED-TAX ROUNDED = NDXD-TAX (NDX) + (.01 * NDXD-PCT (NDX)# * (TAX-
INC - NDXD-INCOME (NDX - 1))).

COMPUTE FED-TAX ROUNDED = 0 + (.01 * 17 * 380.77 - 185.00)).

    SSN          FICA
    123456789    26.80

Entry is from a table and has the value 17.
                                             Change value to 18

WARNING! FED-TAX can also be computed at lines 180 and 186 depending on the
value of TAX-INC.

Accept? __
```

Fig 21. Screen 5C

```
              Session Complete
```

Fig 22. Screen 6

A3 Program Slice

(program taken from [68] - sliced portions are highlighted)

```
 1 IDENTIFICATION DIVISION.
 2 PROGRAM-ID. XXX.
 3 ENVIRONMENT DIVISION.
 4 CONFIGURATION SECTION.
 5 SOURCE-COMPUTER. IBM-PC.
 6 OBJECT-COMPUTER. IBM-PC.
 7 INPUT-OUTPUT SECTION.
 8 FILE-CONTROL.
 9 SELECT  INFILE ASSIGN TO PAYDATA2
10  ORGANIZATION INDEXED
11  ACCESS SEQUENTIAL
12  RECORD KEY IN-SSN.
13 SELECT  YTD-FILE ASSIGN TO YTDFILE
14  ORGANIZATION INDEXED
15  ACCESS RANDOM
16  RECORD KEY YTD-SSN.
17 SELECT  OUTFILE ASSIGN TO PRNT.
18 DATA DIVISION.
19 FILE SECTION.
20 FD  INFILE LABEL RECORD OMITTED.
21 01  IN-RECORD.
22  05  IN-SSN                 PIC X(09).
23  05  IN-NAME                PIC X(18).
24  05  IN-RATE                PIC 99V99.
25  05  IN-HOURS               PIC 99V9.
26  05  FILLER                 PIC X(33).
27  05  IN-DEP                 PIC 9.
28  05  FILLER                 PIC X(12).
29
30 FD  YTD-FILE LABEL RECORD OMITTED.
31 01  YTD-REC.
32  05  YTD-SSN                PIC X(09).
33  05  YTD-PAY                PIC 9(7)V99.
34  05  YTD-TAX                PIC 9(7)V99.
35
36 FD  OUTFILE LABEL RECORD OMITTED.
37 01  OUT-REC                        PIC X(133).
38
39 WORKING-STORAGE SECTION.
40 01  WORK-STORE-RECORD.
41  05  WS-SSN                 PIC X(09).
42  05  WS-NAME                PIC X(18).
43  05  WS-RATE                PIC 99V99.
44  05  WS-HOURS               PIC 99V9.
45  05  FILLER                 PIC X(33).
46  05  WS-DEP                 PIC 9.
```

```
 63  05  FILLER            PIC X(9)    VALUE 'GROSS.PAY'.
 64  05  FILLER            PIC X(13)   VALUE SPACES.
 65  05  FILLER            PIC X(7)    VALUE  'TOT.PAY'.
 66  05  FILLER            PIC X(10)   VALUE SPACES.
 67  05  FILLER            PIC X(6)    VALUE 'TAXES'.
 68  05  FILLER            PIC X(13)   VALUE SPACES.
 69  05  FILLER            PIC X(3)    VALUE 'SSN'.
 70  05  FILLER            PIC X(13)   VALUE SPACES.
 71  05  FILLER            PIC X(4)    VALUE 'FICA'.
 72  05  FILLER            PIC X(27)   VALUE SPACES.
 73 01  DATA-LINE.
 74  05  FILLER                   PIC X   VALUE SPACES.
 75  05  OUT-NAME                 PIC X(20).
 76  05  FILLER                   PIC X(7) VALUE SPACES.
 77  05  OUT-PAY                  PIC $Z(4)999.99.
 78  05  FILLER                   PIC X(7) VALUE SPACES.
 79  05  OUT-GROSS                PIC $Z(4)999.99.
 80  05  FILLER                   PIC X(7) VALUE SPACES.
 81  05  OUT-TAX                  PIC $Z(4)999.99.
 82  05  FILLER                   PIC X(7) VALUE SPACES.
 83  05  OUT-SSN                  PIC X(9).
 84  05  FILLER                   PIC X(7) VALUE SPACES.
 85  05  OUT-FICA                 PIC $Z(4)999.99.
 86  05  FILLER                   PIC X(33) VALUE SPACES.
 87
 88 01  INCOME-TAX-PCT-LIST.
 89  05  ENTRY-1.
 90      10  INC-1           PIC 9(4)V99 VALUE 0046.00.
 91      10  TAX-1           PIC 9(3)V99 VALUE 000.000.
 92      10  PCT-1           PIC 99 VALUE 00.
 93  05  ENTRY-2.
 94      10  INC-2          PIC 9(4)V99 VALUE 0185.00.
 95      10  TAX-2           PIC 9(3)V99 VALUE 000.00.
 96      10  PCT-2           PIC 99 VALUE 12.
 97  05  ENTRY-3.
 98         10  INC-3           PIC 9(4)V99 VALUE 0369.00.
 99         10  TAX-3           PIC 9(3)V99 VALUE 016.68.
100         10  PCT-3           PIC 99 VALUE 17.
101      05  ENTRY-4.
102  10  INC-4             PIC 9(4)V99 VALUE 0454.00.
103  10  TAX-4             PIC 9(3)V99 VALUE 047.96.
104  10  PCT-4             PIC 99 VALUE 22.
105      05  ENTRY-5.
106  10  INC-5             PIC 9(4)V99 VALUE 0556.00.
107  10  TAX-5             PIC 9(3)V99 VALUE 066.66.
108  10  PCT-5             PIC 99 VALUE 25.
109      05  ENTRY-6.
110  10  INC-6             PIC 9(4)V99 VALUE 0658.00.
111  10  TAX-6             PIC 9(3)V99 VALUE 092.16.
112  10  PCT-6             PIC 99 VALUE 28.
```

```
113       05 ENTRY-7.
114   10   INC-7           PIC 9(4)V99 VALUE 0862.00.
115   10   TAX-7           PIC 9(3)V99 VALUE 120.72.
116   10   PCT-7           PIC 99 VALUE 33.
117 01   TAX-TABLE REDEFINES INCOME-TAX-PCT-LIST.
118       05  TAX-GROUP OCCURS 7 TIMES INDEXED BY NDX.
119   10   NDXD-INCOME     PIC 9(4)V99.
120   10   NDXD-TAX        PIC 9(3)V99.
121   10   NDXD-PCT        PIC 99.
122
123 PROCEDURE DIVISION.
124 A-START SECTION.
125 CONTROL-MODULE.
126  PERFORM 100-A-OPEN.
127  PERFORM 200-RANDP THRU 290-UPDA UNTIL NO-MORE-DATA.
128  PERFORM 300-CLOSE-ROUTINE.
129  STOP RUN.
130
131 100-A-OPEN.
132  OPEN INPUT INFILE.
133  OPEN OUTPUT OUTFILE.
134  OPEN I-O YTD-FILE.
135  MOVE ZEROS TO IN-SSN.
136  START INFILE KEY IS NOT LESS THAN IN-SSN
137     INVALID KEY DISPLAY 'BAD START KEY'.
138  READ INFILE INTO WORK-STORE-RECORD
139     AT END DISPLAY 'FIRST RECORD = EOF'
140        MOVE 'NO' TO MORE-DATA-FLAG.
141  IF MORE-DATA PERFORM 150-PRINT-HEADING.
142
143 150-PRINT-HEADING.
144  WRITE OUT-REC FROM HEADING1 AFTER ADVANCING 3 LINES.
145 200-RANDP.
146  MOVE WS-NAME     TO OUT-NAME.
147  MOVE WS-SSN                 TO OUT-SSN
148  COMPUTE GROSS-PAY = WS-RATE * WS-HOURS.
149  MOVE GROSS-PAY   TO OUT-GROSS.
150
151  COMPUTE TAX-INC = GROSS-PAY - DEP-DEDUC * WS-DEP.
152  COMPUTE FICA = FICA-PCT * GROSS-PAY.
153  MOVE FICA              TO OUT-FICA.
154
155  IF    TAX-INC < NDXD-INCOME(2)
156     PERFORM 291-LOW-TAX
157  ELSE   IF    TAX-INC > NDXD-INCOME(7)
158     PERFORM 295-HIGH-TAX    ELSE
159        PERFORM 292-FIND-TAX VARYING NDX FROM 2 BY 1
160     UNTIL TAX-INC NOT > NDXD-INCOME (NDX).
161
162  MOVE TAX-INC       TO OUT-PAY.
```

```
163  MOVE FED-TAX      TO OUT-TAX.
164  WRITE  OUT-REC  FROM  DATA-LINE  AFTER  ADVANCING  2
     LINES.
165  READ INFILE INTO WORK-STORE-RECORD
166  AT END MOVE 'NO' TO MORE-DATA-FLAG.
167 290-UPDA.
168  MOVE WS-SSN TO YTD-SSN.
169  READ YTD-FILE INVALID KEY
170  DISPLAY 'BAD READ YTD KEY- ', YTD-SSN.
171  ADD TAX-INC TO YTD-PAY  ROUNDED.
172  ADD FED-TAX TO YTD-TAX  ROUNDED.
173  WRITE YTD-REC INVALID KEY
174  DISPLAY 'BAD WRITE YTD KEY - ' WS-SSN.
175
176  READ INFILE INTO WORK-STORE-RECORD
177  AT END MOVE 'NO' TO MORE-DATA-FLAG.
178 291-LOW-TAX.
179  IF   TAX-INC < NDXD-INCOME (1) COMPUTE FED-TAX=0
180  ELSE COMPUTE FED-TAX = .12 * (TAX-INC - NDXD-INCOME
     (1)).
181 292-FIND-TAX.
182  COMPUTE FED-TAX ROUNDED = NDXD-TAX (NDX) +
183  (.01 * NDXD-PCT (NDX) *
184  (TAX-INC - NDXD-INCOME (NDX - 1))).
185 295-HIGH-TAX.
186  COMPUTE FED-TAX ROUNDED =
187  188.04 + (.37) * (TAX-INC - NDXD-INCOME (7)).
188
189 300-CLOSE-ROUTINE.
190  CLOSE INFILE.
191  CLOSE OUTFILE.
192  CLOSE YTD-FILE.
193 399.
194  EXIT.
```

A4 Shadow of Influence

(program taken from [68] - influence portions are highlighted)

```
  1 IDENTIFICATION DIVISION.
  2 PROGRAM-ID. XXX.
  3 ENVIRONMENT DIVISION.
  4 CONFIGURATION SECTION.
  5 SOURCE-COMPUTER. IBM-PC.
  6 OBJECT-COMPUTER. IBM-PC.
  7 INPUT-OUTPUT SECTION.
  8 FILE-CONTROL.
  9 SELECT  INFILE ASSIGN TO PAYDATA2
 10  ORGANIZATION INDEXED
 11  ACCESS SEQUENTIAL
 12  RECORD KEY IN-SSN.
```

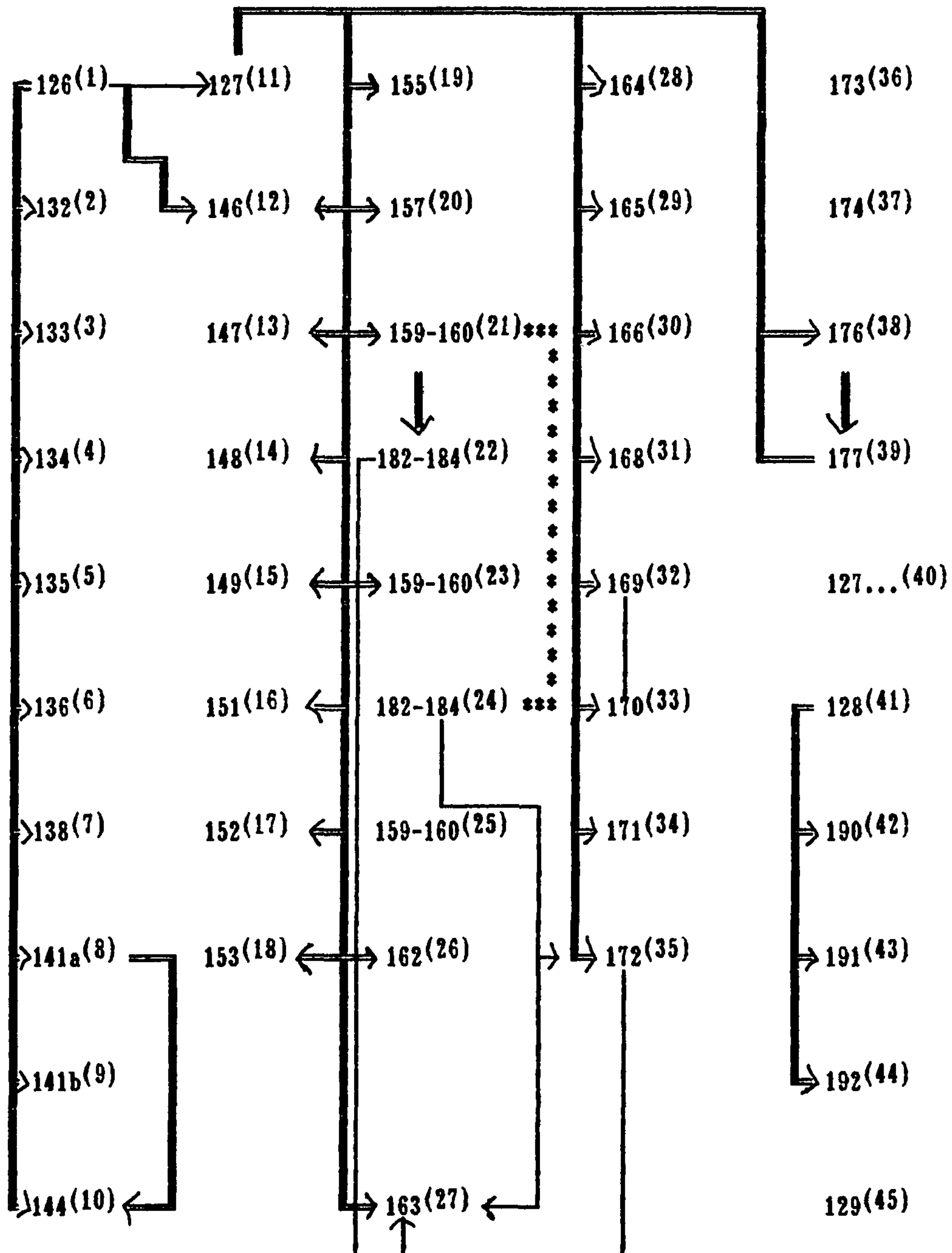
```
13 SELECT  YTD-FILE ASSIGN TO YTDFILE
14  ORGANIZATION INDEXED
15  ACCESS RANDOM
16  RECORD KEY YTD-SSN.
17 SELECT  OUTFILE ASSIGN TO PRNT.
18 DATA DIVISION.
19 FILE SECTION.
20 FD  INFILE LABEL RECORD OMITTED.
21 01  IN-RECORD.
22  05  IN-SSN                          PIC X(09).
23  05  IN-NAME                         PIC X(18).
24  05  IN-RATE                         PIC 99V99.
25  05  IN-HOURS                        PIC 99V9.
26  05  FILLER                          PIC X(33).
27  05  IN-DEP                          PIC 9.
28  05  FILLER                          PIC X(12).
29
30 FD  YTD-FILE LABEL RECORD OMITTED.
31 01  YTD-REC.
32  05  YTD-SSN                         PIC X(09).
33  05  YTD-PAY                         PIC 9(7)V99.
34  05  YTD-TAX                         PIC 9(7)V99.
35
36 FD  OUTFILE LABEL RECORD OMITTED.
37 01  OUT-REC                                PIC X(133).
38
39 WORKING-STORAGE SECTION.
40 01  WORK-STORE-RECORD.
41  05  WS-SSN                          PIC X(09).
42  05  WS-NAME                         PIC X(18).
43  05  WS-RATE                         PIC 99V99.
44  05  WS-HOURS                        PIC 99V9.
45  05  FILLER                          PIC X(33).
46  05  WS-DEP                          PIC 9.
47  05  FILLER                          PIC X(12).
48
49 01  W-S-VARIABLES.
50  05  MORE-DATA-FLAG              PIC XXX VALUE 'YES'.
51  88  MORE-DATA               VALUE 'YES'.
52  88  NO-MORE-DATA            VALUE 'NO'.
53  05  FED-TAX                     PIC 9(7)V99 COMP.
54  05  GROSS-PAY                   PIC 9(7)V99 COMP.
55  05  FICA                        PIC 9(7)V99 COMP.
56  05  FICA-PCT                    PIC 9V999 VALUE 0.067.
57  05  TAX-INC                     PIC 9(7)V99 COMP.
58  05  DEP-DEDUC                 PIC 9(2)V99 VALUE 19.23.
59 01  HEADING1.
60  05  FILLER                  PIC X(7) VALUE SPACES.
61  05  FILLER                  PIC X(4) VALUE 'NAME'.
62  05  FILLER                  PIC X(17)   VALUE SPACES.
```

```
 63  05  FILLER              PIC X(9) VALUE 'GROSS.PAY'.
 64  05  FILLER                PIC X(13)   VALUE SPACES.
 65  05  FILLER                PIC X(7) VALUE 'TOT.PAY'.
 66  05  FILLER                PIC X(10)   VALUE SPACES.
 67  05  FILLER                PIC X(6)    VALUE 'TAXES'.
 68  05  FILLER                PIC X(13)   VALUE SPACES.
 69  05  FILLER                PIC X(3)    VALUE 'SSN'.
 70  05  FILLER                PIC X(13)   VALUE SPACES.
 71  05  FILLER                PIC X(4)    VALUE 'FICA'.
 72  05  FILLER                PIC X(27)   VALUE SPACES.
 73 01  DATA-LINE.
 74  05  FILLER                PIC X   VALUE SPACES.
 75  05  OUT-NAME              PIC X(20).
 76  05  FILLER                PIC X(7) VALUE SPACES.
 77  05  OUT-PAY               PIC $Z(4)999.99.
 78  05  FILLER                PIC X(7) VALUE SPACES.
 79  05  OUT-GROSS             PIC $Z(4)999.99.
 80  05  FILLER                PIC X(7) VALUE SPACES.
 81  05  OUT-TAX               PIC $Z(4)999.99.
 82  05  FILLER                PIC X(7) VALUE SPACES.
 83  05  OUT-SSN               PIC X(9).
 84  05  FILLER                PIC X(7) VALUE SPACES.
 85  05  OUT-FICA              PIC $Z(4)999.99.
 86  05  FILLER                PIC X(33) VALUE SPACES.
 87
 88 01  INCOME-TAX-PCT-LIST.
 89  05  ENTRY-1.
 90      10  INC-1            PIC 9(4)V99 VALUE 0046.00.
 91      10  TAX-1            PIC 9(3)V99 VALUE 000.000.
 92      10  PCT-1            PIC 99 VALUE 00.
 93  05  ENTRY-2.
 94      10  INC-2            PIC 9(4)V99 VALUE 0185.00.
 95      10  TAX-2            PIC 9(3)V99 VALUE 0.00.
 96      10  PCT-2            PIC 99 VALUE 12.
 97  05  ENTRY-3.
 98      10  INC-3            PIC 9(4)V99 VALUE 0369.00.
 99      10  TAX-3            PIC 9(3)V99 VALUE 016.68.
100      10  PCT-3            PIC 99 VALUE 17.
101  05  ENTRY-4.
102      10  INC-4            PIC 9(4)V99 VALUE 0454.00.
103      10  TAX-4            PIC 9(3)V99 VALUE 047.96.
104      10  PCT-4            PIC 99 VALUE 22.
105  05  ENTRY-5.
106      10  INC-5           PIC 9(4)V99 VALUE 0556.00.
107      10  TAX-5            PIC 9(3)V99 VALUE 066.66.
108      10  PCT-5            PIC 99 VALUE 25.
109  05  ENTRY-6.
110      10  INC-6           PIC 9(4)V99 VALUE 0658.00.
111      10  TAX-6            PIC 9(3)V99 VALUE 092.16.
112      10  PCT-6            PIC 99 VALUE 28.
```

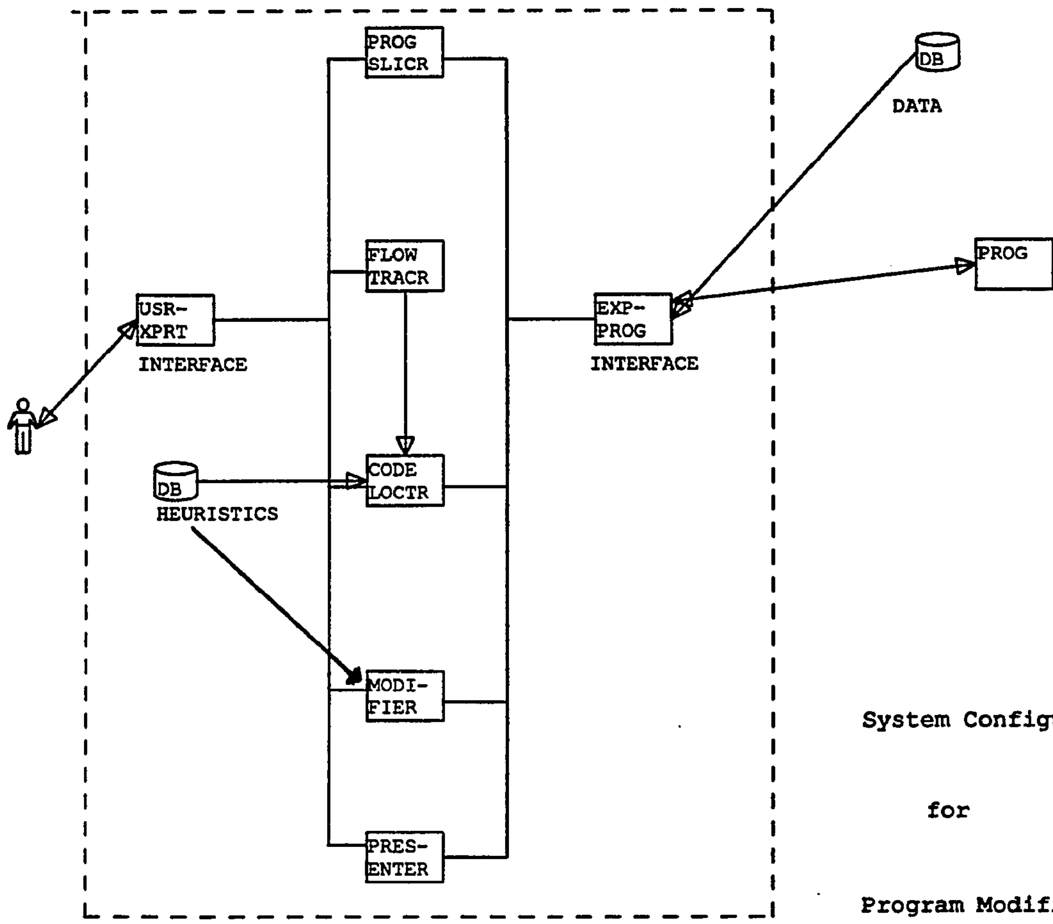
```
113  05  ENTRY-7.
114      10  INC-7            PIC 9(4)V99 VALUE 0862.00.
115      10  TAX-7            PIC 9(3)V99 VALUE 120.72.
116      10  PCT-7            PIC 99 VALUE 33.
117 01  TAX-TABLE REDEFINES INCOME-TAX-PCT-LIST.
118  05  TAX-GROUP OCCURS 7 TIMES INDEXED BY NDX.
119  10  NDXD-INCOME           PIC 9(4)V99.
120  10  NDXD-TAX              PIC 9(3)V99.
121  10  NDXD-PCT              PIC 99.
122
123 PROCEDURE DIVISION.
124 A-START SECTION.
125 CONTROL-MODULE.
126  PERFORM 100-A-OPEN.
127  PERFORM 200-RANDP THRU 290-UPDA UNTIL NO-MORE-DATA.
128  PERFORM 300-CLOSE-ROUTINE.
129  STOP RUN.
130
131 100-A-OPEN.
132  OPEN INPUT INFILE.
133  OPEN OUTPUT OUTFILE.
134  OPEN I-O YTD-FILE.
135  MOVE ZEROS TO IN-SSN.
136  START INFILE KEY IS NOT LESS THAN IN-SSN
137  INVALID KEY DISPLAY 'BAD START KEY'.
138  READ INFILE INTO WORK-STORE-RECORD
139  AT END DISPLAY 'FIRST RECORD = EOF'
140    MOVE 'NO' TO MORE-DATA-FLAG.
141  IF MORE-DATA PERFORM 150-PRINT-HEADING.
142
143 150-PRINT-HEADING.
144  WRITE OUT-REC FROM HEADING1 AFTER ADVANCING 3 LINES.
145 200-RANDP.
146  MOVE WS-NAME    TO OUT-NAME.
147  MOVE WS-SSN     TO OUT-SSN
148  COMPUTE GROSS-PAY = WS-RATE * WS-HOURS.
149  MOVE GROSS-PAY  TO OUT-GROSS.
150
151  COMPUTE TAX-INC = GROSS-PAY - DEP-DEDUC * WS-DEP.
152  COMPUTE FICA = FICA-PCT * GROSS-PAY.
153  MOVE FICA          TO OUT-FICA.
154
155  IF   TAX-INC < NDXD-INCOME(2)
156     PERFORM 291-LOW-TAX
157  ELSE  IF   TAX-INC > NDXD-INCOME(7)
158     PERFORM 295-HIGH-TAX ELSE
159     PERFORM 292-FIND-TAX VARYING NDX FROM 2 BY 1
160     UNTIL TAX-INC NOT > NDXD-INCOME (NDX).
161
162  MOVE TAX-INC    TO OUT-PAY.
```

```
163  MOVE FED-TAX        TO OUT-TAX.
164  WRITE  OUT-REC  FROM  DATA-LINE  AFTER  ADVANCING  2
     LINES.
165  READ INFILE INTO WORK-STORE-RECORD
166  AT END MOVE 'NO' TO MORE-DATA-FLAG.
167 290-UPDA.
168  MOVE WS-SSN TO YTD-SSN.
169  READ YTD-FILE INVALID KEY
170  DISPLAY 'BAD READ YTD KEY- ', YTD-SSN.
171  ADD TAX-INC TO YTD-PAY  ROUNDED.
172  ADD FED-TAX TO YTD-TAX  ROUNDED.
173  WRITE YTD-REC INVALID KEY
174  DISPLAY 'BAD WRITE YTD KEY - ' WS-SSN.
175
176  READ INFILE INTO WORK-STORE-RECORD
177  AT END MOVE 'NO' TO MORE-DATA-FLAG.
178 291-LOW-TAX.
179  IF  TAX-INC < NDXD-INCOME (1) COMPUTE FED-TAX = 0
180  ELSE COMPUTE FED-TAX = .12 * (TAX-INC - NDXD-INCOME
     (1)).
181 292-FIND-TAX.
182  COMPUTE FED-TAX ROUNDED = NDXD-TAX (NDX) +
183  (.01 * NDXD-PCT (NDX) *
184  (TAX-INC - NDXD-INCOME (NDX - 1))).
185 295-HIGH-TAX.
186  COMPUTE FED-TAX ROUNDED =
187  188.04 + (.37) * (TAX-INC - NDXD-INCOME (7)).
188
189 300-CLOSE-ROUTINE.
190  CLOSE INFILE.
191  CLOSE OUTFILE.
192  CLOSE YTD-FILE.
193 399.
194  EXIT.
```

A5 Program Sample Flow Trace



A6 System Configuration for Program Modification

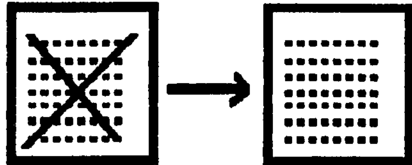


System Configuration

for

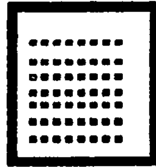
Program Modification

A7 Icons

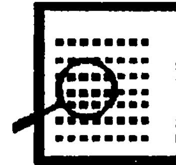


Icon 1. Perform Modification as Specified

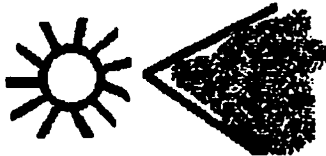
Icon 2. Map Program Flow-graph



Icon 3. Display Program Code



Icon 4. Locate Change Point



Icon 5. Display the Shadow of Influence of the Variable

A8 COBOL BNF Grammar Production Rules

File Modification - portions that don't affect file modification are not expanded

FD description::=

FD <file-name)[*block-clause* | record-clause | label-clause | value clause | data records clause].
Block Clause ::=
BLOCK [<integer> | <integer TO>] [RECORDS|CHARACTERS]
Record description ::=
[<*level-number*> [<data-name> | FILLER] [ε |
***REDEFINES clause* | *BLANK WHEN ZERO clause* |**
***JUSTIFIED clause* | *OCCURS clause* | *PICTURE clause* |**
***SYNCHRONIZE clause* | *USAGE clause* | *VALUE clause*]] |**
[66 <data-name> *RENAMES clause*] |
[88 <condition-name> *VALUE clause*]
level-number::=
<01 | 02 | ... | 65 | 67 | ... | 87 | 89 | ... | 99>
ε::= "empty string"
REDEFINES clause::=
REDEFINES <data-name2>))
note: data-name & data-name2 must be on the same level; data-name2 cannot contain nor be subordinate to a clause that contains an OCCURS clause
BLANK WHEN ZERO clause::=
[BLANK WHEN ZERO | BZ]) note: may be used only on elementary level numeric fields
JUSTIFIED clause::=
[JUSTIFIED | JUST] [ε | RIGHT]
OCCURS clause::=
[OCCURS <integer> [ε | ASCENDING <data-name> | DESCENDING <data-name> | INDEXED <data-name>] |
[OCCURS <integer1> TO <integer2> [ε | DEPENDING <data-name>] [[ASCENDING | DESCENDING] <data-name2>]
| INDEXED <index-name>]
PICTURE clause::=
[PICTURE | PIC] <*character-string*>.
SYNCHRONIZED clause::=
[SYNCHRONIZED | SYNC] [LEFT | RIGHT]
USAGE clause::=
USAGE [DISPLAY | [COMPUTATIONAL | COMP] | INDEX]
VALUE clause::=
VALUE *literal* [ε|THRU *literal2*]*
RENAMES clause::=
RENAMES <data-name2> [ε|THRU <data-name3>]

```
literal::=
   [<number>|"<string>"]
character-string::=
   [ϵ | S] repetition-symbol [ϵ | V | .]
   repetition-symbol [ϵ | CR | DB | + | -]

repetition-symbols::=
   [ϵ | A* | X* | 9* | P* | Z* | ** | B* | 0* | +* | -* |
   $*]

condition::=
   [<data-name> | <arithmetic-expression> [ϵ | NOT]
   [POSITIVE | NEGATIVE | ZERO]] |
   [<subject> [<relational-operator> <object> [ϵ | AND
   | OR [ϵ | NOT]]]+
```

PROCEDURE DIVISION COMMANDS

```
ACCEPT <data-name>

ADD  [[<data-name> | <number>]+ TO [[<data-name2> [ϵ |
ROUNDED]]+ [ϵ | SIZE ERROR <imperative-statement>]] |
   [[<data-name> | <number>]+ GIVING [<data-name3> [ϵ |
   ROUNDED]] [ϵ | SIZE ERROR <imperative-statement>]] |
   [[CORRESPONDING | CORR] <data-name> TO <data-name2>
   [ϵ | ROUNDED]] [ϵ | SIZE ERROR <imperative-
   statement>]]

CALL <program-name> [ϵ | USING <data-name>+]

COMPUTE [<data-name> [ϵ | ROUNDED]]+ [FROM | = |
EQUALS]    [<data-name> | <number> |
           <arithmetic-expression>] [ϵ | SIZE ERROR
           <imperative-statement>]

DISPLAY [<data-name|literal]+

DIVIDE [<data-name> | <number>] [INTO <data-name2> [ϵ |
   ROUNDED]] [ϵ | SIZE) ERROR <imperative-statement>]]
   |
     [[INTO | BY] [<data-name2> | <number2>] GIVING
     <data-name3> [ϵ | ROUNDED] [ϵ | REMAINDER
     <data-name4> [ϵ | SIZE ERROR
     <imperative-statement>]]
```

```
EXAMINE <data-name> [TALLYING [UNTIL FIRST | ALL |
   LEADING] literal [ϵ | REPLACING BY literal2]]|
      [REPLACING) [[ϵ | UNTIL] FIRST | ALL | LEADING]
      literal BY literal2]

GO [<procedure-name>]+ DEPENDING ON <data-name>

MOVE [literal| [[ϵ | CORRESPONDING | CORR] <data-name>]
   TO [<data-name2>]+

MULTIPLY [<data-name> |<number>] [BY <data-name2> [ϵ |
   ROUNDED]] [ϵ | SIZE ERROR <imperative-statement>]] |
       [BY [<data-name2> | <number2>] GIVING
      <data-name3>)  [ϵ | ROUNDED]] [ϵ | SIZE ERROR
      <imperative-statement>]]

PERFORM <procedure> [THRU <procedure2>]
   [[<data-name> | integer] TIMES]| [UNTIL
   <condition>]|) [VARYING [<index> | <data-name>] FROM
   [<index2> | <data-name2>|literal]
       BY [<data-name3> | literal3] UNTIL <condition> [ϵ
      | AFTER [[<index4> | <data-name4>] FROM [<index5>
      | <data-name5> | literal5] BY [<data-name6> |
      literal6] UNTIL <condition2> [ϵ | AFTER [[<index7>
      | <data-name7>] FROM [<index8> | <data-name8> |
      literal8] BY [<data-name9> | literal9] UNTIL
      <condition3>]]]

READ <file-name> [ϵ | INTO <data-name>] [AT END |
INVALID   KEY] <imperative-statement>

RELEASE <record-name> [ϵ | FROM <data-name>]

RETURN <file-name> [ϵ | INTO <data-name>] AT END
   <imperative-statement>

SEARCH <data-name> [ϵ | VARYING [<index> | <data-
name2>]    [AT END <imperative-statement>]) [WHEN
           <condition> [<imperative-statement | NEXT
           SENTENCE]+

SEARCH ALL <data-name> [AT END <imperative-statement>]
   [WHEN <condition> [<imperative-statement | NEXT
   SENTENCE]

SET [[<index> | <data-name>]+ TO [<index2> | <data-
name2> | <number2>]]|
   [<index>+ [UP | DOWN] BY [<data-name2> | <number2>]]
```

SUBTRACT[<data-name> | <number>]$^{+}$ FROM [<data-name2> [ε
| ROUNDED]]]$^{+}$ [ε | SIZE ERROR
<imperative-statement>]] |
[<data-name> | <number>]$^{+}$ FROM [<data-name2 |
<number2> GIVING <data-name3> [ε | ROUNDED] [ε |
SIZE ERROR <imperative-statement>]] |
[[CORRESPONDING | CORR] <data-name> FROM <data-
name2> [ε | ROUNDED] [ε | SIZE ERROR
<imperative-statement>]]

WRITE <record-name> [ε | FROM <data-name>] [ε |
[[BEFORE|AFTER] ADVANCING [<data-name2 | <integer>]
LINES]] [ε | [AT [END-OF-PAGE | EOP]
<imperative-statement>]] | [INVALID KEY
<imperative-statement>

A9 Bibliography

[1] Aggarwal, S., Barbara, D., & Meth, K., "SPANNER: A Tool for the Specification, Analysis, and Evaluation of Protocols", *IEEE Transaction on Software Engineering*, Vol 13 No 12, pp. 1218-1237, Dec 1987

[2] Ambras, J. & O'Day, V., "Microscope: A Knowledge Based Programming Environment", *IEEE Software*, Vol 5 No 3, pp. 50-58, May 1988

[3] Arango, G., Baxter, I., Freeman, P., & Pidgeon, C., "Maintenance and Porting of Software by Design Recovery",in *IEEE Conference on Software Maintenance 1985*, Washington D.C.: IEEE Computer Science Press, 1985, pp. 42-49

[4] Arnold, R., "Software Restructuring", *Proceedings of the IEEE*, Vol 77 No 4, pp. 607-617, Apr 1989

[5] Ashcroft & Manna, "The Translation of 'GOTO' Programs to 'WHILE' Programs", *Proceedings of the IFIP Congress* Vol 1, Amsterdam : North-Holland, 1972, pp. 250-255

[6] Ashcroft & Manna, "Translating Program Schemas to While Schemas", *SIAM Journal of Computing*, Vol 4 No 2, pp. 125-146, Jun 1975

[7] Backhouse, R., *Program Construction and Verification*, London: Prentice Hall International, 1986

[8] Bailin, S., "An Object-Oriented Requirements Specification Method", *Communications ACM*, Vol 32 No 5 pp. 608-623, May 1989

[9] Baker, B., "An Algorithm for Structuring Flowgraphs", *Journal of the ACM*, Vol 24 No 1, pp. 95-120, Jan 1977

[10] Beck, A., Bleicher, M., Crowe, D., *Excursions in Mathematics*, New York: Worth Publishers (experimental edition), 1967

[11] Belady, L., "Evolved Software for the 80's", *Computer*, Vol 24 No 2, pp. 79-82, Feb 1979

[12] Belkhouche, B., & Urban, J., "Direct Implementation of Abstract Data Types from Abstract Specifications", *IEEE Transactions on Software Engineering*, Vol 12 No 5, pp. 649-661, May 1986

[13] Berzins, V., Gray, M., & Naumann, D., "Abstraction-Based Software Development", *Communications of ACM*, Vol 29 No 5 pp. 402-415, May 1986

[14] Boehm, B., "Software Engineering", *IEEE Transactions on Computing*, Vol 25 No 12 pp. 1226-1241, Dec 1976

[15] Boehm, B., Brown, J., Kaspar, H., Lipow, M.,MacLeod, J., & Menit, M., *Characteristics of Software Quality*, New York: North Holland, 1978

[16] Bohm, C. & Jacopini, G., "Flow Diagrams, Turing Machines, and Languages With Only 2 Formation Rules", *Communications of the ACM* Vol 9 No 5, pp. 366-371 May 1966

[17] Brooks, F., "No Silver Bullet", *IEEE Computer*, Vol 4 No 2, pp. 10-19, Apr 1987

[18] Bush, E., "The Automatic Restructuring of COBOL", *IEEE Conference on Software Maintenance 1985*, Washington D.C.: IEEE Computer Science Press, 1985, pp. 35-41

[19] Chapin, N., "Productivity in Software Maintenance", *AFIPS Conference Proceedings on 1981 National Computer Conference* Vol 50, Montvale: AFIPS Press, 1981, pp. 349-352

[20] Charniak, E. & McDermott, D., *Introduction to Artificial Intelligence*, Reading: Addison-Wesley, 1985

[21] Chikofsky, E. & Cross II, J., "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, Vol 7 No 1, pp. 13-17, Jan 1990

[22] Chikofsky, E. & Rubenstein, B., "CASE: Reliability Engineering for Information Systems", *IEEE Software*, Vol 5 No 2, pp. 11-16, Mar 1988

[23] Chikofsky, E., "Application of an Information Systems Analysis and Development Tool to Software Maintenance", Teichrow, G. & David, G. (ed.), *System Description Methodologies*, Amsterdam: North-Holland 1985, pp. 503-514

[24] Cioch, F., "The Impact of Object-Oriented Decomposition on Procedural Abstraction", *Journal of Pascal, ADA & MODULA-2*, Vol 4 No 3, pp. 49-55, May/Jun 1989

[25] Cleaveland, J. C., "Building Application Generators", *IEEE Software*, Vol 3 No 4, pp. 25-33, Jul 1988

[26] Cooper, D., "Bohm and Jacopini's Reduction of Flow Charts", *Communications of the ACM*, Vol 10 No 8, p.463 Aug 1967

[27] Corbi, T., "Program Understanding: Challenge for the 1990s", *IBM Systems Journal*, Vol 28 No 2, pp 294-306, Feb 1989

[28] Date, C., "An Introduction to Data Base Systems", Reading: Addison-Wesley, 1981

[29] DeBalbine, G., "Better Manpower Utilization Using Automatic Restructuring", *AFIPS Proceedings of the 1975 National Computer Conference* Vol 44, Montvale: AFIPS Press, 1975, pp. 319-327

[30] DeMarco, T. & Lister, T., "Software Development: State of the Art vs. State of the Practice", *IEEE Conference on Software Engineering* 1990, Washington, D.C.: IEEE Computer Science Press, 1990, pp. 271-275

[31] DeRose, B. & Nyman, T., "The Software Life Cycle - A Management and Technological Challenge in the Department of Defense", *IEEE Transactions on Software Engineering*, Vol 4 No 4, pp. 309-318, July 1978

[32] Dershowitz, N., *The Evolution of Programs*, Boston: Birkhäuser, 1983

[33] Dijkstra, E., "GOTO Statements Considered Harmful", *Communications of the ACM*, Vol 11 No 3, pp. 147-148, Mar 1968

[34] Dijkstra, E., "Programming Considered as a Human Activity", *Proceedings of the IFIP Congress*, Amsterdam: North-Holland, 1965, pp. 213-214

[35] Ding, C, & Mateti, P., "A Framework for the Automated Drawing of Data Structure Diagrams, *IEEE Transactions on Software Engineering*, Vol 16 No 5, pp. 543-557, May 1990

[36] Feather, M., "Constructing Specifications by Combining Parallel Elaborations", *IEEE Transactions on Software Engineering*, Vol 15 No 2, pp. 198-208, Feb 1989

[37] Ferrante, J., Ottenstein, K., & Warren, J., "The Program Dependence Graph and its Use in Optimization", *ACM Transactions on Programming Languages & Systems*, Vol 9 No 3, pp. 319-349, Jul 1987

[38] Fitzsimmons, A. & Love, T., "A Review & Evaluation of Software Science", *Computer Surveys*, Vol 10 No 1, pp 3-18, Mar 1978

[39] Gallagher, K., "Using Program Slicing in Software Maintenance", working paper, 1989

[40] Gilb, T., "Structured Program Coding: Does it Really Increase Program Maintainability?", *Techniques of Program and System Maintenance*, Cambridge: Winthrop, 1982, pp. 193-195

[41] Harandi, M. & Ning, J., "Knowledge-based Program Analysis", *IEEE Software*, Vol 7 No 1, pp. 74-81, Jan 1990

[42] Harandi, M. "An Experimental COBOL Restructuring System", *Software - Practice and Experience*, Vol 13, pp. 825-846, 1983

[43] Hausler, P., Pleszkoch, M., Linger, R., & Hevner, A., "Using Function Abstraction to Understand Program Behavior", *IEEE Software*, Vol 7 No 1, pp. 55-63, Jan 1990

[44] Hecht, M. & Ullman, J., "Characterizations of Reducible Flow Graphs", *Journal of the ACM*, Vol 21 No 3, pp. 367-375, Jul 1974

[45] Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping", *IEEE Transactions on Software Engineering*, Vol 12 No 2, pp. 241-250, Feb 1986

[46] Hildum & Cohen, "A Language for Specifying Program Transformations", *IEEE Transactions on Software Engineering*, Vol 16 No 6, pp. 630-638, Jun 1990

[47] Hopkins, M., "A Case for the GOTO", *Proceedings of the 25th National ACM Conference*, pp. 787-790, Aug 1972

[48] Humphrey, W., "Characterizing the Software Process: A Maturity Framework", *IEEE Software*, Vol 5 No 2, pp. 27-32, March 1988

[49] Kafura, D. & Reddy, G., "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering*, Vol 13 No 3, pp. 335-343 Mar 1987

[50] Kaiser, G., Feller, P., Popovich, S. "Intelligent Assistance for Software Development and Maintenance", *IEEE Software*, Vol 5 No 3, pp. 40-49, May 1988

[51] Kernighan, B. & Plauger, P., "Programming Style: Example and Counterexample", *ACM Computing Surveys*, Vol 6 No 4, pp. 303-19, Dec 1974

[52] Kishimoto,Z., "Testing in Software Maintenance and Software Maintenance from the Testing Perspective", *IEEE Conference on Software Maintenance* 1983, Washington D.C., IEEE Computer Science Press, 1983, pp. 116-117

[53] Knuth, D. & Floyd, R., "Notes on Avoiding GOTO Statements", *Information Processing Letters* Vol 1, Amsterdam: North Holland, 1971, pp. 23-31

[54] Knuth, D., "Structured Programming with GOTO Statements", *Current Trends in Programming Methodology Vol 1*, Yeh (ed.), Englewood Cliffs: Prentice Hall, 1977,

[55] Korel, B. & Laski, J., "Dynamic Program Slicing", *Information Processing Letters* Vol 29 No 3, pp. 155-163, Oct 1988

[56] Korel, B., "PELAS - Program Error-Locating Assistant System", *IEEE Transactions on Software Engineering*, Vol 14 No 9, pp. 1253-1260, Sep 1988

[57] Kramer, J., Magee, J., Keng, N., "Graphical Configuration Programming", *IEEE Computer*, Vol 5, No 5, pp. 53-65, Oct 1989

[58] Kung, C., "Conceptual Modeling in the Context of Software Development", *IEEE Transactions on Software Engineering*, Vol 15 No 10, pp. 1176-1187, Oct 1989

[59] Leung, H. & Reghbati, H., "Comments on Program Slicing", *IEEE Transactions on Software Engineering*, Vol 13 No 12, p. 1370 Dec 1987

[60] Lientz, B. & Swanson, E. *Software Maintenance Management*, Reading: Addison-Wesley, 1980

[61] Lientz, B. & Swanson, E., "Characteristics of Application Software Maintenance", *Communications of the ACM*, Vol 21 No 6, pp. 466-470, Jun 1978

[62] Lukey, F., "Understanding & Debugging Programs", *International Journal of Man-Machine Studies*, Vol 12, pp. 189-202, 1980

[63] Maggiolo-Schettini, A., Napoli, M., Tortora, G., "Web Structures: A Tool for Representing and Manipulating Programs", *IEEE Transactions on Software Engineering*, Vol 14 No 11, pp. 1621-1639, Nov 1988

[64] Malhorta, A., Markowitz, H., Tsalalikhin, Y., Pazel, D. & Burns, L., "An Entity-Relationship Programming Language", *IEEE Transactions on Software Engineering*, Vol 15 No 9, pp. 1120-1129, Sep 1989

[65] Martin, J. & McCulre, C., "Software Maintenance: The Problem and its Solution", Englewood Cliffs: Prentice Hall, 1983

[66] Mathis, R., "The Last 10 Percent", *IEEE Transactions on Software Engineering*, Vol 12 No 6, pp. 705-712. Jun 1986

[67] McCabe, T., "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol 2 No 4 pp. 308-320, Dec 1976

[68] McCalla, R., *Structured COBOL Programming*, Montery:, Brooks Cole, 1985

[69] McClure, C., *Reducing COBOL Complexity Through Structured Programming*, New York: Van Nostrand Reinhold, 1978

[70] Mikkilineni, R., "Potential Use of the Object Oriented Paradigm for Software Engineering environments in the 1990s", *COMPSAC 88 Proceedings*, 12th annual Intl Computer Software & Applications Conference, pp. 439-440, 1988

[71] Miller, L., "Programming by Non-Programmers", *International Journal on Man-Machine Studies*, Vol 6, pp. 237-260, 1974

[72] Miller, J. C., Personal communication

[73] Mills, H., *Software Productivity*, New York: Little, Brown Computer Systems Series, 1983

[74] Mitchel, J., Urban, J. & McDonald, R., "The Effect of Abstract Data Types on Program Development", *IEEE Computer*, Vol 4 No 4, pp. 85-88, Aug 1987

[75] Moher, T., "PROVIDE: A Process Visualization and Debugging Environment", *IEEE Transactions on Software Engineering*, Vol 14 No 6, pp. 849-857, Jun 1988

[76] Myers, B., "Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints", *ACM Transactions on Programming Languages & Systems*, Vol 12 No 2, pp. 143-177,Apr 1990

[77] Parikh, G., "Some Tips, Techniques, and Guidelines for Program and System Maintenance", *Techniques of Program and System Maintenance*, Cambridge: Winthrop, 1980

[78] Parnas, D., "On the Criteria to be used in Decomposing Systems into Modules", *Communications of the ACM*, Vol 5 No 12, pp. 1053-1058, Dec 1972

[79] Partsch, H. & Steinbruggen, R., "Program Transformation Systems", *Computing Surveys*, Vol 15 No 3, pp. 199-236, Sep 1983

[80] Peterson, W., Kasami, T., & Tokura, N., "On the Capabilities of While, Repeat, and Exit Statements", *Communications of the ACM*, Vol 16 No 8, pp. 503-512, Aug 1973

[81] Prager, J., Lamberti, D., Gardner, D.,Balzak, S., "REASON: An Intelligent User Assistant for Interactive Environments", *IBM Systems Journal*, Vol 29, pp. 141-163, 1990

[82] Presser & Hug, "Change and Configuration Control Tool", *IEEE Conference on Software Maintenance* 1984, Washington D.C.: IEEE Computer Science Press, 1984, pp.271-274

[83] Ramsey, C. & Basili, V., "An Evaluation of Expert Systems for Software Engineering Management", *IEEE Transactions on Software Engineering*, Vol 15 No 6, pp.747-759, Jun 1989

[84] Rich, E., *Artificial Intelligence*, New York: McGraw-Hill, 1983

[85] Rich, C. & Waters, R., "Automatic Programming: Myths & Prospects", *IEEE Computer*, Vol 5 No 4, pp. 40-51, Aug 1988

[86] Richmond, A., "Software Design by Object-oriented Functional Layering", *Computer-Physics Communication*, Vol 41, pp. 377-384, 1986

[87] Rombach, H., "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Transactions on Software Engineering*, Vol 13 No 3, pp. 344-353, Mar 1987

[88] Schneidewind, N., "The State of Software Maintenance", *IEEE Transactions on Software Engineering*, Vol 13 No 3, pp. 303-310, Mar 1987

[89] Seabrook, R. & Shneiderman, B., "The User Interface in a Hypertext, Multiwindow Program Browser", *Interacting with Computers*, Vol 1, pp. 299-337, 1989

[90] Sell, P., *Expert Systems - A Practical Introduction*, New York: Wiley, 1985

[91] Shneiderman, "Empirical Studies of Programmers: The Territory, Paths, and Destinations",Soloway, E. & Sitharma, I. (ed.), *Empiracal Studies of Programmers*, Norwood: Ablex, 1986, pp. 1-12

[92] Shneiderman, B., Schafer, P., Simon, R., & Weldon, L., "Display Strategies for Program Browsing: Concepts and Experiments", *IEEE Software*, Vol 3 No 3, pp. 7-15, May 1986

[93] Shneiderman, B., *Software Psychology : Human Factors in Computer & Information Systems*, Cambridge: Winthrop, 1980

[94] Shneiderman, B., "Overcoming Limitations Imposed by Current Programming Languages", Jernigan, Hamill, & Weintraub (ed.), The Role of Language in Problem Solving, New York: North-Holland, 1985, pp. 253-275

[95] Spohrer, J. & Soloway, E., "Novice Mistakes: Are the Folk Wisdoms Correct?", *Communications of the ACM*, Vol 29 No 7, pp. 624-632, Jul 1986

[96] Stankovic, J.A., "A Technique to Identify Implicit Information Associated With Modified Code", Teichrow, G. & David, G. (ed.), *System Description Methodologies*, Amsterdam: North-Holland 1985, pp. 457-478

[97] Stelovsky, J. & Sugaya, H., "A System for Specification and Rapid Prototyping of Application Command Languages", *IEEE Transactions on Software Engineering*, Vol 14 No 7, pp. 1023-1032, Jul 1988

[98] Stevens, W., Myers, G., & Constantine, L., "Structured Design", *IBM Systems Journal*, Vol 13 No 2, pp. 115-139, May 1974

[99] Tichy, W., "RCS- A System for Version Control", *Software-Practice & Experience* Vol 15 No 7, pp. 637-645, Jul 1985

[100] Van Hoeve, F. & Engmann, R., "An Object-Oriented Approach to Application Generation", *Software Practice & Experience*, Vol 17 No 9, pp. 623-645, Sep 1987

[101] Vessey, I., "Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols", *IEEE Transactions on Systems Man, & Cybernetics*, Vol 16 pp. 621-637, 1987

[102] Weiser, M., "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol 25 No 7, pp. 446-452, Jul 1982

[103] Weiser, M., "Source Code", *IEEE Computer*, Vol 4, No 6, pp. 66-73, Nov 1987

[104] Weiser, M., "Program Slicing", *IEEE Transactions on Software Engineering*, Vol 10 No 4, pp. 352-357, Jul 1984

[105] Williams, M., "Generating Structured Flow Diagrams: The Nature of Unstructuredness", *The Computer Journal*, Vol 20 No 1, pp. 45-90, Jan 1976

[106] Wulf, W., "A Case Against the GOTO", *Proceedings of the 25th National ACM Conference* August 1972, pp.791-797

[107] Yau, S. & Tsai, J., "A Survey of Software Design Techniques", *IEEE Transactions on Software Engineering*, Vol 12 No 3, pp. 713-721, Mar 1986

[108] Yau, S. & Collofello, J., "Some Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, Vol 6 No 6, pp. 545-552, Nov 80