AN APPROACH TO SOFTWARE FAULT LOCALIZATION AND REVALIDATION BASED ON INCREMENTAL DATA FLOW ANALYSIS

By

ABU-BAKR M. TAHA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1991

ACKNOWLEDGEMENTS

This work is dedicated to my parents. I would like to express my deepest appreciation to my Chairman, Prof. Stephen Yau, Cochairman Prof. Sying-Syang Liu, Prof. Randy Chow, Prof. Ronald Rasch, and Prof. Stephen Thebaut for their guidance and invaluable insight throughout this research. I am particularly indebted to Prof. Liu and Prof. Thebaut for their financial support, supportive counsel and for dedicating long hours to discussions and reviews. I am also grateful to my colleagues Roger Ogando and Kurt Johmann who made valuable contributions to this research project. Finally, I would like to thank the government of the Arab Republic of Egypt for the financial support that they provided me during my study.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF TABLES v

LIST OF FIGURES vi

ABSTRACT vii

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND 4

2.1 Data Flow Analysis 4
2.1.1 Program Control Flow Graphs 5
2.1.2 The Data Flow Analysis Problem 5
2.1.3 Data Flow Analysis Techniques 5
2.1.4 Data Flow Testing Criteria 7
2.2 Regression Testing 11
2.3 Fault Localization 12

3 THE PROPOSED APPROACH 14

3.1 The Information Structures 15
3.2 The Process 15
3.3 Conditions for Best Results 18

4 APPLICATIONS OF THE PROPOSED APPROACH 20

4.1 Application 1: Generating New Test Cases 23
4.2 Application 2: Fault Localization 24
4.3 Application 3: Reducing the Number of Test Cases to be Re-run . . . 25

5 TIME AND SPACE COMPLEXITY ANALYSIS 28

5.1 Time Complexity 28
5.2 Space Complexity 30

6 A PROTOTYPE FOR THE PROPOSED APPROACH 31

6.1 Generating Language-Based Programming Environments 32
6.2 The Synthesizer Generator 32
6.3 The Structured Editor of the Prototype 34
6.3.1 The Attribute-Grammar Model of Editing 36
6.3.2 Specification of the Editor 37
6.4 The Incremental Code Generator 44
6.4.1 Incremental Compilation Using Attributes 44
6.4.2 Generating Code Graphs Using SSL 45
6.4.3 Links and Circularities 48
6.5 The Interpreter 51
6.6 The Incremental Data Flow Analyzer 52
6.7 The Run-time Interface 54
6.8 A Sample Session 55
6.9 Related Work 59

7 INTERPROCEDURAL DEFINITION-USE DEPENDENCY ANALYSIS . 61

7.1 Introduction 61
7.2 Background 63
7.3 Approach 64
7.3.1 *Gen* and *Kill* Sets of Self-Recursive Procedures 64
7.3.2 *Gen* and *Kill* Sets of Mutually Recursive Procedures 68
7.3.3 *In* and *Out* Sets 69
7.3.4 Definition-use Chains 70
7.3.5 Aliases 71
7.3.6 Speeding up the Basic Algorithms 72
7.3.7 Discussions 80
7.4 Related Work 81

8 INTERPROCEDURAL DATA FLOW ANOMALY DETECTION 84

8.1 Introduction 84
8.2 Background 86
8.3 Approach 88
8.4 Related Work 90
8.5 The Limits of Data Flow Anomaly Analysis 91

9 CONCLUSIONS AND FURTHER STUDY 92

REFERENCES 95

BIOGRAPHICAL SKETCH 99

LIST OF TABLES

4.1 The definition-use chains required to achieve all-uses coverage. 20
4.2 The contents of Table 2 after running T1, T2, T3, and T4. 23
4.3 The untested chains after running T1, T2, T3, and T4. 23
4.4 The information that is added to Table 2 after running T5. 24

LIST OF FIGURES

2.1 Partial order of the testing criteria 10
4.1 An example program. 21
4.2 The control flow graph. 22
6.1 The block diagram of the prototype. 33
7.1 An example of a self-recursive procedure. 67
7.2 Mutually recursive procedures without aliasing. 69
7.3 Mutually recursive procedures with aliasing. 79
8.1 An example of data flow anomalies. 85

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

AN APPROACH TO SOFTWARE FAULT LOCALIZATION AND REVALIDATION BASED ON INCREMENTAL DATA FLOW ANALYSIS

By

Abu-Bakr M. Taha

May, 1991

Chairman: Stephen S. Yau
Major Department: Computer and Information Sciences

Recent work in the area of software engineering has focused on the development of sophisticated interactive environments to support the development and maintenance of software systems. In such environments, software development usually proceeds in cycles of program modification followed by testing and debugging. The testing and debugging of modified programs is a major factor contributing to the high cost of maintaining evolving software systems. To reduce this cost, program changes must be made and tested in an efficient manner. This dissertation describes a systematic approach for reducing the cost of regression testing and fault localization through the use of incremental data flow analysis. Incremental data flow analysis is used to identify the portion of a program affected by a change so that testing effort may be focused accordingly. The analysis allows the partitioning of existing test cases into relevant, nonrelevant, and invalid classes. This greatly reduces the effort associated with validating a program following modification. New algorithms for definition-use dependency analysis and data flow anomaly detection which is capable of handling recursive procedures are also presented.

CHAPTER 1
INTRODUCTION

Software systems are usually subjected to a series of modifications during both the development and maintenance phases of their life cycles. Modifications may be made to a software system for a variety of reasons. These include correction of errors not discovered during the original validation of the software, changes in the requirements of the system, and enhancements to make the system more efficient.

The cost of such maintenance activities is estimated to be as high as 30 to 80 percent of the total cost of a system [52]. Moreover, Lientz and Swanson [35], in a survey of 487 data processing organizations, reported that about half of an application staff's time is spent on maintenance, with approximately one-half staff-year of effort being allocated to maintain an average system of 23,000 source lines of code annually. The high cost of program modifications is a real problem facing software developers, yet there has been very little research that directly addresses this problem. It is also common knowledge that undesirable side effects are easily introduced into a software system that is undergoing modification. For example, in a study of changes made to a large Centrex-type telephone switching system, Collofello and Buck found that as many as 72 percent of the discovered defects were associated with side effects in unchanged portions of the system [13].

The task of modifying a program to correct a detected fault may be decomposed into three subtasks: *fault localization*, *fault repair*, and *regression testing. Fault localization* is the process of identifying the cause of the anomalous behavior and is considered to be the most difficult and time consuming of the three [50]. *Fault repair* involves modifying the program to eliminate the cause of the discrepancy. *Regression testing* is the selective testing carried out to verify that the modification has not caused adverse side effects and that the fault has been corrected [27]. Typically, regression testing is done many times throughout the life span of a product. For products which have many releases, the total effort in performing regression testing may exceed that of testing during initial development. Therefore, it is essential to have an efficient strategy for regression testing. Effective regression testing involves confining the testing to the affected parts of the program by selecting only the relevant test cases from the existing test suite. This dissertation introduces a systematic approach based on incremental data flow analysis for carrying out each of these tasks.

Data flow analysis has long been a basic tool in building optimizing compilers. It has also been used in detecting program anomalies [40], and in software testing [33, 41]. With the advent of *incremental* data flow analysis techniques [47, 49, 57], additional applications to software engineering have become possible [23, 29]. This dissertation introduces one of them: a systematic approach to increase the efficiency of regression testing and fault localization. Incremental data flow analysis is used to track the changes in *definition-use paths* (see Chapter 2 below) as a program is modified. Modified and newly introduced definition-use paths determine the portion of the program that needs to be retested. Where needed, new test cases that test the affected part of the program can be synthesized. Modified and deleted definition-use chains are used to determine which existing test cases should be rerun. Retesting of the whole program can then be avoided by rerunning only these test cases and new ones to test the modified and/or added definition-use chains, saving time and effort. When an error is detected by a test case, the definition-use paths exercised by it and *not* exercised by any other test case are used to help in locating the fault. If the number of these definition-use paths is large, and the user is not able to determine the exact location, additional test cases can be run to further localize the fault.

The remainder of the dissertation is organized as follows: Chapter 2 provides a brief overview of data flow analysis techniques and the data flow testing criteria used in the following chapters. Chapter 3 introduces a new approach to increase the efficiency of regression testing and fault localization using incremental data flow analysis. Chapter 4 shows how the new approach is applied by way of an example. Chapter 5 discusses the time and space complexity of the new approach. Chapter 6 outlines the implementation of an interactive environment designed to demonstrate the proposed approach. Chapter 7 introduces a new algorithm for interprocedural definition-use dependency analysis. The object of this algorithm is to extend our approach to handle programs with more than one procedure in it. The algorithm introduced in this chapter can handle programs with recursive procedures but is not incremental. To be useful for our approach an incremental version of the algorithm has to be devised. This will be the subject of our future research. Chapter 8 contains a new approach to data flow anomaly detection. Existing data flow anomaly detection algorithms do not handle programs with recursive procedures. The new algorithm does. Conclusions and further study are discussed in Chapter 9.

CHAPTER 2
BACKGROUND

This section summarizes some relevant background information on data flow analysis and fault localization.

2.1 Data Flow Analysis

During program execution, there are three basic actions associated with individual variables: *def*, when a value is assigned to a variable; *ref*, when a variable's value is used; and *undef*, when a variable becomes unavailable (e.g., a local variable on exit from a subprogram). The basic principle of data flow analysis is to study the sequence of such actions on variables along program paths. The original motivation for such work was to improve the efficiency of optimizing compilers. However, this emphasis shifted somewhat with the realization that anomalous sequences of actions (e.g., *undef* followed by *ref*, *def* followed by *undef*, and *def* followed by *def*) could be detected by statically scanning the program source code. While *data flow anomalies* do not necessarily imply a program error, they are clearly worth noting. Early work in this area, particularly that of Fosdick and Osterweil [20], resulted in the *DAVE* software verification tool [40]. Shortly afterwards, Huang showed that the presence of data flow anomalies could be detected through program instrumentation and dynamic execution [26]. More recently, data flow analysis has been used to formulate strategies for the construction and evaluation of test data [33, 41].

2.1.1 Program Control Flow Graphs

In data flow analysis, a directed graph $G = (N, E, n_0)$ called a *control flow graph* is used to represent the program under consideration where N is a set of nodes, E is a set of edges, and n_0 is the initial node in the graph. Each *node* in the graph represents a *basic block*, defined as a maximal sequence of statements where, if any statement in the sequence is executed, all statements in the sequence are executed. Each *edge* in the graph represents a possible transfer of control between two nodes. A *path* is a finite sequence of nodes connected by edges. A *complete path* is a path whose first node is an entry node of the program and whose last node is an exit node.

2.1.2 The Data Flow Analysis Problem

While there are major differences among the algorithms developed for data flow analysis, there is also much similarity. There have been several survey works, notably Kennedy [30] and Ryder and Paull [48], that compare various features of the different data flow algorithms. Each of the algorithms attempts to solve essentially the same problem: Given a control flow graph, the object is to discover the nature of the data flow, *i.e.* which definitions of program quantities can affect which uses, within the program.

2.1.3 Data Flow Analysis Techniques

The gathering of information to solve data flow problems is accomplished in two phases. The program is subdivided into basic blocks, possible block to block transfers are noted, and program loops are found. This phase is known as *control flow analysis.* Next the information about how uses and definitions relate to one another is gleaned in the *global data flow analysis phase.*

Classical data flow analysis techniques can be loosely characterized by the following paradigm:

1. Partition the program into small sections of code. These sections are normally single entry single exit sections of assignment statements called *basic blocks.*

2. Collect local information about each basic block. This information consists of which variables are defined (*i.e.* given values) and which variables are used (*i.e.* referenced) in the block.

3. Build a graph of the program. The nodes of the graph typically correspond to the basic blocks. The edges represent possible transfer between the nodes during execution.

4. Find an ordering for the nodes in the flow graph. This step is where most methods differ from one another.

   In many algorithms this and the next steps are combined. The ordering used in this step is found by doing reductions on the flow graph. As sets of nodes are reduced, the information is propagated for that local area. The common feature of these algorithms is that a single ordering is found. Information in the form of single bit vector is propagated using this ordering.

5. Propagate the local information for each node to the other nodes in the flow graph. This propagation step is done for all variables in parallel. The propagation proceeds using the ordering established in the previous step. The propagation continues until the information stops changing. Several techniques that establish the ordering of the nodes in the graph have the property that after a fixed number of passes the information will have reached a fixed point.

2.1.4 Data Flow Testing Criteria

Rapps and Weyuker [41] divide the *ref* category of possible actions on variables into two classes: *c-use*—when the variable reference occurs in a computation and *p-use*—when the variable reference occurs in a predicate expression. The following examples illustrate the concept:

1. Let *exp* be an arithmetic expression containing the variables $x_1, ..., x_n$, then the assignment statement y := *exp* contains *c-uses* of $x_1, ..., x_n$ followed by a definition of y.

2. The input statement read($x_1, ..., x_n$) contains definitions of $x_1, ..., x_n$.

3. The output statement write($x_1, ..., x_n$) contains *c-uses* of $x_1, ..., x_n$.

4. Let *exp* be a Boolean expression containing the variables $x_1, ..., x_n$, then the conditional statement if *exp* then ... contains *p-uses* of $x_1, ..., x_n$.

Let x be a variable occurring in a program. A path $(n_0, \ldots, n_t), t \geq 1$, containing no definitions of x in the nodes $n_1, ..., n_{t-1}$ is called a *definition-clear path* with respect to x from node n_0 to node n_t. A node n has a *global definition* of a variable x if it has a definition of x and there is a *definition-clear* path from node n to some other node containing a *c-use*, or edge containing a *p-use*, of x.

Rapps and Weyuker [41] define a family of criteria for test case selection based on data flow analysis. These criteria are *all-nodes*, *all-edges*, *all-p-uses*, *all-defs*, *all-c-uses*, *all-c-uses/some-p-uses*, *all-p-uses/some-c-uses*, *all-uses*, *all-du-paths*, and *all-paths*. Precisely, letting P be a set of complete paths through a program flow graph G, then

1. P satisfies the *all-nodes* criterion if every node of G is included in P.

2. P satisfies the *all-edges* criterion if every edge of G is included in P.

3. P satisfies the *all-defs* criterion if for every node i of G and for every variable x which has a global definition in i, P includes a *def-clear path* with respect to x from i to some node or edge where x is used. That is every global definition must be used at least once.

4. P satisfies *all-p-uses* criterion if for every node i and every variable, x, in *def(i)*, P includes a *def-clear* path with respect to x from i to all edges where x is used.

5. P satisfies the *all-c-uses/some-p-uses* criterion if for every node i and every variable x in *def(i)*, P includes some *def-clear path* with respect to x from i to every node where x is used; if no such node exists, then P must include a *def-clear* path with respect to x from i to some edge where x is used.

6. *All-p-uses/some-c-uses* criterion is defined in a similar manner.

7. P satisfies the *all-uses* criterion if for every node i and every variable x which has a global definition in i, P includes *def-clear paths* with respect to x from i to each node or edge where x is used. Thus, P must include a path from every global definition to each of its uses.

8. P satisfies the *all-du-paths* criterion if for every node i and every variable x which has a global definition in i, P includes every *du-path* with respect to x. Thus, if there are multiple *du-paths* from a global definition to a given use, they must all be included in paths of P.

9. P satisfies the *all-complete-paths* criterion if P includes every complete path of G. Note that programs which are represented by graphs containing loops may contain an infinite number of complete paths.

The criteria *all-nodes* (statement coverage) and *all-edges* (branch coverage) are often used in program testing, despite the fact that it is well known that they are weak criteria. Certainly they represent necessary conditions, for if some portion of the program has never been executed, one would not in general feel confident about its behavior. A similar intuition motivated the definition of the *all-defs* criterion, since even if every statement and branch are executed, if the result of some computation has never been used, one would have little evidence that the intended computation has been performed.

The criteria *all-p-uses*, *all-uses*, and *all-du-paths* have been shown to be more powerful than branch testing in the sense that any set of paths that satisfy any of these criteria will also satisfy the branch testing criterion [41].

The family of data flow-based testing criteria is partially ordered by strict inclusion as shown in Figure 2.1 [41]. Criterion C_1 includes criterion C_2 if for every data flow graph G, any set of complete paths of G that satisfies C_1 also satisfies C_2. In Figure 2.1 we write $C_1 \rightarrow C_2$ when criterion C_1 includes C_2. Clarke *et al.* [12] have shown the relationship of the criteria defined by Laski and Korel [33] and Ntafos [39] to the data flow criteria. In addition, Frankl and Weyuker have extended data flow testing criteria to feasible testing criteria by requiring the test data to exercise only those paths which are executable [22].

When selecting a testing criterion, there is, of course, a tradeoff involved. The stronger the selected criterion, the more closely the program is scrutinized in an attempt to locate program faults. A weaker criterion, on the other hand, will usually be satisfied with fewer test cases. The decision as to which criterion to use depends on several factors, including the size of the program, time and cost restrictions, and the consequence of errors.

$$\begin{tikzcd}
& \text{ALL-COMPLETE-PATHS} \arrow[d] & \\
& \text{ALL-DU-PATHS} \arrow[d] & \\
& \text{ALL-USES} \arrow[dl] \arrow[dr] & \\
\text{ALL-C-USES/SOME-P-USES} \arrow[dr] & & \text{ALL-P-USES/SOME-C-USES} \arrow[dl] \arrow[d] \\
& \text{ALL-DEFS} & \text{ALL-P-USES} \arrow[d] \\
& & \text{ALL-EDGES} \arrow[d] \\
& & \text{ALL-NODES}
\end{tikzcd}$$

Figure 2.1. Partial order of the testing criteria

2.2 Regression Testing

Software maintenance involves changing programs as a result of errors, or a change in the user requirements. It is common knowledge that many of the errors appearing in production software have not arisen from the original implementation, but have accidently been incorporated during the post-release modifications, producing unintended side effects. In order to combat such problems, the original implementation of software products should include thorough sets of test cases to exercise all the functional aspects of the programs, together with the capability of retaining and extending these test cases during the software life cycle. Regression testing is the name given to the process of retesting software after modification. A major task in regression testing is to determine which existing test cases should be rerun. Current regression testing tools do not, however, provide any mechanisms for automatically determining which subset of the stored test cases should be rerun after code modifications have been made. Thus, to ensure the verification of the modified segments of code, the user is advised simply to rerun the entire set of existing test cases, or intuitively/randomly select test cases which will exercise the main program features to provide a degree of confidence in the correct operation of the modified software. This approach, of course, can be very wasteful of both time and resources and may frequently be totally impractical for software with a large set of existing test cases.

Leung [34] associate a bit vector with each node in the control graph. If a test case i traversed a node, then the i^{th} bit of the node's bit vector is set to 1. When a node is modified, deleted, or split, the corresponding bit vector determines the test cases that should be rerun.

Another approach, based on data flow testing and incremental data flow analysis, was proposed by Harrold and Soffa [23]. In this approach, the control flow graph is extended to support the needed history information. For each definition in a node in the graph, the nodes and edges that use the definition are attached. In addition, a list containing the nodes where each variable is defined and a list of the definition-use chains that are used to meet the adequacy criterion are maintained. After a program change, the deleted definition-use chains are used to determine the test cases which should be rerun. This approach is similar to Leung's method when the *all-nodes* coverage criterion is used. We extend Harrold and Soffa's approach to help the user in generating new test cases and localizing program errors.

2.3 Fault Localization

Software development, as currently practiced, is a complex and error prone process. As a result, a significant amount of time is spent in debugging. The debugging process itself consists of two activities: fault localization and fault repair. In most situations, much of the debugging effort is associated with fault localization [50].

The fact that software testing and debugging are related is evident and has been formally established [1]. Most existing debugging methodologies that use testing information, however, exploit only information based on the test case that revealed the error and ignore past test cases that executed correctly [2, 31, 32]. This is because of the lack of an environment which maintains the correspondence between the previous test cases and the text of the modified programs. By storing the test cases along with the definition-use chains covered by them and by using incremental data flow analysis to track down the changes in the definition-use chains during program modification, this correspondence can be maintained. When a test case reveals an error, its execution path may be used to guide the search for the fault location. The execution paths of previous test cases may also be used to narrow the scope of the search.

The idea of using the knowledge of existing test cases in fault localization was proposed by Collofello and Cousins [14]. Their approach is based on the analysis of decision-to-decision-path (DD-path) executions. A *DD-path* is a section of straight-line code between predicates in a program—or, more simply, a basic block. The approach suggests that a set of test cases that executes correctly can be utilized to locate DD-paths that contain faults in an incorrect execution path. To support this, execution path information consisting of the DD-paths traversed by each test case is recorded. Various heuristics are applied to compare the execution path of a test case that detects an error with those that do not, in an attempt to locate the DD-paths on the incorrect path associated with the fault. One such heuristic is based on the hypothesis that a DD-path traversed by an error-revealing test case which has not been traversed by successfully run test cases in the database is likely to be associated with the fault. Ten such heuristics are given by Collofello and Cousins [14], and their relative effectiveness in fault localization have been demonstrated using statistical analysis.

Chapter 3 introduces a systematic approach based on incremental data flow analysis for locating faults once their presence is detected during testing. In this approach the definition-use chains that are covered only by the test case which revealed the error are used to help locate the fault.

CHAPTER 3
THE PROPOSED APPROACH

Our approach uses incremental data flow analysis to mitigate the two main problems of program modification and re-validation. One is the fault localization problem and the other is the test suite maintenance problem. In the fault localization problem, we are given a test case that detects a discrepancy between the actual and intended output of a program and we wish to determine which program statements are associated with that discrepancy. The test suite maintenance problem is concerned with developing new test cases as well as partitioning existing test cases into three sets: (1) the set whose members are relevant to the modification—these cases will exercise paths along which changes have been made, if rerun; (2) the set whose members are not relevant to the modification—these cases will exercise paths along which no changes have been made, if rerun; and (3) the set of test cases which are no longer valid, due to changes in program functionality, test case format, etc. These sets are called *retestable, reusable, and obsolete* test cases respectively [34].

Given a solution to the test suite update problem, regression testing is performed as follows: (1) remove the invalid test cases from the original set of test cases; (2) test the modified program using the test cases which are relevant to the modification; and (3) test the modified program using the new set of test cases. If errors are detected, locate and repair the faults. Repeat the process until no errors are detected.

3.1 The Information Structures

Our approach is closely tied to the use of two basic information structures. The first—which we will refer to as "Table 1"—contains the set of *definition-use chains* in a program that must be covered by test cases in order to achieve all-uses coverage. A definition-use chain is represented by a tuple of three elements: a variable name, a number corresponding to a basic block in which the variable is defined, and a number corresponding to a basic block in which the variable is used. Each tuple describes a definition-clear path with respect to the named variable from the definition block to the use block. To make the presentation simpler, each p-use is represented by a c-use in the node that is a target for the edge associated with the p-use.

The second structure—which we will refer to as "Table 2"—contains the following information for each test case:

1. Input value(s), observed output value(s), and a flag indicating whether or not the observed output is correct. (The latter is represented by the field "OK" in Table 4.2.)

2. The definition-use chains covered by each test case.

3.2 The Process

Given a program, the suite of test cases used to validate it, and a specified modification,

1. If Tables 1 and 2 do not already exist, generate them as follows:

   (a) Generate the set of definition-use chains that should be covered to achieve all-uses coverage. This can be done by applying a data flow analysis algorithm such as that given by Allen and Cocke in [4] to determine the set of variable definitions that may reach each node in the control flow graph. The set of definitions reaching a particular node, together with the set of variables used in that node or on the edges leading to that node, can be used to determine the set of chains from other nodes to it. Store the chains in Table 1[1].

(b) Rerun each test case. Store the corresponding definition-use chains covered by it in Table 2.

(c) Delete the definition-use chains covered from those listed in Table 1. Table 1 now lists only those chains which would need to be covered in order to achieve all-uses coverage. Note that complete coverage is not required for the proposed approach and may not even be possible due to the existence of infeasible paths.

2. As the program is being modified, an incremental data flow analysis algorithm [47, 57] is used to maintain the set of reaching definitions to each node in the control flow graph. The deleted, affected, and newly created definition-use chains in the program can be determined from the changes in the reaching definitions. Note that this task can be almost impossible without the use of incremental data flow analysis. A chain (v, n_1, n_2) is *deleted* if (a) all definitions of v have been removed from n_1; (b) all uses of v have been removed from n_2; or (c) a definition of v has been inserted such that there does not exist any definition-clear path with respect to v from n_1 to n_2. A chain (v, n_1, n_2) is *affected* if the value of v on exit from node n_1 could be affected by a change in the node[2]. A chain (v, n_1, n_2) is *created* if (a) a new definition of v has been

[1]In the remainder of this dissertation, the expression "a variable used in a node" will refer to a c-use in that node or a p-use on the edges leading to that node.

[2]This could come about in three ways: (a) a new assignment statement is created for v; (b) the expression part of an assignment to v is changed; or (c) a variable y is defined at statement S_1 and

made in n_1 and v is used in n_2; (b) a new use of v has been made in n_2 and v is defined in n_1; or (c) v is defined in n_1 and used in n_2 respectively, and a definition-clear path with respect to v from n_1 to n_2 has been created due to program changes.

3. Remove the deleted and add the new and affected definition-use chains to Table 1. These are chains which must be covered in order to achieve all-uses coverage of the modified program.

4. Test cases in Table 2 that previously covered an affected or deleted definition-use chain are classified as relevant to the program modification and are rerun. Update Tables 1 and 2 accordingly.

Note that if a test case should fail during this process, the user must determine if an error has been revealed or if the test case is no longer valid due to a change in program specification. If it is found to be invalid, it should either be modified or deleted from the test suite.

If it is determined that the level of coverage provided by the test suite is no longer adequate, information in Tables 1 and 2 may be of help in developing new test cases. Since Table 1 contains the definition-use chains which have not yet been covered, it may be desirable to identify new test cases which will cover these chains, when such test cases exist. The potentially difficult task of identifying input data which will cover a given chain may be simplified by searching Table 2 for a test case with an execution path that includes a subpath from the start node to the definition node of the chain under consideration. In other words, if the chain under consideration is (v, n_1, n_2), Table 2 is searched for a pattern of the form $(v, n_1, *)$ where $*$ matches

later used to define v at statement S_2 where y is not redefined between S_1 and S_2 and the expression part of S_1 has been modified, or y has been redefined between S_1 and S_2.

any node number. If found, the user may take the input values for that test case as a starting point in his search for input values for the new one by changing one value at a time.

When, in the process proposed, a test case is found to reveal an error, a simple heuristic would suggest that the fault is likely to be associated with a definition-use chain exercised solely by this test case. These chains can easily be determined by intersecting those exercised by the revealing test case with those listed in Table 1. If the number of such chains is large, the user may wish to proceed with more testing before employing the strategy. Clearly, as the number of chains covered by successful test cases increases, the number of chains exercised by the revealing test case which have not already been covered will tend to decrease.

3.3 Conditions for Best Results

The proposed approach is particularly useful when the following conditions hold:

1. The program being modified was previously validated using a "good" set of test cases, where goodness is related to the coverage measure (e.g., definition-use chains) employed in the analysis. If the coverage provided by an existing test suite is found to be unsatisfactory with respect to the chosen criterion, the programmer may design additional test cases to improve the level of coverage.

2. The program is being modified to correct a revealed (program) error. Since program requirements remain unchanged in this situation, all of the test cases should remain valid after the modification is made.

3. The program being modified is a single, stand-alone procedure, with no calls to subroutines with side effects. An incremental inter-procedural definition-use dependency analysis algorithm is needed to allow the proposed approach to

handle multi-procedure programs. Chapter 7 introduces a new algorithm that could be helpful in that endeaver.

CHAPTER 4
APPLICATIONS OF THE PROPOSED APPROACH

We illustrate the potential usefulness of our approach by applying it to the program shown in Figure 4.1, which is specified to compute $\sqrt{p}$, for $0 \leq p < 1$ to accuracy e; where $0 < e \leq 1$. Figure 4.2 shows the corresponding control flow graph. The program contains a fault; the statements of Node 5 in Figure 4.2 should be interchanged. This example will be used later to illustrate the usefulness of our approach in discovering and locating program faults.

Now, assume we have a test suite consisting of four cases, T1, T2, T3, and T4. The values for the input variables p and e are given in Table 4.2 for each. We begin by employing standard data flow analysis to determine the definition-use chains that must be covered to achieve all-uses coverage. These chains are shown in Table 4.1. The results of running the four test cases are shown in Table 4.2.

Table 4.1. The definition-use chains required to achieve all-uses coverage.

All definition-use chains in the program
(c, 1, 2), (c, 1, 3), (c, 1, 4), (c, 1, 5), (c, 1, 7), (c, 4, 3), (c, 4, 4), (c, 4, 5), (c, 5, 3), (c, 5, 4), (c, 5, 5), (d, 1, 3), (d, 1, 6), (d, 3, 3), (d, 3, 5), (d, 3, 6), (e, 1, 3), (e, 1, 6), (t, 3, 4), (t, 3, 5), (x, 1, 3), (x, 1, 5), (x, 1, 6), (x, 5, 3), (x, 5, 5), (x, 5, 6)

```
read(p, e);
d := 1.0;
x := 0.0;
c := 2 * p;
if c < 2 then
  begin
    while d > e do
      begin
        d := d / 2.0;
        t := c - (2 * x + d);
        if t < 0 then
          c := 2 * c
        else
          begin
             x := x + d;
             c := 2 * (c - (2 * x + d))
          end;
      end;
    write(x);
  end
else write(-1);
```

Figure 4.1. An example program.

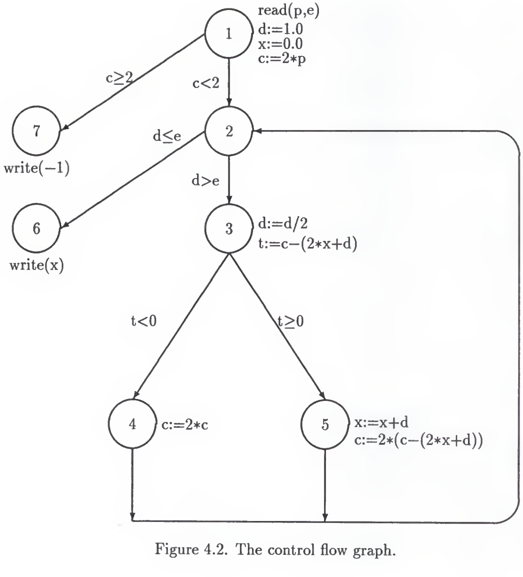

Figure 4.2. The control flow graph.

Table 4.2. The contents of Table 2 after running T1, T2, T3, and T4.

Test case	Inputs p	Inputs e	Output	OK	Definition-use chains covered
T1	2.0	.05	-1.0	yes	(c, 1, 7)
T2	0.5	1.0	0.0	yes	(c, 1, 2), (d, 1, 6), (e, 1, 6), (x, 1, 6)
T3	.16	.3	.25	yes	(c, 1, 2), (c, 1, 3), (c, 1, 4), (c, 4, 3), (c, 4, 5), (d, 1, 3), (d, 3, 3), (d, 3, 5), (d, 3, 6), (e, 1, 3), (t, 3, 4), (t, 3, 5), (x, 1, 3), (x, 1, 5), (x, 5, 6)
T4	.36	.3	0.5	yes	(c, 1, 2), (c, 1, 3), (c, 1, 5), (c, 5, 3), (c, 5, 4), (d, 1, 3), (d, 3, 3), (d, 3, 5), (d, 3, 6), (e, 1, 3), (e, 1, 6), (t, 3, 4), (t, 3, 5), (x, 1, 3), (x, 1, 5), (x, 5, 3), (x, 5, 6)

Table 4.3. The untested chains after running T1, T2, T3, and T4.

The untested definition-use chains
(c, 5, 5), (c, 4, 4), (x, 5, 5)

Table 4.3 indicates the contents of Table 1 after running the test cases T1, T2, T3, and T4, which are the definition-use chains that are not covered yet. Since Table 1 is not empty, new test cases should to be selected to cover these chains.

4.1 Application 1: Generating New Test Cases

The aim in generating new test cases here is to exercise the definition-use chains (c, 4, 4), (c, 5, 5), and (x, 5, 5). To cover (c, 4, 4), we need a test case with an execution path which contains the sub-path (4, 2, 3, 4), so that variable c may be defined and then used later in Node 4. As a first step, we look for an existing test case that has an execution path which contains a definition of the variable c in Node

Table 4.4. The information that is added to Table 2 after running T5.

Test case	Inputs		Output	OK	Definition-use chains covered
	p	e			
T5	.04	.3	0.0	yes	(c, 1, 2), (d, 1, 3), (e, 1, 3), (x, 1, 6), (x, 1, 3), (c, 1, 3), (t, 3, 4), (c, 1, 4), (d, 3, 3), (c, 4, 4), (d, 3, 6), (e, 1, 6)

4. This is done by searching Table 2 for a pattern of the form $(c, 4, *)$. That test case is T3, which has a definition-use chain (c, 4, 3) (or (c, 4, 5)). Now, we change the values of p and e to force the variable c to be used in Node 4. After a few attempts, we find that the values 0.04 and 0.3 for p and e, respectively, will work. We call the new test case T5, and run it. Table 4.4 shows the information that should be added to Table 2 after running T5. Table 1, which contains the chains that need to be covered, now contains only (c, 5, 5) and (x, 5, 5). To cover these chains, we need a test case that executes the sub-path (5, 2, 3, 5). Note that an existing test case with the variable c defined in Node 5 (or the variable x defined in Node 5) is T4. By changing its p value to 0.81 and running the new case, an error is revealed. We call the new test case T6, and add it to the test case set.

4.2 Application 2: Fault Localization

As discussed in Section 2.3, fault localization can be a very time consuming process [50]. In the previous section, we outlined a strategy for fault localization based on incremental data flow analysis. In this section, we illustrate the strategy using the example program.

Since test case T6 revealed an error, our heuristic suggests that the definition-use chains covered solely by T6 are good candidates for initial analysis. These chains are (c, 5, 5) and (x, 5, 5). The statements associated directly with these chains are those contained in Node 5 of the control flow graph (see Figure 4.2). Recall that the error introduced in the program was interchanging the two assignment statements in that node.

While this example illustrates our basic strategy for fault localization, more study is clearly needed to assess its usefulness in application. We believe, however, that it will be more useful than a similar approach shown to have some promise [14] which is based on decision-to-decision-path (DD-path) analysis. Applying DD-path analysis in the strategy illustrated above will produce suspect basic blocks, whereas applying the proposed approach produces both suspect basic blocks and suspect data flows between those basic blocks. The intuition is that an error has been revealed which is associated with a definition-use chain uniquely covered by this test case. This additional information should provide users with more detailed, if still circumstantial, evidence concerning the error.

Another factor affecting the usefulness of the strategy is the level of coverage achieved before the error is revealed. As this level increases, the number of chains covered solely by the revealing test case will tend to decrease, thus reducing the number of chains under suspicion.

The integration of testing and debugging activities through the application of incremental data flow analysis techniques appears to be one of the most interesting approaches in this area. More work is obviously needed to assess its usefulness and limitations.

4.3 Application 3: Reducing the Number of Test Cases to be Re-run

Well managed software development houses usually have sets of test cases that can be used to validate each new version of a system. However, current regression testing tools do not provide efficient mechanisms for automatically determining the subset of cases which should be rerun after program modifications. Since rerunning all test cases is often impractical, programmers are typically left to rerun randomly chosen subsets of cases which are either related to the function being modified, or which exercise modules or procedures believed to be affected by the modification. Even if there is time to run all test cases, it would be desirable to know which are most likely to reveal errors so that they may be run first.

The proposed approach provides a natural mechanism for automatically identifying test cases which should be rerun following program modifications. They are the cases associated with definition-use chains which are deleted or affected during program modification.

To illustrate the concept, consider the interchanging of the two statements in Node 5 of the example program in order to correct the error revealed by test case T6. Interchanging the statements has no effect on the final value of x, but variable c may have a different value on exit from the node. This is because the statement which assigns a value for c is *data dependent* on the statement which define x. Therefore, the definition-use chains (c, 5, 3), (c, 5, 4), and (c, 5, 5) are affected. This determination is made possible by performing the data flow analysis incrementally [47, 57]. The set of affected definition-use chains determines the test cases that should be rerun. By searching Table 2 (Tables 4.2 and 4.4) for test cases that were used to cover these chains, we find that T3 and T4 should be rerun. Obviously, test case T6, which revealed the error, should be rerun also. After rerunning these three test cases, all-uses coverage is satisfied. For our small program example, 3 out of 6 test cases are rerun. For programs with a large number of execution paths, most changes would affect a small fraction of the paths, and hence a smaller percentage of the test cases would need to be rerun. Note that if *all-nodes* coverage was used this approach would degenerate to the approach given by Leung [34].

As a final note, we point out that the proposed approach can also be helpful in maintaining existing programs when no previously developed test cases are available. When a program is modified to enhance performance or to add new capabilities, testing effort should be focused on the affected portions of the program. To isolate these affected portions, data flow analysis is carried out before modifications are made to identify all definition-use chains. As the program is modified, new definition-use chains that either lie wholly in the modified portion of the program or extend from the modified to the unmodified portion, or vice versa, will be introduced and can be identified. By developing test cases to cover these new chains, one may focus the testing on affected portions of the program.

CHAPTER 5
TIME AND SPACE COMPLEXITY ANALYSIS

In this section, the time and space complexities of the proposed approach are discussed.

5.1 Time Complexity

The time complexity of each step in the proposed approach is as follows:

Incremental analysis: As a program is modified, an incremental data flow analysis algorithm is used to determine the deleted, affected, and new definition-use chains. The complexity of such algorithms is $O(N)$, where N is the number of the basic blocks in the program [57].

Updating Table 1: For simplicity, let us assume that on average, program modifications result in changes to a fixed number of nodes, independent of the total number of nodes, N, in the control graph of the program.[1] Thus, changes to, or deletions of, variable definitions can occur in at most a fixed number of basic blocks. Since all variable uses occur in at most N blocks, the number of definition-use chains that may be deleted or modified as a result of a change is $O(vN)$, where v is the number of variables used in the program.

Initially, Table 1 contains the definition-use chains that should be covered to achieve all-uses criterion. Let the number of these chains be K.[2] Therefore, the

[1]Actually, the average number of nodes changed per program modification probably increases slowly with the number of nodes in the program. See, for example, [7].

[2]In the worst case $K = O(N^2)$ [21]. However, for real programs, K will be much smaller.

time required to delete or add a definition-use chain to Table 1 is $O(\log(K))$—assuming that the Table is sorted.[3] Since we have at most $O(N)$ definition-use chains, the time required to update Table 1 is $O(N\log(K))$.

Updating Table 2: Two operations are required to update Table 2: determining the relevant test cases and inserting new test cases. Let the number of test cases in Table 2 be T. Each test case is associated with at most K definition-use chains. To determine which test cases are relevant to a program modification, it is necessary to search through each test cases in Table 2 for each deleted or modified definition-use chain. Therefore, the time complexity to determine the relevant test cases is $T * O(N) * O(\log(K)) = O(TN\log(K))$.

When a new test case is inserted in Table 2, the definition-use chains related to this test case must be sorted. Since there are at most $O(K)$ such chains, the time required to sort them is $O(K\log(K))$. Assuming the maximum number of new test cases to be inserted is C, the time complexity to update Table 2 is $O(TN\log(K) + CK\log(K))$.

Based on this analysis, the time complexity of the proposed approach is $O(TN\log(K) + CK\log(K))$. We are not suggesting that all-uses criterion necessarily be satisfied, since this could involve a great deal of time and effort. Indeed, it would be impossible in some cases due to the existence of infeasible paths. Rather, the aim of our approach is to make the best use of already existing test cases and a limited amount of time.

[3]Time $O(K\log(K))$ is required to sort the chains in Table 1. However, this need be done only once.

5.2 Space Complexity

It is clear that the space required for Tables 1 and 2 is $O(K)$ and $O(TK)$ respectively.

CHAPTER 6
A PROTOTYPE FOR THE PROPOSED APPROACH

Developing reliable programs is a complicated and challenging activity requiring a variety of skills. Software tools have been developed to aid in the process of program development. The move from machine languages to assembly languages and from assembly languages to high level languages were the first significant advances in software tools. The next improvement came with the shift from batch mode to interactive mode. This provided a dramatic decrease in the required time for the usual edit-compile-execute cycle. The advent of screen oriented editors increased the efficiency of the edit phase of this cycle. There is a widespread, although informal, agreement that users of interactive environments tend to be more productive. Integrated programming environments are collections of tools to aid in all phases of program development.

This chapter introduces an interactive programming environment which is designed to demonstrate the approach presented in the last chapters. This interactive environment may support an individual programmer in the development of his/her program by providing integrated tools to create, execute, debug, and revalidate programs. This interactive programming environment is composed of a syntax directed editor, an incremental code generator, an interpreter, an incremental data flow analyzer, and an interface module that maintains a database of test cases to be used for later debugging and regression testing. The tools in the environment are specified in attribute grammar, and is implemented using the Cornell Synthesizer generator

[42, 43, 44]. A block diagram of the prototype of this interactive programming environment is shown in Figure 6.1.

6.1 Generating Language-Based Programming Environments

Today's powerful stand-alone computers provide virtually free processing capacities, which can perform millions of operations between every pair of consecutive keystrokes. This processing power is currently going to waste. Language-based programming environments offer a way to put this capacity to work. The Synthesizer Generator (SG) is one such system for generating language-based programming environments and is used to implement our prototype [44].

Some language-based programming environments already exist, such as Magpie [17], the Incremental Programming Environment (IPE) [37], and the Cornell Program Synthesizer (CPS) [55]. The Cornell Programming Synthesizer was one of the earliest programming systems to incorporate an editor with immediate error analysis capability. In the Synthesizer Generator this capability has been expanded by incorporating a very general mechanism for implementing incremental computations on abstract syntax trees.

6.2 The Synthesizer Generator

The Synthesizer Generator creates a language-specific editor from an input specification that defines a language's abstract syntax, context-sensitive relationships, display format, concrete input syntax, and transformation rules for restructuring objects. From this specification, the Generator creates a display editor for manipulating objects according to these rules.

The treatment of language syntax by the generator is of particular importance. The editor-designer's specification of the language's syntax addresses not only context-free syntax but also such context-sensitive conditions as type correctness. As a user

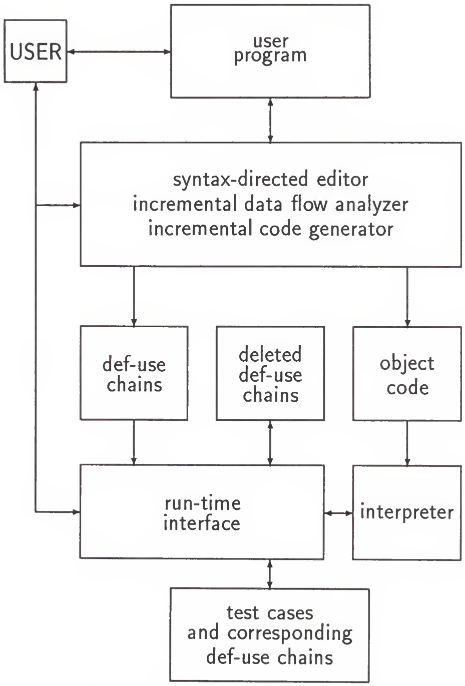

Figure 6.1. The block diagram of the prototype.

creates and modifies objects, the generated editor immediately checks for variations of context conditions that have been specified.

Context conditions are expressed by introducing certain attributes whose attribute equations indicate whether or not a constraint is satisfied. The manner in which objects are annotated with information about variation of context conditions is expressed by the editor's *unparsing* specification, which determines how objects are displayed on the screen. Attributes used in the unparsing specification cause the display to be annotated with values of attribute instances. In particular, the attributes that indicate satisfaction or violation of context-dependent constraints can be used to annotate the display to indicate the presence or absence of errors. If an editing operation modifies an object in such a way that formerly satisfied constraints are now violated (alternatively, formerly violated constraints are now satisfied), the attributes that indicate satisfaction of constraints will receive new values. The changed image of these attributes on the screen provides the user with feedback about new errors introduced and old errors corrected.

Specifications of all the tools in our prototype are written in the Synthesizer Generator Specification Language (SSL), which is built around the concepts of an attribute grammar and a type definition facility.

The Synthesizer Generator is written in C and runs under Berkeley UNIX. Editors can be generated for X Window and for Sun View, as well as for video display terminals. The prototype discribed here supports all three user interfaces.

6.3 The Structured Editor of the Prototype

The syntax-directed editor is used to simplify the user interface and to insure that program modifications are syntactically correct. The editor is generated using the Synthesizer Generator and has the following features:

1. It enforces the view that a program is a hierarchical composition of computational structures. Programs are composed of *templates*, which provide predefined, formatted patterns for each of the constructs in the language. Programs are created top-down by inserting new templates at *placeholders* in the skeleton of previously entered templates.

2. During editing, the current *selection* (*i.e.* insertion point), can be moved from one template to another. The selection is indicated on the screen by highlighting the selected region.

3. The top line of the screen has a highlighted *title bar* displaying the name of the buffer. The rest of the screen is divided into three regions: the *command line*, the *help pane*, and the *object pane*. The command line, just below the title bar, is where commands are echoed and also where the system massages are displayed. The help pane, which takes the last few lines at the bottom, provides information about what constituent is currently selected. The object pane, displaying the buffer's program fragment, covers the remaining of the screen.

4. Templates are inserted into the program by special commands, and the system checks whether the insertion is legal. A menu of the applicable insertion commands is always shown in the help pane at the bottom of the screen.

5. *Transformation* operations in the editor provide a mechanism for making controlled changes in a single step. Construct-to-construct transformation operations emphasize the abstract computational meaning of program units.

6. The editor is not just a structure editor; it also supports character and line-oriented operations to insert and delete text. Text editing is not initiated until the user types or erases a character.

6.3.1 The Attribute-Grammar Model of Editing

The editor is based on the concept of an *attribute grammar*, which provides a powerful mechanism for displaying how widely separated parts of a program's abstract-syntax tree are constrained in the context provided by the rest of the tree. An attribute grammar is a context-free grammar extended by attaching attributes to the nonterminal symbols of the grammar and by supplying *attribute equations* to define attribute values. In every production $p : X_0 \rightarrow X_1 \cdots X_k$, each X_i denotes an *occurrence* of a grammar symbol, and associated with each nonterminal occurrence is a set of *attribute occurrences* corresponding to the nonterminal's attributes.

The attributes of a nonterminal are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes. Each attribute equation defines a value for a synthesized attribute occurrence of the left-hand-side nonterminal or an inherited attribute of a right-hand-side nonterminal.

A derivation-tree node that is an instance of symbol X has an associated set of *attribute instances* corresponding to the attributes of X. An *attributed tree* is a derivation tree together with an assignment of either a value or the special token *null* to each attribute instance of the tree. To analyze a program according to its attribute-grammar specification, first construct its derivation tree; then evaluate the attribute instances using the appropriate equations. The latter process is termed *attribute evaluation.*

Functional dependencies among attribute occurrences in a production p (or attribute instances of a tree T) can be represented by a directed graph, called a *dependence graph*, denoted by $D(p)$ (respectively, $D(T)$).

6.3.2 Specification of the Editor

The editor specification consists of a collection of declarations. These declarations can be decomposed into: abstract-syntax declarations which define the language's underlying structure; attribute declarations and attribute equations which specify the context-sensitive constraints to be enforced; and unparsing declarations which express how programs are formatted on the display screen.

6.3.2.1 The Abstract-Syntax Specification of the Editor

The core of the editor's specification is the definition of the language's abstract-syntax, given as a set of grammar rules. The abstract syntax specification consists of a collection of SSL *productions* of the form $x_0 : op(x_1 x_2 \cdots x_k)$; where *op* is an operator name and each x_i is a nonterminal of the grammar or the name of a *phylum*. The phylum associated with a given nonterminal is the set of derivation trees that can be derived from it. These derivation trees are known as *terms*. With the exception of the operators, whose purpose is to identify the production instances in a derivation tree, the SSL grammar rule acts exactly as the context-free grammar production

$$x_0 \rightarrow x_1 x_2 \cdots x_k.$$

A term is denoted by an expression in which a *k-ary* operator is applied to k constants of the appropriate phyla; for terms of nullary operators, the parentheses may be omitted. The first operator declared for each phylum, such as the operator `Prog` of the phylum `program`, see below, is called the *completing operator* and plays a special role in the editor specification. The completing operator is used to construct a default representative for the phylum called the *completing term*. The completing term is created by applying the completing operator to the completing terms of its argument phyla.

The abstract-syntax rule of a list phylum must have exactly two operators, one being a nullary operator and the other a binary operator that is right recursive. For example, phylum `decList`, see below, is declared to be a list phylum and has operators of the required form; the nullary operator is `DeclListNil` and the binary operator is `DecListPair`.

The abstract syntax of the language accepted by the prototype is defined by the following abstract-syntax declarations, written in SSL:

```
/* A program is a list of variable declarations followed by a list
of statements.*/
root program;
program : Prog(declList stmtList);

/* The list of variable declarations is either empty or is a list
of simple declarations. A variable can be declared to be of type
integer, boolean, or real. */
list declList;
declList: DeclListNil()
        | DeclListPair(decl declList)
        ;
decl    : Declaration(identifier typeExp);
typeExp : EmptyTypeExp()                 /* empty type */
        | IntTypeExp()                   /* integer type */
        | BoolTypeExp()                  /* boolean type */
        | RealTypeExp()                  /* real type */
        ;
identifier: IdentifierNull()
          | Identifier(IDENTIFIER)
          ;
list stmtList;
stmtList: StmtListNil()
        | StmtListPair(stmt stmtList)
        ;

/* The statements supported are empty, putline, get(identifier),
put(exp), assignment statements, if-then-else statements, while
statements, or a composition of them. */
stmt    : EmptyStmt()
        | PutLine()
        | Get(identifier)
        | Put(exp)
        | Assign(identifier exp)
        | IfThenElse(exp stmtList stmtList)
        | While(exp stmtList)
        | Compound(stmtList)
        ;

/* The following expressions are allowed. */
exp     : EmptyExp()
        | IntConst(INTEGER)
        | RealConst(REALTYPE)
        | True()
        | False()
        | Id(identifier)
        | Equal, LT, GT, LE, GE(exp exp)
        | NotEqual(exp exp)
        | Add, Sub, Mult, Div, And, Or(exp exp)
        | Uminus, Not, Abs(exp)
        ;
```

6.3.2.2 Attributes and Attribute Equations

In addition to the grammar rules that define the language's abstract syntax, the editor specification contains declarations that define how to make static inferences about the objects being edited. The following constraints are defined in the specification of our prototype:

1. a declaration must be supplied for all identifiers used in the program,

2. an identifier must be declared at most once, and

3. the constituents of expressions and statement must have compatible types.

In the following declarations, at the root of the program tree, an environment attribute named env contains the type binding of each identifier. In addition, each expression and subexpression has an associated type attribute. The type of an identifier used in an expression is determined by accessing the environment at the root. Local attributes are attributes that are associated with a particular production rather than with each production of a nonterminal. Local attribute declarations differ from those of ordinary attributes in that the keyword `local` is used in place of `synthesized` or `inherited`; they also differ in that local attribute declarations are placed among a production's attribute equations.

The above constraints are specified using attribute equations which are expressed in SSL as follows:

```
program : Prog { local declList env;
                 env = declList;
               };

/* An identifier can only be null or declared only once. */
decl    : Declaration {local STR error;
                       error = (identifier != IdentifierNull
                && NumberOfDecls(identifier, {Prog.env}) > 1)
                ? "{MULTIPLY DECLARED}" : "";
                };
/* If the identifier is null set its type to be empty; otherwise
look up its type in the program environment at the root of the
tree. */
identifier, exp {synthesized typeExp type;};
identifier: IdentifierNull {identifier.type = EmptyTypeExp;}
          | Identifier     {identifier.type =
                          LookupType(identifier,{Prog.env});}
          ;
/* Check the type compatibility in each statement. */
stmt   : Assign {local STR assignError;
                  local STR error;
          assignError = IncompatibleTypes(identifier.type, exp.type)
                     ? "{INCOMPATIBLE TYPES IN :=}" : "";
                  error = (identifier == IdentifierNull ||
                           IsDeclared(identifier, {Prog.env}))
                        ? "" : "{NOT DECLARED}";
                }
        | Get{
                  local STR error;
                  error = (identifier == IdentifierNull ||
                           IsDeclared(identifier, {Prog.env}))
                        ? "" : "{NOT DECLARED}";
                }
        | IfThenElse, While { local STR typeError;
                typeError= IncompatibleTypes(exp.type, BoolTypeExp)
                ? "{BOOLEAN EXP NEEDED}" : "";
                }
        ;
/* Set the type of each expression according to the
```

```
following rules. */
exp     : EmptyExp {exp.type = EmptyTypeExp;}
        | IntConst {exp.type = IntTypeExp;}
        | RealConst{exp.type = RealTypeExp;}
        | True     {exp.type = BoolTypeExp;}
        | False    {exp.type = BoolTypeExp;}
        | Id       {
              local STR error;
              error= (identifier==IdentifierNull ||
                   IsDeclared(identifier, {Prog.env}))
                      ? "" : "{NOT DECLARED}";
              exp.type = identifier.type;
              }
        | Equal, NotEqual, LT, LE, GT, GE{
              local STR error;
              error= IncompatibleTypes(exp$2.type, exp$3.type)
                ? "{INCOMPATIBLE TYPES}" : "";
              exp$1.type = BoolTypeExp;
              }
        | And, Or{
              local STR leftError;
              local STR rightError;
         leftError= IncompatibleTypes(exp$2.type, BoolTypeExp)
                      ? "{BOOL EXP NEEDED}" : "";
         rightError= IncompatibleTypes(exp$3.type, BoolTypeExp)
                      ? "{BOOL EXP NEEDED}" : "";
              exp$1.type = BoolTypeExp;
              }
        | Add, Sub, Mult, Div{
              local STR error;
              error= IncompatibleTypes(exp$2.type, exp$3.type)
                ? "{INCOMPATIBLE TYPES}" : "";
              exp$1.type = exp$2.type;
              }
        | Abs, Uminus{
              exp$1.type = exp$2.type;
              }
        | Not{
              local STR error;
              error= IncompatibleTypes(exp$2.type, BoolTypeExp)
                ? "{BOOL TYPE NEEDED}" : "";
              exp$1.type = BoolTypeExp;
              }
        ;
```

The following auxiliary functions are used in the above declarations:

```
/* Determine the first type bound to identifier i in environment e,
   or EmptyTypeExp otherwise. */
typeExp LookupType(identifier i, declList e) {
        with(e)( DeclListNil : EmptyTypeExp,
                 DeclListPair(Declaration(id, t), dl):
                        (i==id) ? t : LookupType(i, dl))
                };

/* Return true iff there exists a type bound to i in e. */
BOOL IsDeclared(identifier i, declList e){
        with(e)(
                DeclListNil: false,
                DeclListPair(Declaration(id, t), dl):
                  (i==id) ? true : IsDeclared(i, dl))
                };

/* Determine the number of types bound to i in e. */
INT NumberOfDecls(identifier i, declList e){
        with(e)( DeclListNil: 0,
                 DeclListPair(Declaration(id, *), dl):
                  ((i==id)? 1 : 0) + NumberOfDecls(i, dl))
                };

/* Return true iff type t1 is incompatible with type t2. */
BOOL IncompatibleTypes(typeExp t1, typeExp t2){
      (t1 != EmptyTypeExp) && (t2 != EmptyTypeExp) && (t1 != t2)
                };
```

6.3.2.3 The Unparsing Declarations of the Editor

The unparsing rules of the editor define not only the display format, but also which node of the abstract syntax tree are selectable and which productions of an object is editable as text.

The unparsing rules defines a display representation for each production of the abstract syntax declaration. Each rule may take one of the following two forms:

$$symbol : operator[leftside : rightside];$$

$$symbol : operator[leftside ::= rightside];$$

The choice of the symbol `‘‘:’’` or `‘‘::=’’` determines whether or not a production can be edited as text. The symbol `‘‘::=’’` indicates that it is permitted to edit the production's text; the symbol `‘‘:’’` indicates that the production is treated as an indivisible unit. The display is generated by a left-to-right traversal of the tree that interprets the unparsing schemes. Indentation, back indentation, and line breaks are controlled by the control characters ''%t'', ''%b'', and ''%n'' respectively. The selection symbol ''@'' specifies that an occurrence is a resting place; whereas the symbol ''^'' specifies that it is not.

The following are the unparsing rules of our editor written in SSL:

```
program : Prog [@:
    "with TEXT_IO, INTEGER_IO, FLOAT_IO;%n"
    "use  TEXT_IO, INTEGER_IO, FLOAT_IO;%n"
    "procedure Main is%t%n"
                         @ ";"         "%b%n"
                    "begin"         "%t%n"
                         @         "%b%n"
                    "end Main;%n"]
          ;
declList: DeclListNil [@ ::=]
          | DeclListPair [@ ::= ^ [";%n"] @]
          ;
decl    : Declaration [^ : @ error " : " @];
typeExp : EmptyTypeExp [@ : "<type>"]
          | IntTypeExp   [@ : "integer"]
          | BoolTypeExp  [@ : "boolean"]
          | RealTypeExp  [@ : "float"]
          ;
identifier: IdentifierNull [@ ::= "<identifier>"]
            | Identifier     [^ ::= ^]
            ;
stmtList: StmtListNil  [@ :]
          | StmtListPair [@ : ^ ["%b%n"] @]
          ;
stmt        : EmptyStmt [^ ::= "  %t" "<statement>"]
          | PutLine   [^ :   stmtno "  %t" "new_line;"]
          | Get       [^ ::= stmtno "  %t" "get(" @ error ");"]
          | Put       [^ ::= stmtno "  %t" "put(" @ ");"]
           | Assign    [^ ::= stmtno "  %t" @ error " := " @ "; "
                                                              assignError]
      | IfThenElse[^ : stmtno "  %tif " @ typeError " then" "%t%n"
                                  @ "%b%n"
                                  "else%n"
                                  @ "%b%b%nend if;"]
      | While  [^ : stmtno "  %twhile " @ typeError " loop" "%t%n"
                                  @ "%b%b%nend loop;"]
          | Compound  [^ : "begin" "%t%n"
                         @ "%b%b%n"
                         "end;"]
          ;
exp     : EmptyExp [^ ::= "<exp>"]
```

```
    | IntConst [^ ::= ^]
    | RealConst[^ ::= ^]
    | True     [^ ::= "true"]
    | False    [^ ::= "false"]
    | Id       [^ ::= ^ error]
    | Equal    [^ ::= "(" @ " = " error @ ")"]
    | LT       [^ ::= "(" @ " < " error @ ")"]
    | LE       [^ ::= "(" @ " <= " error @ ")"]
    | GT       [^ ::= "(" @ " > " error @ ")"]
    | GE       [^ ::= "(" @ " >= " error @ ")"]
    | NotEqual [^ ::= "(" @ " /= " error @ ")"]
  | And      [^ ::= "(" @ leftError " and " rightError @ ")"]
  | Or       [^ ::= "(" @ leftError " or " rightError @ ")"]
    | Add      [^ ::= "(" @  " + " error @ ")"]
    | Sub      [^ ::= "(" @  " - " error @ ")"]
    | Mult     [^ ::= "(" @  " * " error @ ")"]
    | Div      [^ ::= "(" @  " / " error @ ")"]
    | Not      [^ ::= "not(" @ error")"]
    | Abs      [^ ::= "abs(" @ ")"]
    | Uminus   [^ ::= " -" @ ]
    ;
```

6.4 The Incremental Code Generator

In a system which supports interactive program development, testing and debugging, it is desirable to provide the ability to initiate execution at any time and have the program being executed immediately, with no delay for compilation. Such a system should maintain the program in an executable form at all time and update the program's object code in accordance with changes to the program's source code.

6.4.1 Incremental Compilation Using Attributes

Incremental compilation is an obvious application for an incremental attribute-updating mechanism, such as the one found in the Synthesizer Generator. By incrementally updating attributes whose defining equations express the generation of executable code, a language-based editor can produce and maintain target code as programs are created and modified.

Coupling an incremental compiler with a language-based editor has other benefits as well. Because the editor has knowledge available to it about which portions of the program are incomplete, it can generate **Halt** instructions for those locations. This makes it possible to execute incomplete programs until a **Halt** instruction is encountered.

6.4.2 Generating Code Graphs Using SSL

The chief issue in using an attribute grammar to generate and update object code is how to limit the extent of recompilation. For example, one would like to have the property that when a single expression or assignment statement is modified, updates are limited to just a few of the tree's code fragments. In our implementation, individual code fragments are located in attributes distributed throughout the program tree; however, they are linked together into a *code graph* for the entire program. Components of each fragment provide links to other fragments. The interpreter executes the fragments directly; initially, the interpreter is passed a link to the first fragment of the program; other fragments are accessed, as necessary, by following the appropriate links.

The individual code fragment consists of instructions for an abstract stack machine and is similar to the P-code used by some Pascal compilers. The code's syntax is defined by the following SSL specification of the phylum CODE:

```
/* The abstract-syntax and the unparsing specifications of the
 phylum CODE. */
CODE : Halt()                          [^ : "Halt"]
     | Quit()                          [^ : "Quit"]
     | Putline(LINK LINK)              [^ : "Putline " ^ ^ ]
     | PushVar(identifier LINK)        [^ : "PushVar "^ ^ ]
     | Store(identifier LINK LINK)     [^ : "Store "^ ^  ^ ]
     | GetInt(identifier LINK LINK)    [^ : "GetInt "^ ^ ^ ]
     | PutInt(LINK LINK)               [^ : "PutInt " ^  ^ ]
     | PushInt(INT LINK)               [^ : "PushInt "^ ^ ]
     | IsEqInt(LINK)                   [^ : "IsEqInt "  ^ ]
     | IsNotEqInt(LINK)                [^ : "IsNotEqInt " ^ ]
     | IsLtInt(LINK)                   [^ : "IsLtInt " ^ ]
     | IsLeInt(LINK)                   [^ : "IsLeInt " ^ ]
     | IsGtInt(LINK)                   [^ : "IsGtInt " ^ ]
     | IsGeInt(LINK)                   [^ : "IsGeInt " ^ ]
     | AddInt(LINK)                  [^ : "AddInt" ^ ]
     | SubInt(LINK)                  [^ : "SubInt" ^ ]
     | MultInt(LINK)                 [^ : "MultInt" ^ ]
     | DivInt(LINK)                  [^ : "DivInt" ^ ]
/* similar instructions for type real deleted. */
     | AndBool(LINK)                 [^ : "AndBool" ^ ]
     | OrBool(LINK)                  [^ : "OrBool" ^ ]
     | NotBool(LINK)                 [^ : "NotBool" ^ ]
     | PushBool(BOOL LINK)           [^ : "PushBool "^ ^ ]
     | IsEqBool(LINK)                [^ : "IsEqBool " ^ ]
     | IsNotEqBool(LINK)             [^ : "IsNotEqBool " ^ ]
     | Branch(LINK LINK LINK)        [^ : "Branch "  ^   ^  ^ ]
     ;
```

The construction of the code graph is specified with two sets of attributes named `entry` and `next`, which carry information used to link the fragmented code. The entry attribute at the root of a given program segment is a link to the first instruction to be executed in that segment. The next attribute at the root of a given program segment is a link to the first instruction to be executed after the code for the given segment has been completed.

The values of the `entry` and `next` attributes are incorporated into instructions to establish the code graph. Each `stmt` incorporates the link `stmt.next` into its code fragment, and passes up a link to the first instruction of this fragment in the attribute `stmt.entry`. Intermediate nodes in the abstract-syntax tree, such as each `StmtListPair` node in a statement list, merely pass on linking information in their `entry` and `next` attributes.

Example. Attributes `entry` and `next` are passed through the operators of the phylum `stmtList` as follows:

```
stmtList : StmtListNil{stmtList.entry = stmtList.next;}
        | StmtListPair{stmtList$1.entry = stmt.entry;
               stmt.next = stmtList$2.entry;
               stmtList$2.next = stmtList$1.next:};
```

The equation that defines the entry point of the conditional-statement also illustrates how intermediate nodes in the abstract-syntax tree merely pass on linking information:

```
stmt : IfThenElse{stmt$1.entry = exp.entry;};
```

This equation defines the entry point of the statement to be the entry point of the expression; whatever statement precedes the `IfThenElse` will receive a link to the expression, allowing the interpreter to jump to the expression directly rather than making a jump to the conditional-statement and then jump to the expression.

The semantics of the individual control constructs of the source language are expressed in terms of `CODE` as follows. Each construct has an associated code fragment, which is defined as the value of a local CODE-valued attribute. The link constructed from this fragment is passed to other components of the program using the `entry` and `next` attributes.

Example. The `Prog` operator has one code fragment represented by the local attribute named `code`. The code fragment consists of a single `Quit` instruction. The link formed from the attribute `code` is passed down the abstract-syntax tree, providing access to the final instruction to be executed when the program's body has completed:

```
program : Prog {local CODE code;
                code = Quit;
                stmtList.next = MakeLink(code);
               };
```

The abstract-syntax tree contains error attributes that indicate the presence or absence of errors in the program. Thus, `Assign` operator's code is a `Halt` if either of two error conditions holds, indicated by particular values of the attributes named `assignError` and `identifier.type`:

```
stmt : | Assign{ local CODE code;
         code = (assignError != "" || identifier.type ==
          EmptyTypeExp)
          ? Halt : Store(identifier, stmt.next, stmt.entryno);
         stmt.entry = exp.entry;
         exp.next = MakeLink(code);
             }
```

6.4.3 Links and Circularities

In using attribute grammar to specify the generation of a code graph the question of *circularities* naturally arises. In particular, in creating the code for loops, we have to be careful to avoid circular dependencies in the specification.

Obviously, linked code for loop constructs has to be circular at some level; however, we must be careful not to confuse two different kinds of cicularities. The kind of circularity that causes problems in SSL is a circular dependence in a specification's *attribute equations*; the circularity that is inherent in code for loops is a circular in the representation for the code, *i.e.* a circularity within attribute *values*.

To break up the circularity, we introduced one level of indirection into the generated code by making use of the built-in, primitive phylum ATTR to implement the links. In essence, this breaks the circularity because some of the edges of the dependence cycle are replaced by attribute references; which do not add dependencies between attribute instances.

The following is the specification in SSL of the incremental code generator:

```
/* The following is a mechanism to circumvent the circularity
in the code attribute specification by making use of
indirection. */
%[
#define LINK ATTR
#define MakeLink(x) (&&x)
FOREIGN  CodeForLink(a)          /* SSL type: CODE */
      PROD_INSTANCE a;          /* SSL type: ATTR */
      {
       return(demand_value(AttrValue(a)));
      }
%]
```

```
CODE foreign CodeForLink(LINK a);
/* For each statement and expression define a synthesized
attribute entry and an inherited attribute next. */
stmt, stmtList     { syn LINK entry;
                     inh LINK next;
                    };
exp                { syn LINK entry;
                     inh LINK next;
                    };
program : Prog {local CODE code;
             code = Quit;
             stmtList.next = MakeLink(code);
            };
stmtList : StmtListNil{
                 stmtList.entry = stmtList.next;
                }
         | StmtListPair{
                stmtList$1.entry = stmt.entry;
                stmt.next = stmtList$2.entry;
                stmtList$2.next = stmtList$1.next;
               }
         ;
stmt : EmptyStmt{ local CODE code;
                 code = Halt;
                 stmt.entry = MakeLink(code);
                }
     | PutLine{ local CODE code;
                 code = Putline(stmt.next, stmt.entryno);
                 stmt.entry = MakeLink(code);
                }
     | Assign{ local CODE code;
               code = (assignError != "" || identifier.type ==
               EmptyTypeExp) ? Halt : Store(identifier, stmt.next,
                       stmt.entryno);
               stmt.entry = exp.entry;
               exp.next = MakeLink(code);
              }
     | Get    { local CODE code;
               code = (identifier.type == EmptyTypeExp)
                      ? Halt : (identifier.type == IntTypeExp) ?
                     GetInt(identifier, stmt.next, stmt.entryno):
                     GetReal(identifier, stmt.next, stmt.entryno);
               stmt.entry = MakeLink(code);
              }
     | Put    { local CODE code;
               code = (exp.type == EmptyTypeExp)
                       ? Halt : (exp.type == IntTypeExp) ?
                              PutInt(stmt.next, stmt.entryno):
                               PutReal(stmt.next, stmt.entryno);
               stmt.entry = exp.entry;
               exp.next   = MakeLink(code);
               }
      | IfThenElse{ local CODE code;
                    code = (typeError != "") ? Halt
       : Branch(stmtList$1.entry, stmtList$2.entry, stmt.entryno);
                    stmt.entry = exp.entry;
                    exp.next = MakeLink(code);
                    stmtList$1.next = stmt.next;
                    stmtList$2.next = stmt.next;
                  }
     | While{ local CODE code;
              code = (typeError !="") ? Halt
               : Branch(stmtList.entry, stmt.next, stmt.entryno);
              stmt.entry = exp.entry;
              exp.next = MakeLink(code);
              stmtList.next = exp.entry;
            }
     | Compound { stmt.entry = stmtList.entry;
                  stmtList.next = stmt.next;
                }
     ;
exp  : EmptyExp{ local CODE code;
                 code = Halt;
               }
     | IntConst{ local CODE code;
                 code = PushInt(STRtoINT(INTEGER), exp.next);
                 exp.entry = MakeLink(code);
               }
     | RealConst{ local CODE code;
                 code = PushReal(STRtoREAL(REALTYPE), exp.next);
                 exp.entry = MakeLink(code);
               }
     | True{ local CODE code;
             code = PushBool(true, exp.next);
             exp.entry = MakeLink(code);
           }
     | False{ local CODE code;
              code = PushBool(false, exp.next);
              exp.entry = MakeLink(code);
            }
     | Id{ local CODE code;
           code = (identifier.type== EmptyTypeExp) ? Halt
                  : PushVar(identifier, exp.next);
           exp.entry = MakeLink(code);
         }
     | Equal{ local CODE code;
              code = (error != "") ? Halt
               :(exp$2.type == IntTypeExp) ? IsEqInt(exp$1.next)
             :(exp$2.type == RealTypeExp) ? IsEqReal(exp$1.next)
                         : IsEqBool(exp$1.next);
              exp$1.entry = exp$2.entry;
              exp$2.next = exp$3.entry;
            exp$3.next = MakeLink(code);
          }
/* Similar equations for NotEqual, LT, LE, GT, and GE are deleted. */
   | Add { local CODE code;
            code = (error != "") ? Halt
              :(exp$2.type == IntTypeExp) ? AddInt(exp$1.next)
                       : AddReal(exp$1.next);
            exp$1.entry = exp$2.entry;
            exp$2.next = exp$3.entry;
            exp$3.next = MakeLink(code);
          }
/* Similar equations for Sub, Mult, and Div are deleted. */
    | Abs{ local CODE code;
          code = (exp$2.type == IntTypeExp) ? AbsInt(exp$1.next)
                       : AbsReal(exp$1.next);
            exp$1.entry = exp$2.entry;
            exp$2.next = MakeLink(code);
          }
    | Uminus{ local CODE code;
        code = (exp$2.type == IntTypeExp) ? UminusInt(exp$1.next)
                       : UminusReal(exp$1.next);
            exp$1.entry = exp$2.entry;
            exp$2.next = MakeLink(code);
          }
    | And{ local CODE code;
           code = (leftError != "" || rightError != "") ? Halt
                    : AndBool(exp$1.next);
           exp$1.entry = exp$2.entry;
           exp$2.next = exp$3.entry;
           exp$3.next = MakeLink(code);
         }
\* Similar equations for Or is deleted. */
    | Not{ local CODE code;
           code = (error != "") ? Halt
                    : NotBool(exp$1.next);
           exp$1.entry = exp$2.entry;
           exp$2.next = MakeLink(code);
         }
    ;
```

6.5 The Interpreter

The interpreter is a simple stack machine with a single run-time stack and an environment which is implemented as an SSL list. The environment consists of a list of two elements tuples representing the program's variables and their values. A pointer to the first instruction of the program's object code is passed to the interpreter. The interpreter executes each instruction with the operands being a value read from the keyboard or the value at the top of the stack or both. The result of each instruction is stored on the run-time stack. When the interpreter encounters a `Halt` instruction, it stops the execution and allow the user to examine the values of the variables from the environment list. When a `Quit` instruction is encountered, the values of the variables are written out for the user. The interpreter also produces the execution path taken by a test case as a list of statement numbers, as well as the definition-use chains covered by that test case. This information is used by the run-time environment to update a database of test cases and related information.

6.6 The Incremental Data Flow Analyzer

A *definition* of a variable x is generated by a statement that assigns, or may assign, a value to x. The most common forms of definitions of x are generated by assignment statements to x and statements that read a value into x. A definition of x is called *unambiguous* if x is assigned a value by a statement, such as an assignment or a read statement. A definition of x is called *ambiguous* if x *may* be assigned a value by a statement. An if statement or a call statement to a procedure which may modify the variable x are examples of such statements that cause an ambiguous definition for x. A definition d is said to *reach* a point p in the program if there is a path from the point immediately following d to p such that d is not redefined along that path. A definition of a variable x is *killed* along a path if that path contains an unambiguous definition of x.

A solution of the reaching definitions problem can be described in terms of $Gen(S)$, $Kill(S)$, $Out(S)$, and $In(S)$ of a statement S. Sets $Gen(S)$, $Kill(S)$, and $Out(S)$ are *synthesized attributes* which are computed bottom-up from the leaves of a syntax tree up to its root. A definition d is in $Gen(S)$ if d is generated by statement S. $Kill(S)$ is the set of definitions that never reach any statement following S directly. $Out(S)$ is the set of definitions that may reach the statements that follow S directly. $In(S)$ is an *inherited attribute* which represents the set of definitions that may reach statement S.

The following attribute equations are used to generate the reaching definitions [5, p. 612]:

1. $S \longrightarrow$ "an assignment or a read statement that defines a variable a"

$$\begin{aligned} Gen(S) &= \{d\} \\ Kill(S) &= D_a - \{d\} \\ Out(S) &= Gen(S) \cup (In(S) - Kill(S)) \end{aligned}$$

   where d is the definition of a at statement S, and D_a is the set of all definitions of a in the program.

2. $S \longrightarrow \alpha\beta$

$$\begin{aligned} Gen(S) &= Gen(\beta) \cup (Gen(\alpha) - Kill(\beta)) \\ Kill(S) &= Kill(\beta) \cup (Kill(\alpha) - Gen(\beta)) \\ In(\alpha) &= In(S) \\ In(\beta) &= Out(\alpha) \\ Out(S) &= Out(\beta) \end{aligned}$$

3. $S \longrightarrow \alpha_1|\alpha_2|\cdots|\alpha_n$. For all $1 \leq i \leq n$, we have:

$$Gen(S) = \bigcup_{i=1}^{n} Gen(\alpha_i)$$

$$\begin{aligned}
Kill(S) &= \bigcap_{i=1}^{n} Kill(\alpha_i) \\
In(\alpha_i) &= In(S) \\
Out(S) &= \bigcup_{i=1}^{n} Out(\alpha_i)
\end{aligned}$$

4. $S \longrightarrow \alpha^+$ (Note: $S \longrightarrow \alpha^*$ is equivalent to $S \longrightarrow \varepsilon|\alpha^+$.)

$$\begin{aligned}
Gen(S) &= Gen(\alpha) \\
Kill(S) &= Kill(\alpha) \\
In(\alpha) &= In(S) \cup Gen(\alpha) \\
Out(S) &= Out(\alpha)
\end{aligned}$$

6.7 The Run-time Interface

The run-time interface allow the user to run a test case, know which definition-use chains covered by it, know which definition-use chain are yet to be tested, and query the content of the database. A menu of commands is available to the user at all time. An output buffer records the output from the user's program and queries. This buffer can be stored for later examination.

As a user is modifying a program, the incremental data flow analyzer maintains the deleted and added definition-use chains. The environment maintains the set of definition-use chains that need to be tested to achieve the required coverage criteria. When the user decides to revalidate a program, he can use the system to determine which existing test cases need to be rerun. The system searches the database for existing test cases associated with any of the deleted chains. The interpreter is designed to generate the chains covered by each test case during the execution. The system uses these chains to update the database. Before rerunning an existing test case the user has to decide whether the change has made the test case obsolete or not. In generating new test cases, the user selects an untested chain and asks the system for suggested input values. Assuming the chain under consideration is (v, n_1, n_2), the system will search the database for a test case associated with a tuple of the form $(v, n_1, *)$. The input values for that test case can be used by the user as a starting point in finding the required input values by incrementally changing each input value. When a test case detects an error, the statements corresponding to the chains covered by it alone are highlighted on the screen. The user may proceed with running more test cases if he can not determine the location of the fault.

The system also reports to the user the values of the program variables at the end of each test case execution and the execution path taken by it. In addition, the user can run partial programs; execution halts if a template for a missing program element is encountered.

6.8 A Sample Session

The program below is the same program given in Chapter 4. Remember that the is specified to compute $\sqrt{\mathrm{p}}$, for $0 \leq \mathrm{p} < 1$ to accuracy e; where $0 < \mathrm{e} \leq 1$. The statements of the program are numbered for later reference. The number assigned to each statement appears before it on the line. The program contains a fault; the statements 12 and 13 should be interchanged.

```
1  get(p);
2  get(e);
3  d := 1.0;
4  x := 0.0;
5  c := (2.0 * p);
6  if (c < 2.0) then
    7  while (d > e) loop
        8  d := (d / 2.0);
        9  t := (c - ((2.0 * x) + d));
        10  if (t < 0.0) then
            11  c := (2.0 * c);
            else
```

```
          12  x := (x + d);
          13  c := (2.0 * (c - ((2.0 * x) + d)));
          end if;
        end loop;
      14  put(x);
      15  new_line;
      else
        9  put( -1);
        17  new_line;
        end if;
      end;
    end Main;
```

To see what are the definition-use chains that need to be tested, we issued a `show-all-chains` command. The output was as follows:

```
All definition-use chains:

{<c, 5, 7>, <c, 5, 9>, <c, 5, 11>, <c, 5, 13>, <c, 5, 16>,
<c, 11, 9>, <c, 11, 11>, <c, 11, 13>, <c, 13, 9>, <c, 13, 11>,
<c, 13, 13>, <d, 3, 8>, <d, 3, 14>, <d, 8, 8>, <d, 8, 9>,
<d, 8, 12>, <d, 8, 13>, <d, 8, 14>, <e, 2, 8>, <e, 2, 14>,
<p, 1, 5>, <t, 9, 11>, <t, 9, 12>, <x, 4, 9>, <x, 4, 12>,
<x, 4, 14>, <x, 12, 9>, <x, 12, 12>, <x, 12, 13>, <x, 12, 14>}
```

Then we executed the first test case with $p = 2.0$ and $e = 0.5$ the output of the program was -1 which is correct. The following is a record of the rest of the session with commands marked by a # mark at the beginning of the line.

```
# show-du-chains-covered-by-last-case

The chains covered only by last test case:

{<c, 5, 16>, <p, 1, 5>}

# run-test-case
# 0.5 1.0

0.000000

#show-exit-status

Terminate normally.
```

```
Final state:
<p : 5.000000e-01>
<e : 1.000000e+00>
<d : 1.000000e+00>
<x : 0.000000e+00>
<c : 1.000000e+00>
Execution path:
1, 2, 3, 4, 5, 6, 7, 14, 15,

# run-test-case
# 0.16 0.3

0.250000

# run-test-case
# 0.36 0.3

0.500000

# show-untested-du-chains

The untested chains:

{<c, 11, 11>, <c, 13, 13>, <x, 12, 12>}

# run-test-case
# 0.04 0.3

0.000000

# show-du-chains-covered-by-last-case

The chains covered only by last test case:

{<c, 11, 11>}

# show-untested-du-chains

The untested chains:

{<c, 13, 13>, <x, 12, 12>}

# run-test-case
# 0.9 0.2

0.625000
```

At this stage we recognized that the output is incorrect, so we wanted to know the definition-use chains covered only by the last test case. These chains may be associated with the statements causing the error.

```
# show-du-chains-covered-by-last-case
The chains covered only by last test case:
{<x, 12, 12>}
# show-untested-du-chains
The untested chains:
{<c, 13, 13>}
```

Here we stopped testing and exchanged the statements 12 and 13 to correct the program. Before continuing with testing, we asked the system to determine which existing test cases should be rerun. The answer was the test cases numbered 3, 4, and 6. The system also updated its database to make consistent with the program's code.

```
# show-test-cases-to-be-rerun
The following test cases should be rerun:
3, 4, 6
# show-database-contents
The database contents:
Test case: 1
{<c, 5, 16>, <p, 1, 5>}
Test case: 2
{<c, 5, 7>, <d, 3, 14>, <e, 2, 14>, <p, 1, 5>, <x, 4, 14>}
Test case: 5
{<c, 5, 7>, <c, 5, 9>, <c, 5, 11>, <c, 11, 9>, <c, 11, 11>,
<d, 3, 8>, <d, 8, 8>, <d, 8, 9>, <d, 8, 14>, <e, 2, 8>, <e, 2, 14>,
<p, 1, 5>, <t, 9, 11>, <x, 4, 9>, <x, 4, 14>}
```

When we executed the three test cases again they all executed correctly, completing all-uses coverage criteria.

```
# show-untested-du-chains

The untested chains:

{}

# show-database-contents

The database contents:

Test case: 1
{<c, 5, 16>, <p, 1, 5>}
Test case: 2
{<c, 5, 7>, <d, 3, 14>, <e, 2, 14>, <p, 1, 5>, <x, 4, 14>}
Test case: 5
{<c, 5, 7>, <c, 5, 9>, <c, 5, 11>, <c, 11, 9>, <c, 11, 11>,
<d, 3, 8>, <d, 8, 8>, <d, 8, 9>, <d, 8, 14>, <e, 2, 8>, <e, 2, 14>,
<p, 1, 5>, <t, 9, 11>, <x, 4, 9>, <x, 4, 14>}
Test case: 3
{<c, 5, 7>, <c, 5, 9>, <c, 5, 11>, <c, 11, 9>, <c, 11, 12>,
<d, 3, 8>, <d, 8, 8>, <d, 8, 9>, <d, 8, 13>, <d, 8, 12>, <d, 8, 14>,
<e, 2, 8>, <e, 2, 14>, <p, 1, 5>, <t, 9, 11>, <t, 9, 12>,
<x, 4, 9>, <x, 4, 13>, <x, 4, 12>, <x, 13, 14>}
Test case: 4
{<c, 5, 7>, <c, 5, 9>, <c, 5, 12>, <c, 12, 9>, <c, 12, 11>,
<d, 3, 8>, <d, 8, 8>, <d, 8, 9>, <d, 8, 13>, <d, 8, 12>, <d, 8, 14>,
<e, 2, 8>, <e, 2, 14>, <p, 1, 5>, <t, 9, 11>, <t, 9, 12>, <x, 4, 9>,
<x, 4, 13>, <x, 4, 12>, <x, 13, 9>, <x, 13, 14>}
Test case: 6
{<c, 5, 7>, <c, 5, 9>, <c, 5, 12>, <c, 12, 9>, <c, 12, 12>,
<d, 3, 8>, <d, 8, 8>, <d, 8, 9>, <d, 8, 13>, <d, 8, 12>, <d, 8, 14>,
<e, 2, 8>, <e, 2, 14>, <p, 1, 5>, <t, 9, 12>, <x, 4, 9>, <x, 4, 13>,
<x, 4, 12>, <x, 13, 9>, <x, 13, 13>, <x, 13, 12>, <x, 13, 14>}
```

6.9 Related Work

Recently, there have been a considerable effort to develop such integrated programming environments. Most notable among these are:

1. The Cornell Program Synthesizer(CPS) [55] is a programming environment which includes a set of tools for creating, editing, executing and debugging

programs. It has templates for most syntactic structures but assignment statements and expressions are typed by the user as text. The Cornell Program Synthesizer is implemented for PL/CS, a small subset of PL/I. Execution can be performed at any time during program development. It is possible to run incomplete programs; execution is suspended whenever a placeholder is encountered and can be resumed after the messing program element has been inserted. After most editing changes to a program, it is still possible to resume execution, though certain changes, such as modifying a declaration, destroy the possibility of resuming execution.

2. The Incremental Programming Environment(IPE) [37] provides a template editor environment for all major syntactic structures. The program is manipulated using a syntax directed editor, and its execution is controlled by a language oriented debugger. When changes are made to the source program the smallest unit for which code is generated is the procedure. The program modifications cause the system to incrementally compile those pieces and incorporate them into the executable version of the program

3. The Magpie [17] system does not use the template model, instead it extends the text model of editing to incorporate syntax and semantic analysis. Magpie incrementally analyzes the syntax and static semantics, immediately reporting to the user any errors it finds; the user can fix the errors immediately but is not forced to do so.

Our interactive programming environment [53] takes a step further by aiding the user manage the test cases to be used later in debugging and regression testing.

CHAPTER 7
INTERPROCEDURAL DEFINITION-USE DEPENDENCY ANALYSIS

7.1 Introduction

To extend the approach outlined in the previous chapters to handle programs with more than one procedure, an efficient algorithm for determining the definition-use chains across procedure boundaries is required. This chapter introduces a new algorithm for interprocedural definition-use dependency analysis. Programs with recursive procedures are also supported by the new algorithm. The new algorithm makes use of existing algorithms for aliasing analysis. For the new algorithm to perform the definition-use analysis incrementally, an algorithm for handling aliasing incrementally should be used.

As current trends encourage program modularity, and the number of procedures in a program continues to grow, interest in interprocedural data flow analysis has continued to grow as well. Data flow analysis at the interprocedural level has usually been done by associating with each call site approximate summary information about the effect of the called procedure. This summary information is adequate for some applications such as certain code optimization techniques across call statements. Summary information, however, is insufficient for some other applications such as data flow testing.

In data flow testing, test cases are used to test a specified set of interactions between the statements of the program under test. These interactions occur between two statements when one statement assigns a value to a variable and that variable is later referenced by the other statement before it is redefined. Based on the data flow in the program, test data adequacy criteria are used to select particular definition-use chains that are identified as the test requirement for the program. Thus, data flow testing requires the determination of the definition-use chains in the program [41]. The definition-use chains that exist across procedure boundaries (*i.e.,* interprocedural definition-use chains) are thus needed for interprocedural data flow testing as well.

Analysis techniques to determine intraprocedural definition-use chains are well known [5]. Solving the reaching definitions problem is usually the first step in determining the definition-use chains in a program. Several iterative [57], elimination [48], and syntax-directed [5, 47, 45] methods have been devised to solve the intraprocedural reaching definitions problem. A number of interprocedural data flow analysis techniques have been developed that are useful for parallelization and optimization. Very few of these techniques compute adequate information to solve the interprocedural reaching definitions problem. Some of the existing techniques [3, 6, 15, 25, 38, 46] provide summary data flow information to be used in determining the local effects of called procedures at call sites. However, this information is not enough to determine the interprocedural definition-use chains. Harrold has extended the program summary graph proposed by Callahan [9] to generate the required definition-use chains [25]. The goal of this chapter is to describe a more efficient solution to the interprocedural definition-use dependency analysis problem.

This chapter extends the syntax-directed method for intraprocedural data flow analysis [5, p. 612] to handle the interprocedural reaching definitions problem. The behavior of recursive procedures has been analyzed symbolically in detail. As a result, we were able to simplify the process of deriving the required information for recursive procedures. The derivation process is to convert a set of mutually recursive procedures to self-recursive procedures which are further converted to non-recursive procedures. The set of definitions generated and killed by each procedure is finally derived based on the equivalent non-recursive procedures. Definition-use chains are then derived. Proofs of correctness as well as the analysis of complexity of the algorithms are given.

This chapter is organized as follows: Section 2 provides background information. In Section 3, the details of the proposed algorithm as well as a proof of its correctness are given. In Section 4, related work is summarized.

7.2 Background

To simplify the notations and make the discussion more language-independent, the following notations which are similar to those used in context-free grammars are used throughout the chapter.

Two operators "$\cdot$" and "$|$" are used to represent the syntax of a program. $S_1 \cdot S_2$, or S_1S_2 for short, is a sequence of statements S_1 and S_2. $S_1|S_2$ is a selection between either S_1 or S_2. ε is an *empty* statement, such that $\varepsilon S = S\varepsilon = S$. S^* is $\varepsilon|S|SS|\cdots$ and S^+ is SS^*.

A *pattern* is everything on the right hand side of a production rule. A *production rule* (or simply *production*) "$A \longrightarrow \alpha$" means a pattern α can be generated by A. Clearly, the production process can only be finite for a non-recursive relationship. Therefore, the interesting patterns in the context of this work are those with recursive symbols. A *terminal* is a symbol which never appears on the left hand side of a production rule. A *nonterminal* is a symbol which does appear on the left hand side of some production rules. A *term* is a sequence of terminals and nonterminals. The procedure being analyzed forms the *starting symbol* of the production process. Furthermore, we say two patterns P_1 and P_2 are *equivalent* with respect to a function f, denoted by "$P_1 \equiv P_2$ (modulo f)", if $f(P_1) = f(P_2)$. "$P_1 \equiv P_2$ (modulo $f_1, \ldots, f_n$)" means "$P_1 \equiv P_2$ (modulo f_i) for all $1 \leq i \leq n$". The following laws also apply: (1)

distributive law: $X(Y|Z) = XY|XZ$ and $(X|Y)Z = XZ|YZ$, (2) associative law: $(XY)Z = X(YZ)$ and $(X|Y)|Z = X|(Y|Z)$, and (3) commutative law: $X|Y = Y|X$.

7.3 Approach

In this section, we first deal with procedures without formal parameters. Procedures with formal parameters as well as aliases will be discussed in Subsection 7.3.5. To analyze interprocedural definition-use dependency, we first calculate sets *Gen* and *Kill* which will be used to calculate sets *In* and *Out*. Finally, the definition-use chains in the program will be derived according to the *In* set. Examples will be given to illustrate theorems and algorithms. In Subsections 7.3.1 to 7.3.5, a general methodology to derive precise definition-use dependencies is presented. Subsections 7.3.1 and 7.3.2 deal with *Gen* and *Kill* sets of self and mutually recursive procedures respectively. Subsection 7.3.3 generates the *In* and *Out* sets. Subsection 7.3.4 derives the definition-use chains. Subsection 7.3.5 discusses aliasing. Several algorithms to improve the performance are given in Subsection 7.3.6.

7.3.1 *Gen* and *Kill* Sets of Self-Recursive Procedures

A path from a statement to itself may exist because of loops or recursive calls. Let a path from S back to itself be $S\alpha S$. Since the definitions generated (or killed) by the first S in the term $S\alpha S$ will also be generated (or killed) by the second S, then $S\alpha S \equiv \alpha S$ (modulo $Gen, Kill$). The following lemma provides a formal proof.

Lemma 1 (Duplicate Elimination)

$$S_1S_2S_1 \equiv S_2S_1 \text{ (modulo } Gen, Kill)$$

or

$$Gen(S_1S_2S_1) = Gen(S_2S_1)$$

$$Kill(S_1S_2S_1) = Kill(S_2S_1)$$

Proof. From basic set operations[1]: $Kill(S_2S_1) = Kill(S_2) - Gen(S_1) \cup Kill(S_1) \supseteq Kill(S_1)$. Therefore, $Kill(S_1S_2S_1) = Kill(S_1) - Gen(S_2S_1) \cup Kill(S_2S_1) = Kill(S_2S_1)$. Similarly, $Gen(S_2S_1) = Gen(S_2) - Kill(S_1) \cup Gen(S_1) \supseteq Gen(S_1)$. Thus $Gen(S_1S_2S_1) = Gen(S_1) - Kill(S_2) \cup Gen(S_2S_1) = Gen(S_2S_1)$. □

The following corollary explains how to simplify a loop statement.

Corollary 1 (Loop Elimination)

$$(S)^+ \equiv S \text{ (modulo } Gen, Kill)$$

$$(S)^* \equiv \varepsilon | S \text{ (modulo } Gen, Kill).$$

The approach to determine *Gen* and *Kill* of a self-recursive procedure R is to transform it into a non-recursive procedure which is equivalent to R with respect to *Gen* and *Kill*. This may be done by in-line substituting the body of R for each call statement to itself. However, such in-line substitution process will not terminate. Theorem 1 proves that one in-line substitution is enough for determining *Gen* and *Kill*; provided that after the first in-line substitution each call statement to R is replaced by a statement $\perp$. Statement $\perp$ absorbs all statements on the same execution path, leading to or led by $\perp$, *i.e.*, $S \perp = \perp S = \perp$ and $S \mid \perp = \perp \mid S = S$ for any statement S. Semantically, $Gen(\perp) = \emptyset$ and $Kill(\perp) = \Omega$, where Ω is the set of all definitions in the program.

Theorem 1 (Self-Recursive Procedure) *Let the body of a self-recursive procedure R be represented as $f(R)$, then $R \equiv f(f(\perp))$* (modulo *Gen*, *Kill*).

Proof. Let $f(R)$ be expanded in the form $\mathcal{C}|\Delta_{11}(R)R\delta_{12}|\cdots|\Delta_{n1}(R)R\delta_{n2}$; where $\Delta_{j1}(R)$ may contain zero or more R's but δ_{j2} does not contain any R. Clearly,

$$\mathcal{C} = f(\perp) \tag{7.1}$$

[1]In this chapter, all set operations are computed from left to right unless the order is explicitly indicated by parenthesis.

Let $\delta_{jt} = \Delta_{jt}(\varepsilon)$. Consider a production rule $R \longrightarrow \delta_1 R \delta_2 R \delta_3$. From Lemma 1,

$$\delta_1 R \delta_2 R \delta_3 \equiv \delta_1 \delta_2 R \delta_3 \text{ (modulo } Gen, Kill)$$

which means that only the last self-recursive symbol is important in computing *Gen* and *Kill* sets. Furthermore,

$$Kill(S_1S_2S_3) \supseteq Kill(S_2) \cap Kill(S_1S_3)$$

because

$$\begin{aligned}
Kill(S_2) \cap Kill(S_1S_3) &= Kill(S_2) \cap (Kill(S_1) - Gen(S_3) \cup Kill(S_3)) \\
&= (Kill(S_2) \cap (Kill(S_1) - Gen(S_3))) \cup (Kill(S_2) \cap Kill(S_3)) \\
&\subseteq (Kill(S_2) - Gen(S_3)) \cup Kill(S_3) \\
&= Kill(S_2S_3) \subseteq Kill(S_1S_2S_3).
\end{aligned}$$

Hence, the substitution of a term $\delta_{j1} R \delta_{j2}$ into R in $\delta_{i1} R \delta_{i2}$ can also be ignored because

$$Kill(\delta_{i1}(\delta_{j1} R \delta_{j2})\delta_{i2})) = Kill(\delta_{i1} R(\delta_{j1} R \delta_{j2})\delta_{i2}) \supseteq Kill(\delta_{i1} R \delta_{i2}) \cap Kill(\delta_{j1} R \delta_{j2}).$$

Therefore,

$$Kill(R) = Kill(\mathcal{C}) \cap (\cap_{j=1}^{n} Kill(\delta_{j1} R \delta_{j2})) = Kill(\mathcal{C}) \cap (\cap_{j=1}^{n} Kill(\delta_{j1} \mathcal{C} \delta_{j2})). \quad (7.2)$$

Similarly,

$$Gen(S_1S_2S_3) \subseteq Gen(S_1S_3) \cup Gen(S_2),$$

because

$$\begin{aligned}
Gen(S_1S_2S_3) &= Gen(S_1) - Kill(S_2) \cup Gen(S_2) - Kill(S_3) \cup Gen(S_3) \\
&\subseteq (Gen(S_1) - Kill(S_3) \cup Gen(S_3)) \cup Gen(S_2) \\
&= Gen(S_1S_3) \cup Gen(S_2).
\end{aligned}$$

```
procedure add is
begin
  if x = 1 then
    s := s + 1;        -- 3
  else
    s := s + x;        -- 4
    x := x - 1;        -- 5
    add;               -- 6
    x := x + 1;        -- 7
  end if;
end add;
```

Figure 7.1. An example of a self-recursive procedure.

Hence,

$$Gen(\delta_{i1}(\delta_{j1}R\delta_{j2})\delta_{i2}) = Gen(\delta_{i1}R(\delta_{j1}R\delta_{j2})\delta_{i2}) \subseteq Gen(\delta_{i1}R\delta_{i2}) \cup Gen(\delta_{j1}R\delta_{j2}).$$

Therefore,

$$Gen(R) = Gen(\mathcal{C}) \cup (\cup_{j=1}^{n} Gen(\delta_{j1}R\delta_{j2})) = Gen(\mathcal{C}) \cup (\cup_{j=1}^{n} Gen(\delta_{j1}\mathcal{C}\delta_{j2})). \quad (7.3)$$

From (1), (2), and (3),

$$\begin{aligned} R &\equiv \mathcal{C}|\delta_{11}\mathcal{C}\delta_{12}|\cdots|\delta_{n1}\mathcal{C}\delta_{n2} \equiv \mathcal{C}|\Delta_{11}(\mathcal{C})\mathcal{C}\delta_{12}|\cdots|\Delta_{n1}(\mathcal{C})\mathcal{C}\delta_{n2} \equiv f(\mathcal{C}) \\ &\equiv f(f(\perp)) \text{ (modulo } Gen, Kill). \qquad \Box \end{aligned}$$

The following example illustrates how to apply this theorem.

Example 1 (Self-Recursive Procedure) In Figure 7.1, the execution paths in procedure add can be represented as $S_3|S_4S_5RS_7$. From Theorem 1, $f(\perp) = S_3|S_4S_5\perp S_7 = S_3$ and $R \equiv f(f(\perp)) = S_3|S_4S_5S_3S_7$ (modulo $Gen, Kill$). Hence, $Gen(R) = Gen(S_3) \cup Gen(S_4S_5S_3S_7) = \{(s,3),(x,7)\}$ and $Kill(R) = Kill(S_3) \cap Kill(S_4S_5S_3S_7) = \{(s,i)|i \neq 3\}$. $\Box$

7.3.2 *Gen* and *Kill* Sets of Mutually Recursive Procedures

The following theorem explains how to generate *Gen* and *Kill* for a group of mutually recursive procedures.

Theorem 2 (Mutually Recursive Procedures) Let $R_1, R_2, \ldots,$ and R_n form a group of mutually recursive procedures. Then the following process terminates and generates the right result for $Kill(R_i)$ and $Gen(R_i)$ for all $1 \leq i \leq n$.

1. *For i from 1 to n do: Let the body of R_i be represented as $f_i(R_i)$, then substitute $f_i(f_i(\perp))$ for each R_i in the body of R_j for all $i < j \leq n$.*

2. *For all i from n down to 1 do: Calculate $Gen(R_i)$ and $Kill(R_i)$ sets using the derived $Gen(R_j)$ and $Kill(R_j)$ for all $i < j \leq n$.*

Proof. Since the algorithm only goes through finite steps, it terminates. Furthermore, each substitution generates equivalent patterns for each production rule. Eventually, the final substituted production rules will hold the patterns generated by the original production rules. □

The following example explains how to generate the *Gen* and *Kill* sets of two mutually recursive procedures.

Example 2 (Mutually Recursive Procedures without Aliasing) The program in Figure 7.2 contains two mutually recursive procedures R_1 and R_2. The production rules are $R_1 \longrightarrow S_2S_3(\varepsilon|R_2S_6)$ and $R_2 \longrightarrow \varepsilon|S_9S_{10}R_1S_{12}$. According to Theorem 2:

1. Step 1.(a) is null.

2. Step 1.(b) $R_2 = \varepsilon|S_9S_{10}(S_2S_3(\varepsilon|R_2S_6))S_{12}$. Let $C_2 = \varepsilon|S_9S_{10}(S_2S_3)S_{12}$ by substituting $\perp$ for R_2 in the right-hand side. $R_2 \equiv \varepsilon|S_9S_{10}(S_2S_3(\varepsilon|C_2S_6))S_{12}$.

`procedure R1 is`
`begin`
`  s := s + x;      -- 2`
`  x := x - 1;      -- 3`
`  if x > 0 then`
`    R2;`
`    s := s+1;      -- 6`
`  end if;`
`end R1;`
	`procedure R2 is`
	`begin`
	`  if x /= 0 then`
	`    x := x - 1;    -- 9`
	`    s := s + x;    --10`
	`    R1;`
	`    x := x + 1;    --12`
	`  end if;`
	`end R2;`

Figure 7.2. Mutually recursive procedures without aliasing.

Hence, $Gen(R_2) = \{(s,2),(s,6),(x,12)\}$, and $Kill(R_2) = \{\}$. $Gen(R_2)$, $Kill(R_2)$, and production R_1 are used to generate the following result:
$Gen(R_1) = \{(s,2),(x,3),(s,6),(x,12)\}$, and
$Kill(R_1) = \{(s,i)|i \neq 2,6\} \cup \{(x,i)|i \neq 3,12\}$. □

7.3.3 *In* and *Out* Sets

The following two theorems provide the support for obtaining *In* and *Out* sets for interprocedural analysis.

Theorem 3 (In Set) *The set, $In(R_i)$, of definitions reaching the entry point of procedure R_i is:*

$$In(R_i) = \bigcup_{\substack{t_j = \Delta_{j,1}R_i\Delta_{j,2}R_i\cdots\Delta_{j,\sigma_j}R_i\Delta_{j,\sigma_j+1} \\ \text{where, } t_j \text{ is any term on the right-} \\ \text{hand side of a production rule} \\ \text{such that } R_i \text{ appears in } t_j\text{, and} \\ \Delta_{j,k} \text{ does not contain } R_i \text{ for all } k.}} Gen(\Delta_{j,1}|R_i(\Delta_{j,2}|\cdots|\Delta_{j,\sigma_j}))$$

Proof. A definition can reach the entry point of a procedure only if it reaches the beginning of a call statement to that procedure. In addition, a definition may reach a call statement only if its statement is executed before the call statement and there is a definition-clear path between the two statements. Given that the statements which may be executed before the call statement R_i are those which appear before R_i in a term t_j in a production rule for a procedure in the program under consideration. If *Gen* and *Kill* are known for all procedures in the program, then the set of definitions reaching a call statement from within the calling procedure can be determined. Moreover, any definition in a program has to be defined by either the main procedure or a procedure called by it. Set $In(R_i)$ of definitions reaching the entry point of procedure R_i can be determined by taking the union of all definitions reaching each call statement to R_i. Let t_j be any term on the right-hand side of a production rule such that R_i appears in t_j, and $t_j = \Delta_{j,1}R_i\Delta_{j,2}R_i\cdots\Delta_{j,\sigma_j}R_i\Delta_{j,\sigma_j+1}$, $\Delta_{j,k}$ does not contain R_i for all k. Then $In(R_i)$ is the union of the *Gen* set of the following terms:

$$\begin{aligned}
&\Delta_{j,1}|\Delta_{j,1}R_i\Delta_{j,2}|\cdots|\Delta_{j,1}R_i\Delta_{j,2}\cdots R_i\Delta_{j,\sigma_j}\\
&\equiv\ \Delta_{j,1}|R_i\Delta_{j,2}|\cdots|R_i\Delta_{j,\sigma_j}\\
&\equiv\ \Delta_{j,1}|R_i(\Delta_{j,2}|\cdots|\Delta_{j,\sigma_j})\ (\text{modulo } Gen, Kill). \qquad \square
\end{aligned}$$

For any procedure call statement R, it is easy to prove the following theorem from the definitions.

Theorem 4 (Out Set) $Out(R) = In(R) - Kill(R) \cup Gen(R)$

7.3.4 Definition-use Chains

Given the $Gen(S)$ and $Kill(S)$ sets of definitions for each statement S in the program including the call statements, the sets $Out(S)$ can be determined given that $In(S)$ is known. The set $In(S)$ for the first statement in the body of the main program is the empty set. In addition, the $In(S)$ for the first statement in the body of a procedure can be determined using Theorem 3 above. From $In(S)$, it is easy to determine $In(e)$ for each expression e in statement S. The following algorithm determines the definition-use chains in a program.

Algorithm 1 (Definition-Use Chains from In Set)

```
Input:  A program $P$ where the $In$ set is determined for each expression.
Output: All definition-use chains of program $P$.
procedure Definition-Use-Chains-from-In is
begin
   Def-Use-Chains($P$) := {};
   for each expression $e$ in $P$ loop
      Uses($e$) := the set of variables accessed by $e$;
      From In($e$) and Uses($e$) determine the definition-use chains;
      Add these chains to Def-Use-Chains($P$).
   end loop ;
end Definition-Use-Chains-from-In;
```

Proof. Since the number of expressions in a program is finite, the algorithm terminates after a finite number of steps. The set of definition-use chains in a program is the union of the definition-use chains incident on all expressions in it and hence the algorithm produces all definition-use chains in a program.

Analysis. The algorithm accesses each expression only once and hence its complexity is $O(l)$, where l is the size of the program. In the algorithm, an expression is assumed to have no side effect. Expressions with side effects such as the ones which contain function calls are decomposed into subexpressions such that each function call is in a subexpression by itself. Given Gen and $Kill$ of each function call, the set of definitions reaching each subexpression can be determined in a straight-forward manner. □

7.3.5 Aliases

A major complication to the gathering of precise data flow information is created when the same memory location is referred to by different names. These names are called *aliases.* There are two ways in which aliases can be introduced. First, aliases are introduced when reference parameter-passing maps two distinct variables to the same memory location at the same time. Second, the use of pointers introduces aliases even in the absence of procedure calls.

To perform precise data flow analysis, the aliases of all variables in a program have to be determined. In the past, several interprocedural data flow analysis algorithms have been developed to compute data flow information to various degrees of precision. As an example, Allen's algorithm propagates information by processing the procedures in reverse invocation order [3]. The premise of this approach is that if the side effects of all called procedures are known, the side effects of the calling procedure can also be determined. However, this approach does not apply to programs with recursive procedures. Barth gathers data flow information in a single pass over the program [6]. His method is easy to implement and is quite efficient, but is imprecise in the presence of reference parameters. Chow presents an algorithm for determining aliases in programs that employ a rich set of parameter passing mechanisms and pointer data types [8]. His approach handles the use of pointers bounded to a data type as in Pascal, as well as unbounded pointers that can point to the same locations to which variables map.

For the purpose of solving the reaching definitions problem, we assume that the aliases of each variable in the program are already determined using an algorithm such as the one mentioned in [8]. Each definition of a variable is considered a definition for all of its aliases. Each use of a variable is treated as a use of all of its aliases as well.

7.3.6 Speeding up the Basic Algorithms

Up to this point, we have proposed a methodology to derive precise definition-use chains for a program with recursive procedures. In this subsection, we will propose several algorithms to improve the performance.

Procedure Definition-Use-Chains in Algorithm 2 determines all definition-use chains in a program. The process can be summarized as follows: (1) Construct a call graph from the source program. (2) Find all strongly connected components. Each component represents either a single non-recursive procedure, a self-recursive procedure, or a group of mutually recursive procedures. (3) For all strongly connected components in invocation order, derive *Gen* and *Kill* sets for all procedures in the components. (4) For all strongly connected components in invocation order, use *Gen* and *Kill* sets to find the *In* set for each expression. (5) For each expression, use *In* set to determine the definition-use chains to that expression.

In addition to Procedure Definition-Use-Chains, two more procedures, Gen-Kill-Sets-of-Recursions and In-Set-of-Recursions are used in procedure Reaching-Definition-Chains. Gen-Kill-Sets-of-Recursions derives the same result as that given by Theorem 2. However, the time-consuming substitution process in Theorem 2 has been eliminated. Procedure In-Set-of-Recursions is an indirect result from Theorem 3. Instead of finding all execution paths which may take a very long time to do, each *In* set in a strongly connected component is updated iteratively. By doing this, the information derived from various paths can be shared and the performance can be greatly improved.

The correctness of Algorithm 2 is based on the three procedures used in it. Except for procedure Gen-Kill-Sets-of-Recursions, they are straightforward. The correctness of Gen-Kill-Sets-of-Recursions is proved right after it is given. The worst case of procedure In-Set-of-Recursions is linear and the worst case of procedure Gen-Kill-Sets-of-Recursions is exponential in terms of the number of recursive procedures under analysis. However, in a practical situation, this number is usually bounded by a very small constant and is independent of program size. Both procedures have linear complexity in terms of program length with operations on bit-vectors of size equal to program length and are practical for most real applications.

Algorithm 2 (Definition-Use Chains)

```
    Input:  A program $P$.
    Output: Definition-use chains in program $P$.
    procedure Definition-Use-Chains is
    begin
      Construct a call graph of the program.
      -- $Gen$ and $Kill$ sets
      for each strongly connected component in the graph in reverse
            invocation order loop
         if only a single non-recursive procedure is involved in this component then
            derive its $Gen$ and $Kill$ directly.
         else
            Call Gen-Kill-Sets-of-Recursions to get $Gen(R_i)$ and $Kill(R_i)$.
         end if;
      end loop;
      -- $In$ set
      for each strongly connected component in the call graph in invocation order
loop
        for each procedure $R_i$ in the current group loop
          Let $A$ be the set of procedures $p$ that calls $R_i$ and does not form a recursive
              relationship with $R_i$.
            $In(R_i) := \bigcup_{p \in A} Out(p)$.
            Call $In$-Set-of-Recursions to update $In(R_i)$.
        end loop;
      end loop;
      Call Definition-Use-Chains-from-In to determine the definition-use chains.
    end Definition-Use-Chains;
```

Algorithm 3 (Gen, Kill and In Sets of Recursive Procedures)

```
Input: A set of procedures $R_1, R_2, \ldots, R_n$ forming a mutually recursive relationship
    or a self-recursive procedure if $n = 1$ such that $R_i \longrightarrow f_i(R_1, R_2, \ldots, R_n)$
    for all $1 \leq i \leq n$.
Output: $Gen(R_i)$ and $Kill(R_i)$ for all $1 \leq i \leq n$.
procedure Gen-Kill-Sets-of-Recursions is
   procedure Gen-Kill-Iterative(k) is
   begin
      if $k = 1$ then
         $r_1 := f_1(r_1, r_2, \ldots, r_n)$;
         $r_1 := f_1(r_1, r_2, \ldots, r_n)$;
      else
         Gen-Kill-Iterative($k-1$);
         $r_k := f_k(r_1, r_2, \ldots, r_n)$;
         for i in 1..$k-1$ loop  $r_i := \perp$; end loop ;
         Gen-Kill-Iterative($k-1$);
         $r_k := f_k(r_1, r_2, \ldots, r_n)$;
      end if;
   end Gen-Kill-Iterative;
begin
   for k in reverse 1..n loop
     for i in 1..k loop  $r_i := \perp$; end loop ;
     Gen-Kill-Iterative(k);
   end loop ;
   for k in reverse 1..n loop
     Calculate $Gen(R_k)$ and $Kill(R_k)$ based on $r_k$.
   end loop ;
end Gen-Kill-Sets-of-Recursions;
```

Proof. Let r_i be the intermediate value in computing R_i. From Theorem 1, for each self-recursive procedure $R = f(R)$, we need two steps to get an equivalent non-recursive procedure to R with respect to *Gen* and *Kill*. *I.e.,* $c = f(\perp)$ and $r = f(c)$. Throughout the proof, we will always use c_i to denote $f(\ldots, r_{i-1}, \perp, r_{i+1}, \ldots)$ which will be used to derive an intermediate value of r_i. We start with the elimination process described in Theorem 2. The goal is to carry out the computation process without substitution. Consider the case of $n = 3$. Let's start with $r_1 = f_1(r_1, r_2, r_3)$. From Theorem 1, $c_1 = f_1(\perp, r_2, r_3)$ and $r_1 = f_1(c_1, r_2, r_3)$. Now substituting r_1 in $r_2 = f_2(r_1, r_2, r_3)$ results in $r_2 = f_2(f_1(c_1, r_2, r_3), r_2, r_3)$ and $c_2 = f_2(f_1(c_1, \perp, r_3), \perp, r_3)$ where r_2 is substituted by $\perp$. Therefore, $c_1 = f_1(\perp, \perp, r_3)$ and $r_1 = f_1(c_1, \perp, r_3)$.

Furthermore, $r_2 = f_2(f_1(c_1, c_2, r_3), c_2, r_3)$. Again the c_1 and r_1 corresponding to this r_2 are $c_1 = f_1(\perp, c_2, r_3)$ and $r_1 = f_1(c_1, c_2, r_3)$. Using the same idea we can continue the process to obtain r_n. In the following table a derivation process for $n = 3$ is given.

	1	2	3	4	5	6	7
c_1	$f_1(\perp, r_2, r_3)$	$f_1(\perp, \perp, r_3)$	$f_1(\perp, c_2, r_3)$	$f_1(\perp, \perp, \perp)$	$f_1(\perp, c_2, \perp)$	$f_1(\perp, \perp, c_3)$	$f_1(\perp, c_2, c_3)$
r_1	$f_1(c_1, r_2, r_3)$	$f_1(c_1, \perp, r_3)$	$f_1(c_1, c_2, r_3)$	$f_1(c_1, \perp, \perp)$	$f_1(c_1, c_2, \perp)$	$f_1(c_1, \perp, c_3)$	$f_1(c_1, c_2, c_3)$
c_2		$f_2(r_1, \perp, r_3)$		$f_2(r_1, \perp, \perp)$		$f_2(r_1, \perp, c_3)$	
r_2			$f_2(r_1, c_2, r_3)$		$f_2(r_1, c_2, \perp)$		$f_2(r_1, c_2, c_3)$
c_3					$f_3(r_1, r_2, \perp)$		
r_3							$f_3(r_1, r_2, c_3)$

The execution order is from top to bottom and then from left to right. Please note that c_i and r_i need not exist at the same time (*i.e.*, they can share the same memory locations). In general, the relation through the iteration can be recursively described as follows: Assume $A(k)$ is a procedure to calculate r_k if $k \geq 1$, and $A(0)$ is an empty statement. Then we have the following recursive relationship:

```
begin
    $r_i = \perp \quad \forall\, 1 \leq i \leq k$              -- initialize $r_i$.
    $A(k-1)$                                 -- get $r_1, r_2, \ldots, r_{k-1}$.
    $r_k = f_k(r_1, r_2, \ldots, r_k)$             -- get first $r_k$ in a column.
    $r_i = \perp \quad \forall i \neq k$                 -- reinitialize $r_i$.
    $A(k-1)$
    $r_k = f_k(r_1, r_2, \ldots, r_k)$
end ;
```

This recursive relationship derives procedure Gen-Kill-Iterative(k) except that the first initialization process "$r_i = \perp$ for $1 \leq i \leq k$" has been moved out to improve the performance. The other statements in procedure Gen-Kill-Sets-of-Recursions simply go through each r_k for $1 \leq k \leq n$, and the derivation of $Gen(k)$ and $Kill(k)$ sets is based on r_k.

Analysis. Let $h(k)$ be the number of function operations $f_i(r_1 \ldots r_n)$ required in Gen-Kill-Iterative(k), then $h(k) = 2h(k-1) + 2$; where $h(1) = 2$. Hence, $h(k) = 2^{k+1} - 2$. In addition, in Gen-Kill-of-Recursions, there are $O(n)$ operations to calculate Gen and $Kill$ sets. Each operation needs to manipulate a program up to a length l_Σ; where $l_\Sigma = \Sigma_{i=1}^n l_i$ and l_i is the program size of R_i. Therefore the total complexity of procedure Gen-Kill-of-Recursions is less than $O((\Sigma_{k=1}^n (2^{k+1} - 2) + n) l_\Sigma) = O(2^{n+2} l_\Sigma)$.

Discussion. Since R_i will be processed almost twice as often as R_{i+1}, it is desirable to rearrange $R_1, R_2, \cdots, R_n$ such that $l_1 \leq l_2 \leq \cdots \leq l_n$ before applying Gen-Kill-of-Recursions. If we are only interested in calculating *Gen* and *Kill* sets, we can use prederived $Gen(r_j)$ and $Kill(r_j)$ for all $j > k$, to calculate $Gen(r_k)$ and $Kill(r_k)$, instead of substituting the pattern of r_j into r_k. By doing this, each procedure R_i is completely bounded by its own body. *I.e.* the interface to other procedures is performed by going through *Gen* and *Kill* sets only. Eventually, we can reduce the complexity to $O(\Sigma_{i=1}^{n} 2^{n-i+1} l_i) \leq O((l_\Sigma/n)2^{n+2})$.□

<u>*Algorithm 4 (In Set of Recursions)*</u>

```
Input:
  (1) A set of procedures $R_1, R_2, \ldots, R_n$ forming a
      mutually recursive relationship or a self-recursive procedure if $n = 1$.
  (2) $Gen(R_i)$, $Kill(R_i)$ and an initial $In$ set $In(R_i)$ for all $1 \leq i \leq n$.
Output: $In(R_i)$ for all $1 \leq i \leq n$.
procedure In-Set-of-Recursions is
begin
  loop
    for i in 1..n loop
        Use syntax-directed method to propagate $In(R_i)$
        If $R_j$ is invoked in $R_i$ then
           Let $In_i(R_j)$ be the $In$ set of $R_j$ in $R_i$, and
            $In(R_j) := In_i(R_j) \cup In(R_j)$.
    end loop ;
    exit when no change in $In(R_i)$ for all $1 \leq i \leq n$;
  end loop ;
end In-Set-of-Recursions;
```

Proof. Since $In(R_i)$ is not decreasing through the iteration process and there is a finite upper bound for each of them, the algorithm will terminate eventually with the expected result.

Analysis. Each iteration will propagate information down one call level. To have an *In* set to be propagated to every procedure forming mutually recursive procedures, we need at most n iterations. From Theorem 1, a call to itself does not generate any new *Gen* set. Therefore, the worst case is $n \cdot l_\Sigma$, where $l_\Sigma = \Sigma_{i=1}^{n} l_i$ and l_i is the program size of R_i. Again in real applications n is usually bounded by a very small constant. This algorithm has a linear complexity in terms of program size. $\square$

Example 3 (Mutually Recursive Procedures with Aliasing) The program in Figure 7.3 contains two mutually recursive procedures R_1 and R_2. The production rules for the program are $R \longrightarrow S_{14}S_{15}R_1S_{17}S_{18}$, $R_1 \longrightarrow S_2S_3(\varepsilon|R_1R_2S_6)$, and $R_2 \longrightarrow \varepsilon|S_9S_{10}R_2R_1S_{12}$ where R represents the body of the main procedure. Applying Algorithm 3 to the mutually recursive procedures R_1 and R_2, we have the following derivation table:

	1	2	3
c_1	$f_1(\perp, r_2)$	$f_1(\perp, \perp)$	$f_1(\perp, c_2)$
r_1	$f_1(c_1, r_2)$	$f_1(c_1, \perp)$	$f_1(c_1, c_2)$
c_2		$f_2(r_1, \perp)$	
r_2			$f_2(r_1, c_2)$

From this table we can easily generate the following table:

	1	2	3
c_1	S_2S_3	S_2S_3	S_2S_3
r_1	$S_2S_3(\varepsilon \mid c_1r_2S_6)$	S_2S_3	$S_2S_3(\varepsilon \mid c_1c_2S_6)$
c_2		ε	
r_2			$\varepsilon \mid S_9S_{10}c_2r_1S_{12}$

Before generating the *Gen* and *Kill* sets, an algorithm like the one presented in [8] should be used to determine the aliases in the program. It is easy to show that the variables x, $x1$, and $x2$ are aliases. Hence, any definition to any one of these variable must be considered as a definition of the three of them. To simplify the discussion we use $\overline{x}$ to denote x, $x1$, and $x2$ and it should be obvious from the location of the definition which specific variable name $\overline{x}$ refers to. The variable name $\overline{s}$ is also used to denote the three aliases s, $s1$, and $s2$ in the program. From the value of r_2 in the table above, $Gen(R_2)$ and $Kill(R_2)$ can then be expressed as follows:

$$\begin{aligned} Gen(R_2) &= Gen(\varepsilon|S_9S_{10}c_2r_1S_{12}) = \{(\overline{s},2),(\overline{s},6),(\overline{x},12)\} \\ Kill(R_2) &= Kill(\varepsilon|S_9S_{10}c_2r_1S_{12}) = \{\} \end{aligned}$$

```
with Text_IO;
procedure R is
  s,x:integer;
  package Int_IO is new Text_IO.Integer_IO(Integer);

  procedure R2 (s2, x2:in out integer);
  procedure R1 (s1, x1:in out integer) is
  begin
    s1 := s1 + x1; x1 := x1 - 1;           -- 2,3
    if x1>0 then
      R1 (s1, x1); R2 (s1, x1);
      s1 := s1 + 1;                        -- 6
    end if;
  end R1;

  procedure R2 (s2, x2:in out integer) is
  begin
    if x2 /= 0 then
      x2 := x2 - 1; s2 := s2 + x2;         --9,10
      R2 (s2, x2); R1 (s2, x2);
      x2 := x2 + 1;                        --12
    end if;
  end R2;

begin
  Int_IO.Get(x); s:=0; R1(s,x);           --14,15,16
  Int_IO.Put(x); Int_IO.Put(s);           --17,18
end R3;
```

Figure 7.3. Mutually recursive procedures with aliasing.

where $c_2 = \varepsilon$, $r_1 = S_2S_3(\varepsilon|c_1c_2S_6)$ and $c_1 = S_2S_3$. To determine $Gen(R_1)$ and $Kill(R_1)$, use the following value of r_1 generated by Algorithm 3: $r_1 = S_2S_3(\varepsilon|c_1r_2S_6)$; where $c_1 = S_2S_3$. With the prederived $Gen(R_2)$ and $Kill(R_2)$, we have

$$\begin{aligned} Gen(R_1) &= \{(\overline{s},2),(\overline{x},3),(\overline{s},6),(\overline{x},12)\} \\ Kill(R_1) &= \{(\overline{s},i)|i \neq 2,6\} \cup \{(\overline{x},i)|i \neq 3,12\} \end{aligned}$$

The set of definitions reaching the entry point of each procedure can be determined as follows:

$$\begin{aligned} In(R_1) &= Gen(S_{14}S_{15}) \cup Gen(S_2S_3) \cup Gen(S_9S_{10}R_2) \\ &= \{(\overline{s},2),(\overline{s},6),(\overline{s},10),(\overline{s},15),(\overline{x},3),(\overline{x},9),(\overline{x},12),(\overline{x},14)\} \\ In(R_2) &= Gen(S_2S_3R_1) \cup Gen(S_9S_{10}) \\ &= \{(\overline{s},2),(\overline{s},6),(\overline{s},10),(\overline{x},3),(\overline{x},9),(\overline{x},12)\} \end{aligned}$$

From these two sets, are as follows in terms of the original variable names:

$$\begin{aligned} In(R_1) &= \{(s1,2),(s1,6),(s2,10),(s,15),(x1,3),(x2,9),(x2,12),(x,14)\} \\ In(R_2) &= \{(s1,2),(s1,6),(s2,10),(x1,3),(x2,9),(x2,12)\} \end{aligned}$$

The *In* and *Out* of each statement in the program can easily be found using Theorem 4 and the formulae given in Section 7.2. Once the *In* set of reaching definitions has been determined for each statement in the program, building the definition-use chains is a straightforward operation and is described by Algorithm 1.

□

7.3.7 Discussions

In this section we have provided a mathematical support for the interprocedural definition-use dependency analysis for recursive procedures. We were able to reduce an infinite number of execution paths for recursive procedures to only a finite number of execution paths and make the generation of the precise data flow information for recursive procedures feasible.

For a real programming language, such as Ada, C, COBOL, or Pascal, there are several situations that must be handled. The first is the function call. Each function call can be considered as a procedure call along with the generation of a value. Therefore the analysis of recursive functions is similar to what we have done for the analysis of recursive procedures. The execution order can also be decided from the program context. For example, if a function is used in an expression, then the function should be handled first before the expression can be analyzed.

The second situation is the passing mechanism of formal parameters. Only call-by-value, call-by-value-result, and call-by-reference are discussed here. Call-by-value passing mechanism can be considered as a virtual assignment to copy the value of an actual parameter to a virtual memory before we process the interprocedural data flow analysis. Nothing will be returned at the return point from the procedure for call-by-value formal parameters. Call-by-value-result passing mechanism is similar to the call-by-value except the value at the virtual memory will be copied back to the formal parameter at the return point. Call-by-reference passing mechanism deals with the virtual memory exactly the same as the actual parameter. By combining the techniques on handling aliases, we are able to generate the right aliasing classes which can be used to generate precise definition-use dependency analysis.

7.4 Related Work

Although interprocedural data flow analysis algorithms do exist [3, 6, 9, 10, 15, 38, 46], they do not provide detailed information (*i.e.,* the locations of definitions and uses that reach across procedure boundaries) needed for interprocedural data flow testing. Existing interprocedural data flow analysis algorithms can be classified into three categories: in-line substitution, flow-insensitive analysis, and flow sensitive analysis.

In *in-line substitution*, a call statement is substituted by the body of the called procedure [3]. In addition to the obvious problem of memory requirements, in-line substitution has other inherent problems. Both scoping of local variables in procedures and binding of formal and actual parameters are difficult because the entire module is viewed as a single procedure. Additionally, recursive procedures cannot be represented.

A problem is *flow-insensitive* if information about control internal to subroutines is not needed to compute the final data flow solution. The program representation widely used to solve such problems is the *call-graph* in which each node represents a procedure and each edge represents a procedure call site [3, 6, 10, 15, 38, 46]. The call graph is not sufficient for computing the definition-use information across procedure boundaries because it has no information about the control flow in individual procedures. These algorithms are used in applications such as determining the set of variables that *may* be defined, used or modified by a procedure call statement.

In *flow-sensitive* data flow analysis, the control flow information of the called procedure is accounted for [9]. Flow-sensitive data flow analysis produces information such as the set of variables that *must* be defined, used or modified by the called procedure. These variables have to be defined, used or modified along *all* paths (as opposed to along *some* paths in the case of flow-insensitive analysis) in the called procedure. Callahan [9] uses a *program summary graph* to represent a program. The program summary graph summarizes some of the required information at call sites but this information does not indicate the locations of definitions and uses that reach across procedure boundaries.

There have been very few attempts in the past to solve the interprocedural reaching definitions problem [24, 51]. In [51], the *call-strings* approach that can be used to solve the interprocedural reaching definitions problem is presented. The call-strings approach, however, cannot effectively handle programs with recursive procedures. Harrold and Soffa [25] have developed a method to solve the reaching definition problem using an extended program summary graph. Her method, however, requires the transformation of the program into intermediate code and has a complexity[2] of $O(n^2)$ on bit-vector operations with size l; where l is the program length and n is the size of the program summary graph [9, 25]. The proposed algorithm is faster than Harrold's approach in most cases because the proposed approach has linear complexity in terms of program length with operations on bit-vectors of size equal to program length for for all cases assuming the number of nodes of any strongly connected component in the program call graph is bounded by a constant. The new algorithm also has the advantage of being syntax-driven. The interaction between the user and the environment can be in terms of the source code instead of an intermediate form of representation. This greatly simplifies the user interface. However, Harrold's algorithm can be applied incrementally, while the proposed approach requires incremental aliasing analysis which is still under investigation.

[2]The theoretical worst case of n is $O((l + c_m^2)(v_p * v_g))$; where c_m is the maximum number of call sites in any procedure, v_g is the total number of global variables, and v_p is the average number of actual parameters at call sites. However, n is expected to be $O(l \cdot v_g)$ for real programs [9].

CHAPTER 8
INTERPROCEDURAL DATA FLOW ANOMALY DETECTION

8.1 Introduction

A data flow anomaly is the phenomenon that happens between definitions, undefinitions, and references of identifiers. There are three common data flow anomalies: a *defined-undefined* anomaly occurs when a variable is defined but never used within its scope; a *defined-defined* anomaly occurs when a variable is defined and defined again before it is used; and an *undefined-referenced* anomaly occurs when a variable is used before it has been defined.

Methods for the detection of data flow anomalies are usually limited to single procedure or non-recursive interprocedural situation [19, 20, 28, 56]. Fairfield and Hennel [18] proposed a method which claimed to to detect the data flow anomalies of recursive procedures. However, his approach is based on the assumption that all recursive calls will be substituted by all possible nonrecursive execution paths. Calling context is ignored and the effect of the other paths containing recursive calls is not accounted for in his approach. Fairfield and Hennel's approach may sometimes generate improper results as illustrated in the example in Figure 8.1. The regular expression denoting the set of possible paths in the example can be stated as follows:

$$R = S_2(S_3|S_4S_5RS_7|S_8S_9RS_{11})$$

Fairfield and Hennel's approach will generate the following regular expression[1]:

$$R = S_2S_3|(S_2S_4S_5)^+S_2S_3S_7^+|(S_2S_8S_9)^+S_2S_3S_{11}^+$$

[1] S^+ denotes S, SS, $\ldots$, etc.

```
-- a,b are globals
procedure R is
begin
  case a is                                       -- S2
    when 1       => Put(b);                       -- S3
    when -20..0  => b := a;                       -- S4
                    a := b + 2;                   -- S5
                    R;                            -- S6
                    Put(b);                       -- S7
    when   2..20 => a := b;                       -- S8
                    b := a - 3;                   -- S9
                    R;                            -- S10
                    Put(a);                       -- S11
  end case;
end R;
```

Figure 8.1. An example of data flow anomalies.

which is wrong because of the following two reasons:

1. Calling context is ignored since neither $(S_2S_4S_5)^iS_2S_3S_7^j$ nor $(S_2S_8S_9)^iS_2S_3S_{11}^j$ is a real execution path if $i \neq j$.

2. Execution paths which pass through both recursive calls, such as S_2-S_8-S_9-S_{10}-S_2-S_4-S_5-S_6-S_2- S_3-S_7-S_{11}, are ignored.

Although the first case does not introduce improper results, the second case does and is manifested by the defined-defined anomaly with respect to b from statement S_9 to statement S_4, which can not be detected by Fairfield and Hennel's approach.

This chapter introduces an approach to detect data flow anomaly in the interprocedural level, including self-recursive and mutually-recursive procedures. The chapter is organized as follows: Section 8.1 introduces the problem. Section 8.2 summarizes relevant contributions by others and their restrictions. Section 8.3 introduces our approach.

8.2 Background

The detection of data flow anomalies to improve program reliability is an important technique, and its methods are still being refined. Fosdick and Osterweil [20] first approached the problem. In their DAVE system [40], the problem of detecting data flow anomalies is solved using depth first search. More recent solutions use standard global data flow analysis algorithms and are being to concurrent programs [54].

Both static and dynamic data flow analyses are useful tools for data flow anomalies detection [11, 18, 19, 20, 26, 28, 56]. Static data flow analysis methods have problems with array subscript evaluation which dynamic data flow analysis methods does not. On the other hand, static analysis will reveal all data flow anomalies in a program, while dynamic approach is capable of detecting only data flow anomalies along those paths that are actually executed.

Existing static data flow anomaly analysis techniques analyze in three phases [18, 19, 20, 28, 56]. The first phase detects the data flow anomalies. Two different techniques are employed for data flow anomaly detection: path expressions and standard data flow analysis techniques. In path expressions, actions on each variable along the program execution paths are expressed in terms of regular expressions. The regular expressions are then searched for patterns of data flow anomalies with respect to each variable [18, 19]. Standard algorithms from compiler code optimization for solving the live variable problem and the availability problem are also used to solve the data flow anomaly detection problem [20, 54, 56]. This chapter extends the approach of Jachner and Agarwal [28] to handle programs with recursive subprograms.

Let referenced, defined, undefined, and null be four types of data actions applied on a variable. A variable is *referenced* (r) when its value is used, *defined* (d) when a value is assigned to it, *undefined* (u) when it has no known value, and *null* (l) if none of the other three actions take place. For instance, local variables are undefined at the beginning and at the end of their scope; variable `d` is defined, and variables `u` and `v` are used in the assignment statement `d:=u-v`. All variables except `d`, `u` and `v` have a null action in this assignment. A *path expression* is a sequence of actions on a variable along a path in a program written from left-to-right corresponding to the order in which these actions occur along the path. The data flow anomalies can now be expressed in terms of path expressions. For instance, a reference to an undefined variable corresponds to a path expression of the form $\rho_1 ur \rho_2$, where ρ_1 and ρ_2 stand for arbitrary path expressions [28].

In our approach, a flow graph $G(N, E, n_0)$ is used to represent the program under consideration; where N is the set of nodes, E is the set of edges in the graph, and $n_0 \in N$ is a unique entry node. The edges in G correspond to control paths among the nodes. Each node of G is either a simple statement, a logical expression, or a call statement. Let the term, *token*, represent a simple variable, a group of aliased variables, or an entire array. Moreover, let $P(n;\alpha)$, $P(\rightarrow n;\alpha)$, and $P(n \rightarrow;\alpha)$ denote path expressions of a token α at node n, on paths entering node n, and on paths leaving node n, respectively. Furthermore, suppose that $P(n;\alpha)$, $P(\rightarrow n;\alpha)$, and $P(n \rightarrow;\alpha)$ have been determined for all nodes $n \in N$ and for all tokens in the program. Then, there is an anomaly on all paths through n if the following equation holds:

$$P(\rightarrow n;\alpha)P(n;\alpha)P(n \rightarrow;\alpha) \quad = \quad \rho_1 xy \rho_2 + \rho_3; \text{ where } xy \in \{ur, dd, du\}$$

It is seen from this equation that data flow anomalies can be detected by first considering each individual node of the flow graph, followed by an evaluation of path expressions at node n and on paths leaving and entering node n. A useful outcome of this approach is that all tokens which have the same form of path expressions at n, on paths leaving n, and on paths entering n will have the same anomalies, if any. Thus it is convenient to group such tokens into a set, called a *path set* and process all the token in the set collectively.

Jachner and Agarwal [28] showed that the path sets can be calculated using the *Avail*, *Live*, and *Reach* algorithms as follows: (1) Define three mutually exclusive sets $gen(n)$, $kill(n)$, and $null(n)$ of tokens at node n. $Kill(n)$ undoes whatever action $gen(n)$ represents. No action at all is represented by $null(n)$. (2) Avail, Reach, and Live can then be viewed as algorithms for determining sets of tokens whose path expression on paths leaving node n is $g\rho + \rho_1$, and on paths entering node n is $\rho g + \rho_1$, irrespective of how g, k, and l are defined, as long as the following properties hold:

$$gen(n) \cap kill(n) = gen(n) \cap null(n) = kill(n) \cap null(n) = \emptyset$$

$$gl = lg = g; kl = lk = k; l + l = ll = l.$$

(3) A judicious definition of g, k, l in terms of the data actions r, d, u, and l allows the use of Avail, Live, and Reach algorithms to determine the required path sets.

8.3 Approach

In this section, a general methodology for data flow anomaly detection in the presence of recursive procedures is presented. The essence here is to devise algorithms for Reach, Avail and Live analysis capable of handling recursive procedures.

A two step approach is used to carry out the Reach, Avail and Live-variable problems. In the first step the *Gen* and *Kill* sets are determined using the approach presented in Chapter 7. In the second step, after the *Gen* and *Kill* sets have been determined for each node on the control flow graph of each procedure, Algorithm 5 solves the Available definitions problem, Algorithm 7 solves the Reach definitions problem, and Algorithm 6 solves the live-variable problem [5].

Algorithm 5 (Avail)

```
    Input: A control flow graph with e_gen and e_kill determined for each node;
        where $Gen(n)$ is the set of expressions generated at node $n$ and
        $Kill(n)$ is the set of expressions killed at node $n$.  The initial node $n_0$.
    Output: The set  Avail for each node in the graph.
     procedure Avail  is
     begin
        $Avail(n_0) = \emptyset$;
        for each node $j \neq n_0$ in the control flow graph  loop
            $Avail(j) := \Omega$;
             end loop;
        – where Omega is the set of all expressions in the control flow graph.
          for each node $j \neq n_0$ in the control flow graph  loop
              $Avail(j) := \bigcap_{k \text{ is a predecessor of } j}(Avail(k) - Kill(k) \cup Gen(k))$;
          exit when no changes to Avail's occur;
        end loop;
     end Avail;
```

Algorithm 6 (Live-variable)

```
    Input: A control flow graph with Gen and Kill determined for each node.
         where $Kill(k)$ is set of variables definitely assigned values in $k$ prior to
           any use of that variable in $k$, and $Gen(k)$ is the set of variables whose
values
         may be used in $k$ prior to nay definition of that variable.
    Output: The set  Live for each node in the graph.
     procedure Live  is
     begin
        for each node $j$ in the control flow graph  loop
           $Live(j) := \emptyset$;
           end loop;
        for each node $j$ in the control flow graph  loop
           $Live(j) := \bigcup_{k \text{ is a successor of } j}(Live(k) - Kill(k) \cup Gen(k))$;
           exit when no changes to Live's occur;
        end loop;
     end Live;
```

Algorithm 7 (Reach)

```
Input: A control flow graph with $Gen$ and $Kill$ determined for each node.
Output: The set $Reach$ for each node in the graph.
 procedure Reach is
 begin
     for each node $j$ in the control flow graph loop $Reach(j) := \emptyset$;
        end loop;
     loop
        for each node $j$ in the control flow graph loop
             $Reach(j) := \bigcup_{k \text{ is a predecessor of } j}(Reach(k) - Kill(k) \cup Gen(k))$;
        exit when no changes to Reach's occur;
     end loop;
 end Reach;
```

For the purpose of solving the Avail, Live, and Reach problems, we assume that the aliases of each variable in the program are already determined using an algorithm such as the one mentioned in [8]. Each definition of a variable is considered a definition for all of its aliases. Each use of a variable is treated as a use of all of its aliases as well.

8.4 Related Work

Early work in the data flow anomaly analysis area, particularly that of Fosdick and Osterweil, resulted in the *Dave* software verification tool [40]. This tool is batch oriented and requires that the entire source code to be processed each time any of it changes. The basic scheme of *Dave* is to use data flow analysis techniques to process the entire program to attempt to detect paths of execution that may result in the program producing anomalous results. If a data flow anomaly is found, the path of the anomaly is reconstructed and printed out.

Shortly afterwards, Huang showed that the presence of data flow anomalies could be detected through program instrumentation and dynamic execution [26].

Moreover, Masinter at Xerox has added several types of program checking to the LISP based *Scope Programming Environment* [36]. In this environment, several types of errors, including type errors, can be detected. The intent of *Scope* is much different from that of *Dave.* The analysis in *Scope* is totally query driven. No information is available to the user unless the user specifically asks for it. The strategy of *lazy evaluation* was used to defer any overhead until the user actually requests the information. The undesirable side effect of a lazy evaluation based system is that no checking is automatically done. The programmer must guess what harmful side effects he may have introduced by making a change. He must then ask *Scope* to check a particular condition. No secondary effects are found automatically. On the hand, the user does have the ability to query an interactive facility of the same power as the *Dave* system.

8.5 The Limits of Data Flow Anomaly Analysis

Data flow anomaly analysis cannot determine that a program is correct, it can only indicate that the program may be incorrect. This distinction is significant. Data flow anomaly analysis is capable of determining that if the code is executed under a given set of conditions, the code will produce erroneous results. Data flow anomaly analysis cannot discover, in general, that the program will ever execute in that manner, nor can it prove that if it executes in another manner it will produce correct results. A similar problem occurs while detecting syntax errors in a normal compiler; a program that is syntactically correct will not necessarily run correctly, but one with syntax errors will definitely run incorrectly. The main purpose of providing the output of a data flow anomaly analysis to the user is not to prove that the program is correct, but to find as many mistakes in the program at the earliest possible point in the development cycle.

CHAPTER 9
CONCLUSIONS AND FURTHER STUDY

This dissertation has described a strategy based on incremental data flow analysis for maintaining the consistency of a suite of test cases and related data flow information with the state of a software system being developed or maintained. Based on this approach, the impact of change on the system can be measured by the number of altered definition-use paths. The number of such paths might be used in choosing between different implementations of a change, or in allocating resources for testing. Knowledge of the location of altered paths can aid in designing new test cases and in locating faults once errors are discovered.

Data flow testing criteria are based on the tracking of variable values through a program. These values are identified by the variable's name and definition locations. When a variable name denotes multiple values, however, they can be difficult to track. This problem comes about, for example, with the use of arrays and pointers. Extending data flow based testing to languages which allow arrays poses interesting problems. Ideally, one would like to treat each array element as a separate variable. However, it is not in general possible to statically determine the particular element to which the array occurrence, A[i], refers. The simplest solution is to treat the entire array as a single element. Each definition or use of an array element is treated as a definition or use of the entire array. While this method is easy to implement, it fails to take into account any information about particular array elements. This can lead to test cases being considered adequate even though intuitively they fail to exercise the program thoroughly. Further study is clearly needed to identify acceptable strategies for dealing with this problem.

The failure to properly manage and make effective use of previously developed test cases is a major impediment to ensuring the quality of modified programs. We are optimistic that a tool based on the strategies described in this dissertation can efficiently maintain a useful set of test cases and provide the means for rapidly assessing the impact of program modifications.

In addition, we have derived a new algorithm to generate precise interprocedural definition-use dependency information. The proposed algorithm has linear complexity for most cases and is practical in most software applications. Handling aliasing incrementally is still a problem that has to be addressed in order for the algorithms developed in this dissertation to support inter-procedural definition-use dependency analysis incrementally.

An efficient approach to detect data flow anomalies at the interprocedural level has been presented. Unlike existing approaches, our approach can handle self and mutually recursive procedures. The proposed method has linear complexity in terms of program length with operations on bit-vectors of size equal to program length for most cases and is practical in most software applications. This approach uses static analysis and hence inherits its limitations. The major problem seems to be the approach's inadequate handling of array and pointer variables. Further research on these problems can be of significant importance.

Furthermore, an interactive programming environment which includes tools for regression testing and fault localization has been prototyped. The tools in the environment are specified in terms of attribute grammars, and the Cornell Synthesizer Generator [43] has been used to generate them. Expanding this approach to handle programs with more than one procedure was the motivation behind the development of the definition-use dependency analysis algorithm in Chapter 7. The evaluation of these tools is still in an early stage, and more work is needed to assess their usefulness in real applications.

REFERENCES

[1] H. Agrawal, R.A. DeMillo, and E. H. Spafford, "A Process State Model to Relate Testing and Debugging," SERC-TR-27-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1988.

[2] H. Agrawal and E. H. Spafford, "An Execution Backtracking Approach to Program Debugging," *SERC-TR-22-P Software Engineering Research Center*, Purdue University, West Lafayette, IN, 1988.

[3] F. E. Allen, "Interprocedural Data Flow Analysis," *IFIP Information Processing 74,* North-Holland Publishing Company, Amsterdam, 1974, pp. 398–402.

[4] F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Communications of ACM*, Vol. 19, No. 3, March 1976, pp. 137–147.

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

[6] J. M. Barth, "A Practical Interprocedural Data Flow Analysis Algorithm," *Communications of ACM*, Vol. 21, No. 9, September 1978, pp. 724–736.

[7] L. A. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, Vol. 3, 1976, pp. 225–252.

[8] M. Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis," *ACM Trans. on Programming Languages and Systems*, Vol. 12, No. 3, July 1990, pp. 341–395.

[9] D. Callahan, "The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation,* Atlanta, GA, June 1988, pp. 47–56.

[10] M. D. Carroll and B. G. Ryder, "An Incremental Algorithm for Software Analysis," *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* also *SIGPLAN Notices,* Vol. 22, No. 1, Jan. 1988, pp. 30–40.

[11] F. T. Chan and T. Y. Chen, "AIDA–A Dynamic Data Flow Anomaly Detection System for Pascal Programs," *Software–Practice and Experience,* Vol. 17, No. 3, March 1987, pp. 227–239.

[12] L. A. Clarke, A. Podgurski, and D. J. Richardson, "A Comparison of Data Flow Path Selection Criteria," *Proc. 8th IEEE Int. Conf. Software Eng.*, London, UK, Aug. 1985, pp. 244–251.

[13] J. S. Collofello and J. J. Buck, "Software Quality Assurance for Maintenance," *IEEE Software*, Vol. 4, No. 5, Sep. 1987, pp. 46–51.

[14] J. S. Collofello and L. Cousins, "Towards Automatic Software Fault Location through Decision-to-Decision Path Analysis," *National Computer Conference*, Vol. 56, 1987, pp. 539–544.

[15] K. Cooper and K. Kennedy, "Interprocedural Side Effect Analysis in Linear Time," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation,* Atlanta, GA, June 1988, pp. 57–66.

[16] Keith D. Cooper, Ken Kennedy, "Efficient Computation of Flow-Insensitive Interprocedural Summary Information–A Correction," *ACM SIGPLAN Notices* Vol. 23, No. 4, 1988, pp 35–42.

[17] N. M. Delisle, D. Menicosy, and M. Schwartz, "Viewing a Programming Environment as a Single Tool," *ACM SIGPLAN Notices,* Vol. 19, No. 5, 1984, pp. 49–56.

[18] P. Fairfield and M. A. Hennel, "Data Flow Analysis of Recursive Procedures," *ACM SIGPLAN Notices*, Vol. 23, No. 1, Jan. 1988, pp. 48–57.

[19] I. R. Forman, "An Algebra for Data Flow Anomaly Detection," *7th Int'l Conf. on Software Engineering*, March 1984, pp. 278–286.

[20] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," *ACM Computing Surveys*, Vol. 8, No. 3, Sep. 1976, pp. 305–330.

[21] P. G. Frankl and E. J. Weyuker, "A Data Flow Testing Tool," *Softfair II: A Second Conference on Software Development Tools, Techniques and Alternatives*, San Francisco, CA, Dec. 1985, pp. 46–53.

[22] P. G. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. on Software Engineering*, Vol. 14, No. 10, Oct. 1988, pp. 1483–1489.

[23] M. J. Harrold and M. L. Soffa, "An Incremental Approach to Unit Testing during Maintenance," *Proc. Conf. on Software Maintenance,* Phoenix Arizona, Oct. 1988, pp. 362–367.

[24] M. J. Harrold and M. L. Soffa, "Interprocedural Data Flow Testing," *Proc. ACM SIGSOFT'89 3rd Symp. on Software Testing, Analysis, and Verification,* Key West, Florida, 1989, pp. 158–167.

[25] M. J. Harrold and M. L. Soffa, "Computation of Interprocedural Definition and Use Dependencies," submitted.

[26] J. C. Huang, "Detection of Data Flow Anomaly Through Program Instrumentation," *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp. 227–236.

[27] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 729-1983, IEEE Press, 1983.

[28] J. Jachner and V. Agarwal, "Data Flow Anomaly Detection," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 4, July 1984, pp. 432–437.

[29] J. Keables, K. Robertson, and A. Mayrhauser, "Data Flow Analysis and its Application to Software Maintenance," *Proceedings of Conference on Software Maintenance,* Phoenix Arizona, 1988, pp. 335–347.

[30] K. Kennedy, "A Survey of Data Flow Analysis Techniques," in S. S. Muchnick and M. D. Jones, editors, *Program Flow Analysis: Theory and Applications,* Prentice-Hall, Englewood Cliffs, New Jersey, 1981, pp. 189–232.

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

[32] B. Korel and J. Laski, "STAD–A System For Testing And Debugging: User Perspective," *Second IEEE Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, Canada, July 1988, pp. 13–20.

[33] J. Laski and B. Korel, "A Data Flow Oriented Testing Strategy," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 3, May 1983, pp. 347–354.

[34] H. K. N. Leung, *A Study of Regression Testing, TR 88-15, Department of Computing Science, The University of Alberta,* Edmonton, Alberta, Canada, 1988.

[35] B. P. Lientz and E. B. Swanson, "Problems in Application Software Maintenance," *Communications of ACM*, Vol. 24, No. 11, Nov. 1981, pp. 763–769.

[36] L. M. Masinter, "Global Program Analysis in an Interactive Environment," *Xerox Report* SSL–80–1, January 1980.

[37] R. Medina-Mora and P. H. Feiler, "An Incremental Programming Environment," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. , 1981, pp. 472–482.

[38] E. W. Myers, "A Precise Interprocedural Data Flow Algorithm," *Conference Record of the Eight Annual ACM Symposium on Principles of Programming Languages,* Williamsburg, VA, January 1981, pp. 219–230.

[39] S. Ntafos, "On Required Element Testing," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. , Nov. 1984, pp. 795–803.

[40] L. J. Osterweil and L. D. Fosdick, "DAVE—A Validation Error Detection and Documentation System for Fortran Programs," *Software Practice and Experience*, Vol. 6, No. 4, Sep. 1976, pp. 473–486.

[41] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 4, April 1985, pp. 367–374.

[42] T. Reps, *Generating Language-Based Environments*, M.I.T. Press, Cambridge, MA, 1984.

[43] T. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual,* Springer-Verlag, New York, 3rd ed., 1988.

[44] T. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editor*, Springer-Verlag, New York, 1989.

[45] B. K. Rosen, "High-Level Data Flow Analysis," *Communications of ACM,* Vol. 20, No. 10, October 1977, pp. 712–724.

[46] B. K. Rosen, "Data Flow Analysis for Interprocedural Languages," *Journal of the ACM,* Vol. 26, No. 2, April 1979, pp. 322–344.

[47] B. G. Ryder and M. D. Carroll, "Incremental Data Flow Analysis via Attributes," Tech. Rep. LCSR-TR-93, Dept. of Computer Science, Rutgers Univ., New Brunswick, NJ, June 1987.

[48] B. G. Ryder and M. C. Paull, "Elimination Algorithms for Data Flow Analysis," *ACM Computing Surveys*, Vol. 18, No. 3, Sep. 1986, pp. 277–316.

[49] B. G. Ryder and M. C. Paull, "Incremental Data-Flow Analysis," *ACM Trans. on Programming Languages and Systems*, Vol. 10, No. 1, Jan. 1988, pp. 1–50.

[50] R. L. Sedlmeyer, W. B. Thompson, and P. Johnson, "Knowledge-based Fault Localization in debugging," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging*, March 1983, pp. 25–31.

[51] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis," in S. S. Muchnick and M. D. Jones, editors, *Program Flow Analysis: Theory and Applications,* Prentice-Hall, Englewood Cliffs, New Jersey, 1981, pp. 189–232.

[52] M. L. Shooman, *Software Engineering: Design, Reliability, and Management,* McGraw-Hill Book Company, New York, 1983.

[53] A. Taha, S. Thebaut, and S. S. Liu, "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis," *Proc. COMPSAC 89*, Orlando, FL, September 1989, pp. 527–534.

[54] R. N. Taylor and L. J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 3, May 1980, pp. 265–278.

[55] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of ACM*, Vol. 24, No. 9, September 1981, pp. 563–573.

[56] C. Wilson and L. Osterweil, "Omega—A Data Flow Analysis Tool for the C Programming Language," *Proc. COMPSAC 82*, Nov. 1982, pp. 9–18.

[57] F. K. Zadeck, "Incremental Data Flow Analysis in a Structured Program Editor," *Proceeding of the ACM SIGPLAN '84 Symposium on Compiler Construction*, also in *SIGPLAN Notices*, Vol. 19, No. 6, June 1984, pp. 132–143.

BIOGRAPHICAL SKETCH

Abu-Bakr M. Taha was born on July 3, 1954, in the Arab Republic of Egypt. He received a B.S. and M.S. degree in electrical engineering from Menoufia University, Egypt, in 1977 and 1982 respectively. Mr. Taha also received an M.S. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1985. Between 1977 and 1983, he worked in the College of Electronic Engineering, Menoufia University, Egypt. He began his Ph.D. program in computer and information sciences at the University of Florida in Fall 1986.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Stephen S. Yau, Chairman
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Sying-Syang Liu, Cochairman
Visiting Assistant Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Randy Chow

Randy Chow
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Stephen Thebaut
Assistant Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Ronald Rasch
Assistant Professor of Accounting

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

May, 1991

Hubert A. Bevis
for Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School