DATA-FLOW ANALYSIS OF SOFTWARE CHANGE

James N. Campbell
B. S., Rose-Hulman Institute of Technology, 1967

A thesis submitted to the faculty
of the Oregon Graduate Institute
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

November, 1990

The dissertation "Data-Flow Analysis of Software Change" by James N. Campbell has been examined and approved by the following Examination Committee:

Richard G. Hamlet
Professor
Thesis Advisor

David Maier
Professor

Warren Harrison
Associate Professor
Portland State University

ACKNOWLEDGEMENTS

Dr. Richard Hamlet provided the guidance and advice that made this work possible. Doctors Warren Harrison and David Maier made it better with their generous criticism and comments. I would also like to thank the administration and faculty of OGI, who allowed me more than normal consideration in its preparation.

DEDICATION

This thesis is dedicated to the two people who most influenced me in its preparation: Dick Hamlet, who told me that it would be difficult and did his best to make it less so, and my wife Janice, whose love and support provide the context for my life.

TABLE OF CONTENTS

1. Introduction 1

2. Background 7

3. Terminology 23

4. Generic Programming Language 25

5. Dependency Chains 29

6. Change Primitives 38

7. Dependency Chain Analysis 60

8. Examples 73

9. Conclusions 93

References 95

Appendix: Example II Detail 99

Biographical Note 111

ABSTRACT

Data-flow Analysis of Software Change

James N. Campbell, M. S.
Oregon Graduate Institute, 1990

Supervising Professor: Richard G. Hamlet

Structural testing techniques have been the subject of extensive analysis. Until recently, all research has been oriented to the newly-written program; no consideration has been given the modification of existing software.

A new path descriptor, the dependency chain, was proposed, and its use described in gauging the test requirements of a program change. A set of essential changes, called change primitives, was described. Generating the change in dependency chains for these change primitives lead to a method for generating test data for larger changes. The technique shows promise of extension.

1. INTRODUCTION

It has been reported that fifty percent or more of software development costs are spent in maintenance [Brooks, Andr]. It has been suggested that at least half of the software effort should be spent in testing [Myers]. These two statements taken together mean that twenty-five percent of software resources are expended on testing modifications to software. One-quarter of the software effort is spent testing program changes. In spite of the importance of maintenance testing, little has been done to study it and develop techniques for doing it better.

Maintenance projects are traditionally allocated minimal resources. They do not usually attract the brightest and most creative people. They are usually scheduled over-optimistically, sometimes whimsically so. Development activities attract the people, time, and money.

Testing by its nature is scheduled late in the project. Any schedule slip in any of the previous phases cheats the testing phase. Any problem at any time chisels its little bit from testing. All these tendencies conspire to deprive maintenance testing of adequate resources. According to [Boeh], as paraphrased in [Andr]:

"One of the most prevalent and costly mistakes made on software projects today is to defer the activity of detecting and correcting software problems until late in the project."

Other phases of the development cycle have established methodologies. They have automated tools supporting them. The oldest computer manual that comes to hand says of testing, ". . . a trial run should be made with simple numbers so that the machine answer(s) can be checked by hand." [Wyan] It also suggests "a reasonable degree of humility." The thirty or so years that have passed since this was written have not presented too much more than that as testing methodology. Try some stuff and be prepared for the worst.

Considering the plight of the maintenance programmer, what are the attributes of a software aid? Whatever methodology is used, there isn't time to do a manual effort of test selection in parallel with the programming effort. The solution should be automated. As tests are accomplished, they should be accounted for automatically; the solution should be instrumented. Ideally, the management control structure would like to know the scope of the testing effort and be advised of progress; the solution should be measurable. Since a maintenance task usually involves some specific sub-set of the program and there are limited

resources (always!), the solution should be directed and efficient. The testing effort should be directed to just that required by the changes to the program, and provide the best coverage with the fewest cases.

This paper describes a technique for generating test cases that satisfies these requirements. It can be implemented as a compiler option, since it is based on compiler optimization techniques. The results can be stored and as tests are accomplished, instrumentation embedded in the subject program can check off the test cases satisfied. It is oriented toward those test cases that correlate with the changes in the program; any unchanged code is not considered. The technique is automated, instrumentable, measurable, systematic, and efficient.

On the other hand, this technique is not radical or revolutionary; it arises from work already done in the field. In brief, this technique looks at every input variable to the program. When this variable is used in determining the value of another variable, that one is traced until it is used to set the value of another, and so on until the last variable is output from the program. This series of variables receiving values and being used is called a dependency chain. The dependency chain defines a set of paths through the program that have to be executed to test the program's response to one set of input data. Dependency chains are changed by changes to the program. Changes to the dependency chain can be evaluated with regard to changes in test requirements. The dependency chain is a device to reveal changes in

testing required by changes in the program. Again, there is nothing novel in this approach. What has been achieved is the realization that the change to the program can be identified with a change in the set of dependency chains and can be used to define a set of test cases as a direct result of the change itself.

Mark Weiser [Weis82, Weis84] has developed the program slice, which is similar to the dependency chain. A program slice is a minimal subset of program instructions that contribute to one function. Weiser uses it to study the mental habits of programmers while debugging.

The use of dependency chains is a structural testing technique. It provides knowledge of interrelationships over long stretches of code that would be hard to recognize otherwise. It provides a useful set of test cases that has to be considered the "starter set". Added to this should be tests of program functions changed, and those tests that the programmer decides are necessary from previous knowledge of the application and the programming environment.

When we change a program, we want to test it as thoroughly as possible. We see intuitively that for any change, there are two sets of statements in the program: those affected by the change, which have to be retested, and those unaffected by the change, for which further testing would be less useful. Since we are always in a condition of limited resources, we would prefer to concentrate our attention where it would be most useful. To explore this requires answering two questions:

What do we mean that the change "affects" a statement, and how do we find such statements?

When we change a program, we intuitively feel that the testing activity should be proportional to the size and complexity of the change. If we make a small change in the format of a field on a print line, we expect to be able to test the change with a simple test case or two. If we make a more radical or complex change, say, one involving several database accesses and a complicated logic path change, we would expect to have to run many tests, requiring considerable thought in their preparation, before feeling that the program had been adequately tested. To explore this aspect of the problem requires a definition of "size" and "complexity" in software.

It has to be noted that idea of concentrating attention on the change made is not a universal opinion. Many authors recommend extensive testing after modification, among them [Glass] and [Dunn]. They address the concern that limiting the testing to known effects disregards the (likely) possibility of inadvertent and unplanned changes in areas of the program that should not be affected. For those agreeing with this point of view, the technique will assist maintenance of the regression libraries necessary for full system tests. Changes to the program can generate changes to the regression library.

The next chapter provides an overview of data-path techniques and some related writings. Chapter 3 defines terminology. Chapter 4 presents a small language used for examples. Chapter 5 shows a way of relating the effects different parts of the program have on one another, the dependency chain. Chapter 6 describes the set of "change primitives" that have been derived to help define the nature of a program change. Chapter 7 uses material from all the previous chapters to present the analysis. Chapter 8 shows its use in examples.

2. BACKGROUND

There are many methods of testing software reliability. One method is described by Andrion, et al. [Andr]:

> "A complete verification of a program, at any stage in the life cycle, can be obtained by performing the test process for every element of the domain. If each instance succeeds, the program is verified, otherwise an error has been found. This testing method is known as *exhaustive testing* and is the only dynamic analysis technique that will guarantee the validity of the program. Unfortunately, this technique is not practical. Frequently functional domains are infinite, or if not infinite very large, so as to make the number of required test instances infeasible."

With the goal of generating small test cases in order to finish in a reasonable time, most testing methods partition the input domain of the program into smaller sub-domains and choose one element to represent each subdomain. The program is run for each representative and the output compared to a known correct value. If the output matches for each of the test cases, the program is said to be valid.

2.1 FUNCTIONAL TESTING VS. STRUCTURAL TESTING

There are two primary techniques used to generate data sets to test software, functional testing and structural testing. Functional testing, also known as "black box" testing, concentrates on the proposed function of the program. It uses the program specification to generate test data. From the specification, ranges of valid inputs, their interrelationships, and the corresponding proper outputs are determined. The program is run, and the outputs are checked against the set of valid outputs to verify the program's performance. One important technique used in functional testing is cause-effect graphing [Myers].

Structural testing is concerned with looking into the program and determining test data from analysis of the structure of the program. There are several methods used in structural testing. One that has provided a rich field of creative research is error-based testing [Myers], which is oriented toward providing tests that uncover particular error conditions or classes of error conditions. While the tendency to anticipate certain types of errors and test software for these errors is traditional and wide-spread in an informal way, the attempt here is to systematize the effort. One example of this is Weyuker and Ostrand [OW] and their method of "revealing subdomains" of

the program's input domain. Another class of error-based testing methods is "mutation testing" techniques, which create modified versions of the program, with minor changes [Hamlet]. The basis for the technique is to create test suites that distinguish the original program from the modified ones. Mutation testing takes advantage of the observation that programmers write programs that are at least almost correct.

It has long been felt that software validation requires both functional and structural testing [Andr]. A quick analysis should show that this is reasonable. One taxonomy of errors [Good] identifies three types of coding errors: a missing path, an incorrectly chosen path, and an erroneous path. It can be seen that both functional and structural methods can find the erroneous path. However, functional testing could be fooled by an incorrectly chosen path that failed only under rare circumstances. On the other hand, structural techniques are blind to the problem of missing paths. In order to cover all three error types, both methods are required.

2.2 STRUCTURAL TESTING METHODS

Other structural testing methods are called "path-based" or "flow analysis" methods [RW], based on graph-oriented methods [Hecht]. These are based on selecting paths through the program, either "data-flow" or "control-flow" or some combination. Data-flow techniques

depend on the values bound to variables and their use. Control-flow techniques depend on the statement ordering and branching within the program. Zeil [Zeil] has shown that data-flow methods can be more selective in particular cases, that is, they can uncover more errors in fewer test cases than control-flow methods. With all the path-flow techniques, there is one area that has had little attention--program modification. With few exceptions, the work has been done only in the testing of new programs. This thesis investigates a path-based analysis for testing the effect of a program change.

Test criteria are those rules by which a testing strategy determines how much testing has been done, rules that determine to what level the program is tested. One such criterion is *all-statements*. *All-statements* designates that the test cases must be chosen so that when testing is complete, every statement has been executed at least once. A statement, in this context, means anything that changes data or reads or writes data; a branch is not considered a statement for purposes of this criterion. *All-branches* specifies that the every unconditional and conditional branch in the program has to have been executed at least once by the time all the test cases have been considered. *All-branches* is considered "stronger" than *all-statements* because satisfying *all-branches* forces execution of all the statements in the program and both branches resulting from conditions. *All-statements* could have left one of these branches undone. As an example, a small program from Rapps

and Weyuker [RW], in the GPL language, to be defined in detail in Chapter 4:

```
1       read x, y
2       if x ≤ 1000 goto a
3       y <- x + y
4 a:    print x, y
```

The path <1, 2, 3, 4> satisfies the *all-statements* criterion, but does not exercise the branch from 2 to 4. *All-branches* forces a requirement to execute a path that includes the sub-path <2, 4>. Every path that is required to satisfy *all-statements* is also required to satisfy *all-branches*; we say that *all-branches* "subsumes" *all-statements*. Since a case has be shown where *all-branches* requires a path that *all-statements* did not (<1, 2, 4>), *all-statements* does not subsume *all-branches*. Because of the graphic orientation of structural testing, *all-statements* is also called *all-nodes* and *all-branches* is called *all-edges* [RW]. Though *all-statements* or *all-branches* have been recommended as minimal requirements in setting up software assurance plans [Andr], they are regarded as "weaker" criteria [RW]. The "strongest" in that it subsumes the others is *all-paths*, which calls for executing every possible path through the program. The glaring problem with this criterion is that any program with a loop allows for a poten-

tially unlimited number of possible paths through the program. To quit the testing after a finite number of passes through the loop, say k, allows a possible error to occur when the processing of the program passes through it k + 1 times.

It is well to remember that discussing "weak" and "strong" methods may obscure that the processes are oriented to establishing test requirements, and these requirements have to be met by test cases. The quality of these test cases in uncovering program errors can have more to do with the ability of the person creating the test cases than the method generating the requirement. It is easy to overload the words "weak" and "strong" with more meaning than they deserve.

The study of data-path testing methods is concerned with extending test data selection criteria past the control-flow analysis criteria such as *all-paths*, *all-statements*, and *all-branches*. Supplementing these criteria has been the subject of study by three different groups of researchers: Rapps and Weyuker [RW], using Def-use paths, Ntafos [Ntaf], using required element testing, and Laski and Korel [Lask], using data environment and data context. An important analysis by Clarke, et al. [Clar], studies these three families, looking at each for their realization of their original intent and compares them, after resolving them to a common notation.

Comparing selection criteria requires some additional concepts and accompanying notation. A <u>module</u> M is a main program or subroutine.

A path P is a subset of the set of all paths through M. A path criterion C assigns a truth value to each pair (M, P). A pair (M, P) is said to satisfy a criterion C iff C(M, P) is true. A criterion C1 subsumes a criterion C2 iff every pair (M, P) that satisfies C2 also satisfies C1. Two criteria that subsume each other are equivalent. C1 strictly subsumes C2 iff C1 subsumes C2, but C2 does not subsume C1. Two criteria are incomparable iff neither subsumes the other. If A subsumes B and B subsumes C, A subsumes C.

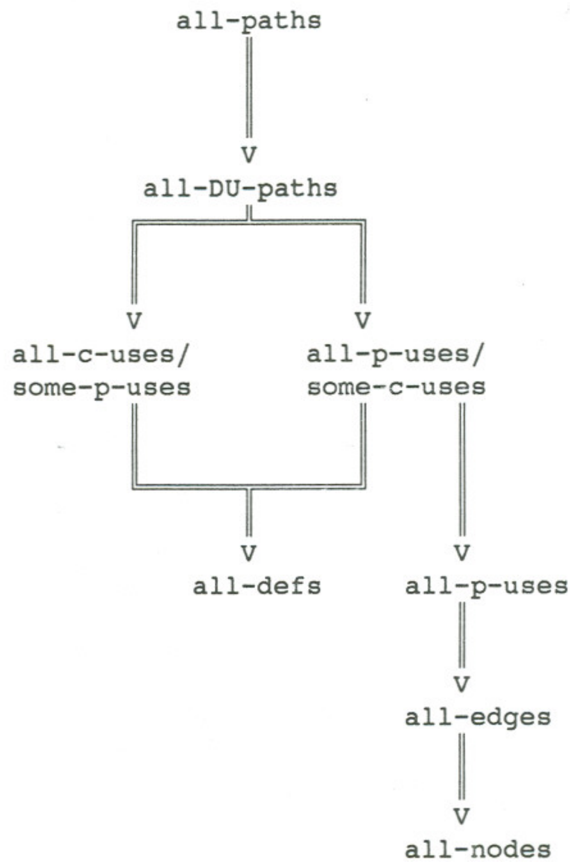
2.2.1 DEF-USE GRAPH

Rapps and Weyuker base their analysis on the def-use graph, which divides occurrences of variables into definitions, computational uses, and predicate uses. A definition, or def, is any place at which variable takes on a value, by assignment or input statement. A computational use, or c-use, is an occurrence of the variable in the calculation of a value bound to a variable or is an output of the program. A predicate use, or p-use, is an occurrence of the variable in the test expression portion of a conditional branch; its value at the time of execution of the statement affects the flow of control through the program.

A def-use path or DU-path is a path through the program that originates at a def of a variable and terminates at a c-use or p-use of

that variable, without encountering another def of the same variable. They define a hierarchy of path selection criteria that subsume *all-statements* and *all-branches*. They define new selection criteria *all-defs*, *all-uses*, *all-c-uses*, *all-p-uses*, *all-c-uses/some-p-uses*, *all-p-uses/some-c-uses*, and *all-DU-paths*, which they rank between the rather weak *all-statements* and the essentially impossible *all-paths*. *All-defs* is satisfied by tests that execute every def statement; *all-DU-paths* is satisfied by tests that execute every DU-path. *All-uses* is satisfied if there is a path executed from every def to every use of the variable. *All-c-uses* is satisfied if all the paths are covered between every def and all c-uses of all variables. *All-p-uses* is satisfied by coverage of the paths between each def and all the p-uses of all variables. *All-c-uses/some-p-uses* requires that paths from all defs to all c-uses be covered and at least one p-use of any variable not used anywhere in a c-use. *All-p-uses/some-c-uses* requires that paths from all defs to all p-uses be covered and at least one c-use of any variable not used anywhere in a p-use. The hierarchy is established by proving that one criterion subsumes another. A criteria that subsumes another is said to be "stronger" or "stricter" than the other, since it demands more knowledge about the program's behavior (more tests) to satisfy the requirements it generates. Rapps and Weyuker point out that stricter criteria tend to demand more tests and practical considerations may indicate that a less strict criterion be used.

Using "A ==> B" to stand for "A subsumes B", Rapps and Weyuker summarize their results with the following diagram.



Harrold and Soffa [Harr] have extended Rapps' and Weyuker's work into the area of software maintenance, recognizing many of the same concerns expressed in this paper, but concentrate on two different topics: interprocedural flow and the effects of the program change on the test history, the accumulated information gained in previous tests of the program.

2.2.2 REQUIRED ELEMENT TESTING

Ntafos defines required-element testing, which can be called a hybrid of functional testing and structural testing. Required elements are expressed as a structural component, such as a branch or path, and an assertation of a condition that the test must satisfy. A set of required elements is used as the selection criterion. He uses a control structure similar to that of Rapps and Weyuker, but divides the data occurrences differently. Data can be defined (d), undefined (u), and referenced (r). A definition is the assignment of a value to the variable at that statement. An undefinition is the return of the variable to a state of having no value assigned to it, such as a local variable on exit from the segment in which it is defined or a loop control variable on exit from the loop. A reference is his term for a use--any occurrence of the variable as in a function calculating a value, in a conditional, or an output statement. An interaction is a reference (r) and a definition (d) that reaches that reference; this is also called a dr interaction (another name for the DU-path.) The basic structural unit that Ntafos uses is the k-dr interaction, which consists of k statements in which the reference to each variable in order is involved in the definition of the next. A class of strategies called "required k-tuples" is built by consideration of the k-dr interactions in the program. The

k-dr interaction consists of k statements and k-1 variables such that there is a path that visits the statements in the order given, and each statement after the first contains a reference of the variable defined in the previous statement.

A required element is of the form {S;F}, where S is some structural element to be tested and F is a functional component to be observed. Structural elements are such things as branches and paths. Functional components are assertions to be satisfied. Within this general format many different testing strategies can be described. A test case is said to cover the required element if it causes S to be executed and satisfies F. For example, consider the statement S: X = Y / 2. If we wish to test for an odd value for Y, we include a required element {V, P(Y)}, where V is a program segment containing statement S and P(Y) is true if Y is odd. As examples of the extremes that can be described with this notation, consider the "pure" functional testing required element {[s_0, t_0]; F), where s_0 is the first statement of the program, t_0 is the last, and [s_0, t_0] defines all the paths between them, and F is any functional requirement. On the other hand, the "pure" structural testing *all-edges* is $\{\{(v_i,v_j) \mid (v_i,v_j) \in E\} ; \}$, where E is the set of edges in the graph.

An example of the power of the assertion portion of the required element can be seen in considering the conditional IF X ≥ Y . . . . The normal testing procedure would be to choose the path X greater than

or equal to Y and the path X less than Y. A required element such as $\{(v_i,v_j);X=Y\}$ could be included in the required elements, where (v_i,v_j) includes the conditional branch statement, and the assertion specifies that X must equal Y, a potential source of error that might otherwise be overlooked.

Two different strategies are specified: required k-tuples and required k-tuples with assertions. Required k-tuples is a strictly structural testing strategy that is the result of static analysis that can be automated. The functional component of the required element is nil. In the required k-tuples with assertions, the assertions generally add the intuition and experience factor, as the programmer describes various potential error conditions that have to be checked out. As such, the assertions do not lend themselves to automation or instrumentation, since they don't flow directly from the structure of the program. Ntafos suggests that it would be appropriate to implement this part as an on-line, interactive application, with the programmer changing and adding to the assertions from one set of tests to another.

Ntafos studies the tradeoffs between required k-tuples and required m-tuples with assertions, where m is less than k. The former technique specifies tests covering larger portions of the program with more interactions, while the latter allows detection of blunders that may be missed by the other. Ntafos' study so far has been concentrated on 3-tuples and 2-tuples with assertions. He has outlined plans to

study higher values of m and k.

2.2.3 DATA ENVIRONMENT/DATA CONTEXT

Using an example program, Laski and Korel show three control-flow test strategies and discuss their reliability for uncovering errors. They say that "all-paths" is viable, but the number of paths increases dramatically for a program of any size and soon testing becomes impractical. They claim that "all-branches" is not viable because paths can be generated that satisfy this criterion but do not uncover the some errors. Similarly, they show that "boundary-interior" testing produces unreliable tests. "Boundary-interior" testing relates to loops in the program: at least one test must immediately satisfy the condition terminating the loop (the "boundary"), and at least one test must execute the statements making up the body of the loop (the "interior").

Laski and Korel base their analysis on the concepts of the "data environment" and the "data context". Let x_k represent the definition of variable x at statement k. We say that x_k is live at statement i if there is a control path from k to i free of another definition of x. Another way of saying this is that there is a path from i to k that is def-clear with respect to x. The data environment is defined at a statement and is the set of all live definitions of all variables used in the statement.

An elementary data context is a n-tuple of definitions that are live when a particular path from the beginning of the program to statement i is designated; n is the number of variables input to statement i. In other words, for a given path that reaches statement i, the elementary data context is the list of definitions of the variables in i that are live when statement i is reached. The data context DC(i) of statement i is the set of all elementary data contexts at i. In other words, it is the set of all n-tuples that can be generated by taking paths through the program from the beginning to statement i, such that each item in the n-tuple is a live definition of a variable used in statement i.

Testing strategies are derived from these concepts. The first is to require that each definition of the data environment of every statement be tested at least once. The second is to test the data context of every statement at least once.

Laski and Korel use these data-flow strategies to test the same example program and determine that the first, definition liveness, is inadequate, but that the second, based on the data context, is reliable for that program. They claim that this demonstrates the power of data-flow over control-flow methods. They follow this with a discussion of error types (corresponding to [Good]), and the reliability of the techniques, when used to attempt detection of these error types.

They emphasize in concluding that data-flow analysis lends

itself to being automated; the techniques are related to compiler code optimization techniques.

2.3 SUMMARY OF STRUCTURAL METHODS

There are some common features to data-flow testing methods. They are best at detecting a particular set of errors, transference errors. A transference error is one in which a variable is bound to an incorrect value and causes another variable to be incorrect when used in assignment or calculation. While a data-flow technique can identify a path, it can't always detect whether the path is feasible, that is, whether there exists any combination of values of the program's variables that will allow the path to be executed completely.

The ideal is to construct error-revealing paths; these are paths that reliably expose errors every time they are executed. Since the program is not subject to change by the tester, the best that can be achieved in the general case is a set of error-sensitive paths, paths that will expose the error if executed with the correct revealing values bound to the variables involved. An example of an error-sensitive path is a statement with a potential zero-divide error that may be executed many times before the denominator takes on the value zero.

One fundamental limitation of structural methods is that they may test a particular program well, but they do nothing to show that the

program does what it was intended to do. Functional testing methods are established to do this. However, experience has shown that structural methods complement functional methods, since the two tend to reveal a different set of errors. Howden says [How], "A programmer has three sources of information for constructing test data: the program to be tested, its specification, and his knowledge of commonly occurring programming errors.".

3. TERMINOLOGY

Structural control-flow testing techniques typically start by generating a directed graph or digraph from the program code. A digraph is a structure <N, E>, where N is a set of nodes and E is a set of edges in N x N.

A node is a program segment consisting of a set of program statements that necessarily execute together, in the normal course of the program's execution. A group of statements will execute together if there are no control mechanisms that can interrupt or redirect program flow once their execution has begun. A node is defined by a conditional branch immediately before it or a branch instruction passing control to its first instruction, and a branch instruction passing control out of it. More formally, a node is a series of consecutive program statements $<s_1, s_2, \ldots, s_n>$ such that, when $n > 1$, for $i = 2, \ldots, n$, s_i is always executed immediately after s_{i-1} and s_{i-1} is always executed immediately before s_i.

An edge is a relationship between two nodes, i and j, such that the last statement of node i can be immediately followed in execution by the first statement of node j. The edge corresponds to control flow.

A <u>path</u> is a sequence of nodes <n_1, n_2, . . ., n_k> such that there is an edge from n_{i-1} to n_i for i = 2, 3, . . ., k. A <u>simple path</u> is a path such that $n_i \neq n_j$ for any $i \neq j$.

A <u>definition</u> of a variable at a node occurs when the value of the variable is modified there by a read or assignment statemnt. DEF(s) represents the set of all variables defined at node s.

A <u>use</u> of a variable occurs at a node when the value of the variable is required to execute the statement in which it occurs. A use is any occurrence of the variable that is not a definition, coming in two categories: predicate use (p-use), which is the occurrence of the variable in an expression associated with a condition, and computational use (c-use), which is the occurrence of the variable in an expression that determines the value of a variable. However, the c-use will be the only use considered here. USE(s) is the set of c-use variables used at node s.

The predicate portion of the conditional will be treated as a simple boolean switch for the purposes of this analysis. The variable involved in the condition does not enter into the def-use couple or dependency chain.

A <u>definition clear path</u>, or def-clear path, is defined with respect to a particular variable. We say that a path is def-clear with respect to x if it contains no definition of x.

4. GPL--GENERIC PROGRAMMING LANGUAGE

For the purposes of exposition, a simple language, based on the formal programming language of Rapps and Weyuker [RW], is used to show the technique without being so complicated itself that it interferes with the examples. It is called GPL, or Generic Programming Language. There are several reasons for using this rather primitive language. It doesn't seem useful to present the examples in an actual programming language in common use. If a reader doesn't know the language, the example has no force. If the reader knows the language well, it may become the focus of attention: Are the examples in good form for the language? Are the best language features used for the sample program? Is the code efficient? These problems are avoided by using a language that anyone can understand, but that no one will become attached to. At the same time, the language has the major contructs that are found in any higher-level programming language.

There are facilities of more powerful languages missing from GPL: arrays, pointers, and procedure calls, for instance. This is deliberate, in the interest of limiting the scope of the thesis. They provide some interesting topics for further research.

4.1 TYPING

There is only one type, real, and no declaration mechanism.

4.2 I/O

GPL has two I/O statements. The Read statement is the only input statement. It consists of the keyword *read* followed by any number of variables, separated by commas. The Print statment is the only output statement. It consists of the keyword *print* followed by any number of variables, separated by commas.

4.3 BRANCHING

There are two ways to control the program flow, conditional and unconditional branches. Any statement can have a label, which precedes the statement and is separated from it by a colon. Every label must be the target of a branch statement somewhere in the program.

The unconditional branch is the keyword *goto* followed by any valid label.

The conditional branch consists of five elements:

if cond *then goto* lab

if,*then*, and *goto* are keywords, lab is any label in the program, and cond is a condition of the form

operand1 operator operand2,

where operand1 and operand2 can be variable names or numeric values, and operator is any of the standard relational operators (=, <, >, ≥, ≤, and ≠).

4.4 ASSIGNMENT

The assignment statment is

var <- val

where var is any variable, '<-' is the assignment operator, and val is a variable, numeric literal, or some computation of the form

val1 numop val2,

where val1 and val2 are variables or numeric literals and numop is any of the standard arithmetic operators (+, -, *, /, and %).

4.5 PROCESSING HALT

The keyword *stop* indicates the end of processing.

4.6 EXAMPLE

A small example will help to understand GPL.

```
a:    read e, c
      if e = -1 then goto x
      f <- c * 9
      f <- f / 5
      f <- f + 32
      print c, f
      goto a
x:    stop
```

This program reads in an end-of-data indicator and a temperature in degrees Celsius. It converts this to degrees Fahrenheit and prints both numbers. It repeats this until the end-of-data indicator equals -1, when it halts processing.

5. DEPENDENCY CHAINS

There is much information in the structure of a program. It is assumed that the program has been written and tested in development. A change is made. What is required to take advantage of our previous knowledge of the program to somehow limit the testing to just that required by the change? There has to be a way to relate the interdependencies of the variables and the values bound to them at various points in the program until there is a complete transference from the program's inputs to its outputs. There has to be an analysis of changes in general so that algorithms can be defined for each potential change. Finally, there has to be an algorithmic approach to relating the change in the program to a change in the testing requirements. This last is to demonstrate that with the proper compiler support and instrumentation, the whole process can be automated.

Note that there are two possible strategies to isolating the test requirements of a given change, given a technique for generating test requirements of a given program. One is to analyze the program before and after the change, store the requirements, and throw out the

intersection of the two sets from the second set. An example will show some of the problems associated with this approach. Consider the program:

```
10          read x, y
20          if x > y then goto a
30          q <- x + x
40          goto b
50    a:    q <- y * y
60    b:    print q
70          stop
```

If the criterion *all-statements* is used to generate requirements for testing this program, the following set is the result.

a. Execute statement 10

b. Execute statement 20

c. Execute statement 30

d. Execute statement 40

e. Execute statement 50

f. Execute statement 60

g. Execute statement 70

Two tests provide complete coverage of the requirements.

1. x = 5, y = 3

Satisfies a, b, e, f, and g

2. x = 5, y = 10

Satisfies a, b, c, d, f, and g

Change the program by changing statement 30 and adding two statements, 61 and 62.

```
10          read x, y
20          if x > y then goto a
30          q <- x * x
40          goto b
50    a:    q <- y * y
60    b:    print q
61          y <- y - 1
62          if y > 0 then goto a
70          stop
```

The test requirements generated from the changed program:

a. Execute statement 10

b. Execute statement 20

c. Execute statement 30

d. Execute statement 40

e. Execute statement 50

f. Execute statement 60

g. Execute statement 70

h. Execute statement 61

i. Execute statement 62

Striking the intersection of the two sets produces a set of test requirements generated from the changes:

h. Execute statement 61

i. Execute statement 62

Running test 2 will satisfy these requirements. However, there has been no test of the changed statement 30. This method is not dependable.

The second strategy is to derive the new requirements from the original program and the changes that transformed it to the new version. This method has been chosen.

The dependency chain is the essential construct in this analysis. The first link is a statement at which a variable is bound to a value; the value is independent of any variable in the program, either read at the statement or a literal assignment. The next link is a statement at which this variable is used in calculating the value of some other variable. This other variable is used in the calculation of another, and so on. Eventually there is a statement that puts data out of the program, the end of the dependency chain. The strategy for choosing test data is to select data that will execute the statements in the sequence that they appear in the dependency chain.

5.1 VARIABLE-STATEMENT PAIRS

In order to define the dependency chain, some basic definitions are required.

Let V be the set of all variables in the program, and S be the set of all statements in the program. A variable-statement pair, (x,i) is any member of V x S.

A def-use couple, denoted by [(v,s) (v,t)], where (v,s) and (v,t) are variable-statement pairs, is the set of simple paths p such that $v \in DEF(s)$, $v \in USE(t)$, and p is def-clear with respect to v between s and t.

IN is the set of all statements s such that DEF(s) $\neq \phi$ and USE (s) = ϕ; i. e. a variable is defined, but without dependence on the value of any other variable, as in a read statement or literal assignment.

OUT is the set of all statements s such that DEF(s) = ϕ and USE(s) $\neq \phi$; i. e., a variable is used but no variable receives a value, as in an output statement.

5.2 FORMAL DEFINITION OF DEPENDENCY CHAINS

A dependency chain is a sequence of variable-statement couples $<(v_1,s_1), (v_1,t_1), (v_2,s_2), (v_2,t_2), \ldots, (v_n,s_n), (v_n,t_n)>$, such that:

$s_1 \in$ IN,

$t_n \in$ OUT,

$t_{i-1} = s_i$, for $1 < i \leq n$, and

there is a def-use couple, $[(v_i,s_i), (v_i,t_i)]$,

for $1 \leq i \leq n$.

Dave Maier has pointed out that this formalism lacks the ability to define a def-use couple [(d,i) (d,i)], where d is used at statement i in the calculation of a new value of d. This is a flaw. When this instance was originally considered, it was felt to be a spe-

cial case of looping, which is not included in this study. The solution would be to redefine the simple path (section 5.1) to allow a statement to occur more than once, when it is the first and last statement of the path. With this redefinition, the def-use couple and dependency chains would fall into place and process this case correctly.

5.3 BUILDING DEPENDENCY CHAINS

Once the def-use couples are determined, dependency chains can be formed. The first step is to find all the variables v such that $v \in DEF(i)$, where $i \in IN$. Each input variable may be used in the definition of another variable, this forms a def-use couple. The new variable is in turn used somewhere else, and so on. Eventually there is a use with no definition, an output statement, which is the end of the dependency chain. The result of this analysis is that the program is considered as a set of interlocking pairs of defs and uses, similar to the required element testing of Ntafos [Ntaf].

EXAMPLE:

```
10   read x, y
20   if x < y then goto a
30   r <- x * x
50   goto b
```

```
60    a: r <- y * y
70    b: print r
80    stop
```

```
IN = {10}
OUT = {70}
```

```
DEF(10) = {x, y}
DEF(30) = {r}
DEF(60) = {r}
USE(30) = {x}
USE(60) = {y}
USE(70) = {r}.
```

The def-use couples are:

```
[(x,10) (x,30)]
[(y,10) (y,60)]
[(r,30) (r,70)]
[(r,60) (r,70)].
```

The dependency chains are:

<(x,10), (x,30), (r,30), (r,70)>, and

<(y,10), (y,60), (r,60), (r,70)>.

From these dependency chains, the following paths are built:

<10, 20, 30, 50, 70>

and

<10, 20, 60, 70>.

Tests have to be developed that will execute the statements in these paths.

6. CHANGE PRIMITIVES

The change primitives are the smallest essential changes that can be made to a program. In order to define change primitives, the consideration is that there are only two things that can be done to data, definition and use, and there are only two ways to affect the flow of control, conditionally and unconditionally. There are only three operations that can be performed on programming entities--add, change, and delete. Thus, the change primitives are defined as a completely exhaustive set of changes.

The use of change primitives allows the sketching of an inductive proof that the use of dependency chains will provide consistently good results for changes of any size. Consider that any change is composed of individual, "atomic" changes. Each of these can be applied individually. Make the first of these changes and apply the techniques to generate a set of tests. Make the next change and add the resulting changes to the first set. Repeat this for all the small changes that make up the original change. It can be seen that there may be too many tests in the final set -- some later change could obviate a test required by an earlier change -- but there will be no tests miss-

ing. Each required test for the whole change has to be generated by one of the detail changes.

Each change primitive is discussed in this chapter and analyzed for dependency chain requirements in Chapter 7. The following table should assist in navigating through these two sections.

	Add	Change	Delete
Definition	6.2.1, 7.2	6.2.2, 7.3	6.2.3, 7.4
Use	6.3.1, 7.5	6.3.2	6.3.3
Unconditional Branch	6.4.1, 7.6	6.4.1	6.4.1
Conditional Branch	6.4.2.1	6.4.2.2	6.4.2.3

6.1 MOTIVATION

In order to compute the activity required to re-test the program, we have to consider the change primitives and look at the activity necessary for each. We consider each from the point of view that the original program was tested to the point that every dependency chain was exercised and we plan to test to the same standard only the added test requirements after the change.

6.2 DEFINITION

6.2.1 CREATION

Introducing a new definition into a program has two possible impacts: it can create new def-use couples, and it may truncate existing ones. As an example of both of these effects, consider this program segment before and after a single definition is inserted into the code at line 12:

```
10 read x              10 read x

11 y <- x + 3          11 y <- x + 3

13 . . .               12 x <- 2

 .                     13 . . .

 .                          .

 .                          .


20 z <- x * x          20 z <- x * x
```

Assuming that these are the only references to x in the program, the first program has two def-use couples defined for x, [(x,10) (x,11)] and [(x,10) (x,20)]. The introduction of statement 12 kills [(x,10) (x,20)] and creates [(x,12) (x,20)]. The couple [(x,10) (x,11)] is not affected.

An added definition can be of three forms: 1) the assignment of a literal constant, 2) an input statement, or 3) a computation in terms of another variable; for example:

```
1)  x <- 100

2)  read x

3)  x <- y + 3
```

In cases 1) and 2), the definition involves no other variable. Therefore, this definition and any uses it reaches form the first link of new dependency chains. These chains must be tested.

Example:

Starting with the program:

```
10        read x, y
20        print y
30        if y > 0 then goto a
40        print x
50        goto b
60   a:   print x
70   b:   y <- x * x
80        print y
90        stop
```

Add statement 35, x <- 9000.

```
10        read x, y
20        print y
30        if y > 0 then goto a
35        x <- 9000
40        print x
50        goto b
60   a:   print x
70   b:   y <- x * x
80        print y
90        stop
```

In this case, the uses of x that can be reached by the new definition are at 40 and 70, with def-use couples [(x,35) (x,40)] and [(x,35) (x,70)]. The new dependency chains are <(x,35),(x,40)> and <(x,35),(x,70),(y,70),(y,80)>. These new chains have to be tested and make up the test requirements due to the change.

In case 3), the definition depends on use variables. These variables have definitions, and subchains can be traced backward from them to input variables. These subchains form the prefixes of the new dependency chains. The suffixes are calculated as in case 1).

Example:

```
10        read x, y
20        print x
30        print y
40        if x > 0 then goto a
50        print y
60   a:   stop
```

Add statement 45:

```
10        read x, y
20        print x
30        print y
40        if x > 0 then goto a
45        y <- x * x
50        print y
60   a:   stop
```

The newly-defined y at 45 uses the x at 45, for which the def is at 10. Since this is the only definition of x that reaches statement 45, every new dependency chain will begin with the subchain <(x,10), (y,45)>. The only suffix to be considered is <(y,45), (y,50)>. This new dependency chain is <(x,10), (x,45), (y,45), (y,50)>. Testing this chain is the added testing requirement caused by the added statement.

6.2.2 CHANGE

There are two ways that a definition can be changed: 1) a function change in terms of the same use variable, and 2) a change in variable for the same or a different function. This can be represented:

1) y <- f(x) => y <- g(x), and

2) y <- f(x) => y <- h(z).

In this case 1), the important factor is that nothing has changed in any dependency chain leading up to this statement. Only the portion following the changed statement can be affected, and then new chains will be formed only because of the data transfers that depend on the value of y.

Case 2) can be treated as a deletion of the first variable and addition of the new one. As such, it will be treated in Section 6.3 concerned with use variables.

Example of case 1):

```
10          read u, v, w, x
20          if x = 1 then goto a
30          if x = 2 then goto b
40          if x = 3 then goto c
50    a:    y <- u
60          goto d
70    b:    y <- v
80          goto d
90    c:    y <- w
100   d:    z <- y * y
110         print z
120         stop
```

Change the statement at 100:

```
10          read u, v, w, x
20          if x = 1 then goto a
30          if x = 2 then goto b
40          if x = 3 then goto c
50    a:    y <- u
60          goto d
70    b:    y <- v
80          goto d
```

```
90   c:   y <- w
100  d:   z <- y + y
110       print z
120       stop
```

The function of y at statement 100 has been changed. The value of y could have been set at statement 50, 70, or 90. However, the paths from these defs to 100 have been tested and are not changed.

To test this change, use the dependency chains

<(u,10), (u,50), (y,50), (y,100), (z,100), (z,110)>,

<(v,10), (v,70), (y,70), (y,100), (z,100), (z,110)>,

and

<(w,10), (w,90), (y,90), (u,100), (z,100), (z,110)>.

6.2.3 DELETION

It is obvious that deleting a definition from a program removes all the def-use couples that contained it. However, deleting a definition can also create def-use couples.

Consider the following example, in which statement 21 is removed.

```
10 read x, y            10 read x, y
11 goto 20              11 goto 20
12 . . .                12 . . .
13 . . .                13 . . .
14 . . .                14 . . .
15 x <- y + 3           15 x <- y + 3
16 goto 20              16 goto 20
17 . . .                17 . . .
18 . . .                18 . . .
19 . . .                19 . . .
20 read y               20 read y
21 x <- y * y
22 r <- x * x           22 r <- x * x
```

If these are the only references to x in the program, there is just one def-use couple that contains x, [(x,21) (x,22)]. In the changed program, deleting statement 21 has eliminated that path, but has "uncovered" [(x,10) (x,22)] and [(x,15) (x,22)].

The deletion of a definition is potentially the most extensive change that can be made as far as the testing that it demands. When a definition is deleted, any other definition of the same variable that is still live at the point of the deleted definition now has potential

impact on the uses of the variable. These are called "exposed" definitions.

This is an important point to study for the reason that this effect is usually not planned by the programmer. The def deleted is usually the primary focus of attention. The danger is that the programmer may forget the new def-use couples created by this act.

When a definition is deleted, a new def-use couple is created for every definition of the same variable that reaches the place of the deleted definition and every use that was reached by the deleted definition.

As an example of the impact of these exposed definitions and the new def-use couples, consider this code portion.

```
      .
      .
20  x <- q
21  goto 100
      .
30  x <- r
31  goto 100
      .
40  x <- s
41  goto 100
```

```
       .
50  x <- t
51  goto 100
       .
60  x <- u
61  goto 100
       .
       .
100 x <- y * 2
       .
200  a <- x
201  print a
       .
       .
300  b <- x
301  print b
       .
       .
400  c <- x
401  print c
       .
       .
500  d <- x
```

```
501  print d
      .
      .
600  e <- x
601  print e
```

In this example, there is a definition of x at statement 100, five other definitions of x that are live at statement 100, and five uses of x that are reached from statement 100. These are the only references to x in the program. Assume that the path to each definition of x is simple, with just one unique prefix each. The testing consists of five dependency chains, with common proper prefixes through statement 100, then to each of the five uses of x.

Should the definition of x at 100 be deleted, each of the five definitions of x can serve each of the uses of x, for a total of twenty-five new dependency chains to be tested.

6.3 USE

Changes due to uses differ from changes due to definitions, because the effects of uses on paths are different than the effects of definitions. A definition starts a new path; as shown above, it can interrupt a path. A use is not as powerful--a def-use path can go through a use. For instance, in the program segment:

```
10  read x
11  y <- x + 5
12  z <- x * x
```

the path [(x,10) (x,11)] does not interrupt [(x,10) (x,12)].

6.3.1 ADDITION

Adding a use creates new paths, as many paths as there are definitions that reach the new use.

When adding a use, the key is to find all the definitions of the use variable that reach the changed statement and combine each with the new use to form new def-use couples. These def-use couples are then new prefixes that are then matched up with the suffixes that depend on the def-use paths originating at the definition variable in the changed statement. For example, consider:

```
10       read x, y
11   a:  z <- x + 7
12       v <- z * 2
13       print v
14       y <- z + 1
15       if x > y then goto a
16       stop
```

<(x,10), (x,11), (z,11), (z,12), (v,12), (v,13)> is the only dependency chain.

Add a use by changing statement 11:

```
11   a:  z <- x + y + 7
```

This results in two new dependency chains being added to the set that has to be exercised:
<(x,10), (x,11), (z,11), (z,14), (y,14), (y,11), (z,11), (z,12), (v,12), (v,13)>, and <(y,10), (y,11), (z,11), (z,12), (v,12), (v,13)>.

6.3.2 CHANGE

In Section 6.2.2, the replacement

```
1)   y <- f(x)  =>  y <- g(x)
```

was treated. The other change that could be considered,

```
2)  y <- f(x)  =>  y <- h (z)
```

can be seen to be a simultaneous addition of the use of z, treated in

6.3.1 above, and deletion of the use of x, discussed in 6.3.3 below.

6.3.3 DELETION

No existing dependency chains are changed deleting a use, and none are added. At least one is lost, but there is nothing to test in response to this.

6.4 BRANCHES

6.4.1 UNCONDITIONAL

Adding, changing, and deleting an unconditional branch are all equivalent operations. Consider that every statement carries an implicit branch to the next, unless specifically overridden.

```
10        start
20        read x
30        print x
40        stop
```

is really

```
10          start
19          goto a
20    a:    read x
29          goto b
30    b:    print x
39          goto c
40    c:    stop
```

Changing the original program by inserting a branch statement can be seen to be changing one of the implicit branches, as in 39 below.

```
10          start
19          goto a
20    a:    read x
29          goto b
30    b:    print x
39          goto a
40    c:    stop
```

Similarly, deleting a branch is actually changing a branch to point to the next sequential instruction. For example, the program:

```
10      start
20      read x, y
30      print x
40      goto z
50      print y
60  z:  stop
```

is equivalent to:

```
10      start
19      goto a
20  a:  read x, y
29      goto b
30  b:  print x
39      goto c
40  c:  goto z
49      goto d
50  d:  print y
59      goto e
60 e: z: stop
```

Deleting statement 40 is the same as changing it to

```
40      goto d
```

Since changing a branch can be seen as adding a branch and deleting can be seen as changing a branch, adding unconditional branches alone will be treated.

An unconditional branch can lead to the deletion of a dependency chain or a creation of one. Consider the example program:

```
10        read x, y
20        z <- x * 2
30        print z
40        goto 60
50        print y
60        stop
```

The only dependency chain is <(x,10), (x,20), (z,20), (z,30)>. Adding a branch,

```
21        goto 60
```

deletes this dependency chain, since there is no longer any chain that reaches an output statement. Adding this statement to the original program:

```
31        goto 50
```

results in adding a new dependency chain:

<(y,10), (y,50)>.

6.4.2 CONDITIONAL

6.4.2.1 ADDITION

Adding a conditional branch has the same impact on testing requirements as adding an unconditional branch. There are two paths after the conditional branch is added. The first is the "true" case, the one that will be followed if the predicate is true; this is the same as if the branch had been unconditional. The other branch is the "false" branch, the one that will be followed if the predicate is false; this has already been tested.

6.4.2.2 CHANGE

Changing the conditional branch is the same as changing the equivalent unconditional branch, by similar argument.

6.4.2.3 DELETION

Deleting a conditional branch has no impact on the testing requirements of the program, since the false branch, which has been tested,is now taken unconditionally.

Any add, change, or delete in any branch, conditional or unconditional, impacts test requirements through the changes in def-use paths that result.

6.5 SUMMARY

In order to analyze software changes, the concept of the change primitive was introduced. Any change can be decomposed into a set of change primitives. Twelve candidate change primitives were proposed in this chapter, but analysis demonstrated that only five are required for further consideration. They will be treated in more detail in Chapter 7.

7. DEPENDENCY CHAIN ANALYSIS

Since an important aspect of this presentation is the eventual automatic generation and instrumentation of dependency-chain-based test cases, it is necessary to show that there are algorithms to govern the process. That is, there are consistent rules that can be applied to a set of dependency chains based on a given change. These rules then result in a new set of dependency chains that relate the program changes to test requirement changes.

This section uses the analysis of Chapter 6 and the definitions of Chapter 5 to derive the appropriate test response for each change requirement.

7.1 COMPONENTS OF DEPENDENCY CHAIN FORMATION

The following are needed for the formation of dependency chains to be tested.

| is the def-use couple formation operator, forming a def-use couple from two variable-statement pairs.

|| is the concatenation operator, building longer dependency chains from two smaller sub-chains, under the condition that the last variable-statement pair of the first sub-chain is the same as the first of the second sub-chain. Often the concatenation operator is used to build a new dependency chain from a sub-chain and a def-use couple; for this purpose, [(x,i) (x,j)] will also stand for <(x,i), (x,j)>. || takes two operands, but it is associative and can be applied successively to build up longer chains:
a || b || c || d = a || (b || (c || d)).

DefUse(s)

The set of def-use couples [(x,i) (x,j)] such that statement s is in a path between i and j.

DepChain(v,s,t)

The set of dependency chains that contain the def-use couple [(v,s) (v,t)]. Dependency chain D <u>contains</u> the def-use couple [(x,i) (x,j)] if the two variable-statement pairs (x,i) and (x,j) appear in that order together in D.

Reach(s,x)

The set of variable-statements pairs (x,j) such that a definition of variable x at j is "live" at statement s (that is, there is a def-clear path with respect to x from j to s.)

Pref(v,s,w,t,u)

The set of sub-chains that are the prefixes of the dependency chains from (v,s) to (w,t), truncated after statement u.

Source(s)

The set of variable-statement pairs (y,j) such that there is a def-use couple [(y,j) (y,s)], $y \in$ USE(s), and $y \in$ DEF(j).

Suff(v,s,w,t,u)

The set of sub-chains that are the suffixes of the dependency chains from (v,s) to (w,t), truncated before statement u.

First(d)

The variable-statement pair that is the "definition" pair (x,i) of def-use couple d, when d is written as [(x,i) (x,j)].

Last(d)

The variable-statement pair that is the "use" pair (x,j) of the

def-use couple d, when d is written [(x,i) (x,j)].

Orig(s,x)

The set of def-use couples [(x,j) (x,k)] such that [(x,j) (x,k)] ϵ DefUse(s). In other words, Orig is the set of def-use couples through s that define and use x.

BuildChain(x,i)

The set of dependency sub-chains generated from the def-use couples [(x,i) (x,j)] for all j such that x ϵ USE(j). Note that these may be complete dependency chains if i ϵ IN.

In some of the following algorithms, sets are easy to define, but individual elements are easier to use in computation. To get around this in the formal proofs, there is a notational convenience, a "while" construction, and a function "elt_of", which will select one item from a set, delete it from the set, and return it as the value of the function. It returns a special value (symbolized by "¤"), to indicate that the set is exhausted. The type returned by "elt_of" is that of the set that it is working on; applied to a set of def-use couples, it returns a def-use couple, and so on. To understand its use, consider an example using the function DefUse, defined above, which returns a set of def-use couples through s. In order to deal with these, the function "elt_of" working

on DefUse selects one def-use couple from this set, returns it as the value of the function, and deletes it from the set. Eventually the function returns a null value ("☼") when the set is depleted.

The while controls the use of the function:

```
while ([(x,i) (x,j)] = elt_of(DefUse(s)) ≠ ☼)
  do
        .
        .
    some process on x, i, and j
        .
        .
  enddo
```

Once a dependency chain is created or changed in the process, the function "collect" is used to record it in a set of dependency chains. This set is the result of the process, when it is finished.

7.2 ADDING A DEFINITION

In the following analysis, a previously existing statement s' is used to retrieve information about the program as it was before the change. The selection of s' is done in the following manner. In most

cases, s' is the statement immediately preceding the added statement s. If s is the first statement in a multi-line node, s' is the statement immediately after s. If s is a single statement node, s' is the final statement of every node that can execute before s; the analysis has to be repeated for each occurrence of s'.

In generating the new dependency chains to test, there are two cases, based on whether the added statement s ∊ IN.

7.2.1 Case I: s ∊ IN

```
while (x = elt_of(DEF(s)) ≠ ☼)              ;find a variable
 do                                          ; defined at s
 while ([(x,a) (x,t)] = elt_of(Orig(s',x)) ≠ ☼) ;find a def-use
  do                                         ; through s, with an
                                             ; origin at a def of x
  d = (x,s) | (x,t)                          ; create a new d-u couple
                                             ;   from new def, old use
  while (c = elt_of(DepChain(x,a,t)) ≠ ☼) ; find
   do                                        ; dep chain from old d-u
      c' = d || Suff(c,s')                   ; create new dep chain
    collect c'                               ; record the new dep chain
   enddo
  enddo
 enddo
```

In less formal terms, this case starts with the new definition, which must be the head of each new dependency chain. Find all the uses reached by this new definition. The new dependency chain is formed by "grafting" the new definition onto the front of the old chains.

7.1.2 Case II: s ∉ IN

```
while (x = elt_of(DEF(s)) ≠ ☼)                       ; find a variable
 do                                                   ; defined at s
 while ([(x,a) (x,t)] = elt_of(Orig(s',x)) ≠ ☼)      ; find a d-u couple
  do                                                  ; thru s' originating
                                                      ; at a def of x
  d = (x,s) | (x,t)                                   ; make new d-u couple
  while (c = elt_of(DepChain(x,a,x,t)) ≠ ☼)          ; find an old
   do                                                 ; dependency chain
    while (y = elt_of(USE(s)) ≠ ☼)                   ; find var used in def
     do
     while ((y,u) = elt_of(Reach(s,y)) ≠ ☼)          ; find reaching def
      do                                              ; of that variable
      d' = (y,u) | (y,s)                              ; create new d-u couple
      [(y,u) (y,v)] = Orig(u,y)                       ; find d-u couple thru
                                                      ; var-stmt pair
      while (c' = elt_of(DepChain(y,u,y,v)) ≠ ☼)        ; find dep chain
```

```
          do
          c''= Pref(c',n) || d' || d || Suff(c,s') ;build new dep chain
          collect c''                               ;record the new chain
         enddo
        enddo
       enddo
      enddo
     enddo
    enddo
```

This case is similar to the previous one, except that the new definition now becomes (with each use it reaches) a connecting link between the first part of old dependency chains leading to it and the last part of the dependency chains dependent on its use.

7.3 CHANGING A DEFINITION

Since none of the dependency chains are affected by this change, the reaction is to find all the dependency chains that contain the variable-statement pair that represent the changed definition, and test them. In order to find them, the first step is to find the def-use couples through the changed statement for the use variables in that

statement. From these we can find all the dependency chains that contain these def-use couples. We test these chains.

```
while ([(v,s) (v,t)] = elt_of(Orig (s,y)) ≠ ∅);find all def-use couples
  do                                              ; thru s that use y
  while (d = elt_of(DepChain(v,s,t)) ≠ ∅) ; find the
   do                                             ; dep chains thru them
   d' = d
   collect d'                                     ; record these chains
  enddo
 enddo
```

No dependency chains are created or destroyed. This process finds the dependency chains "containing" the changed definition. Note this this primitive could be consolidated to the application of a deletion of a definition and addition of a definition. However, this would be at the expense of added complexity, with no real benefit.

7.4 DELETING A DEFINITION

The impact of deleting a definition of variable x from statement s is to expose its uses to definitions that were stopped by the one deleted. The strategy is to find all the uses reached by the definition deleted, associate them with all the definitions of the same variable

that now reach those uses, and form new dependency chains around the def-use couples thus created.

```
while ((x,i) = elt_of(Reach(s,x) ≠ ☼)        ; find all defs of x
  do                                            ; that reach s
  while ([(x,j) (x,k)] = elt_of(Orig(s,x)) ≠ ☼); find d-u couples
   do                                           ; originating at def of
                                                ; x and going thru s
   o' = (x,i) | (x,k)                           ; create new d-u couples
   while ([v,t) (v,u)] = elt_of(Source(i)) ≠ ☼); find d-u leading to i
    do
    while (c = elt_of(DepChain(v,t,u)) ≠ ☼) ; find dep chains
     do                                         ; for those d-u couples
     while (c' = elt_of(DepChain(x,j,k)) ≠ ☼) ; find dep chains
      do                                        ; for d-u couples above
      d = Pref (c,i) || o' || Suff (c',k)       ; create new dep chains
      collect d                                 ; record these chains
      enddo
     enddo
    enddo
   enddo
  enddo
```

The definitions that were stopped from reaching uses by the deleted definitions can now form new dependency chains. The result could be thought of as multiple added definitions.

7.5 ADDING A USE

This section describes the situation in which an existing definition in statement s calculated in terms of one or more variables is changed to a function of a different use variable. In order to find the set of dependency chains to be tested, we find their prefixes by working back from the new use and the suffixes of the old set of dependency chains. The original use variable is y, the new is y'.

```
while ((v,t) = elt_of(Reach(s,y')) ≠ ☼)      ; find all the defs
 do                                              ; of y' reaching s
 while (p = elt_of(Orig(v,t)) ≠ ☼)           ; find d-u couples
  do                                             ; beginning at defs
  while (d = elt_of(DepChain(p)) ≠ ☼)        ; find dep chains
   do                                           ; containing d-u couples
   while (b = elt_of(DefUse(s)) ≠ ☼)         ; find dep chains
    do                                           ; thru stmt s
    while (d' = elt_of(DepChain(b)) ≠ ☼)     ; generate
     do
     c = Pref(d,s) || Suff(d',s)                 ; new dep chains
     collect c                                   ; record chains
     enddo
```

```
    enddo
   enddo
  enddo
 enddo
```

This process can be thought of as finding a whole new set of "heads" for the "tails" of the previously existing dependency chains.

7.6 ADDING A BRANCH

A new branch creates new dependency chains by grafting new suffixes to the chains going through the new branch statement.

```
while (d = elt_of(DefUse(s)) ≠ ☼)          ; find d-u couples
 do                                          ; thru stmt s
 while (c = elt_of(DepChain(d)) ≠ ☼)       ; find dep chain
  do                                         ; containing couples
  c' = Pref(c,s)                             ; get prefix of chain
  f = First(d)                               ; get var-stmt pair
  while (c" = elt_of(BuildChain(f)) ≠ ☼)   ; build sub-chain
   do                                        ; dependent on pair
   n = c' || c"                              ; create new dep chain
   collect n                                 ; record the new chain
   enddo
  enddo
```

```
enddo
```

Since the dependency chains that existed before are still the same up to the changed branch statement, this process can be thought of as grafting new "tails" onto the previously existing "heads".

7.7 SUMMARY

This chapter demonstrated the algorithms that an automated system could use to respond to any of the change primitives described in Chapter 6. These algorithms, with information stored at the time of the creation of the program, will generate the test requirements necessary to test the program change.

8. EXAMPLES

These examples attempt to illustrate the usefulness of dependency chains in focusing the testing effort on portions of a program affected by a change. Where the effects of the change are small and local to a few statements, that portion of the program is identified for scrutiny. When the effects of a change are more pervasive, more of the program is selected for testing.

8.1 EXAMPLE I

The first example is from Rapps and Weyuker [RW], a program that computes x^y.

```
10      start
20      read x, y
30      if y < 0 then goto a
40      pow <- y
50      goto b
```

```
60 a:  pow <- -y

70 b:  z <- 1

80 c:  if pow = 0 then goto d

90     z <- z * x

100    pow <- pow - 1

110    goto c

120 d: if y >= 0 then goto e

130    z <- 1 / z

140 e: answer <- z

150    print answer

160    stop
```

The variable-statement pairs in this program are:

(x,20)

(y,20)

(pow,40)

(y,40)

(pow,60)

(y,60)

(z,70)

(z,90)

(x,90)

(pow,100)

(z,130)

(answer,140)

(z,140)

(answer,150)

The def-use couples in this program are:

[(x,20) (x,90)]

[(y,20) (y,40)]

[(y,20) (y,60)]

[(pow,40) (pow,100)]

[(pow,60) (pow,100)]

[(z,70) (z,90)]

[(z,70) (z,130)]

[(z,70) (z,140)]

[(z,90) (z,130)]

[(z,90) (z,140)]

[(z,130) (z,140)]

[(answer,140) (answer,150)]

The dependency chains are:

1. <(x,20), (x,90), (z,90), (z,130), (z,130), (z,140), (answer,140) (answer,150)>

2. <(x,20), (x,90), (x,90), (z,90), (z,140), (answer,140), (answer,150)>

3. <(z,70), (z,90), (z,90), (z,130), (z,130), (z,140), (answer,140), (answer,150)>

4. <(z,70), (z,90), (z,90), (z,140), (answer,140), (answer,150)>

5. <(z,70), (z,130), (z,130), (z,140), (answer,140), (answer,150)>

6. <(z,70), (z,140), (answer,140), (answer,150)>

Paths corresponding to these chains:

1. a. 20, 40, 70, 90, 100, 130, 140, 150 *

1. b. 20, 60, 70, 90, 100, 130, 140, 150

2. a. 20, 40, 70, 90, 100, 140, 150

2. b. 20, 60, 70, 90, 100, 140, 150 *

3. 70, 90, 100, 130, 140, 150

4. 70, 90, 100, 140, 150

5. 70, 130, 140, 150 *

6. 70, 140, 150

* Infeasible

In these examples, the feasible paths were found manually. There is no manual or automated method to resolve feasibility in the general case.

A test satisfies a path if it traverses the statements of the path in the order specified with no intervening "foreign" statements. Looping requires that some subset of the statements may be repeated.

Suppose that when the program is developed, the following tests are done:

Test I: x = 2, y = 2

Path traversed: 20, 40, 70, 90, 100, 90, 100, 140, 150

Cases satisfied: 2.a., 4.

Test II: x = 2, y = -2

Path traversed: 20, 60, 70, 90, 100, 90, 100, 130, 140, 150

Cases satisfied: 1.b., 3.

Test III: x = 2, y = 0

Path traversed: 20, 40, 70, 140, 150

Case satisfied: 6.

With three tests, we have satisfied five of eight required cases. On inspection, we can see that the other three cannot be satisfied, since they depend on values that are mutually exclusive to be bound to the same variable. In 1.a., y must take on values both less than and greater than zero, with no intervening assignement statement. In 2.b., y must take on similarly conflicting values. In case 5., pow must be equal to zero, y must be less than zero, and yet the two must be equal.

The program is now considered complete and put into production(of course, these are not the only test cases run, merely the ones required by dependency chain analysis). The final compile includes pre- and post-processor routines necessary to record the internal relationships required for the analysis: for every statement, the def-use paths containing it and for every def-use path, the dependency chains containing it. We now consider two different and independent changes to the program and the corresponding changes in the test scenarios.

8.1.1 A SMALL CHANGE

Introduce a small change.

Change

```
130 z <- 1 / z
```

to

```
130 z <- 1 / x
```

In order to find the new paths, we follow the algorithm of section 7.5. This is an example of an added use.

The only def of x reaching 130 is (x,20). The only def-use couple that starts at (x, 20) is [(x,20) (x,90)]. The two dependency chains (call them "d") containing this def-use couple are:

1. <(x,20), (x,90), (z,90), (z,130), (z,130), (answer,140), (answer,150)>, and
2. <(x,20), (x,90), (z,90), (z,140), (answer,140), (answer,150)>.

There are three dependency chains (call them "d'") going through statement 130:

1. <(x,20), (x,90), (z,90), (z,130), (z,140), (answer,140), (answer, 150)>,

3. <(z,70), (z,90), (z,90), (z,130), (z,140), (answer,140), (answer, 150)>, and

5. <(z,70), (z,130), (z,140), (answer,140), (answer,150)>

The new dependency chains are generated from the prefix of the d chains above, truncated at statement 130 and the suffixes of the d' chains, truncated at statement 130. Dependency chain 2 of (d) is not involved because it does not go through statement 130. Combining chain 1. (from (d)) with 1, 3, and 5 (from (d')) gives:

1/1 <(x,20), (x,130), (z,130), (z,140), (answer,140), (answer, 150)>,

1/3 <(x,20), (x,130), (z,130), (z,140), (answer,140), (answer, 150)>, and

1/5 <(x,20), (x,130), (z,130), (z,140), (answer,140), (answer,150)>.

This one dependency chain requires us to execute the paths:

20, 30, 40, 50, 70, 80, 120, 130, 140, 150 *

20, 30, 60, 70, 80, 120, 130, 140, 150 *

20, 30, 40, 50, 70, 80, 90, 100, 110, 80, 120, 130, 140, 150 *

and 20, 30, 60, 70, 80, 90, 100, 110, 80, 120, 130, 140, 150

The only feasible path is

20, 30, 60, 70, 80, 90, 100, 110, 80, 120, 130, 140, 150.

In order to execute this path, we must execute the program with the values

$y < 0$ and any x.

Try $x = 2$ and $y = -2$.

The value of answer that prints is 1/2, not 1/4, which is the correct value. This test uncovers the error inserted by this change.

With no testing methodology at all, the programmer could choose to run just one test, Test I from Section 8.1 ($x = 2$, $y = 2$.) This produces the correct answer, but does not test the change and does not reveal the error. Another test suite would be the repetition of the

whole set of three tests from 8.1. Two of these tests are not productive in that they do not exercise the changed code. This is not a great expenditure of resources, but consider that a larger program may have a regression library of hundreds of cases, with only a few specific to a small change like this one. Many of the test cases waste the effort required to run them and evaluate their results.

8.1.2 ANOTHER CHANGE

Going back to the original program, we look at a change with more impact than the one in 8.1.1.

```
70  b:  z <- y
```

Analyzed using the analysis of Section 7.3, this process generates a much larger set of new dependency chains.

9. <(y,20), (y,70), (z,70), (z,90), (z,90), (z,130), (z,140), (answer,140), (answer,150)>
10. <(y,20), (y,70), (z,70), (z,90), (z,90), (z,140), (answer,140), (answer,150)>
11. <(y,20), (y,70), (z,70), (z,130), (z,140), (answer,140), (answer, 150)>
12. <(y,20), (y,70), (z,70), (z,140), (answer,140), (answer,150)>

The paths required by these chains:

9.a. 20, 40, 70, 90, 100, 130, 140, 150 *

9.b. 20, 60, 70, 90, 100, 130, 140, 150

10.a. 20, 40, 70, 90, 100, 140, 150

10.b. 20, 60, 70, 90, 100, 140, 150 *

11.a. 20, 40, 70, 130, 140, 150 *

11.b. 20, 60, 70, 130, 140, 150 *

12.a. 20, 40, 70, 140, 150

12.b. 20, 60, 70, 140, 150 *

Eight paths are generated by these dependency chains, but three survive after incompatible conditions are considered.

In order to test these paths, the following test cases must be considered.

9.b.:	$y < 0$ and any x	(1)
10.a.:	$y > 0$ and any x	(2)
12.a.:	$y = 0$ and any x	(3)

In response to these requirements, run the test values and receive the following results.

	x	y	calculated answer	correct answer
(1)	2	-3	- 1 / 64	1 / 8
(2)	2	3	64	8
(3)	2	0	0	1

Note that mere observation of the sample program, with the conditions on the value of y at 30 and 120 would have lead to the same or similar requirements, but this ability to define test requirements from simple observation is difficult with programs of any size.

In this example, we see a one-character change in the program that has produced requirements for the same number of tests as the original program required. The two examples drawn from this fairly simple program show that the size of the test requirements is appropriate to the intuitive feel for the "size" of the change. The first, which can be seen to have a relatively small impact on the program, required less testing than the second, which effected a more influential variable at a more critical point of the program.

8.2 EXAMPLE PROGRAM II

A slightly more complex example, a bill of materials master file update pre-edit. The appendix contains a more complete analysis.

```
1          start
2       a: read func, parent, comp, qty, um
3          if parent = 999999999 then goto z
4          if func = 0 then goto b
5          if func = 1 then goto c
6          if func = 2 then goto d
7          if func = 3 then goto e
8          if func = 9 then goto f
9          errmess <- 1
10         print func, parent, comp, qty, um, errmess
11         goto a
12      b: delparent <- parent
13         delcomp <- comp
14         goto a
15      c: addparent <- parent
16         addcomp <- comp
17         addqty <- qty
18         addum <- um
19         goto a
```

```
20    d: chgparent <- parent
21       chgcomp <- comp
22       chgqty <- qty
23       chgum <- um
24       goto a
25    e: inqparent <- parent
26       inqcomp <- comp
27       goto a
28    f: if chgparent = parent then goto g
29       goto i
30    g: if chgcomp = comp then goto h
31       goto i
32    h: chgparent <- 0
33       chgcomp <- 0
34       qty <- chgqty
35       um <- chgum
36    i: if chgparent = 0 then goto j
37       errmess <- 2
38       print chgparent, chgcomp, errmess
39       chgparent <- 0
40    j: if chgcomp = 0 then goto k
41       chgcomp <- 0
42       errmess <- 3
```

```
43        print chgparent, chgcomp, errmess
44     k: if delparent = parent then goto l
45        goto n
46     l: if delcomp = comp then goto m
47        goto n
48     m: delparent <- 0
49        delcomp <- 0
50        goto a
51     n: if delparent = 0 then goto o
52        errmess <- 4
53        print delparent, delcomp, errmess
54     o: if inqparent = parent then goto p
55        goto r
56     p: if inqcomp = comp then goto q
57        goto r
58     q: print parent, comp, qty, um
59     r: if addparent = parent then goto s
60        goto u
61     s: if addcomp = comp then goto t
62        goto u
63     t: errmess <- 5
64        print addparent, addcomp, errmess
65        addparent <- 0
```

```
66          addcomp <- 0
67          addqty <- 0
68          addum <- 0
69      u: if addparent > 0 then goto v
70          goto w
71      v: print addparent, addcomp, addqty, addum
72          addparent <- 0
73          addcomp <- 0
74          addqty <- 0
75          addum <- 0
76          goto a
77      w: print parent, comp, qty, um
78          goto a
79      z: stop
```

8.2.1. CHANGE 1

Delete statement 41.

The defs of chgcomp that reach the use at 43 are at 21 and 33.

The variable-statement pairs added are [(chgcomp, 21) (chgcomp, 43)] and [(chgcomp, 33) (chgcomp, 43)].

In order to test this change, we have to exercise the paths:

1. 2,4,5,6,20,2,4,5,6,7,8,28,36,40,42,43
2. 2,4,5,6,20,2,4,12,2,4,5,6,7,8,28,36,40,42,43
3. 2,4,5,6,20,2,4,5,15,2,4,5,6,7,8,28,36,40,42,43
4. 2,4,5,6,20,2,4,5,7,25,2,4,5,6,7,8,28,36,40,42,43
5. 2,4,5,6,20,2,4,5,8,9,2,4,5,6,7,8,28,36,40,42,43
6. 2,4,5,6,20,2,4,5,6,7,8,28,30,36,40,42,43
7. 2,4,5,6,20,2,4,5,6,7,8,28,36,37,40,42,43
8. 2,4,5,6,7,8,28,30,32,36,37,40,42,43

In order to exercise these paths, the following sets of data are input.

	func	parent	comp	qty	um
1.	2	0	1	10	1
	9	1	1	10	1
2.	2	0	1	10	1
	0	0	1	0	0
	9	1	1	10	1
3.	2	0	1	10	1
	1	0	1	0	0
	9	1	1	10	1
4.	2	0	2	10	1
	3	0	2	0	0
	9	1	2	10	2
5.	2	0	2	10	1
	5	0	2	0	0
	9	1	2	10	2
6.	2	0	2	10	1

	9	0	3	10	2
7.	2	1	2	10	1
	9	2	2	10	2
8.	2	1	2	10	1
	9	1	2	10	2

The results of tests show that there is no error from this change. It has achieved its purpose, correcting a coding error that obscured some important information supporting an error type 3, which means that there was a change put in for a missing parent/component pair.

In this case, the regression library may contain hundreds of test cases, but the changed can be tested with just eight test cases.

8.2.2. CHANGE 2

```
7.1         if func = 4 then goto ea


27.1 ea:    oldparent <- parent
27.2        newparent <- comp
27.3        goto a


77      w:  if oldparent = 0 then goto y
77.1        if oldparent = parent then goto x
77.2        goto a
77.3    x:  parent <- newparent
77.4    y:  print parent, comp, qty, um
```

In effect, adding this function calls for an almost total re-test of the whole program. To see this, consider that the addition of the branch at 7.1 calls for testing that sub-path for every dependency chain that depends on the read of a variable at statement 2 and a use of that variable anywhere after statement 7. The first twenty dependency chains satisfy this requirement; each will require at least one path through the new statements. Of course, not all of these new path requirements are unique. However, each existing function has to be re-tested, in conjunction with the new one. Again, the level of response is proportional to the "size" of the change.

9. CONCLUSIONS

This thesis has presented a structural testing technique for describing the effects of program changes on program testing requirements. The entity introduced to facilitate this technique is the dependency chain. A dependency chain is a set of sub-paths within the program, described by def-use couples, which represent places in the program that a variable receives a value and is used. The dependency chain represents all the simple paths between those places. Testing program changes is made more manageable by the use of dependency chains. The program change is described in terms of changes to dependency chains and these changes to the chains are reflected in changes to test requirements.

In order to study program changes, the concept of change primitives was introduced. These are the smallest changes that can be made to a program and serve to categorize any program change. Once the change primitives were established, algorithms were described for each change primitive, to demonstrate the process for changing the dependency chain to reflect the program change.

This research is only a beginning and several topics suggested themselves but were set aside in the interest of completion of the main topic. One that comes immediately to mind is to enlarge the concept of the def-use couple by allowing for a def-p-use couple, tracing the path between a variable definition and its use in a conditional predicate. One that would seem to have real value is a p-use-def couple, which would trace a dependency of a definition on some conditional that exercised control over whether of not a definition is even executed. Enlarging the concept of the dependency chain to allow for stored values would lead to studying interactions in a system of programs working with the same data files. The dependency chain would seem to have potential as a metric of program complexity; it would be worth determining whether a program is easier to understand and maintain if it has a smaller number of longer dependency chains, or more chains with fewer links.

REFERENCES

[Andr] W. Andrion, M. A. Branstad, and J. C. Cherniavski, "Validation, verification, and testing of computer software", *Computer Surveys*, Vol 14, No. 2, pp. 159-92, June, 1982.

[Brooks] F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975.

[Clar] L. A. Clarke, A. Podjursky, D. Richardson, S. J. Zeil, "An Investigation of Data Flow Path Selection Criteria", *IEEE Computer Society Proceedings of the Workshop on Software Testing*, pp. 23-32, July, 1986.

[Good] J. B. Goodenough and S. L. Gerhart, "Toward a theory of testing: Data selection and criteria", *Current Trends in Programming Methodology*, Vol. 2, R. T. Yeh, Ed., Prentice-Hall, pp. 44-79, Englewood Cliffs, N. J., 1977.

[Hamlet] R. G. Hamlet, "Testing Programs with the Aid of a Compiler", *IEEE Transactions on Software Engineering*, Vol SE-3, No. 4, pp. 279-90, July, 1977.

[Harr] M. J. Harrold and M. L. Soffa,"An Incremental Data Flow Testing Tool", *Proceedings of the Sixth International Conference on Testing Computer Software*, Washington, D. C., May, 1989.

[Hecht] M. S. Hecht, *Flow Analysis of Computer Programs*, The Computer Science Library, North-Holland, Inc. New York, N. Y., 1977.

[How] W. E. Howden, "Reliability of the Path Analysis Testing Strategy", *IEEE Transactions on Software Engineering*, Vol. SE-2, pp. 208-14, Sept., 1976.

[Lask] J. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, pp. 347-54, May, 1983.

[Myers] G. J. Myers, *The Art of Software Testing*, A Wiley-Interscience Publication, John Wiley & Sons, New York, 1979.

[Ntaf] S. C. Ntafos, "On Required Element Testing", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, pp. 795-803, November, 1984.

[OW] T. J. Ostrand and E. J. Weyuker, "Theories of Program Testing and Revealing Subdomains", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, pp. 236-46, May 1980

[RW] S. Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No.4, pp. 367-75, April, 1985.

[Weis82] M. Weiser, "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, pp. 446-52, July, 1982.

[Wis84] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, Vol. SE-10, pp. 352-7, July, 1984

[Wyan] L. Wyant and J. Howell, *Intercom Programming for the Bendix G-15 Computer*, Wm. C. Brown Company Publishers, Dubuque, Iowa, 1961.

[Zeil] S. J. Zeil, "Selectivity of Data-Flow and Control-Flow Path Criteria", *IEEE Transactions on Software Engineering*, Vol. SE-13, pp. 216-22, July, 1987.

APPENDIX: EXAMPLE II DETAIL

This program preedits the changes for a bill of material file update. Because of the limitations of GPL, the update transactions are imbedded in the incoming file with the master file itself. There are four fields, the assembly (parent), the part in the assembly (component), the quantity (qty) and unit of measure (um). There are four update functions (func), delete (0), add (1), change (2), and inquiry (3). The master record itself has a 9 in the function code.

```
1          start
2       a: read func, parent, comp, qty, um
3          if parent = 999999999 then goto z
4          if func = 0 then goto b
5          if func = 1 then goto c
6          if func = 2 then goto d
7          if func = 3 then goto e
8          if func = 9 then goto f
```

```
9           errmess <- 1
10          print func, parent, comp, qty, um, errmess
11          goto a
12       b: delparent <- parent
13          delcomp <- comp
14          goto a
15       c: addparent <- parent
16          addcomp <- comp
17          addqty <- qty
18          addum <- um
19          goto a
20       d: chgparent <- parent
21          chgcomp <- comp
22          chgqty <- qty
23          chgum <- um
24          goto a
25       e: inqparent <- parent
26          inqcomp <- comp
27          goto a
28       f: if chgparent = parent then goto g
29          goto i
30       g: if chgcomp = comp then goto h
31          goto i
```

```
32      h: chgparent <- 0
33         chgcomp <- 0
34         qty <- chgqty
35         um <- chgum
36      i: if chgparent = 0 then goto j
37         errmess <- 2
38         print chgparent, chgcomp, errmess
39         chgparent <- 0
40      j: if chgcomp = 0 then goto k
41         chgcomp <- 0
42         errmess <- 3
43         print chgparent, chgcomp, errmess
44      k: if delparent = parent then goto l
45         goto n
46      l: if delcomp = comp then goto m
47         goto n
48      m: delparent <- 0
49         delcomp <- 0
50         goto a
51      n: if delparent = 0 then goto o
52         errmess <- 4
53         print delparent, delcomp, errmess
54      o: if inqparent = parent then goto p
```

```
55        goto r
56     p: if inqcomp = comp then goto q
57        goto r
58     q: print parent, comp, qty, um
59     r: if addparent = parent then goto s
60        goto u
61     s: if addcomp = comp then goto t
62        goto u
63     t: errmess <- 5
64        print addparent, addcomp, errmess
65        addparent <- 0
66        addcomp <- 0
67        addqty <- 0
68        addum <- 0
69     u: if addparent > 0 then goto v
70        goto w
71     v: print addparent, addcomp, addqty, addum
72        addparent <- 0
73        addcomp <- 0
74        addqty <- 0
75        addum <- 0
76        goto a
77     w: print parent, comp, qty, um
```

```
78        goto a
79     z: stop
```

The variable-statement pairs in the program are:

(func,2)

(parent,2)

(comp,2)

(qty,2)

(um,2)

(errmess,9)

(func,10)

(parent,10)

(comp,10)

(qty,10)

(um,10)

(errmess,10)

(delparent,12

(parent,12)

(delcomp,13)

(comp,13)

(addparent,15)

(parent,15)

(addcomp,16)

(comp,16)

(addqty,17)

(qty,17)

(addum,18)

(um,18)

(chgparent,20)

(parent,20)

(chgcomp,21)

(parent,21)

(chgqty,22)

(qty,22)

(chgum,23)

(um,23)

(inqparent,25)

(parent,25)

(inqcomp,26)

(comp,26)

(chgparent,32)

(chgcomp,33)

(qty,34)

(chgqty,34)

(um,35)

(chgum,35)

(errmess,37)

(chgparent,38)

(chgcomp,38)

(errmess,38)

(chgparent,39)

(chgcomp,41)

(errmess,42)

(chgparent,43)

(chgcomp,43)

(errmess,43)

(delparent,48)

(delcomp,49)

(errmess,52)

(delparent,53)

(delcomp,53)

(errmess,53)

(parent,58)

(comp,58)

(qty,58)

(um,58)

(errmess,63)

(addparent,64)

(addcomp,64)

  (errmess,64)

(addparent,65)

(addcomp,66)

(addqty,67)

(addum,68)

(addparent,71)

(addcomp,71)

(addqty,71)

(addum,71)

(addparent,72)

(addcomp,73)

(addqty,74)

(addum,75)

(parent,77)

(comp,77)

(qty,77)

(um,77)

The def-use couples in this example are:

```
[(func,2) (func,10)]
[(parent,2) (parent,10)]
[(comp,2) (comp,10)]
[(qty,2) (qty,10)]
[(um,2) (qty,10)]
[(parent,2) (parent,12)]
[(comp,2) (comp,13)]
[(parent,2) (parent,15)]
[(parent,2) (parent,20)]
[(parent,2) (parent,25)]
[(parent,2) (parent,58)]
[(parent,2) (parent,77)]
[(comp,2) (comp,16)]
[(comp,2) (comp,21)]
[(comp,2) (comp,26)]
[(comp,2) (comp,58)]
[(comp,2) (comp,77)]
[(qty,2) (qty,17)]
[(qty,2) (qty,58)]
[(qty,2) (qty,77)]
[(um,2) (um,18)]
[(um,2) (um,23)]
```

```
[(um,2) (um,58)]
[(um,2) (um,77)]
[(errmess,9) (errmess,10)]
[(delparent,12) (delparent,53)]
[(delcomp,13) (delcomp,53)]
[(addparent,15) (addparent,64)]
[(addparent,15) (addparent,71)]
[(addcomp,16) (addcomp,64)]
[(addcomp,16) (addcomp,71)]
[(addqty,17) (addqty,71)]
[(addum,18) (addum,71)]
[(chgparent,20) (chgparent,38)]
[(chgparent,20) (chgparent,43)]
[(chgcomp,21) (chgcomp,38)]
[(chgcomp,21) (chgcomp,43)]
[(chgparent,32) (chgparent,38)]
[(chgcomp,33) (chgcomp,38)]
[(chgcomp,41) (chgcomp,43)]
[(chgparent,39) (chgparent,43)]
[(qty,34) (qty,58)]
[(qty,34) (qty,58)]
[(um,35) (um,58)]
[(um,35) (um,77)]
```

[(errmess,37) (errmess,38)]

[(errmess,42) (errmess,43)]

[(errmess,52) (errmess,53)]

[(errmess,63) (errmess,64)]

The dependency chains generated by this program are:

1. <(parent,2) (delparent,12) (delparent,53)>
2. <(comp,2) (delcomp,13) (delcomp,53)>
3. <(parent,2) (addparent,15) (addparent,64)>
4. <(parent,2) (addparent,15) (addparent,71)>
5. <(comp,2) (addcomp,16) (addcomp,64)>
6. <(comp,2) (addcomp,16) (addcomp,71)>
7. <(qty,2) (addqty,17) (addqty,71)>
8. <(um,2) (addum,18) (addum,71)>
9. <(parent,2) (chgparent,20) (chgparent,38)>
10. <(parent,2) (chgparent,20) (chgparent,43)>
11. <(comp,2) (chgcomp,21) (chgcomp,38)>
12. <(parent,2) (parent,58)>
13. <(parent,2) (parent,77)>
14. <(comp,2) (comp,58)>
15. <(comp,2) (comp,77)>
16. <(qty,2) (qty,58)>

17. <(qty,2) (qty,77)>

18. <(um,2) (um,58)>

19. <(um,2) (um,77)>

20. <(errmess,9) (errmess,10)>

21. <(errmess,37) (errmess,38)>

22. <(chgparent,32) (chgparent,38)>

23. <(chgcomp,33) (chgcomp,38)>

24. <(chgcomp,41) (chgcomp,43)>

25. <(chgparent,39) (chgparent,43)>

As an example of path requirements generated by these dependency chains, number 1 calls for execution of these paths:

1.a. 2,4,12,2,5,6,7,8,28,36,40,44,51,52

1.b. 2,4,12,2,5,15,2,6,7,8,28,36,40,44,51,52

1.c. 2,4,12,2,5,6,20,2,7,8,28,36,40,44,51,52

1.d. 2,4,12,2,5,6,7,25,2,8,28,36,40,44,51,52

1.e. 2,4,12,2,5,6,7,8,9,2,28,36,40,44,51,52

1.f. 2,4,12,2,5,6,7,8,28,30,36,40,44,51,52

1.g. 2,4,12,2,5,6,7,8,28,30,32,36,37,40,44,51,52

1.h. 2,4,12,2,5,6,7,8,28,36,40,41,44,51,52

1.i. 2,4,12,2,5,6,7,8,28,36,40,44,46,51,52

BIOGRAPHICAL NOTE

The author was born June 25, 1945 in Terre Haute, Indiana. He attended parochial grammar school in that city. He attended St. Meinrad Seminary High School in St. Meinrad, Indiana, graduating in 1963. He studied at St. Meinrad College, transferring to Rose-Hulman Institute of Technology in Terre Haute in 1965. He graduated from Rose-Hulman June 10, 1967.

He entered the U. S. Army June 12, 1967. He served in Germany and Viet Nam and left the Army in 1969.

He has worked in data processing for more than twenty years as a programmer-analyst at twelve different companies in Indiana, Illinois, Oregon, and Washington. His experience has been almost entirely on IBM mainframes. He now works as a data processing consultant.

He currently lives north of Seattle in equal partnership with his beautiful wife and a cat with an attitude.