National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

A TEST SELECTION STRATEGY
BASED ON INPUT-OUTPUT RELATION ANALYSIS

by

Bo Yang

A thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Science

University of Ottawa
Ottawa, Ontario, Canada
March 1988

© Bo Yang, Ottawa, Canada, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-46882-3

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

Dynamic program testing is widely used to affirm the quality of software by executing the software under test in a controlled environment over a finite set of tests. It generally falls into two categories: structural testing and functional testing. A structural testing strategy consists of two seemingly distinct phases: test path selection and test data selection.

Most path selection criteria are based on control flow analysis which examines the branch and loop structure of a program. Previous work has shown that path selection criteria based merely on control flow information are not enough to insure the quality of software under test.

In this thesis, a class of data flow oriented path selection criteria is proposed. They are based on the identification of associations between each output variable and all input variables that influence that output variable in a source program. These criteria require each such association to be examined at least once during testing.

It is shown that these criteria are stronger than other data flow oriented criteria. These criteria also bridge the gap between the relatively weak branch coverage and the impractical path coverage. More importantly, an effort is made toward using program semantics such as input-output relations in the path selection process. This is an improvement over using program syntax alone.

The complexity of the proposed criteria is then discussed, and a brief discussion on test data selection is given.

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to my thesis supervisor, Dr. Hasan Ural, for his constructive advice, guidance, time, support, encouragement and patience throughout my graduate studies. It was a great pleasure working with him.

I wish to thank Drs. T. Y. Cheung and R..L. Probert for their stimulating course instructions.

My parents deserve the most thanks. Their infinite encouragement, moral support and understanding have encouraged me to pursue my education at this level.

I am very grateful to my grandparents for their love and financial support and for offering me the opportunity to come and study in Canada.

TABLE OF CONTENTS

Content	Page
Abstract	i
Acknowledgements	ii
Table of Contents	iii
List of Figures	v
List of Tables	vi
Chapter 1 Introduction	**1**
1.1 Background and Related Work	1
1.2 Motivation	3
1.3 Major Contributions of the Thesis	4
1.4 Outline of the Thesis	5
Chapter 2 Test Path Selection Criteria	**6**
2.1 Terminology	6
2.1.1 Control Flow Related Concepts	6
2.1.2 Data Flow Related Concepts	8
2.1.3 Concepts Related to Input-Output Relations	9
2.1.4 Coverage Related Concepts	11
2.2 Assumptions	12
2.3 Commonly-Referenced Control Flow-Based Criteria	14

2.4 Commonly-Referenced Data Flow-Based Criteria 16
2.4.1 Rapps and Weyuker's Family of Criteria 16
2.4.2 Ntafos's Required K-Tuples Criteria 18
2.4.3 Laski and Korel's Criteria 18
2.4.4 A Comparison 19
2.5 The Proposed Criteria 20
2.5.1 Definitions of the OI-Paths Criteria 20
2.5.2 Properties of the OI-Paths Criteria 20
2.5.3 Complexity of the OI-Paths Criteria 22
Chapter 3 A Comparison of the Path Selection Criteria 25
3.1 Relative Strength of the Existing Criteria 25
3.2 OI-Paths *vs* Control Flow-Based Criteria 28
3.3 OI-Paths *vs* Data Flow-Based Criteria 32
3.3.1 All Simple OI-Paths *vs* All Du-Paths 32
3.3.2 OI-Paths *vs* Required K-Tuples 36
3.3.3 OI-Paths *vs* Ordered Context Coverage 37
Chapter 4 Two Detailed Examples 41
4.1 Illustration of the All Simple OI-Paths Criterion 41
4.2 Illustration of the All k-Iteration OI-Paths Criterion 47
Chapter 5 Conclusions 56
5.1 Summary of Contributions 56
5.2 Future Research Directions 58
References 59

LIST OF FIGURES

Figure		Page
Figure 1	Inclusion Hierarchy of the Commonly-Referenced Control Flow-Based Criteria	15
Figure 2	Inclusion Hierarchy of Rapps and Weyuker's Family of Criteria	17
Figure 3	Inclusion Hierarchy of the Commonly-Referenced Data Flow-Based Criteria	19
Figure 4	Inclusion Hierarchy of the Proposed Criteria	21
Figure 5a	The Worst Case for the OI-Paths Criteria (except *all 0-iteration OI-paths*)	23
Figure 5b	The Worst Case for *all 0-iteration OI-paths*	24
Figure 6	Inclusion Hierarchy of the Commonly-Referenced Criteria	26
Figure 7	Inclusion Hierarchy of the Modified Criteria	27
Figure 8	The OI-Paths Criteria *vs* the Control Flow-Based Criteria	31
Figure 9	Example E1 and Its Corresponding Flowgraph	35
Figure 10	Example E2 and Its Corresponding Flowgraph	37
Figure 11	The Complete Inclusion Hierarchy	40
Figure 12	Example E3 and Its Corresponding Flowgraph	42
Figure 13a	Example E4 - A Program Schmata	48
Figure 13b	The Flowgraph of Example E4	49

LIST OF TABLES

Table		Page
Table 1	The Set of Complete Paths in E1	35
Table 2a	All the DU-Paths in E3	43
Table 2b	Non-Inclusive DU-Paths in E3	44
Table 2c	A Set of Complete Paths Satisfying All DU-Paths	44
Table 3a	All the Simple OI-Paths in E3	45
Table 3b	Non-Inclusive Simple OI-Paths in E3	46
Table 3c	A Set of Complete Paths Satisfying All Simple OI-Paths	46
Table 4a	All the Ordered Data Context in E4	50
Table 4b	A Set of Complete Paths Satisfying Ordered Context Coverage	50
Table 5a	All the 2-dr Interactions in E4	51
Table 5b	Subpaths to be Covered by Required 2-Tuples Criterion	51
Table 6a	All the 3-dr Interactions in E4	52
Table 6b	Subpaths to be Covered by Required 3-Tuples Criterion	52
Table 7a	All the 4-dr Interactions in E4	53
Table 7b	Subpaths to be Covered by Required 4-Tuples Criterion	53
Table 8a	All the 5-dr Interactions in E4	53
Table 8b	Subpaths to be Covered by Required 5-Tuples Criterion	53
Table 9a	All the 3-Iteration OI-Paths in E4	54
Table 9b	A Set of Complete Paths Satisfying All 3-Iteration OI-Paths	55

Chapter 1

Introduction

1.1 Background and Related Work

Dynamic program testing is the most widely used method for affirming the quality of software, because the applicability of formal validation methods (e.g., inductive assertion technique for proving the correctness of programs) is currently confined to the validation of relatively small combinatorial algorithms [3,7]. The general aim of dynamic program testing is to reveal the existence of errors in a program by executing it in a controlled environment over a finite set of test cases. Strategies for selecting such test cases generally fall into two categories: functional testing and structural testing.

In functional testing, programs are viewed as interacting collections of functions [3, 4], and the specification of a program is used in test selection [9]. The strongest functional testing strategy is *exhaustive domain testing* in which a program is tested for all possible input values of its input domain. Surveys of functional testing strategies can be found in [3, 4, 7, 18, 19, 20].

In structural testing the flowgraph of a program is used in test selection [9]. The

strongest structural testing strategy is *exhaustive path testing* in which test data are selected to impose execution on all control paths in the program's flowgraph during testing. Surveys of structural testing strategies can be found in [6, 7, 8, 9, 10, 11, 13, 16, 17, 21, 22].

It is well known that both *exhaustive domain testing* (in terms of functions of a program) and *exhaustive path testing* (in terms of structures of a program) are usually impractical due to the large, possibly infinite, number of input values and paths, respectively. Thus, researchers have been concerned with developing feasible, yet effective, testing strategies.

Given a program flowgraph, a structural testing strategy may be divided into two seemingly distinct phases: test path selection and test case generation (also referred to as test data selection) [9]. In the test path selection phase (or, in short, path selection), a finite set of control paths is selected from the flowgraph to, in general, satisfy a test selection criterion, which is defined in terms of control and/or data flow information contained in the flowgraph. In the test case generation phase, a finite set of test data is chosen from the program input domain in order to impose the execution on selected paths.

Most path selection criteria are based on control flow analysis which examines the branch and loop structure of a program [10]. The commonly-referenced control flow based path selection criteria are statement, branch, and path *coverage* criteria [7]. For example, branch coverage criterion requires the selection of test data that result in traversing those paths which cover all edges in the flowgraph at least once. Although it is necessary to exercise a certain degree of control structures [10, 15], path selection criteria, based merely on control flow information, are not enough to insure the quality of software under test

[10, 13, 14]. The commonly-referenced data flow based structural test selection criteria are Rapps and Weyuker's *all du-paths* criterion [10,12], Ntafos's *required K-tuples* criterion [8], and Laski and Korel's *(ordered) context coverage* criterion [6]. These three criteria (and their derivatives) focus on tracing the flow of data in a flowgraph through the associations between assignments of values to variables (i.e., definitions of variables) and the uses of these variables in either assigning values to other variables or determining the outcome of conditional branching.

From the point of view of testing the embedded functions in a program, structural testing strategies based on data flow are perhaps more suitable than those based on control flow because the former identifies data dependencies and thus implicitly requires testing of functional segments [3].

1.2 Motivation

It is important to note that *all du-paths* criterion considers individual associations between definitions and uses of variables. On the other hand, *required K-tuples* and *(ordered) context coverage* criteria consider the chains of associations between definitions and uses of variables that relate to each other.

The structural units that serve as the basis of these test selection criteria are a du-path [10], a k-dr interaction [8], and an ordered elementary data context [6]. None of these structural units is indicative of the implemented program functionality which can be viewed as a set of functions from the program input to the program output. It was Korel who suggested that the identification of all input variables that influence a specific output

variable may be helpful in providing better understanding of the implemented program functionality and in checking the consistency of the program with its specification [5]. It was our feeling that such identification could be used to form a structural unit which is indicative of the implemented program functionality and thus serves as a basis of a more effective test selection criterion than other data flow based criteria.

1.3 Major Contributions of the Thesis

In this thesis, we propose a class of data flow oriented path selection criteria, called *the OI-paths* criteria, that are based on the analysis of the effects of program inputs on program outputs. These criteria require that each critical association between each input variable and the output variable that is influenced by this input variable is examined during testing. The structural unit which serves as the basis of these criteria is an OI-path. This structural unit is a chain of definition and use pairs which starts with a program input and ends with a program output. An OI-path is a more comprehensive structural unit than a du-path, a k-dr interaction, and an ordered elementary data context in the sense that it relates to the implemented program functionality.

We prove that *all simple OI-paths* is stronger than *all du-paths* criterion and incomparable with both *required K-tuples* and *(ordered) context coverage* criteria. We also show that *all k-iteration OI-paths* is stronger than *all simple OI-paths*. It is then proved that *all k-iteration OI-paths* is stronger than *required K-tuples* and (*ordered*) *context coverage* criteria for some integer *k*.

1.4 Outline of the Thesis

The thesis proceeds as follows. In Chapter 2 we review some of the most often cited path selection criteria and introduce definitions of new terms used throughout this thesis. Then, we formally define our criteria and explore their properties as well as their complexity. Chapter 3 demonstrates the strength of the new criteria by comparing them with existing ones. In Chapter 4, some examples are presented to illustrate the proposed criteria in detail. Chapter 5 concludes the thesis and points out new directions for future research.

Chapter 2

Test Path Selection Criteria

2.1 Terminology

To facilitate the precise description of the path selection criteria given later in the thesis and to formalize the discussion, we now define a set of terms used throughout this thesis.

2.1.1 Control Flow Related Concepts

A **program** is either a main program or a subprogram (i.e. a procedure or a function) and has one **entry statement** and one **exit statement**.

To apply a path selection criterion, a program is represented by a digraph *G(V, E)* called **program flowgraph** (or, in short, flowgraph), where *V* is a set of nodes, each representing a statement or a block of statements, and *E* is a set of edges which represent the control flow between nodes.

Every statement block has the following two properties:

a) Whenever the first statement is executed, the rest are executed subsequently in the given physical order;

b) The first statement is the only statement which may be executed directly after execution of a statement in another block.

In a flowgraph, the entry and exit statements of a program are represented by two separate nodes called **entry node** and **exit node**, respectively.

Note that the terms *program* and (its corresponding) *flowgraph* are used interchangeably in this thesis, as long as such usage does not cause confusion.

A **subpath** is a finite, possibly empty, sequence of nodes $(n_1, n_2, ..., n_m)$, where there is an edge (n_i, n_{i+1}) for $1 \le i \le m-1$ and $m \ge 2$. The number of nodes in a subpath is referred as the **length of the subpath**. We call a subpath with length 2 or more a **path**.

A **complete path** is a path whose first node is the entry node and whose last node is the exit node of a flowgraph G.

A **loop-free path** is a path $(n_1, n_2, ..., n_m)$ in which all nodes are distinct.

A **simple cycle** is a path in which all nodes except the first and the last are distinct.

A **loop** of a program flowgraph *G* is a strongly connected subgraph of *G* corresponding to a looping construct in a program.

The **loop entry** of a loop L is a node n in L such that there is an edge (m, n) in *G*, where m is not in L. A **loop exit** of a loop L is a node m in L such that there is an edge (m, n) in *G*, where n is not in L. In this thesis, we assume that every loop in *G* has single entry and single exit.

A **conditional-entry loop** is a loop whose loop entry and loop exit are the same node.

The **loop body** of a loop L is a subgraph of L which consists of

a) (if L is a conditional-entry loop) all nodes and edges in L except the loop

entry, the incoming edge and outgoing edge of the loop entry;

b) (if L is not a conditional-entry loop) all nodes and edges in L except the loop exit and the outgoing edge of the loop exit.

k iterations of a loop ($k \geq 0$, k is integer) L is a subpath, from the loop entry to the loop exit, which contains exactly k+1 occurrences of the loop entry.

A loop is said to be **traversed** k times ($k \geq 0$, k is integer) in a path if the path includes k iterations of the loop.

2.1.2 Data Flow Related Concepts

A variable occurrence in a program is said to be a **definition** if it is

1) on the LHS of an assignment statement, or

2) in an input statement from which it obtains a value, or

3) an output or input/output parameter in a subprogram call.

A variable occurrence in a program is said to be a **use** if it is not a definition. A use is further classified as a computational-use (c-use) or a predicate-use (p-use). A **c-use** is a use which directly affects the computation being performed (e.g., on the RHS of an assignment statement) or allows one to see the result of some earlier definition (e.g., in the list of variables of an output statement). A **p-use** is a use which directly affects the control flow of the program (e.g., in a predicate portion of a conditional transfer statement) [10].

Every p-use is associated with an edge in the flowgraph.

A path $(n_1, n_2, ..., n_{m-1}, n_m)$ is a **def-clear path** with respect to (wrt) a variable x from node n_1 to node n_m or to edge (n_{m-1}, n_m) if there are no definitions of x from node

n_2 to node n_{m-1} (inclusively).

A path $(n_1, n_2, ..., n_{m-1}, n_m)$ is a **du-path** wrt a variable x if n_1 has a definition of x and either: 1) n_m has a c-use of x and $(n_1, ..., n_m)$ is a def-clear loop-free path wrt x or a def-clear simple cycle wrt x, or 2) (n_{m-1}, n_m) has a p-use of x and $(n_1, ..., n_{m-1})$ is a def-clear loop-free path wrt x .

The use of a variable y is **affected** by the definition of a variable x if

either a) x and y are the same variable and the use of y is reached by the definition of x through a def-clear path wrt x

or b) the definition of a variable z is given in terms of the use of x at a node which is reached by the definition of x through a def-clear path wrt x, and the use of y is affected by the definition of z.

A conditional-entry loop is said to be **marked** if there exists a definition of a variable which succeeds a use of the same variable in the loop body.

A **simple path** is a path in which every loop is traversed zero time, or

a) twice, if it is a marked conditional-entry loop, or

b) once, if it is not a marked conditional-entry loop.

A **k-iteration path** ($k \geq 0$, k is integer) is a path in which every loop (if there exists any) is traversed up to k times and at least one loop in the path is traversed exactly k times.

2.1.3 Concepts Related to Input-Output Relations

An **input** of the program is the definition of a variable which

1) occurs in an input statement, or

2) is defined by an assignment statement whose RHS contains only constants, or
3) is defined by a subprogram call whose parameter list contains the variable as an output parameter (i.e. the variable is not defined before the call, but it will obtain a value from the called subprogram).

An **output** of the program is either

1) a c-use of a variable in an output statement or
2) an output statement containing only constants which must be reached from the entry node through a path that contains at least one p-use.

An **output-free path** is a path in which none of its nodes contains an output.

A definition of a variable x (denoted by d_x) **influences** an output O if

a) O is a c-use of a variable in an output statement and the c-use is affected by d_x, or

b) O is an output statement containing only constants and the last p-use which leads the control flow to O is affected by d_x, or

c) A definition of variable y, d_y, influences O, and the last p-use (if there is any) which leads the control flow to d_y is affected by d_x.

An **output-input path** is a path (n_1, ..., n_m) where n_1 contains an input I and n_m contains an output O which is influenced by I.

An **external p-use** is a p-use which is not in any output-input path.

A **redundant use** is a use which has no effect on any program output and is not an external p-use.

An **extended output-input path** is an output-input path followed by a path which

terminates with an edge containing an external p-use.

For short, we call an (simple/k-iteration) output-input path or, an extended (simple/k-iteration) output-input path an (simple/k-iteration) **OI-path.**

2.1.4 Coverage Related Concepts

Let P be a set of complete paths of a program's flowgraph. We say that **a node i is covered** by P if P contains a path $(n_1, \ldots, n_m)$ such that $i = n_j$ for some j, $1 \leq j \leq m$. Similarly, **an edge (i_1, i_2) is covered** by P if P contains a path $(n_1, \ldots, n_m)$ such that $i_1 = n_j$ and $i_2 = n_{j+1}$, for some j, $1 \leq j \leq m-1$. **A path $(i_1, \ldots i_k)$ is covered** by P if P contains a path $(n_1, \ldots, n_m)$ such that $i_1 = n_j$ and $i_2 = n_{j+1}, \ldots, i_k = n_{j+k-1}$ for some j, $1 \leq j \leq m-k+1$.

Path selection criterion A **includes** path selection criterion B iff for any given flowgraph, any set of complete paths that satisfies A also satisfies B. "*A includes B*" is represented by $A => B$. Path selection criterion A is **equivalent** to path selection criterion B iff $A => B$ and $B => A$. "*A* is equivalent to *B*" is represented by $A = B$. Path selection criterion A **strictly includes** path selection criterion B if for any given flowgraph, A **includes** B but B does not **include** "*A strictly includes B*" is represented by $A \longrightarrow B$. Two path selection criteria A and B are **incomparable** if neither $A => B$ nor $B => A$. "*A and B are incomparable*" is represented by $A \neq B$.

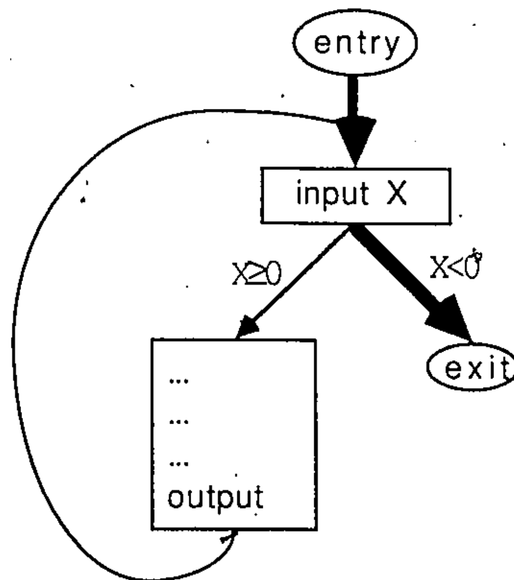
2.2 Assumptions

In order to apply a data flow-based test selection criterion to a given program, the program should meet some constraints on program structure. This may include data structure constraints and/or control structure constraints. In general, these constraints are so weak that the applicability of the criterion is not affected even if the program fails to satisfy them. Furthermore, there exist techniques (e.g., data flow analysis) to check the violations of the constraints. These constraints guarantee that the applied criterion is valid and meaningful for the program. For instance, if a definition of a variable can not reach a use of that variable through any executable path, then the du-path is meaningless for the definition. In this thesis, we call a program to be **well-formed** if it meets the following assumptions (constraints):

1. All statements are reachable from the program entry;
2. Every definition of a variable reaches some use of that variable through a def-clear path;
3. Every use of a variable is reached by some definition of that variable through a def-clear path;
4. There is no redundant use;
4. Every definition that reaches an output through a path must influence that output;
5. If a complete path is an output-free path, and in addition, if the absence of output in the path is not treated as an anomaly, then the first node in the path is considered to contain an output-input path.

In this thesis, only well-formed programs are considered.

Assumption 5 deals with the case that there is no output in a complete path. Such paths are usually those that are similar to the path (marked by boldface lines) in the following program. As we see, assumption 5 guarantees that this path is tested.



2.3 Commonly-Referenced Control Flow-Based Criteria

First, we define the three most commonly-referenced control flow criteria: *all nodes* (statement coverage), *all edges* (branch coverage), and *all paths* (path coverage). These three criteria are solely based on the control structure of a program. The objective of these criteria is to provide tests that will cover certain control flow-based structural components of a given source code.

Let G be a program's flowgraph, P be a set of complete paths of G, and PATHS(G) be the set of all possible complete paths of G. Then,

CF1. P satisfies the ***all-nodes*** criterion if P covers every node of G.

CF2. P satisfies the ***all-edges*** criterion if P covers every edge of G.

CF3. P satisfies the ***all-paths*** criterion if $P =$ PATHS(G).

It is well known that *all-nodes* is a rather weak criterion, representing necessary but by no means sufficient conditions for obtaining a reasonable test. *All-edges* criterion, however, implies *all-nodes* and has come to be regarded as a minimal standard of achievement in structural testing (i.e., minimally thorough [7]). *All-paths* is the strongest structural testing criterion. These three criteria are partially ordered by strict inclusion as shown in Figure 1. Since for most programs (which contain looping constructs) PATHS is infinite, thus, *all-paths* is often not practical. On the other hand, important combinations of nodes and/or edges may be overlooked by *all-nodes* and/or *all-edges*. The data flow based criteria described in the rest of this chapter distinguish those combinations that are important in terms of the flow of data through a program.

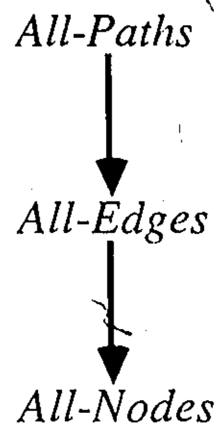


Figure 1: Inclusion Hierarchy of the Commonly-Referenced Control Flow-Based Criteria

2.4 Commonly-Referenced Data Flow-Based Criteria

In this section, we define the family of path selection criteria proposed by Rapps and Weyuker, *required k-tuples* proposed by Ntafos, and *context coverage* and *ordered context coverage* criteria proposed by Laski and Korel.

2.4.1 Rapps and Weyuker's Family of Criteria

Rapps and Weyuker define a family of path selection criteria based on data flow analysis and analyze these criteria in an attempt to specify the inclusion relationships that exist among the members of the family [10, 12]. We now present these criteria using our terminology.

DF1. *P* satisfies the ***all-p-uses*** criterion if for every node i of G and every definition of a variable x at node i, *P* covers a def-clear path wrt x from i to each edge containing a p-use of x.

DF2. *P* satisfies the ***all-defs*** criterion if for every node i of G and every definition of a variable x at node i, *P* covers a def-clear path wrt x from i to some node containing a c-use of x or to some edge containing a p-use of x.

DF3. *P* satisfies the ***all-p-uses/some-c-uses*** criterion if for every node i of G and every definition of a variable x at node i, *P* covers a def-clear path wrt x from i to each edge containing a p-use of x; if such a node does not exist, then *P* must cover a def-clear path wrt x from i to some node containing a c-use of x.

DF4. *P* satisfies the ***all-c-uses/some-p-uses*** criterion if for every node i of G and every definition of a variable x at node i, *P* covers a def-clear path wrt x from i to each node containing a c-use of x; if such a node does not exist, then *P* must cover a def-clear path wrt x from i to some edge containing a p-use of x.

DF5. *P* satisfies the ***all-uses*** criterion if for every node i of G and every definition of a variable x at node i, *P* covers a def-clear path wrt x from i to each node containing a c-use of x and to each edge containing a p-use of x.

DF6. *P* satisfies the ***all-du-paths*** criterion if for every node i of G and every definition of a variable x at node i, *P* covers every du-path wrt x.

Rapps and Weyuker [10] have proved that the above criteria are partially ordered by strict inclusion as shown in Figure 2.

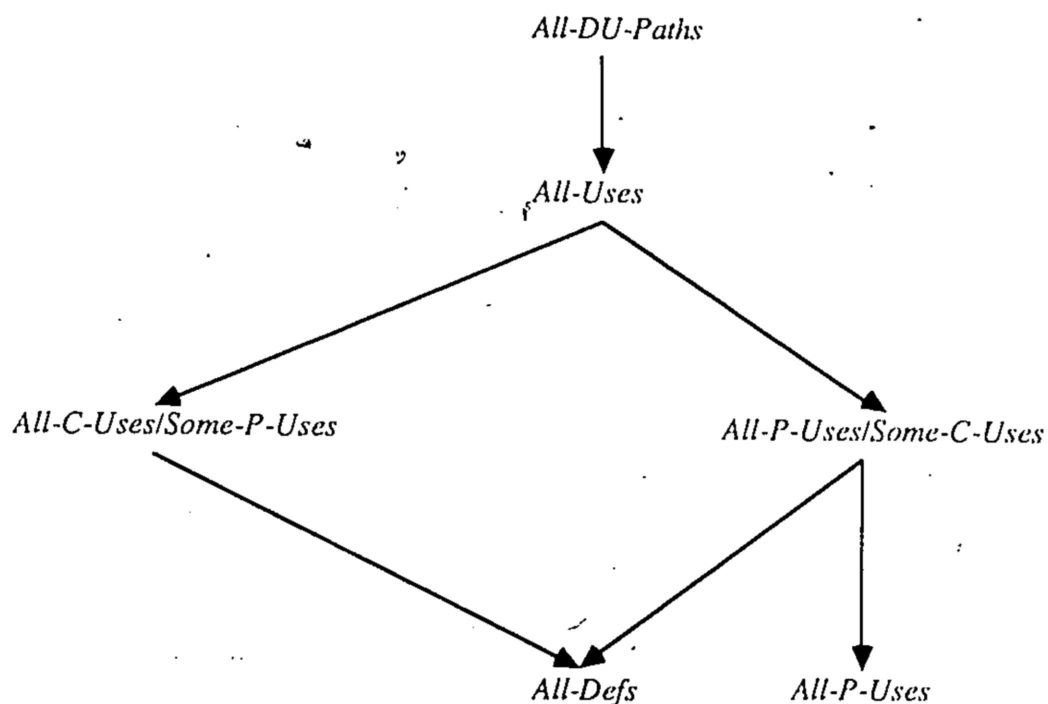


Figure 2: Inclusion Hierarchy of Rapps and Weyuker's Family of Criteria

2.4.2 Ntafos's Required k-Tuples Criteria

Required k-tuples is a class of program path selection criteria. These criteria require that a set of complete paths covers chains of alternating definitions and uses, called k-dr interactions. A k-dr ($k \geq 2$) interaction is a sequence of k-1 definitions and k-1 uses $[d_1(x_1), u_2(x_1), d_2(x_2), u_3(x_2), ..., d_{k-1}(x_{k-1}), u_k(x_{k-1})]$ which are associated with k distinct nodes $n_1, n_2, ..., n_k$, where, for all i, $1 \leq i \leq k$, the i-th definition $d_i(x_i)$ at node n_i reaches the i-th use $u_{i+1}(x_i)$ at node n_{i+1} through a def-clear path. Note that, the variables $x_1, x_2, ..., x_{k-1}$ need not be distinct. Note also that, as a restriction required by these criteria, no nodes in a flowgraph contain data flow interactions of the form *dr* or compound statements (e.g., conditional transfer statements with compound predicates). *Required k-tuples* requires that a) each k-dr interaction not involving branches or loops be covered at least once, b) each k-dr interaction with its last use involving in a conditional transfer be covered at least twice - each represents a different outcome of the conditional transfer, c) each k-dr interaction with its first definition or last use involving in a loop to be covered at least twice - each represents a different iteration count corresponding the boundary condition of the loop [8].

2.4.3 Laski and Korel's Criteria

(Ordered) context coverage deals with liveness of vectors of variables, called (ordered) elementary data context. It requires that each (ordered) elementary data context of every node be covered at least once [6].

A definition of a variable x at a node i is **live** at a node j if there exists a def-clear path with respect to x from i to j. An **(ordered) elementary data context** of a node i that uses a vector of variables $X(i) = [x_1, x_2, ..., x_n]$ is a (ordered) set of definitions ec(i) $= [def(x_1), def(x_2), ..., def(x_n)]$ of all variables from X(i) such that there exists a control path from the entry node to node i and all definitions from ec(i) are live at i when the path reaches i.

2.4.4 A Comparison

It is shown [2, 10] that these data flow-based path selection criteria are partially ordered by strict inclusion as shown in Figure 3.

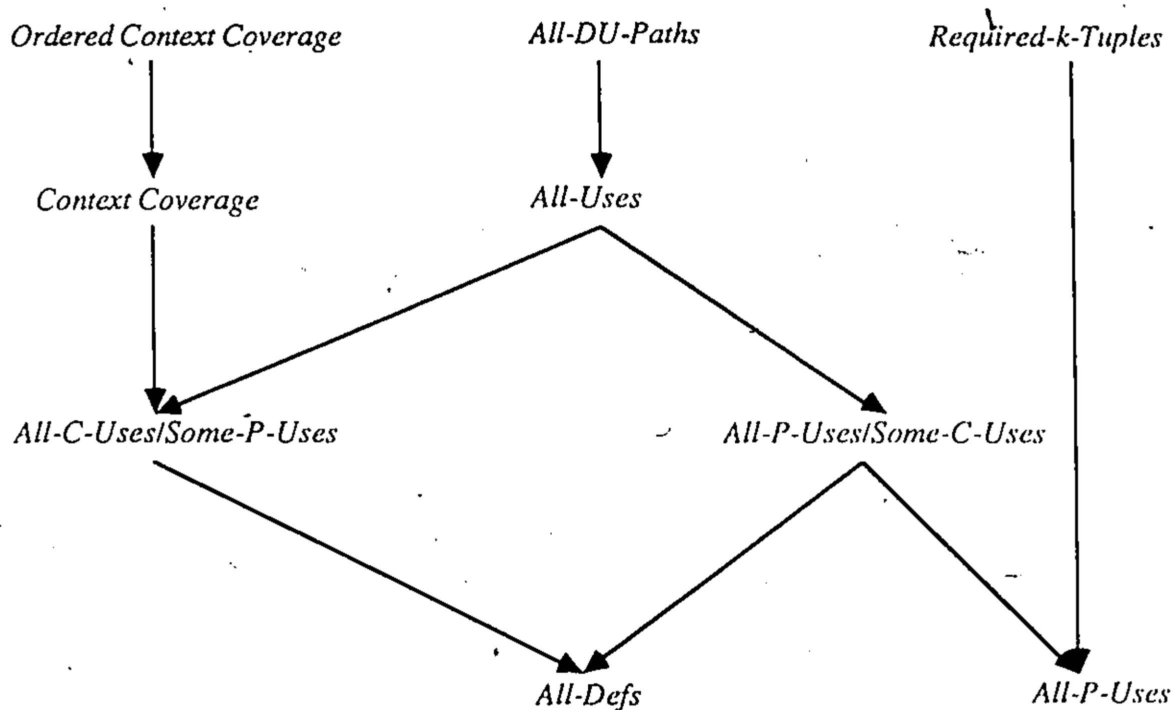


Figure 3: Inclusion Hierarchy of the Commonly-Referenced Data Flow-Based Criteria

2.5 The Proposed Criteria

2.5.1 Definitions of the OI-Paths Criteria

We now introduce a class of new path selection criteria called *the OI-paths* criteria. These criteria are based on input-output relation analysis. The basic unit to be tested is an OI-path.

Let G be a program's flowgraph, and P be a set of complete paths of G. Then,

OI1. P satisfies the ***all simple OI-paths*** if P covers every simple OI-path in G.

OI2. P satisfies the ***all k-iteration OI-paths*** if P covers every k-iteration OI-path in G, and if $(k\text{-}1)\geq 0$, P satisfies *all (k-1)-iteration OI-paths*, where k is an integer and $k\geq 0$.

2.5.2 Properties of the OI-Paths Criteria

Lemma 1: *All m-iteration OI-paths* criterion strictly includes *all n-iteration OI-paths* criterion, where *m, n are integers* and $m>n\geq 0$.

Proof: The proof is immediate from the definitions of the these criteria.

Lemma 2: *All 2-iteration OI-paths* criterion strictly includes *all simple OI-paths* criterion, and *all simple OI-paths* criterion strictly includes *all 0-iteration OI-paths* criterion

Proof: The proof is immediate from the definitions of the these criteria.

Lemma 3: *All simple OI-paths* criterion is incompariable with *all 1-iteration OI-paths*

criterion,

Proof: The proof is straightforward, since a complete path, in a program, which covers a simple OI-path containing 2-iterations of a conditional-entry loop, does not necessarily cover a 1-iteration path containing 1-iteration of the same loop.

The above results are graphically shown in Figure 4.

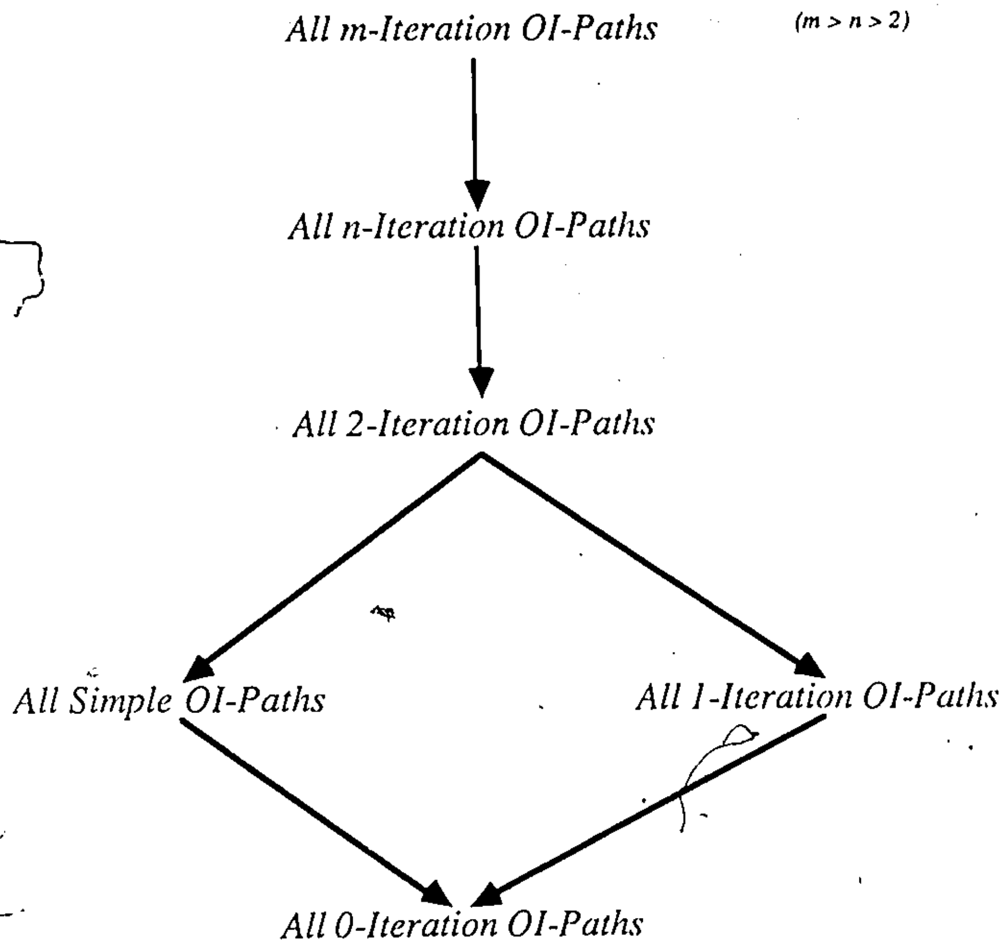


Figure 4: Inclusion Hierarchy of the Proposed Criteria.

2.5.3 Complexity of the OI-Paths Criteria

We now determine the upper bound on the amount of complete paths needed to satisfy each OI-paths criterion. We call such an upper bound the complexity of the criterion.

Theorem 1: The complexity of *all 0-iteration OI-paths* is $O(2^t)$, and the complexity of *all simple OI-paths* is $O(2^{2t})$, and the complexity of *all k-iteration OI-paths* is $O(2^{kt})$, where t is the number of conditional transfers in a program and k is an integer, $k \geq 1$.

Proof: In order to compare the complexity of our criteria with that of *all du-paths,* we assume that no nested loop exists in the program. The worst case for these criteria is a flowgraph which maximizes the number of OI-paths. This occurs when a flowgraph has the form outlined in Figure 5a (or Figure 5b, for *all 0-iteration OI-paths*), where every path in the flowgraph is an OI-path.

In Figure 5a, there is only one loop in the flowgraph and all other conditional transfers are contained in the loop. Thus, the amount of complete paths needed for *all simple OI-paths* criterion is $1+(2^{t-1})*(2^{t-1}) = O(2^{2t})$. Similarly, the amount of test cases needed for *all k-iteration OI-paths* criterion is $1+(2^{t-1})^2+ \ldots + (2^{t-1})^k = [(2^{t-1})^{k+1} -1]/[(2^{t-1})-1] = O(2^{kt})$.

In Figure 5b, there is only one loop in the flowgraph. The loop body does not contain any conditional transfer. Thus, the number of 0-iteration OI-paths is $2^{t-1}=O(2^t)$. We complete the proof.

We would like to point out that the mere fact that these criteria have such bounds,

does not necessarily rule them out for practical consideration. The cost of a criterion should be considered for an individual program, rather than dismissing it based on the upper bound [12].

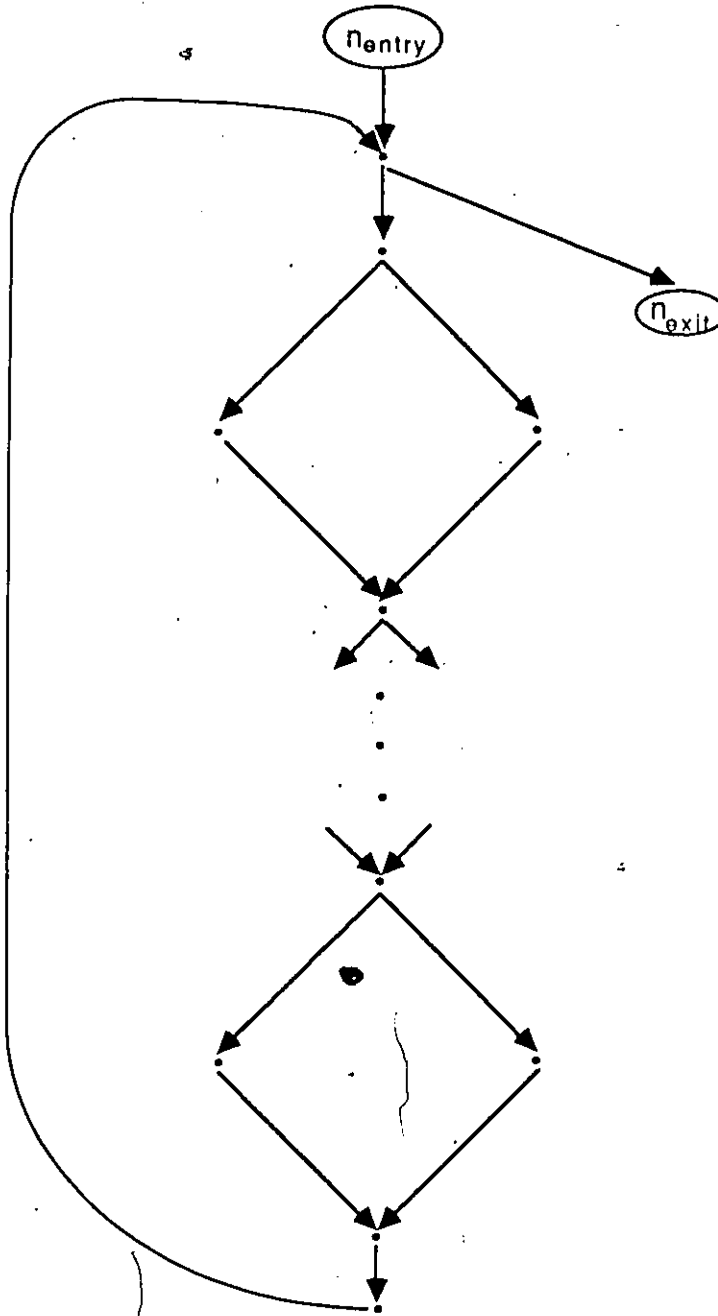


Figure 5a: The Worst Case for the OI-Paths Criteria (except *all 0-iteration OI-paths*)

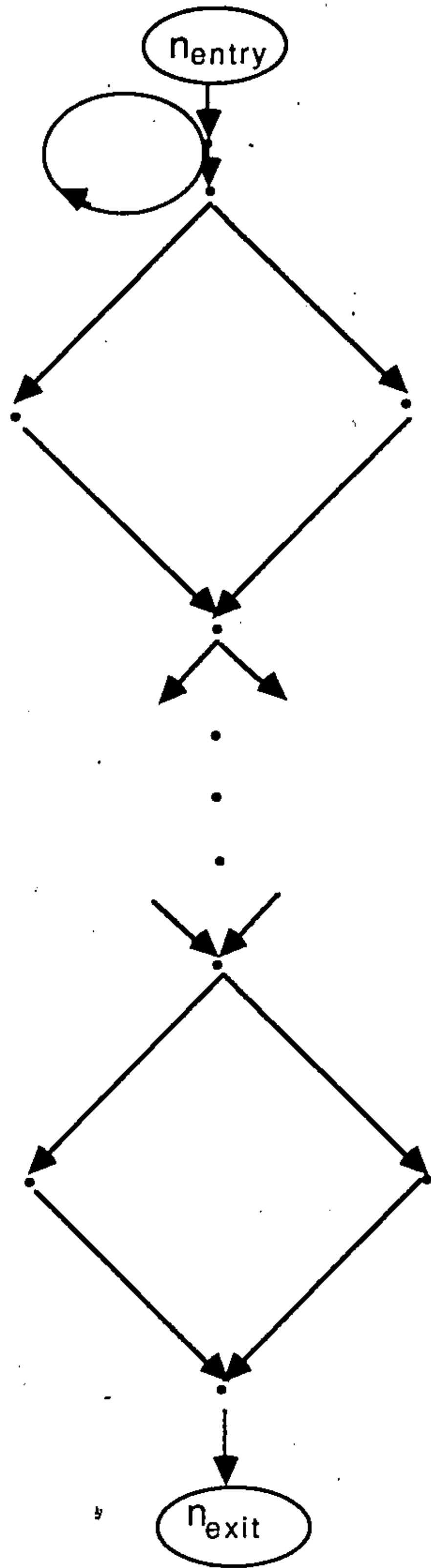


Figure 5b: The Worst Case for *all 0-iteration OI-paths*

Chapter 3

A Comparison of the Path Selection Criteria

In this chapter, we prove that *all simple OI-paths* criterion strictly includes *all du-paths* and is incomparable with both *required k-tuples* and (*ordered*) *context coverage* criteria. We also prove that *all k-iteration OI-paths* strictly includes *required k-tuples* and (*ordered*) *context coverage* criteria for some integer *k*.

3.1 Relative Strength of the Existing Criteria

To determine what would be an effective path selection criterion for revealing errors in programs, Clarke and others [2] have undertaken an evaluation of the criteria presented in section 2.3 and section 2.4. The evaluation is based on the inclusion relationships between two criteria. The results of the evaluation are shown in Figure 6. They found that the strongest data flow-based criteria in the three families mentioned in section 2.4 are incomparable and that, in two of the families, the strongest criteria fail to satisfy certain minimal coverage requirements. They then introduce minor changes to the original criteria to assure that they each satisfy these minimal coverage requirements and show how the

modified criteria related to each other (see Figure 7).

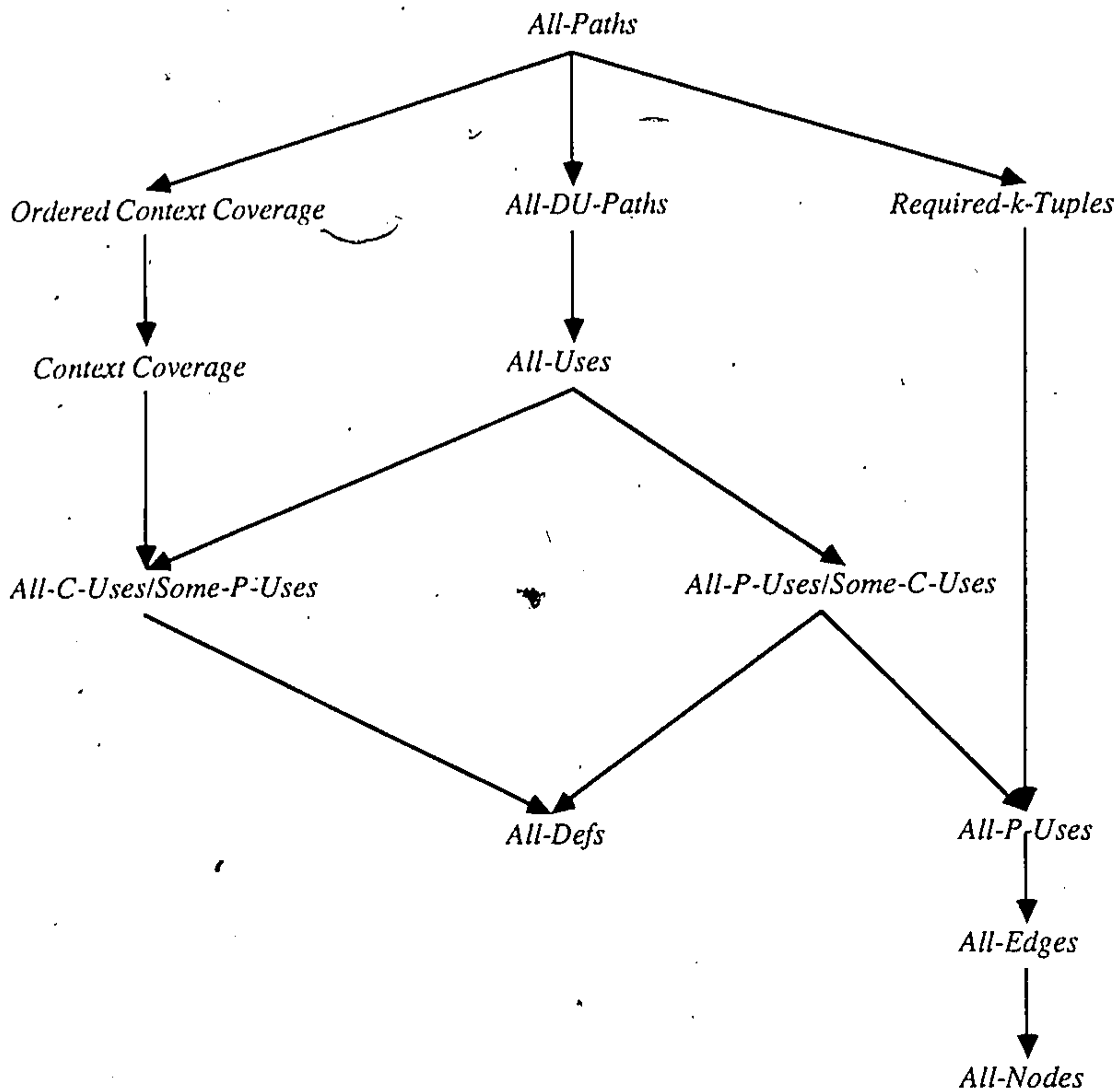


Figure 6: Inclusion Hierarchy of the Commonly-Referenced Criteria

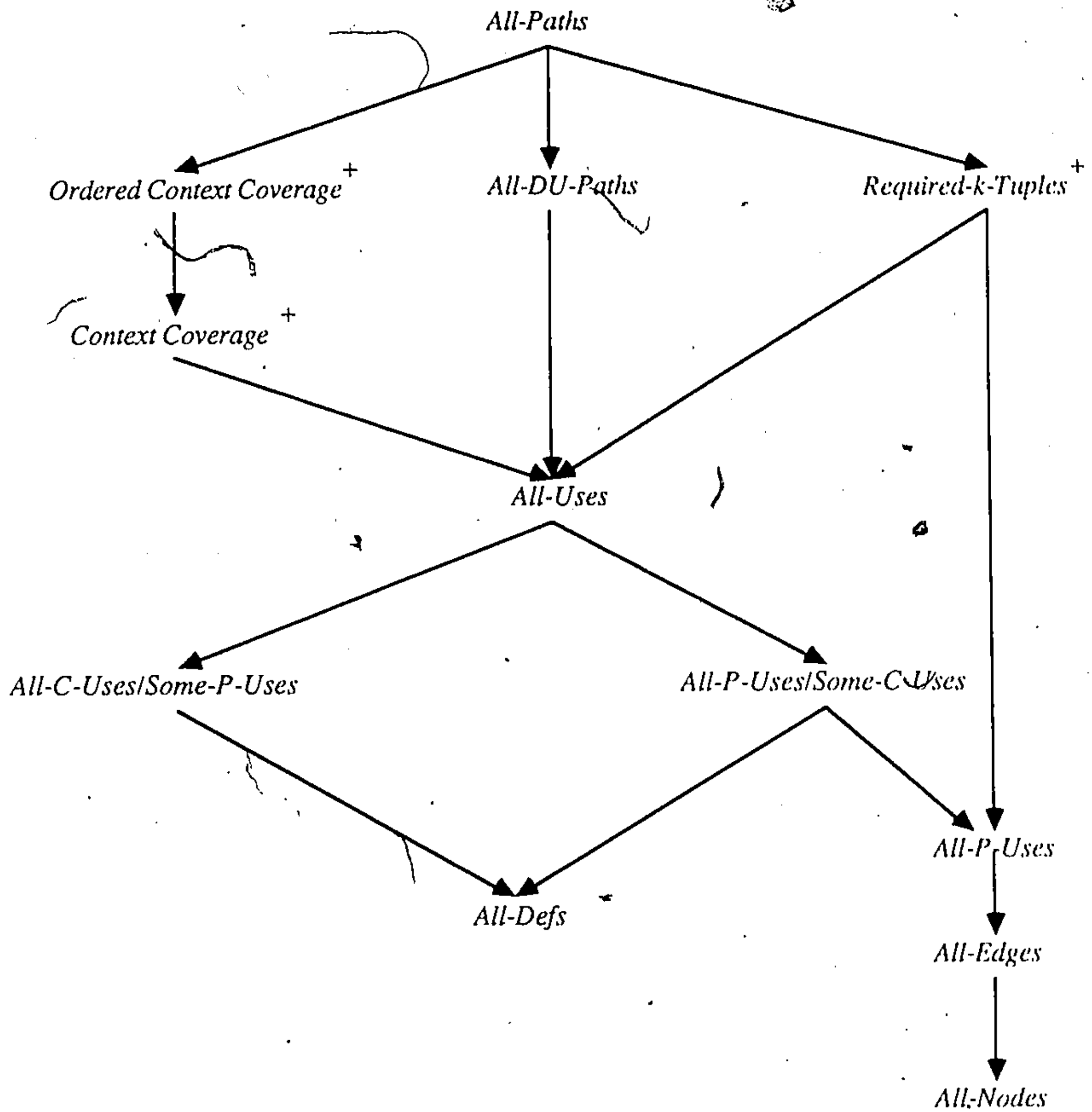


Figure 7: Inclusion Hierarchy of the Modified Criteria.

3.2 OI-Paths *vs* Control Flow-Based Criteria

Lemma 4: *All-paths* criterion strictly includes *all k-iteration OI-paths* criterion, where k is an integer and $k \geq 0$.

Proof: The proof is immediate from the definitions of the two criteria.

Lemma 5: Every use in a well-formed program is affected by some input(s) of the program.

Proof: Since the program is well-formed, any use of a variable x (denoted by u_x) is reached through a def-clear path wrt x by some definition of x (denoted by d_x) which precedes u_x.

1. if d_x is an input, u_x is affected by d_x;
2. otherwise, there must be a use of another variable, say y, which defines d_x. The above arguments are now applied wrt y.

These arguments succeed at step 1 in finite iterations for well-formed programs. Therefore, we complete the proof.

Theorem 2: Every edge of a flowgraph G is covered by some simple OI-path as well as by some 1-iteration OI-path.

Proof: This proof consists of two parts:

1. we prove that every p-use p in G is covered by some OI-path;
2. we prove that if a p-use is covered by an OI-path, this p-use is also covered

by some 1-iteration OI-path as well as by some simple OI-path.

Part 1: We assume the opposite to be true, i.e. there exists a p-use p which can not be covered by any OI-path of G, then p must be an external p-use which can not be reached by any output-input path.

There are two cases:

a) Any complete path that covers p is an output-free path. However, according to our assumption 5 given in section 2.2, any such path is an extended output-input path. Contradiction!

b) There exists a complete path, containing an output, which covers p, but it does not cover an output-input path. However, this is impossible, because that output must be influenced by some input along the path in a well-formed program.

Part 2: From the viewpoint of syntax, the function of a loop is to use some variables defined outside the loop to redefine some variables previously defined outside the loop through du-paths residing in the loop body. Note that in a well-formed program, no matter how many times the loop is traversed, these variables are still defined. That is, if an OI-path contains traversals of some loops, then no matter how many times these loops are traversed, the path remains to be an OI-path. Hence, if a p-use is covered by an OI-path from node n_1 to node n_m, this p-use is also covered by some 1-iteration OI-path as well as by some simple OI-path from node n_1 to node n_m.

Corollary 1: Every use in a flowgraph G is covered by some simple OI-path.

Proof: Since any use is either a p-use (associated with some edge) or a c-use (associated with some node), and from Theorem 2 all the edges and therefore all the nodes in G are

covered by some simple OI-paths, we complete the proof.

Corollary 2: *All simple OI-paths* criterion strictly includes *all-edges* criterion, and *all 1-iteration OI-paths* criterion strictly includes *all-edges* criterion.

Proof: Immediate from Theorem 2.

Theorem 3: *All 0-iteration OI-paths* criterion is incomparable with both *all-nodes* criterion and *all-edges* criterion.

Proof: *All 0-iteration OI-paths* criterion does not include both *all-nodes* criterion and *all-edges* criterion, because the former does not require edges and nodes in a loop to be tested. *All-nodes and all-edges* does not include *all 0-iteration OI-paths,* because they do not require all the of loop-free OI-paths (i.e., 0-iteration OI-paths) to be tested.

These results are graphically shown in Figure 8.

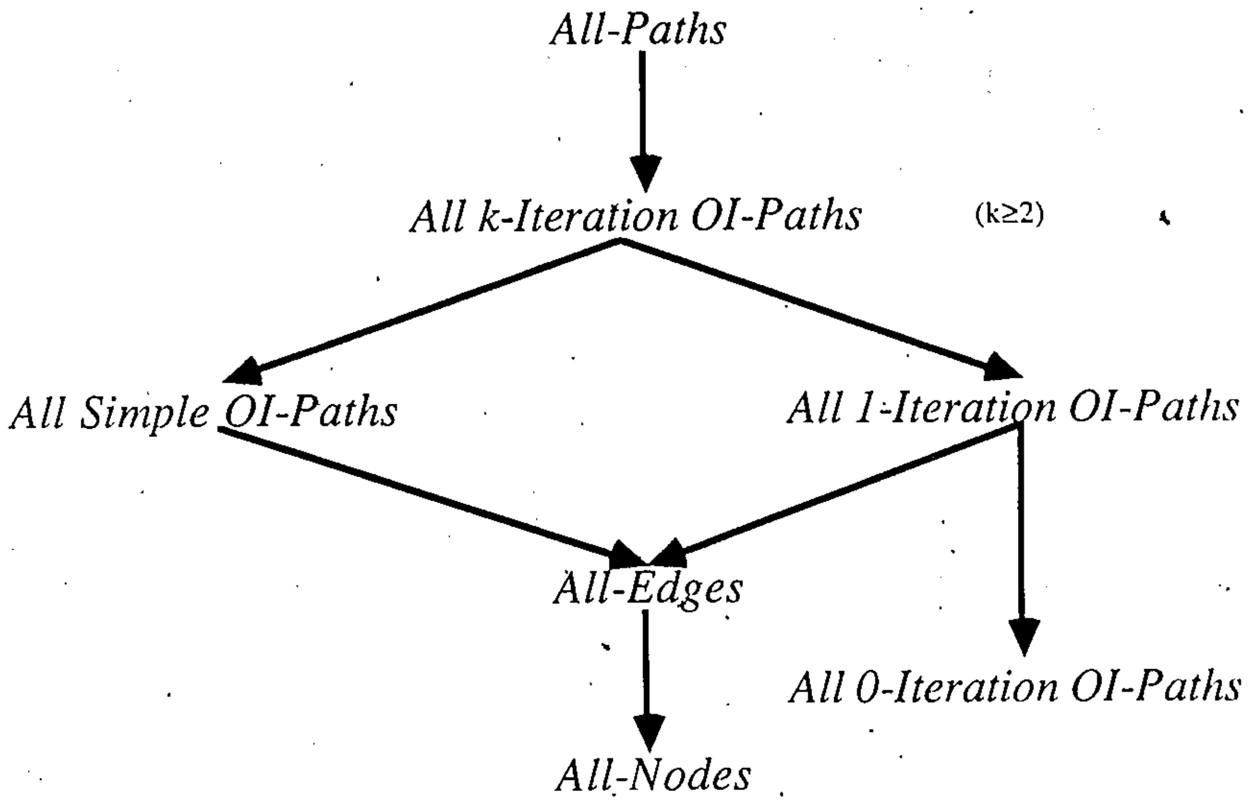


Figure 8: The OI-Paths Criteria *vs* the Control Flow-Based Criteria

3.3 OI-Paths *vs* Data Flow-Based Criteria

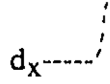
3.3.1 All Simple OI-Paths *vs* All Du-Paths

Lemma 6: If the use of a variable x (denoted by u_X) of a du-path wrt x is covered by a simple output-input path, say I----------->O_I, then the du-path is covered by some simple output-input path.

Proof: From the given condition I----->u_X ----->O_I, we can separate the proof into three parts according to the location of d_X.

1: In case that I--------->d_X--------->u_X--------->O_I, the proof for this case is immediate: the du-path is covered by this simple output-input path or (if u_X is in a loop which is traversed less than twice by the path) by the simple output-input path obtained from expansion of the original simple output-input path such that that loop is traversed twice.

2: In case that I----------->u_X--------->O_I, we apply the following arguments:



a) If d_X itself is an input, d_X---------->u_X------->O_I must be an output-input path since there is no redundant use in a well-formed program (thus, d_X must influence O_I through u_X). The path can be reduced to a simple path by removing some iterations of each loop (if there is any) within path u_X---------->O_I. The new path from d_X to O_I is a simple output-input path and it covers the du-path

d_x--------->u_x.

b) If d_x is not an input, then it must be defined in terms of some other variable, say y. From Lemma 5, there must be some input I_y which affects the use of y (denoted by u_y). So we have I_y--------->(u_y,d_x)------->u_x--------->O_I. Since there is no redundant use in a well-formed program, I_y must influence O_I through u_x, thus this path is an output-input path. The path can be reduced to a simple path by removing some iterations of each loop (if there is any). The new path obtained is a simple output-input path and it covers the du-path d_x--------->u_x.

3: In case that d_x------->I--------->u_x------------->O_I, the proof is similar to 2. Q.E.D.

Lemma 7: If the use of a variable x of a du-path wrt x is covered only by a simple extended output-input path, then the du-path must be covered by some simple extended output-input path.

Proof: From the given condition I--------->O_I------>u_x----------->*P*, where *P* is an external p-use that leads the control flow to the exit node, we can separate the proof into three parts according to the location of d_x which reaches u_x through the du-path (denoted by d_x----->u_x).

1: In case that I----->O_I----->d_x------>u_x----->*P* or I----->d_x----->O_I----->u_x----->*P*, the proof for this case is immediate: the du-path is covered by the this simple extended output-input path or (if u_x is in a loop which is traversed less than twice by the path) by the

simple extended output-input path obtained from expansion of the original simple extended output-input path such that that loop is traversed twice.

2: In case that d_x------>I------>O_I-------->u_x------->P, since the program is well-formed, d_x must influence O_I through this path. We apply the following arguments:

a) If d_x itself is an input, d_x--------->I------>O_I is an output-input path. Therefore, the du-path is covered by the simple extended output-input path. d_x------>I------>$\overline{O_I}$-------->u_x----->P.

b) If d_x is not an input, then it must be defined in terms of some other variable, say y. From Lemma 5, there must be some input I_y which affects the use of y (denoted by u_y). So we have I_y--------->(u_y,d_x)----->I-------->O_I-------->u_x----->P. Since the program is well-formed, I_y must influence O_I, thus this path is an extended output-input path. The path can be reduced to a simple path by removing some iterations of each loop (if there is any). The new path obtained is a simple extended output-input path and it covers the du-path d_x--------->u_x.

3: Case: d_x is not covered by the path: I---->O_I---->u_x. If d_x itself is an input, then either d_x---->u_x---->P is an extended output-input path or entry---->d_x---->u_x---->P---->exit is an output-free path (thus, entry----->d_x----->u_x----->P is an extended output-input path by assumption 5 given in section 2.2). Otherwise, the arguments for 2b can be similarly applied. Therefore, the du-path is covered by some extended output-input path which can be reduced to a simple extended output-input path covering the du-path.

Theorem 4: *All simple OI-paths* criterion strictly includes *all du-paths* criterion.

Proof: All simple OI-paths criterion => *all du-paths* criterion can be immediately proved by using Corollary 1, Lemma 6, Lemma 7. The strictness of the inclusion can be proved by considering the program and its flowgraph given in Figure 9.

```
nentry   begin
n1       input (x);
n2       if even(x) then
n3            L := x / 2
         else
n4            L := (x-1) / 2;
n5       z := f (L);
n6       if not integer (z) then
n7            z := round (z);
n8       output ('z =', z);
nexit    end.
```

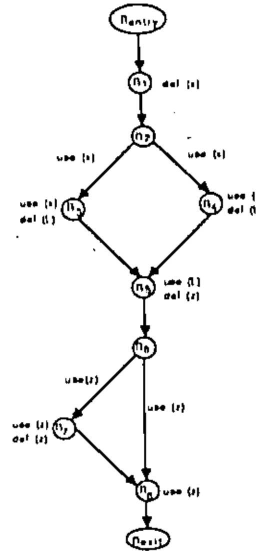


Figure 9: Example E1 and Its Corresponding Flowgraph

There are four complete paths in the example (see Table 1). All these paths are simple paths.

Table 1: The Set of Complete Paths in Example E1

p1 = (nentry, n1, n2, n3, n5, n6, n7, n8, nexit),
p2 = (nentry, n1, n2, n3, n5, n6, n8, nexit),
p3 = (nentry, n1, n2, n4, n5, n6, n7, n8, nexit),
p4 = (nentry, n1, n2, n4, n5, n6, n8, nexit).

The path set P = {p1, p4} satisfies the *all du-paths* but P does not satisfy the *all simple OI-paths* which requires all four of p1, p2, p3, p4 being tested.

3.3.2 OI-Paths *vs* Required K-Tuples

Theorem 5: *All simple OI-paths* criterion is incomparable with *required k-tuples* criterion.

Proof: First, we note that *required k-tuples* does not include *all simple OI-paths*, because the former does not consider the case when the definition of a variable and its use occur at the same node [1], whereas the latter requires the case be covered. Furthermore, *all simple OI-paths* requires that not only the direct effect (see definition of affect in section 2.1.2) but also the indirect effect (see definition of influence in section 2.1.3) to be tested.

It is obvious that *all simple OI-paths* does not include *required k-tuples*, since the latter requires loops in a program be tested for two different iteration counts whereas the former does not. However, we believe that *all simple OI-paths* is inherently "stronger" than *required k-tuples*, because it is straightforward to see that every k-dr interaction [8] is covered by some simple OI-path(s). In fact, by including an additional condition to the former, it become "stronger" than the latter. The added condition is fairly reasonable since it asks that, while generating a set of complete paths, each loop is entered at least twice (corresponding to the boundary condition of the loop): one minimum, one some larger iteration. Of course, the resulting OI-paths are not simple paths any more.

Theorem 6: *All t-iteration OI-paths* criterion strictly includes *required k-tuples* criterion, where t = max(iteration count corresponding to the boundary conditions of any loop).

Proof: It is straightforward to see that every k-dr interaction is covered by some simple OI-path. If the first definition or the last use in a k-dr interaction occurs in a loop, then two distinct iteration counts (which correspond to the boundary conditions of the loop) are

produced for the loop in *required k-tuples* criteria. The same conditions can be also incorporated into the simple OI-path which covers the k-dr interaction. The simple OI-path before incorporation is, of course, a simple OI-path which traverses the loop for the minimum (i.e., one) iteration count, and the simple OI-path after incorporation is an s-iteration OI-path which traverses the loop for s iteration count, where s corresponds to the other (larger iteration count) boundary condition and $1<s\leq t$.

The strictness of the inclusion can be seen from Lemma 1 and Theorem 5. Q.E.D.

3.3.3 OI-Paths *vs* Ordered Context Coverage

Theorem 7: *All simple OI-paths* criterion is incomparable with both *context coverage* criterion (Strategy II) and *ordered context coverage* criterion (modified Strategy II).

Proof : First we show that *ordered context coverage* (and therefore *context coverage*) does not include *all simple OI-paths* by the following example:

n_{entry}	begin
n_1	input (x);
n_2	if odd(x) then
n_3	output (1)
	else
n_4	output (0);
n_{exit}	end.

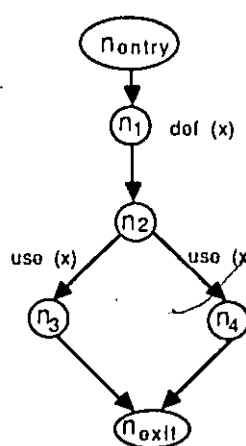


Figure 10: Example E2 and Its Corresponding Flowgraph

The only ordered elementary data context associated with the flowgraph (shown in Figure 10) is ODC(n_2)=[def(x) at n_1]. Thus path P_1= (n_{entry}, n_1, n_2, n_3, n_{exit}) satisfies *ordered context coverage* but does not satisfy *all simple OI-paths* which requires both P_1 and P_2= (n_{entry}, n_1, n_2, n_4, n_{exit}) to be tested.

It is obvious that *all simple OI-paths* does not include *context coverage* (and therefore *ordered context coverage*), because the former only considers simple paths.

Theorem 8: *All (m+1)-iteration OI-path* criterion strictly includes both *context coverage* and *ordered context coverage* criteria, where *m* = max(number of uses at any node).

Proof: First of all, we shall show that each ordered elementary data context (and therefore each elementary data context) is covered by some k-iteration OI-path, where k≤max(number of iterations of any loop).

Let i be a node that uses a set of variables X(i) = [x_1, x_2, ..., x_n] and ec(i) = [d_{i1}, d_{i2}, ..., d_{in}] be an ordered elementary data context of i, where subscripts (i1, i2, ..., in) is a 1-1 mapping of subscripts (1, 2, ..., n) and d_{ir} denotes the definition of x_{ir} (1≤r≤n) and these definitions are reached in the same order as they are given in ec(i) by a control path from the node containing d_{i1} to node i. By the definition of an ordered elementary data context, such a control path exists. i.e., we have the path d_{i1}---->d_{i2}---->......---->d_{in}---->i. It is obvious that this path is covered by some OI-path. Of course, this OI-path must be a k-iteration OI-path, where k is an integer.

Next, we prove that k can be determined (i.e., k≤m+1).

There are two cases for the proof:

1. If the path d_{i1}---->d_{i2}---->......---->d_{in}---->i is loop-free (i.e., it is a du-path wrt x_{i1}), then from Lemma 2, we know that it is covered by some simple OI-path. i.e. $k=1\leq m+1$.

2. If the path d_{i1}---->d_{i2}---->......---->d_{in}---->i contains some loops, and

   a) there is only one occurrence of node i in the path, then at most a-iterations are needed to reach each d_{ir} at least once (so, it becomes defined), where a=max(number of elements in a subset of ec(i) contained in a loop) and $1\leq r\leq n$. Note that $a\leq n\leq m$, thus $k=a\leq m$.

   or

   b) the first occurrence of i precedes the node containing $d_{i(s+1)}$, where $1\leq s<n$, then at most k=max(b, c+1) iterations needed to reach each definition in ec(i) once and then to reach the last occurrence of node i, where b=max(number of elements in a subset of $[d_{i1}, d_{i2}, ..., d_{is}]$ contained in a loop) and c=max(number of elements in a subset of $[d_{i(s+1)}, d_{i(s+2)}, ..., d_{in}]$ contained in a loop).

The strictness of the inclusion can be seen from Lemma 1 and Theorem 7.

The complete results obtained in this section are graphically shown in Figure 11.

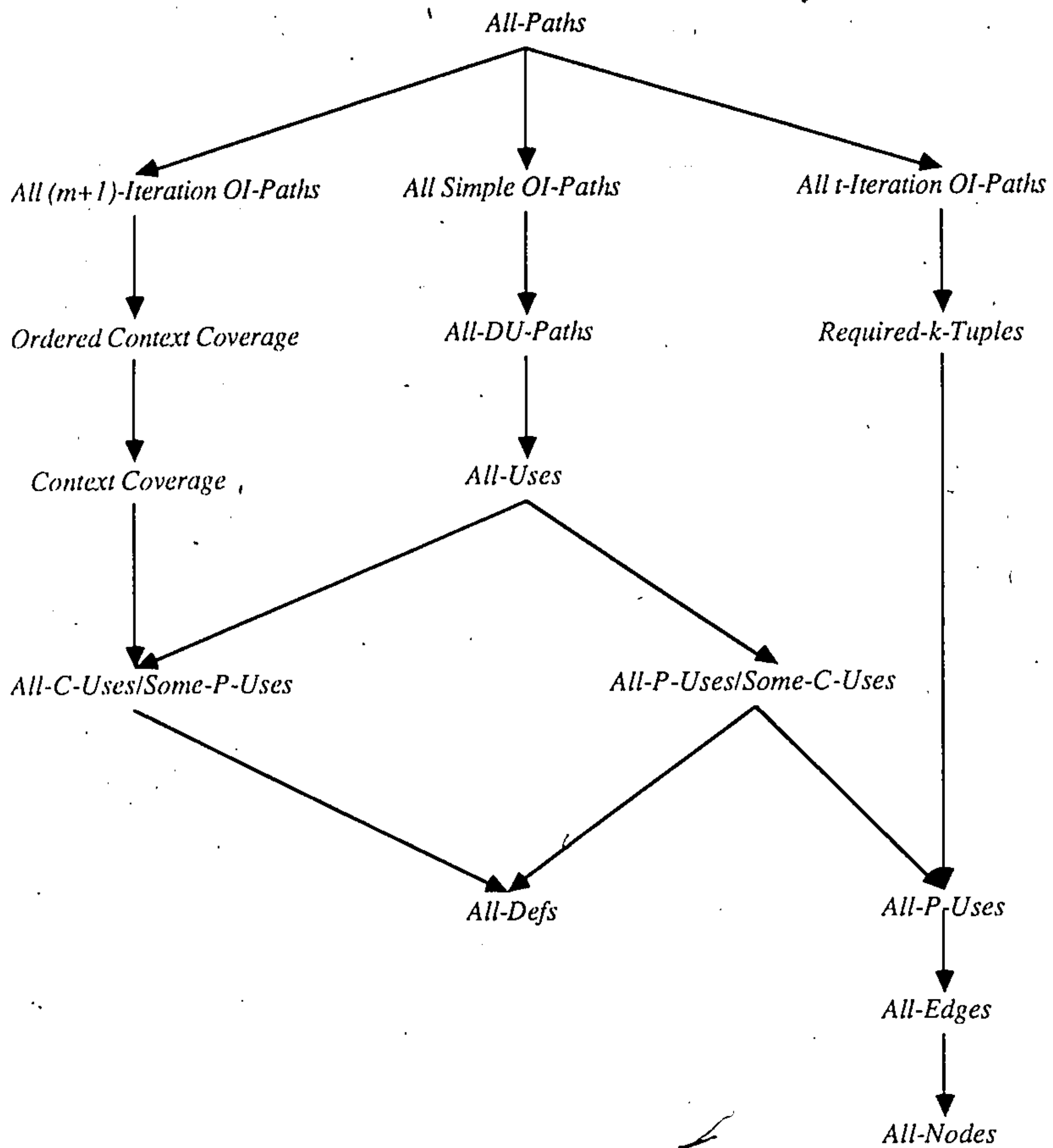


Figure 11: The Complete Inclusion Hierarchy

Chapter 4

Two Detailed Examples

4.1 Illustration of the All Simple OI-Paths Criterion

In this section, we demonstrate the coverage of *all simple OI-paths* criterion with respect to *all du-paths* criterion.

The simple program shown in Figure 12 is equivalent to the one used Rapps and Weyuker [10, 12]. This program computes X to the power of Y for any real number X and any integer number Y.

From Tables 2 and 3, it is straightforward to see that *all simple OI-paths* coverage is more comprehensive than that of *all du-paths*. That is, every du-path is covered by some simple OI-path, but not *vice versa* (see Tables 2a, 2b, 3a, and 3b). As shown in Tables 2c and 3c, *all simple OI-paths* requires more extensive path coverage by testing the loop over 4 tests where as *all du-paths* only requires testing over 2 tests (moreover, only one test will drive the program to traverse the loop twice).

All simple OI-paths is more thorough and informative than *all du-paths*. To see this,

note that every simple OI-path is actually a series of definition-use chains (i.e., du chains) from an input to an influenced output. Each such chain better capture the data flow in the corresponding functional segment of the specification. A simple OI-paths extends the coverage of several du-paths to a larger, more meaningful functional segment. Furthermore, local du-chains are overlooked by the all du-paths but not by all simple OI-paths. These du-chains as well as other non-local du-chains play important roles in the input-output relation analysis.

```
n_entry  begin
n1       input (Y);
n2       POW := Y;
n3       if Y< 0 then
n4           Y := -Y
n5       Z := 1;
         input (X);
n6       while Y ≠ 0 do
n7           Z := Z*X;
             Y := Y-1
         end;
n8       if POW < 0 then
n9           Z := 1/Z;
n10      ANSWER := Z+1;
n11      output (ANSWER)
n_exit   end.
```

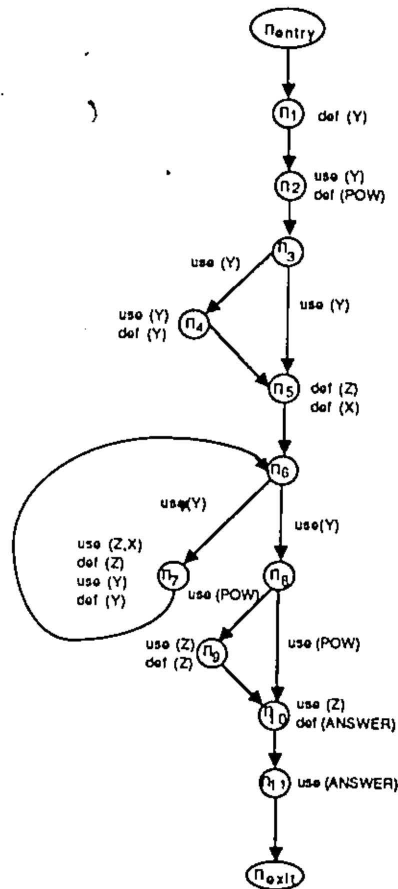


Figure 12: Example E3 and Its Corresponding Flowgraph.

Table 2a: All the DU-Paths in E3

	du-path	wrt	type of use
du1:	(n_1, n_2)	Y	c
du2:	(n_1, n_2, n_3, n_4)	Y	p, c
du3:	(n_1, n_2, n_3, n_5)	Y	p
du4:	$(n_1, n_2, n_3, n_5, n_6, n_7)$	Y	p, c
du5:	(n_4, n_5, n_6, n_7)	Y	p, c
du6:	$(n_1, n_2, n_3, n_5, n_6, n_8)$	Y	p
du7:	(n_4, n_5, n_6, n_8)	Y	p
du8:	(n_7, n_6, n_7)	Y	p, c
du9:	(n_7, n_6, n_8)	Y	p
du10:	$(n_2, n_3, n_4, n_5, n_6, n_8, n_9)$	POW	p
du11:	$(n_2, n_3, n_4, n_5, n_6, n_8, n_{10})$	POW	p
du12:	$(n_2, n_3, n_5, n_6, n_8, n_9)$	POW	p
du13:	$(n_2, n_3, n_5, n_6, n_8, n_{10})$	POW	p
du14:	(n_5, n_6, n_7)	X	c
du15:	(n_5, n_6, n_7)	Z	c
du16:	(n_5, n_6, n_8, n_9)	Z	c
du17:	(n_5, n_6, n_8, n_{10})	Z	c
du18:	(n_7, n_6, n_7)	Z	c
du19:	(n_7, n_6, n_8, n_9)	Z	c
du20:	(n_7, n_6, n_8, n_{10})	Z	c
du21:	(n_9, n_{10})	Z	c
du22:	(n_{10}, n_{11})	ANSWER	c

Table 2b: Non-Inclusive DU-Paths in E3

ndu1:	$(n_1, n_2, n_3, n_5, n_6, n_7)$	ndu8:	$(n_2, n_3, n_5, n_6, n_8, n_{10})$
ndu2:	(n_1, n_2, n_3, n_4)	ndu9:	(n_7, n_6, n_7)
ndu3:	$(n_1, n_2, n_3, n_5, n_6, n_8)$	ndu10:	(n_7, n_6, n_8, n_9)
ndu4:	(n_4, n_5, n_6, n_7)	ndu11:	(n_7, n_6, n_8, n_{10})
ndu5:	$(n_2, n_3, n_4, n_5, n_6, n_8, n_9)$	ndu12:	(n_9, n_{10})
ndu6:	$(n_2, n_3, n_4, n_5, n_6, n_8, n_{10})$	ndu13:	(n_{10}, n_{11})
ndu7:	$(n_2, n_3, n_5, n_6, n_8, n_9)$		

Table 2c: A Set of Complete Paths Satisfying All DU-Paths

$(n_{entry}, n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_9, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_8, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_8, n_9, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11}, n_{exit})$

Table 3a: All the Simple OI-Paths in E3

simple OI-path	input / output
$(n_5, n_6, n_8, n_9, n_{10}, n_{11})$	Z / ANSWER
$(n_5, n_6, n_8, n_{10}, n_{11})$	Z / ANSWER
$(n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	Z / ANSWER
$(n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	Z / ANSWER
$(n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_1, n_2, n_3, n_5, n_6, n_8, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_9, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_1, n_2, n_3, n_5, n_6, n_8, n_9, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	Y, Z / ANSWER
$(n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	X / ANSWER
$(n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	X / ANSWER
$(n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	Y, X / ANSWER
$(n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	Y, X / ANSWER
$(n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	Y, X / ANSWER
$(n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	Y, X / ANSWER

Table 3b: Non-Inclusive Simple OI-Paths in E3

simple OI-path	ndu-paths covered
$(n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_{10}, n_{11})$	4, 6, 13
$(n_1, n_2, n_3, n_5, n_6, n_8, n_{10}, n_{11})$	3, 8, 13
$(n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_9, n_{10}, n_{11})$	4, 5, 12, 13
$(n_1, n_2, n_3, n_5, n_6, n_8, n_9, n_{10}, n_{11})$	3, 7, 12, 13
$(n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	2, 9, 11, 13
$(n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11})$	1, 9, 11, 13
$(n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	2, 9, 10, 12, 13
$(n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11})$	1, 9, 10, 12, 13

Table 3c: A Set of Complete Paths Satisfying All Simple OI-Paths

$(n_{entry}, n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_8, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_9, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_8, n_9, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11}, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_7, n_6, n_7, n_6, n_8, n_9, n_{10}, n_{11}, n_{exit})$

4.2 Illustration of the All k-Iteration OI-Paths Criterion

In this section, we demonstrate the coverage of *all 3-iteration OI-paths* criterion with respect to *ordered context coverage* criterion as well as some *required K-tuples* criteria (for K = 2, 3, 4, and 5, respectively). This demonstration is based on the program shown in Figure 13a which is equivalent to the one used by Laski and Korel [6].

From Tables 4 to 9, it is straightforward to see that *all 3-iteration OI-paths* coverage is more comprehensive than those of *required K-tuples* and *ordered context coverage*.

Every ordered elementary data context is covered by some i-iteration OI-path (see Tables 4a, 4b, 9a, and 9b), where $1 \leq i \leq 3$. This, in another way, shows that our proof for Theorem 8 (m = 2 for this program) is correct. *Ordered context coverage* only requires 3 tests whereas *all 3-iteration OI-paths* requires more tests. It is important to note that every k-iteration OI-path (k is some integer) is actually a series of ordered elementary data contexts which represents a larger, more meaningful functional segment than the ones represented by the individual ordered elementary data contexts.

There are four *required K-tuples* criteria which can be applied to example E4. i.e., for K = 2, 3, 4, and 5, respectively. This is due to the fact that no K-dr interactions exist for $K \geq 6$. The coverage of *all 3-iteration OI-paths* is more thorough than that of any such *required K-tuples* (see Tables 5b, 6b, 7b, and 8b). It is noticed that *required K-tuples* criteria do not test those du-chains where definitions and uses that are reached from these definitions are at the same nodes. Furthermore, every k-iteration OI-path (k is some integer) is actually a series of K-dr interactions which represents a larger, more meaningful

functional segment than the ones represented by the individual K-dr interactions.

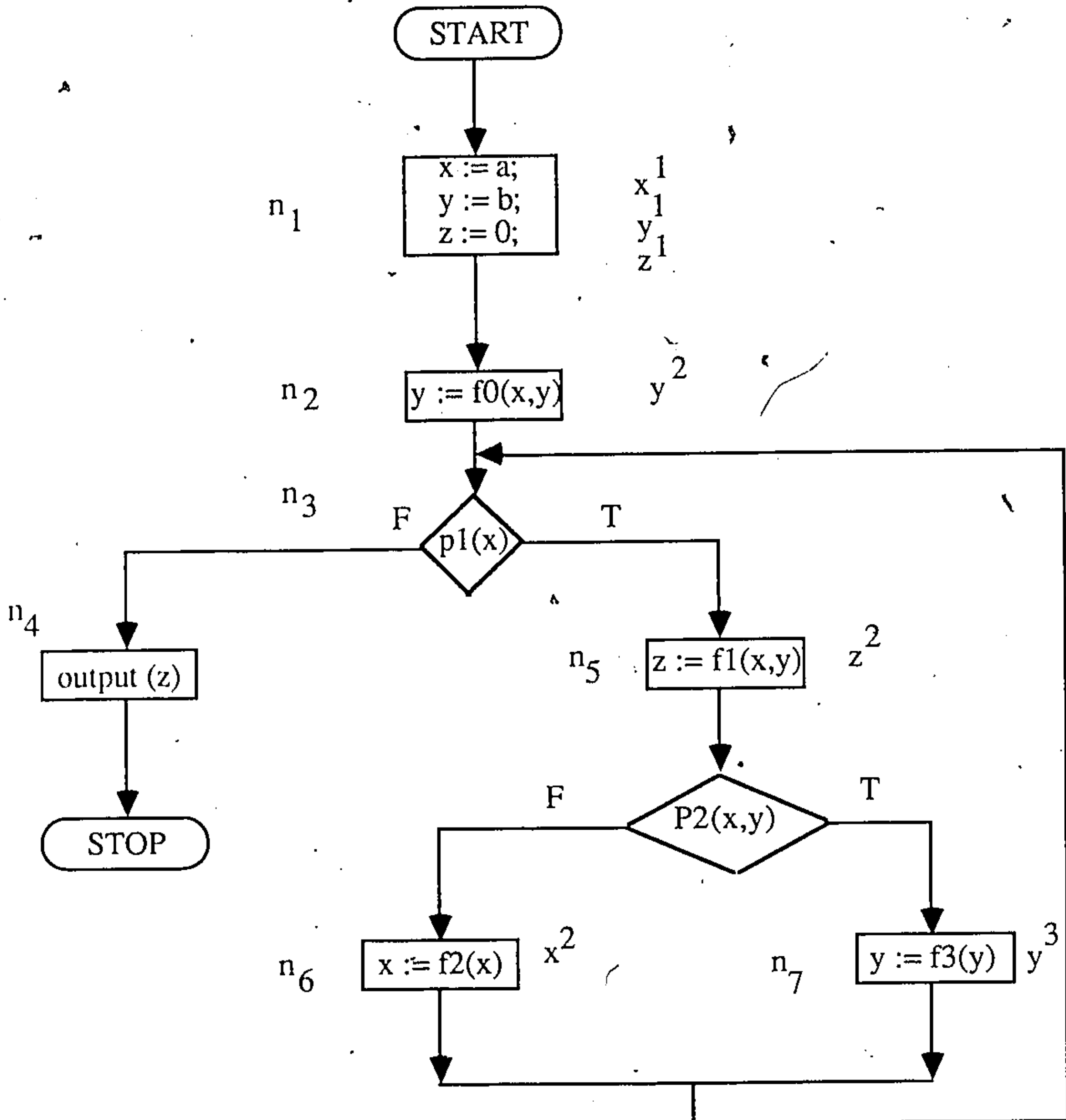


Figure 13a: Example E4 - A Program Schemata

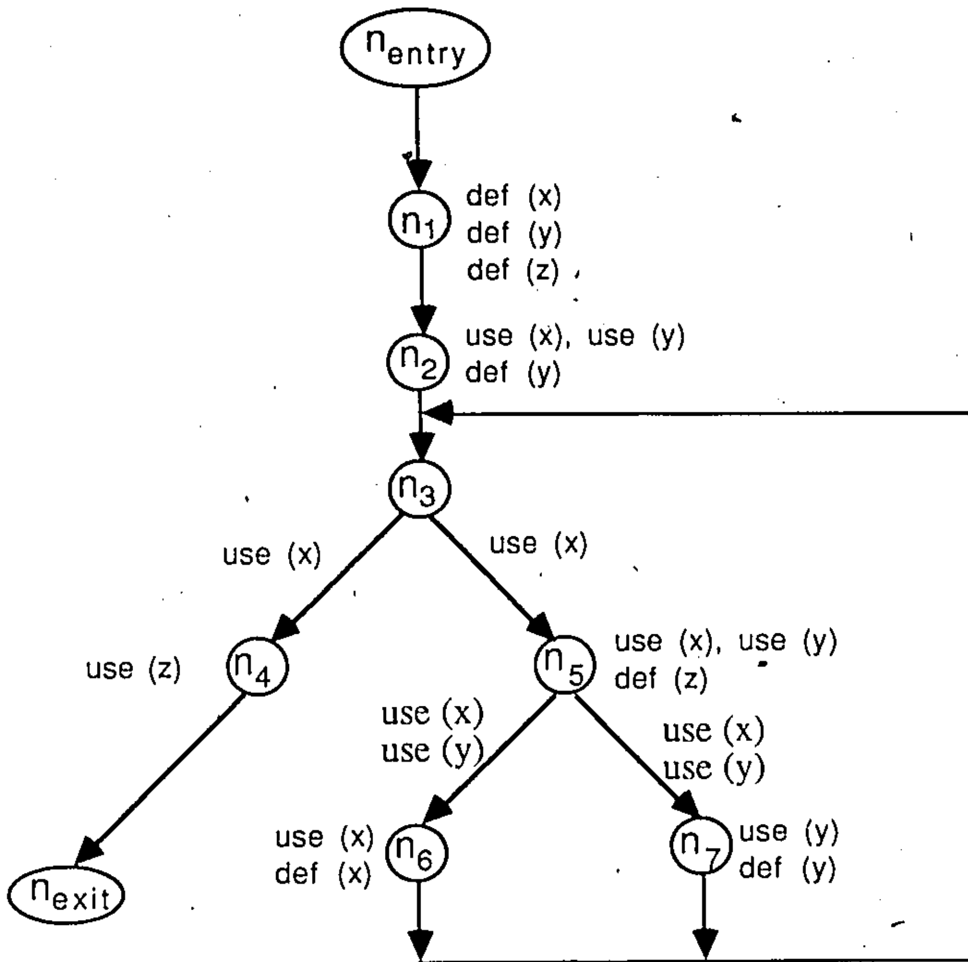


Figure 13b: The Flowgraph of Example E4

Table 4a: All the Ordered Data Context in E4

$ODC(n_2) = \{(x^1, y^1)\}$

$ODC(n_3) = \{(x^1)\}$

$ODC(n_4) = \{(z^1), (z^2)\}$

$ODC(n_5) = \{(x^1, y^2), (y^2, x^2), (x^1, y^3), (x^2, y^3), (y^3, x^2)\}$

$ODC(n_6) = \{(x^1), (x^2)\}$

$ODC(n_7) = \{(y^2), (y^3)\}$

Table 4b: A Set of Complete Paths Satisfying Ordered Context Coverage

Path 1: $(n_{entry}, n_1, n_2, n_3, n_4, n_{exit})$

This path covers ordered (elementary) data context: $ODC(n_2)$, $ODC(n_3)$, (z^1) in $ODC(n_4)$.

Path 2: $(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4, n_{exit})$

This path covers ordered (elementary) data context: $ODC(n_2)$, $ODC(n_3)$, (z^2) in $ODC(n_4)$, (x^1, y^2), (x^1, y^3), (y^3, x^2) in $ODC(n_5)$, (x^1) in $ODC(n_6)$, and $ODC(n_7)$.

Path 3: $(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4, n_{exit})$

This path covers ordered (elementary) data context: $ODC(n_2)$, $ODC(n_3)$, (z^2) in $ODC(n_4)$, (x^1, y^2), (y^2, x^2), (x^2, y^3), $ODC(n_6)$, and (y^2) in $ODC(n_7)$.

Table 5a: All the 2-dr Interactions in E4

2-dr interaction	definitions involved	first definition	last use
(n_1, n_2)	x^1 / y^1		c-use
(n_1, n_3)	x^1		p-use
(n_1, n_4)	z^1		c-use
(n_1, n_5)	x^1		c-use, p-use, in the loop
(n_1, n_6)	x^1		c-use, in the loop
(n_2, n_5)	y^2		c-use, p-use, in the loop
(n_2, n_7)	y^2		c-use, in the loop
(n_5, n_4)	z^2	in the loop	c-use
(n_6, n_3)	x^2	in the loop	p-use, in the loop
(n_6, n_5)	x^2	in the loop	c-use, p-use, in the loop
(n_7, n_5)	y^2	in the loop	c-use, p-use, in the loop

Table 5b: Subpaths to be Covered by Required 2-Tuples Criterion

(n_1, n_2)	(n_1, n_2, n_3, n_4)
(n_1, n_2, n_3, n_5)	$(n_1, n_2, n_3, n_5, n_6)$
$(n_1, n_2, n_3, n_5, n_7)$	(n_2, n_3, n_5)
(n_2, n_3, n_5, n_6)	(n_2, n_3, n_5, n_7)
$(n_5, n_6/n_7, n_3, n_4)$	(n_6, n_3, n_5)
(n_6, n_3, n_4)	(n_6, n_3, n_5, n_6)
(n_6, n_3, n_5, n_7)	(n_7, n_3, n_5)
(n_7, n_3, n_5, n_6)	(n_7, n_3, n_5, n_6)

Table 6a: All the 3-dr Interactions in E4

3-dr interaction	definitions involved	first definition	last use
(n_1, n_2, n_5)	$x^1 / y^1, y^2$		c-use, p-use, in the loop
(n_1, n_2, n_7)	$x^1 / y^1, y^2$		c-use, in the loop
(n_1, n_5, n_4)	x^1, z^2		c-use
(n_1, n_6, n_3)	x^1, x^2		p-use, in the loop
(n_1, n_6, n_5)	x^1, x^2		p-use, in the loop
(n_2, n_5, n_4)	y^2, z^2		c-use
(n_2, n_7, n_5)	y^2, y^3		c-use, p-use, in the loop
(n_6, n_5, n_4)	x^2, z^2	in the loop	c-use
(n_7, n_5, n_4)	y^3, z^2	in the loop	c-use

Table 6b: Subpaths to be Covered by Required 3-Tuples Criterion

(n_1, n_2, n_3, n_5)

$(n_1, n_2, n_3, n_5, n_6)$

$(n_1, n_2, n_3, n_5, n_7)$

$(n_1, n_2, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5)$

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_6)$

$(n_2, n_3, n_5, n_7, n_3, n_5)$

$(n_2, n_3, n_5, n_7, n_3, n_5, n_6)$

$(n_2, n_3, n_5, n_7, n_3, n_5, n_7)$

$(n_6, n_3, n_5, n_7, n_3, n_4)$

$(n_7, n_3, n_5, n_6, n_3, n_4)$

Table 7a: All the 4-dr Interactions in E4

4-dr interaction	definitions involved	first definition	last use
(n_1, n_2, n_5, n_4)	$x^1/y^1, y^2, z^2$		c-use
(n_1, n_2, n_7, n_5)	$x^1/y^1, y^2, y^3$		c-use, p-use, in the loop
(n_1, n_6, n_5, n_4)	x^1, x^2, z^2		c-use
(n_1, n_7, n_5, n_4)	y^2, y^3, z^2		c-use

Table 7b: Subpaths to be Covered by Required 4-Tuples Criterion

$(n_1, n_2, n_3, n_5, n_6, n_3, n_4)$	$(n_1, n_2, n_3, n_5, n_7, n_3, n_5)$
$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6)$	$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7)$
$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4)$	$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_4)$

Table 8a: All the 5-dr Interactions in E4

5-dr interaction	definitions involved	last use
$(n_1, n_2, n_7, n_5, n_4)$	$x^1/y^1, y^2, y^3, z^2$	c-use

Table 8b: Subpaths to be Covered by Required 5-Tuples Criterion

$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_4)$

Table 9a: All the 3-Iteration OI-Paths in E4

loop traversed zero time:

(n_1, n_2, n_3, n_4)

loop traversed once:

$(n_1, n_2, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_7, n_3, n_4)$

loop traversed twice:

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_4)$

loop traversed three times:

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_4)$

$(n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_4)$

Table 9b: A Set of Complete Paths Satisfying All 3-Iteration OI-Paths

loop traversed zero time:

$(n_{entry}, n_1, n_2, n_3, n_4, n_{exit})$

loop traversed once:

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_4, n_{exit})$

loop traversed twice:

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_4, n_{exit})$

loop traversedthree times:

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_5, n_6, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_5, n_7, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_5, n_6, n_3, n_4, n_{exit})$

$(n_{entry}, n_1, n_2, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_5, n_7, n_3, n_4, n_{exit})$

Chapter 5

Conclusions

5.1 Summary of Contributions

We have proposed a class of data flow oriented path selection criteria, called the *OI-paths* criteria that are based on the identification of associations between each output variable and all input variables that influence the output variable in a source program. These criteria require each such association to be examined at least once during testing. Hence, more insight can be gained on a better understanding of the functions a program performs as well as the consistency of the program with its specification, in comparison with other data flow oriented criteria.

It has been shown that these criteria are stronger than other data flow oriented criteria. These criteria also bridge the gap between the weak branch coverage and the impractical path coverage. More importantly, an effort has been made toward using program semantics such as input-output relations in guidance to the path selection process. This is an improvement over using program syntax alone. The *OI-Paths* criteria are more thorough and informative than the existing criteria presented in Chapter 2. To see this, note that

every OI-path is actually a series of definition-use chains (i.e., du-chains) from an input to an influenced output. Each such chain better capture the data flow in the corresponding functional segment of the specification. For example, a simple OI-Path extends the coverage of several du-paths to a larger, more meaningful functional segment.

As shown in section 2.5.3, in the worst cases, the complexity of the *all 0-iteration, all simple OI-paths* and *all k-iteration OI-paths (k≥1)* criteria are $O(2^t)$, $O(2^{2t})$, and $O(2^{kt})$, respectively. We would like to point out that the mere fact that these criteria have such bounds, does not necessarily rule them out for practical consideration. The cost of a criterion should be considered for an individual program, rather than dismissing it based on the upper bound [12].

It should be mentioned that most of the ideas related to the criteria are independent of the details of the programming language in which the source code is given.

In this thesis, we solely concentrate on the test path selection phase. It has been assumed that a reasonable test data selection criterion will be used in the test case generation phase. In general, once a finite set of complete paths P that satifies a given criterion is chosen, one enters the test data selection phase, wherein a finite set of test data are chosen from the program input domain in order to impose the execution of the selected paths P. As classified in [13], a computer program contains, in general, two types of errors which have been identified as computation errors and domain errors [16, 17]. Each proposed test data selection strategy shows how to select a set of test data to cope with effective detection of those errors. In our opinion, it is necessary that both structural and functional testing principles should be incorporated into the testing process in order to effectively reveal domain errors as well as computation errors.

5.2 Future Research Directions

It should be pointed out that like other structural test selection criteria, our criteria are also hampered by its reliance on the syntactic selection of paths to be traversed during testing. It is well known that not necessarily all paths in a program are feasible (executable) and that the identification of feasible or infeasible paths is an undecidable problem. That is, the syntactic information in a flowgraph, representing the source code of a program, is not sufficient to determine whether a particular path is feasible (or infeasible). Thus, a set of paths selected to satisfy any of our criterion may also contain infeasible paths.

There seems to be an interplay between the selection of paths to satisfy a certain criterion and the search for input values to enable the program to follow each selected path. A recent study [9] exploits this interplay by using prefixes of previously selected paths to select subsequent paths. This is an interesting approach that needs further study.

A software tool need to be developed in order to automate the identification of the input-output relations in a program and generation of simple OI-paths and k-iteration OI-paths in the path selection phase.

The proposed criteria result in the identification and generation of input-output relations through static analysis of the source code of an implementation of a given specification. The specification also defines functionality in terms of input-output relations. It will be worthwhile to investigate some methods to compare the input-output relations identified in the source code with those relations defined in the specification.

References

[1] L.A. Clarke *et al.* , An Investigation of Data Flow Path Selection Criteria, in: Workshop on Software Testing, Banff, Canada (1986) 23-32.

[2] L.A. Clarke *et al.* , A Comparison of Data Flow Path Selection Criteria, in: 8th Int'l Conf. on Software Engrg., USA (1985) 244-251.

[3] W. E. Howden, Functional Program Testing and Analysis, (McGraw Hill, New York, 1987).

[4] W. E. Howden, A Functional Approach to Program Testing and Analysis, IEEE Trans. Software Engrg. SE-12 (10) (1986), 997-1005.

[5] B. Korel, The Program Dependence Graph in Static Program Testing, Info. Proc. Lett. 24 (2) (1987) 103-108.

[6] J. W. Laski and B. Korel , A Data Flow Oriented Program Testing Method , IEEE Trans. Software Engrg. SE-9 (3) (1983) 347-354.

[7] E. Miller and W. E. Howden, Software Testing and Validation Techniques, IEEE Tutorial, second edition, (IEEE Computer Society Press, Los Alamitos, CA, 1981).

[8] S. C. Ntafos, On Required Element Testing, IEEE Trans. Software Engrg. SE-10 (6) (1984) 795-803.

[9] R.E. Prather and J. P. Myers Jr., The Path Prefix Software Testing Strategy, IEEE Trans. Software Engrg. SE-13 (7) (1987) 761-766.

[10] S. Rapps and E. Weyuker, Selecting Software Test Data Using Data Flow Information, IEEE Trans. Software Engrg. SE-11 (4) (1985) 367-375.

[11] M.D. Welser, J.D. Gannon and P.R. McMullin, Comparison of Structural Test

Coverage Metrics, IEEE Software, 2 (2) (1985) 80-85.

[12] E. J. Weyuker, The Complexity of Data Flow Criteria for Test Data Selection, Info. Proc. Lett., 19 (2) (1984) 103-109.

[13] W. E. Howden, Reliability of the Path Analysis Testing Strategy, IEEE Trans. Software Engrg., SE-2 (3) (1976) 208-215.

[14] H. M. Sneed, Data Coverage Measurement in Program Testing, in: Workshop on Software Testing, Banff, Canada (1986) 34-40.

[15] L. G. Stucki, Automatic Generation of Self-Metric Software, Recordings 1973 IEEE Symp. Software Reliability, April, 1973, 94-100.

[16] L. J. White and E. I. Cohen, A Domain Strategy for Computer Program Testing, IEEE Trans. Software Engrg., SE-6 (3) (1980) 247-257.

[17] L. A. Clarke, *et al.*, A Close Look at Domain Testing, IEEE Trans. Software Engrg., SE-8 (4) (1982) 380-390.

[18] W. E. Howden, Methodology for the Generation of Program Test Data, IEEE Trans. Comput., C-24 (1975) 554-559.

[19] E. J. Weyuker and T. J. Ostrand, Theories of Program Testing and the Application of Revealing Subdomains, IEEE Trans. Software Engrg., SE-6 (1980) 236-246.

[20] E. J. Weyuker and T. J. Ostrand, Current Directions in the Theory of Testing, in Proc. Computer Software and applications Conf. (COMPSAC 80), IEEE, Chicago, IL, (1980) 386-389.

[21] J. C. Huang, An Approach to Program Testing, Comput. Surveys, 7 (3) (1975) 114-128.

[22] M. R. Paige, An Analytic Approach to Software Testing, in Proc. Computer Software and applications Conf. IEEE, Chicago, IL, (1978) 527-531.