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.

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

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.

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

SOFTWARE TEST GENERATION BASED ON FLOW MODELS

by

Bo Yang

A thesis submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering

University of Ottawa
Ottawa, Ontario, Canada
March 1991

ISBN 0-315-70474-8

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis. I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Bo Yang

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Bo Yang

to my mother
and
to the memory of my father

Abstract

Software testing is one of the most widely used quality assurance methodologies. A large software system usually has a hierarchical structure: for example, system, subsystems (programs), subprograms, and procedures, where a subsystem is composed of a number of subprograms, each of which in turn is composed of a number of procedures. Testing of a system can be done at different levels with different emphases. In this thesis, we focus on testing at the two lowest levels, namely procedure and subprogram levels.

Until recently, many testing techniques used control flow graph or its variants for selecting the tests. It is known that such a flow model is only capable of capturing the control flow. For data flow oriented testing, the def-use graph is used to represent both control flow and data flow. Based on this model, we propose a class of data flow oriented test selection criteria using input-output and input-predicate relations. These criteria are shown to have certain merits over the existing test selection criteria.

However, a def-use graph can be only used to capture control flow and data flow within a procedure. Since control flow and data flow also exist among interacting procedures in a subprogram, a more expressive model is necessary in order to perform testing at subprogram level. Such a model is proposed by extending the def-use graph model. By using this model, the tests generated using a test selection criterion are shown to be more meaningful and, likely, more effective in detecting errors.

Acknowledgements

I would like to express my sincerest gratitude to my thesis supervisor, Dr. Hasan Ural, for his advice, encouragement, commitment, understanding, and support during my studies.

I would like to thank Professor R. L. Probert for many useful comments and discussions. He has been one of the most influential people in my thinking.

I would like to thank Professor S. Das for his stimulating course instructions which, in particular, had deepened my understanding of finite-state machine based testing.

I would like to take this opportunity to thank all of my teachers, colleagues, and friends, from whom I have gradually built up my "knowledge base".

I am most grateful to my family, my grandparents and other relatives for their love, constant encouragement, tolerance, and support in various forms.

Special thanks go to Q. Chen who was very helpful and supportive in the final stage of the thesis.

Finally, I would like to acknowledge the financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Telecommunications Research Institute of Ontario (TRIO).

Table of Contents

Abstract iv
Acknowledgements v
List of Figures viii
List of Tables ix

Chapter 1. Introduction 1
1.1 Background 1
1.2 Motivations 6
1.3 Major Contributions 9
1.4 Organization of the Thesis 11

Chapter 2. Flow Model Based Software Testing: An Overview 13
2.1 Introduction 14
2.2 Finite State Machine Testing 15
2.2.1 The FSM Model and Its Graph Representation 18
2.2.2 Overview of FSM Based Testing Methods 19
2.2.3 Optimization vs. Approximation 20
2.3 Path Selection Criteria 24
2.3.1 Control Flow Oriented Path Selection 24
2.3.2 Data Flow Oriented Path Selection 27

Chapter 3. Intra-Procedural Data Flow Testing 29
3.1 Background 30
3.2 The Def-Use Graph Model 33
3.3 Representing Pascal Procedures with Def-Use Graphs 35
3.4 Literature Review and Critique 41

Chapter 4. IP/O-Chains Criteria 47
4.1 Motivation 48
4.2 Definitions of the IP/O-Chains Criteria 52
4.3 Comparisons 56
4.3.1 IP/O-Chains vs. Some Control Flow-Oriented Criteria 59
4.3.2 IP/O-Chains vs. the Rapps-Weyuker Criteria 60
4.3.3 IP/O-Chains vs. Required k-Tuples Criteria 66
4.3.4 IP/O-Chains vs. Context Coverage Criteria 68

Chapter 5. Variations of the IP/O-Chains Criteria.....71
5.1 Definition of the IP/O2-Chains+ Criterion.....72
5.2 Definition of the IP/O2-Chains* Criterion.....78

Chapter 6. Inter-Procedural Data Flow Testing.....84
6.1 Background and Motivation.....86
6.2 Inter-Procedural Data Flow Modeling and Analysis.....92
6.2.1 The Inter-Procedural Flow Graph Model.....92
6.2.2 Computing Inter-Procedural Data Dependencies.....96
6.3 Extended Def-Use Graph: The Proposed Flow Model.....98
6.3.1 The Super Graph Model.....98
6.3.2 The EDUG Model.....101
6.3.3 Representing Pascal Programs with EDUG.....101
6.4 Data Flow Testing Using the EDUG Model.....105

Chapter 7. Conclusions and Future Research.....118
7.1 Summary of the Thesis.....119
7.2 Future Research.....121

References.....123

List of Figures

Figure	Description	Page
Figure 2.1	An Example FSM	16
Figure 3.1	Procedure A and Its Def-Use Graph	33
Figure 3.2	Procedure B and Its Def-Use Graph	43
Figure 4.1	Procedure C and Its Def-Use Graph	48
Figure 4.2	A Segment of the Triangle Classification Procedure & Its Def-Use Graph	51
Figure 4.3	Procedure D and Its Def-Use Graph	53
Figure 4.4	Procedure E and Its Def-Use Graph	61
Figure 4.5	Procedure F and Its Def-Use Graph	63
Figure 4.6	Procedure G and Its Def-Use Graph	69
Figure 4.7	The Partial Inclusion Hierarchy	70
Figure 5.1	Procedure H and Its Def-Use Graph	72
Figure 5.2	The Complete Inclusion Hierarchy	83
Figure 6.1	A Pascal Subprogram Listing	88
Figure 6.2	DUGs for Main, GetMax, and PairMax	89
Figure 6.3	The IFG for the Pascal Example in Fig.6.1	95
Figure 6.4	The Super Graph for the Pascal Example in Fig.6.1	100
Figure 6.5	The EDUG for the Pascal Example in Fig.6.1	104

List of Tables

Table	Description	Page
Table 2.1	Minimum-Length UIO Sequences for Fig.2.1.	17
Table 3.1a.	Defs & Uses of Non-Simple Variables	36
Table 3.1b.	Interpretation of Predefined Procedures	37
Table 3.2.	Translation Rules for Simple Statements	38
Table 3.3.	Translation Rules for Conditional Statements	39
Table 3.4.	Translation Rules for Repetitive Statements	40
Table 4.1	A Set of Paths Covering IP/O_1-Chains in Procedure E	61
Table 4.2	Du-Pairs and Du-Paths in Procedure F	64
Table 4.3	IP/O_1-Chains in Procedure F	65
Table 4.4	IP/O_2-Chains in Procedure F	66
Table 4.5	A Set of Paths Satisfying IP/O_2-Chains Coverage in Procedure F	66
Table 5.1	IP/O_1-Chains in Procedure H	72
Table 5.2.	IP/O_2-Chains in Procedure H	73
Table 5.3.	Data Contexts in Procedure H	74
Table 5.4.	Paths Required by IP/O_2-Chains* for Procedure F	79
Table 5.5.	Data Context in Procedure F	81
Table 5.6.	Paths Required by IP/O_2-Chains$^+$ for Procedure F	81
Table 5.7	Du-Pairs and Du-Paths in Procedure H	82
Table 6.1.	Inter-Procedural Data Dependencies among Main, GetMax, and PairMax	97
Table 6.2.	Du-Pairs in Procedure Main of Fig.6.2	105
Table 6.3.	Du-Pairs in Procedure GetMax of Fig.6.2	105
Table 6.4.	Du-Pairs in Procedure PairMax of Fig.6.2	106
Table 6.5.	Du-Pairs in Fig.6.3	106
Table 6.6	IP/O_1-Chains & IP/O_2-Chains in Main	108
Table 6.7	IP/O_1-Chains & IP/O_2-Chains in GetMax	109
Table 6.8	IP/O_1-Chains & IP/O_2-Chains in PairMax	110
Table 6.9	IP/O_1-Chains & IP/O_2-Chains in Fig.6.3	111

Chapter 1

Introduction

1.1 Background

The quality of software systems has become a crucial factor for customer satisfaction, and thus high-quality software has become an important goal.

Software quality can be defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software [Pres 87].

There are numerous factors which affect software quality: its operational characteristics (e.g., operational correctness), its ability to undergo change (e.g., enhanceability and maintainability), and its adaptability to new environments (e.g, portability and reusability) [McRi 77]. Among them, the most important factor is perhaps the operational correctness of the software, because the first thing the users want is that the software works as intended. This is particularly the case for software systems that handle critical situations such as nuclear power plant control, space missions, and telecommunication systems.

As the size and complexity of software systems have continued to grow drastically over the last two decades, it has become more and more important to assess their operational correctness in an automatic manner by means of software validation and verification (V & V) activities which include testing, analysis, and proof of correctness.

This thesis focuses on software testing which is one of the most widely used V & V activities for checking the operational correctness of software [MiHo 81, Howd 87]. The general aim of software testing is to reveal the existence of errors in a software entity under test by executing it in a controlled environment over a finite set of tests. One of the most challenging tasks in software testing is to generate the test set such that it can be effectively used to detect the errors in a software entity under test. Techniques for selecting such tests generally fall into three broad categories: black-box testing, grey-box testing, and white-box testing. In black-box testing [Myer 79, MiHo 81] the specification of the system is used to select the tests, in grey-box testing [Prob 82] the design of the system is used to select the tests, and in white-box testing [Myer 79, MiHo 81] the source code of the system is used to select the tests.

Specification, design, and implementation of a software system represent the functionality of the system in increasing levels of detail. Moreover, they may be given in different formalism, notations, and languages which depict different aspects of a software system in different depth. For example, control flow graph [Hech 77] can be used to model the sequential structure in a program segment, call graph [Hech 77] can be used to model control and data dependencies across program segments, and Petri net [Pete 77, Pete 81] is well suited for describing the timing, synchronization, and resource contention aspects of software systems.

A variety of testing methods have been devised for generating tests based on these and other models. It is widely acknowledged that tests that are derived from specifications,

designs, and implementations by using these methods differ from each other in their coverage, purpose, and testing effectiveness, and therefore are considered as complementary to each other.

In this thesis, we focus on the white-box test generation methods which utilize the source code of a program to select the tests. A program may be viewed as exhibiting two types of flows: flow of control and flow of data. **Control flow** is concerned with sequencing of events, actions, or program statements. **Data flow** is concerned with the way the data values are created, accessed, and abandoned. Both control flow and data flow can be generally modeled by **directed graphs**, or digraphs. We will refer to such a graph as a **flow model**. The well-known flow models are: control flow graph, call graphs, data flow graph [MiHo 81], def-use graphs [RaWe 85], and program dependence graphs [FeOt 87, Kore 87], and a variety of transition systems such as finite state machines (FSMs) [Gill 62, Koha 78], extended finite state machines (EFSMs) [BoSu 80, BoSu 83], and Petri nets [Pete 77].

Once a model is conceived, a variety of white-box testing methods may be used. For example, a program may be modeled by an EFSM in which the control portion is modeled by a deterministic finite-state machine, and the data portion is modeled by program segments. FSM-based testing techniques as described in [AhDa 88, ShLo 89, SiLe 89, YaUr 90a] can be used to generate test sequences for the control portion, whereas the data-flow based techniques [LaKo 83, Ntaf 84, RaWe 85, Ural 87, SaBo 87, UrYa 88, UrYa 91] can be used to generate test sequences for the data portion.

Given a flow model of a program, a white-box test generation technique may be divided into two seemingly distinct phases: (test) path selection and test data generation [PrMy 87]. In the path selection phase, a finite set of paths is selected from the flow model

to, in general, satisfy a test generation criterion which is defined in terms of control and/or data flow information contained in the flow model.

Most path selection criteria are based on control flow analysis which examines the branch and loop structures of an program. The well-known control flow based path selection criteria are node coverage, branch coverage, and path coverage criteria which require the selection of test data that result in traversing those paths which cover all nodes, edges, and paths at least once in the flow model of a given program, respectively [Myer 79, MiHo 81]. Depending on the particular flow model used, these criteria may have different meanings. For example, when flow graph model is used branch coverage implies the condition coverage of the program. On the other hand, branch coverage can imply the coverage of all the transitions if an FSM is used to model the functionality of a system.

Control flow oriented test path selection criteria such as branch coverage, boundary-interior testing [Howd 75], and path coverage, require examinations of the control and loop structures of a program being tested. Although it is necessary to exercise a certain degree of control structures during testing [Stuc 73], path selection criteria based merely on control flow information are not enough to insure the quality of software under test [RaWe 85, Howd 87].

It is argued that data flow oriented path selection should therefore be considered in addition to exercising control flow structures [RaWe 85, Howd 87]. The well-known data flow oriented test path generation criteria are Laski and Korel's context coverage criteria [LaKo 83], Ntafos's required k-tuples criteria [Ntaf 84], and Rapps and Weyuker's all-uses and all du-paths criteria [RaWe 85]. These three families of criteria focus on tracing the flow of data in a flow model 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.

In the test data generation phase, a finite set of test data is chosen from the input domain of the program in order to impose executions on the selected paths. However, this is not a trivial problem. In fact, frequently there exists no input data for traversing a selected path. Such a path is called an **infeasible** or **non-executable path**. It's well-known that the problem of determining the feasibility of a given path through a given program is unsolvable being equivalent to the Halting Problem. This is certainly an unfortunate characteristic of white-box testing, which affects its accuracy and applicability [FrWe 88]. In one study [WoHe 80], it is found that only a very small percentage of all paths are feasible. Some effort has been made to use program semantics to suppress the generation of infeasible paths [KrSm 73, Oste 77]. In some cases, symbolic evaluation [Clar 76, Fran 90] has been shown to be an effective means to derive test data.

1.2 Motivations

In this thesis, we will focus on test path selection based on data flow information. Data flow oriented test path selection focuses on tracing the flow of data in the program through the associations between variable definitions and their uses in either assigning values to other variables or determining the outcome of conditional branching. Associations between definitions and uses of variables can be established on the basis of a structural unit called definition-use pair, or simply du-pair. A du-pair is a tuple (d(v), u(v)), where v is a variable, d(v) is a definition of v, and u(v) is a use of v such that d(v) reaches u(v) through a path without encountering another definition of v.

For the well-known data flow oriented path selection criteria, namely all-du-paths criterion, context coverage criteria, and required k-tuples criteria, the structural components (henceforth referred to as test units) that serve as the basis of these path selection criteria are: du-path, elementary data context, and k-dr interaction, respectively. A du-path is related to a single du-pair and thus, considers only individual associations between definitions and uses of variables. An (ordered) elementary data context is related to a group of du-pairs whose uses occur in the same statement and whose definitions reach that statement (in some fixed order) via a common control path. A k-dr interaction ($k \geq 2$) is related to a chain of k-1 du-pairs in which the use in each du-pair is directly employed for the definition associated with the succeeding du-pair.

None of these test units explicitly identify the program functionality which can be viewed as a set of functions defined over program inputs [Kore 87, Howd 87]. In [Kore 87, UrYa 88], it was shown that the implemented functionality of a program can in part be revealed by identifying specific input variables affecting specific outputs in the program through input-output relation analysis.

Such analysis is used in Chapter 4 of this thesis in conjunction with input-predicate relation analysis to form a larger structural component called IP/O-chain which is indicative of the program functionality. Each IP/O-chain is a sequence of du-pairs through which the effects of a program input on the control flow and the data flow in a given program can be traced. The class of data flow oriented path selection criteria proposed in this thesis is based on the coverage of IP/O-chains and its variants identified in a given program.

Block-structured or procedural languages remain to be the most widely used computer languages. Programs written in these languages usually consist of a main program and a number of procedures. As current trends in software development encourage a high degree of modularity, the number of procedure calls and returns executed in a module maintain its growth. This increase in use of procedures mandates the efficient testing of data dependencies among procedures.

Data dependencies exist not only within procedures, but also across procedure boundaries. So far, the use of the data flow oriented test generation methods has been restricted to testing intra-procedural data dependencies modeled by a def-use graph (e.g., [FrWe 88]). Inter-procedural data dependencies that exist among procedures are not representable in the def-use graph. Inter-procedural data flow testing requires and employs additional information about the data flow across procedure boundaries, and thus tends to be more accurate. This information provides the locations of variable defs and their uses reached across procedure boundaries, and is gathered by inter-procedural data flow analysis [HaSo 89].

In accordance with the classification of data flow analysis techniques, data flow oriented test path generation methods can be further classified into **intra-procedural data flow testing** and **inter-procedural data flow testing**. Intra-procedural data flow testing handles procedures and treats them as nearly independent functional units. On the

other hand, inter-procedural data flow testing considers procedures as interrelated functional units. The former approach is less complex, however the latter approach is more complete and accurate. Chapters 3, 4 and 5 discuss intra-procedural data flow testing in detail, and Chapter 6 discusses the inter-procedural data flow testing.

1.3 Major Contributions

The major contributions of this thesis include the following:

1) Conception and formalization of the notion "a variable definition affects a variable use". With this notion, we are able to clearly and precisely define data flow chains.

2) Emphasis of the importance of identifying how inputs affect outputs or predicates in a program. Definition of input-output and input-predicate relations are given using the "affect" notion. We argue that it is necessary to make use of these two relations as basis of test path selection.

3) Introduction of IP/O-chains criteria. These criteria are designed to select paths such that a) every element in the input-output and input-predicate relations is covered at least once; b) minimum data flow requirements are satisfied (i.e., every definition-use pair is covered at least once); and c) minimum control flow requirements are satisfied (i.e., every branch in a flow model is covered at least once).

4) Comparisons of IP/O-chains criteria to the well-known data flow oriented path selection criteria. The IP/O-chains criteria are shown to strictly include the all-uses criterion, but they are incomparable with the all-du-paths criterion and the context coverage criteria. It is shown that, for any k, there exists an IP/O-chains criterion which strictly includes the required k-tuples criterion.

5) Introduction of some useful variations of the IP/O-chains criteria, namely, IP/O_2-chains$^+$ criterion and IP/O_2-chains* criterion, both meet the fore-mentioned design objectives. The IP/O_2-chains$^+$ criterion is shown to strictly include the context coverage criterion, and the IP/O_2-chains* criterion is shown to strictly include the all-du-paths criterion

6) Introduction of the extended def-use graph model for testing at subprogram level. We believe that it is more meaningful, while still manageable, to perform data flow testing at subprogram level. The extended def-use graph model is shown to be suitable for modelling and testing both intra-procedural and inter-procedural control and data flow dependencies. It is shown that existing data flow oriented test path selection criteria may use this model to yield more meaningful and more accurate test paths.

1.4 Organization of the Thesis

The following is a brief description of each of the remaining chapters of this thesis:

Chapter 2 details the discussion of flow model based testing. The state-of-the-art techniques are reviewed. This includes a section on FSM testing, where an extensive survey of the existing FSM testing techniques is given, and another section on path selection criteria, where control flow oriented path selection is reviewed and necessity of data flow oriented coverage is discussed.

Chapter 3 first defines some related terminology necessary for further discussions, then describes the def-use graph model which can be used for modelling both the control flow and the data flow in a procedure. As an example, it is shown how essential Pascal programming constructs can be modelled as components of a def-use graph. Finally, a literature review and critique of the well-known data flow oriented path selection criteria is conducted, possible improvements which is one of the motivating forces for this thesis are also pointed out.

Chapter 4 introduces the proposed IP/O-chains criteria. It is observed that the well-known data flow oriented path selection criteria, i.e., data context coverage criteria [LaKo 83], required k-tuples criteria [Ntaf 84], and all-uses and all-du-paths criteria [RaWe 85], fail to abstract and cover program functional components which relate inputs to outputs or predicates. The IP/O-chains criteria are designed to overcome this by utilizing input-output and input-predicate relations. The relative strength of IP/O-chains criteria is studied. The results show that the IP/O-chains criteria guarantee the all-uses coverage which we consider as the minimum data flow oriented path coverage criterion. Furthermore, we prove that for any constant k, there exists a constant n such that IP/O_n-chains criterion is stronger than the

required k-tuples criterion. However, it is shown that IP/O-chains criteria are incomparable with either the all-du-paths criterion or the data context coverage criteria.

Chapter 5 examines the incomparability between the IP/O-chains criteria and the all-du-paths criterion as well as the data context coverage criteria. Based on the results of the examinations, some useful variations of the IP/O_2-chains criterion, called IP/O_2-chains* criteria and IP/O_2-chains$^+$ criteria, are proposed. The former guarantees the all-du-paths coverage, whereas the latter guarantees the data context coverage.

In Chapters 3, 4, and 5, the underlying flow model is def-use graph which is only capable of modelling control and data dependencies within procedures (i.e., intra-procedural control and data flow). Control and data dependencies across procedure boundaries (i.e., inter-procedural control and data flow) have not been taken into account. This will affect the accuracy of the particular data flow oriented path selection criterion being used.

Chapter 6 investigates how to model both intra-procedural and inter-procedural control and data flow. An existing model is studied and shown not suitable for testing purposes. A new model is then proposed. It is derived from the def-use graph model and the super graph, but capable of modelling the intra-procedural data flow as well as the inter-procedural data flow in a program. As an example, we show how Pascal programs can be modelled using the proposed model. With this model, the data flow coverage criteria as described in Chapters 3, 4, 5 can be used to select more meaningful test paths.

Chapter 7 summarizes the thesis and points out directions for future research.

Chapter 2

Flow Model Based Software Testing: An Overview

In this chapter, we give an overview of flow model based testing. This includes a section on FSM testing, where an extensive survey of the existing FSM testing techniques is given, and another section on path selection criteria, where control flow oriented path selection is reviewed and necessity of data flow oriented path selection is discussed.

2.1 Introduction

Recall that control flow and/or data flow can be modeled by a variety of flow models. For the purpose of software testing, the following models are the most widely used: control flow graphs and finite state machines which respectively model only the control flow in a program and a specification; and def-use graphs which model the data flow as well as the control flow. Based on these flow models, various testing techniques have been developed. These techniques can be categorized as being either FSM testing or path selection criteria. The latter can be further classified as being control flow oriented or data flow oriented.

Section 2.2 will give an extensive review of the existing FSM testing techniques. Control flow oriented testing techniques are summarized in Section 2.3.1. A brief introduction to data flow oriented testing techniques is given in Section 2.3.2. Data flow oriented testing techniques will be discussed in detail in the remainder of the thesis as the need arises.

2.2 Finite State Machine Testing

Finite state machines have long been used in the development of both computer hardware and software. For instance, digital computers inevitably contains circuits known as sequential circuits which are capable of storing data as well as performing some logical, mathematical and other operations upon these data. The outputs of these circuits at any time are functions of the inputs and the stored data. The FSM model is the classic means of describing, designing, and testing sequential circuits [Henn 64, Gone 70, Koha 78].

Important applications of the FSM model can be also seen in the development of communications software, where much work of the design and validation of protocols has been done based on a variety of FSM models [Boch 78, West 78, Zafi 78, BoSu 80]. For example, a protocol can be modeled as an extended finite-state machine [Boch 78, BoSu 83] in which the control portion is specified by an FSM, and the data portion is specified by program segments.

FSM testing is concerned with checking whether an implementation of an FSM conforms to (or behaves in accordance with) the FSM representing a specification. Test sequences (i.e., sequences of inputs possibly interleaved with expected outputs) derived from a given FSM specification are used to determine the conformance of an implementation of the FSM. As specifications are becoming more and more complex, ad hoc methods for test sequence generation which suffer from being poor in error detection and efficiency can no longer serve the purpose. Formal methods [Henn 64, Gone 70, Chow 78, DaSh 79, Koha 78, NaTs 81, SaBo 84, KaLo 86, UyDa 86, SaDa 88, AhDa 88, ShLo 89, ChVu 89, YaUr 90a] have emerged to meet the new demands with the intention of generating a set of test sequences which, when applied to an implementation of the FSM, are capable of verifying all the states and the specified transitions. These methods are based on the so-called state-identification (SI) sequences of a given FSM. An SI

sequence for a state is an input/output (I/O) sequence which uniquely identifies the state and, thus, has the capability of revealing the unique behavior of the state. The known SI sequences are distinguishing sequences [Henn 64, Gone 70, DaSh 79], characterizing sequences [Chow 78], and unique input/output (UIO) sequences [SaDa 85, SaDa 88]. A **distinguishing sequence** is a sequence of inputs to which the response from distinct states are also distinct. A **characterizing sequence** set is a set of input sequences to each of which the response from each state is unique. A **UIO sequence** for a state is an input/output sequence which is unique to that state. For a state, a **minimum-length UIO sequence** is a UIO sequence which does not contain any other UIO sequence. The minimum-length UIO sequences for each state in Fig. 2.1 are given in Table 2.1. As it shows, there may be more than one minimum-length UIO sequences for a state. In this example, all states but state 2 have multiple minimum-length UIO sequences.

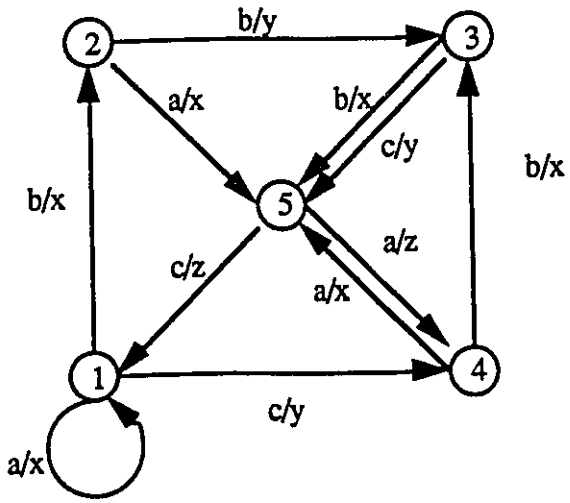


Figure 2.1. An Example FSM

Table 2.1. Minimum-Length UIO Sequences for Fig. 2.1

State	UIO
1	a/x, a/x a/x, b/x a/x, c/y b/x, a/x b/x, b/y c/y, a/x c/y, b/x
2	b/y
3	b/x, a/z b/x, c/z c/y, a/z c/y, c/z
4	b/x, b/x b/x, c/y
5	a/z c/z

Test generation methods using these SI sequences are known as D-method, W-method, and U-method [SiLe 89]. The so-called T-method [NaTs 81], which is described in Section 2.2.2 has its origin from software testing and does not make use of SI sequences. In the following section, we review these four types of methods and present a further classification.

2.2.1 The FSM Model and Its Graph Representation

An FSM can be represented by a labeled digraph G(V, E), where $V=\{v_1, ..., v_n\}$ is a finite set of vertices such that v_1 corresponds to the initial state and the other vertices correspond to the other states of the FSM. E is a set of edges corresponding to the specified state transitions in the FSM, where every edge $(v_i, v_j) \in E$ has a label i_{ij}/o_{ij} consisting of an input i_{ij} and an output o_{ij}.

To determine whether the behavior of an implementation of a given FSM is as specified by the FSM, a test sequence is derived from the digraph G representing the FSM. A **test sequence** is a sequence of inputs possibly interleaved with the expected outputs. The construction of a test sequence is based on the following procedure for verifying whether all the transitions in G are implemented correctly by the implementation of the correponding FSM. A transition $(v_i,v_j,i_{ij}/o_{ij})$ is verified by a 3-step procedure:

Step 1: The FSM implementation is brought to state v_i;

Step 2: Input i_{ij} is applied and the output is verified to be o_{ij};

Step 3: The new state of the FSM implementation is verified to be state v_j, using an SI sequence.

An input sequence that verifies a transition $(v_i,v_j,i_{ij}/o_{ij})$ is called a **test subsequence** for the transition and consists of an optional transfer sequence[1], the input i_{ij}, and an SI sequence for state v_j for realizing steps 1, 2, and 3 of the above procedure, respectively.

In order for an FSM to possess a certain type of SI sequences, i.e., every state has at least one SI sequence of the type, it is necessary to require the FSM be minimized and strongly connected. An FSM G(V, E) is said to be **minimized** iff any two states in the

[1] A transfer sequence is a shortest sequence of inputs that brings the implementation from its current state to state v_i.

FSM are distinguishable. G is said to be **strongly connected** iff for any pair of vertices v_i and v_j, there exists a path in G from v_i to v_j.

A test sequence for an FSM starts with the initial state v_1 consists of at least one occurrence of one test subsequence for each transition and, possibly, transfer sequences which may be necessary to jump from the end state of a test subsequence for one transition to the start state of the succeeding test subsequence for another transition.

2.2.2 Overview of FSM Based Testing Methods

T-Method

T-method [NaTs 81] assumes a strongly connected FSM which may not possess any type of SI sequences. It is a rather simple method which generates a transition tour (i.e., a path in G that starts at v_1, traverses every edge in G, and returns to v_1). On the other hand, although it is capable of detecting all output errors (i.e., errors related to not producing the expected output), it may not be capable of detecting transfer errors (i.e., errors caused by transfering to an incorrect state). One exception is when the FSM implementation has a hidden transition (i.e., status message [UyDa 86]) for each state. In this case, the method in [UyDa 86] can be used to derive an optimum test sequence[2] using a solution to the Chinese postman problem [Kuan 62], such as the one given in [EdJo 73].

D-Method

D-method [Gone 70, Koha 78, DaSh 79] assumes a minimal, strongly connected, completely specified FSM which possesses at least one distinguishing sequence. Every transition of an FSM implementation is verified by a test sequence which consists of test subsequences constructed as defined in section 2.2.1 using a distinguishing sequence as

---

2 A test sequence for an FSM is said to be an **optimum** test sequence if it is a minimum length test sequence.

the SI sequence. The problem with the D-method is that very few FSMs have distinguishing sequences. In this case, the following methods may be used.

W-Method

W-method [Chow 78] assumes a minimal, strongly connected, completely specified FSM possessing at least one set of characterizing sequences. Every transition of an FSM implementation is verified by a test sequence which consists of test subsequences constructed as defined in section 2.2.1 using a set of characterizing sequences as the SI sequence. The problem with W-method is that the test sequences it generates are often too lengthy, therefore executions of these test sequences are, if practical at all, very costly.

U-Method

Among the well-known test generation methods in the FSM testing literature, the U-method [SaDa 85, SaDa 88] is relatively new. UIO sequences have certain advantages over distinguishing sequences [AhDa 88]: the length of a UIO sequence is never more than that of a distinguishing sequence and nearly all FSMs have UIO sequences for each state while few have distinguishing sequences. It has been shown [SaDa 88, SiLe 89] that the use of UIO sequences as SI sequences yields shorter test sequences than those generated by the D- and W-methods and better error coverage than the T-method.

2.2.3 Optimization vs. Approximation

FSM-based test generation methods can be viewed as being twofold: In the first phase, one must determine and derive SI sequences for all states in the FSM. The second phase deals with the formation of the test sequence which consists of at least one occurrence of each transition and subjects to possible constraints from the first phase (i.e., the transition is immediately followed by an SI sequence for the end state of the transition).

The second phase of most known methods are ad hoc. This phase had not received much attention before Uyar, Dahbura [UyDa 86] and later Aho *et al.* [AhDa 88] noticed its importance and introduced their optimization techniques.

By this classification, T-method [NaTs 81] can be described as having only the second phase, thus there are no constraints, the resulting test sequence traverses each transition at least once.

For a given FSM and one or more SI sequences for every state in the FSM, the **optimum test sequence generation problem** (OTSP) [YaUr 90c, BoUr 90] is concerned with finding the optimum (minimum-length) test sequence for the FSM using these SI sequences. Recall that a test sequence for an FSM is said to be an optimum test sequence if it is a minimum length test sequence. The length of a test sequence is usually considered as the number of inputs it contains.

In graph-theoretic terms, OTSP can be defined as follows: given a digraph G(V, E) representing an FSM and, for each state, one or more SI subpaths which start from that node and correspond to a certain type of SI sequences for that state, find the minimum-length traversal of G starting with initial state v_1 and contains at least one occurrence of every transition $e=(v_i, v_j) \in E$ such that the edge is immediately followed by an SI subpath of v_i.

An optimum test sequence (with expected outputs) for Fig. 2.1 is: a/x b/x b/y b/x c/z c/y b/x c/y a/z b/x c/y a/z a/x c/z b/x a/x c/z. The corresponding path is: v_1, v_1, v_2, v_3, v_5, v_1, v_4, v_3, v_5, v_4, v_3, v_5, v_4, v_5, v_1, v_2, v_5, v_1.

The test sequence consists of 17 inputs, the third occurrence of input a is a transfer sequence from state v_5 to state v_4, and the second last input c and the last input b make up a transfer sequence from state v_1 to state v_2.

Shorter test sequences generally result in shorter execution time for the test sequences and, therefore, are desirable. However, it is proved [BoUr 90] that the OTSP is, NP-complete. Therefore, an efficient solution to the problem may not likely be expected in the general case. This is to say that it is unlikely that the length of the test sequence for an FSM can be minimized to the full extent in an efficient algorithmic manner.

As a special case where the FSM implementation being tested has a status message for each state, Uyar and Dahbura [UyDa 86] describe an optimization technique using the well-studied **Chinese Postman Problem** [Kuan 62], for which efficient algorithms exist [EdKa 72, EdJo 73]. However, the applicability of the technique is limited to FSM implementations possessing such messages.

Aho et al. [AhDa 88] proposed a more general approach by relaxing the requirements. A minimum-length UIO sequence is used for each state in the first phase to replace the special status message. An FSM G(V, E) is augmented by appending a set of new transitions E_c which correspond to the resulting test subsequences. The second phase of the test sequence generation is then reduced to finding a **Rural Chinese Postman Tour** (RCPT) in the graph G'(V', E') over E_c,where V'=V, E'=E $\cup$ E_c. An RCPT in G' over E_c is a minimum-length traversal of E_c in G'. Shen et al. [ShLo 89] further improved Aho's approach by using multiple minimum-length UIO sequences for each state. The result is a reduction between 4% to 37% in the length of the test sequences required in [AhDa 88] with no noticeable increase in the time required to generate the test sequence.

It is important to point out that the both Aho's [AhDa 88] and Shen's [ShLo 89] are not truly optimization methods, rather they are approximation methods. The way the new transitions are constructed implies that the test subsequences which the new transitions represent are not overlappable. This is illustrated as follows: for transitions (v1, v1, a/x) and (v1, v2, b/x) in Fig. 2.1, the test subsequences a/x b/x b/y and b/x b/y can be

overlapped as a/x b/x b/y, therefore the length is reduced by 2. If a transition representing the latter is used instead of constructing two transitions for the former, the resulting test sequence can be further reduced by a length of at least 2.

Noticing the differences between optimization and approximation in the context of FSM-based test generation methods, we classify these methods as either heuristic (approximation) or optimal (optimization). Thus, the methods in [AhDa 88, ShLo 89] are approximation methods, whereas the method in [UyDa 86] is an optimization method.

Although the OTSP is NP-complete, further reductions in the length of a test sequence can be achieved by overlapping test subsequences. [Chen 90], [MiPa 90], [YaUr 90a] describe heuristic methods for reducing the length of test sequences by overlapping test subsequences obtained using single or multiple minimum-length UIO sequences.

2.3 Path Selection Criteria

The most widely used systematic testing methods are path selection criteria employed in white-box testing methods [Howd 87]. A test path selection criterion is satisfied if certain aspects of a program have been exercised at least once by the corresponding selected test paths. Different criteria correspond to different aspects of the program's source code. Two kinds of path selection criteria can be identified by their orientation towards the control flow and data flow aspects of a given program.

2.3.1 Control Flow Oriented Path Selection

Statement coverage (node coverage) is the most intuitive control flow oriented path selection criterion which requires the selection of a set of paths to cover all the statements in a given program. An obvious shortcoming of statement coverage is its failure to test alternative outcomes of branching conditions. For example, the coverage of an IF-THEN statement is satisfied when the THEN branch is taken which leaves the implied ELSE branch not covered.

Branch coverage is a refinement of statement coverage which requires that all branches in a given flow model be tested at least once. For example, branch coverage requires both the THEN branch and the implied ELSE branch of an IF-THEN statement be covered. Branch coverage is considered to be a minimum requirement for structural testing. There are several variants of branch coverage based on how a branch is defined in the flow model [Myer 79]. In an FSM, a branch (edge) represents a transition. Therefore, the T-method guarantees the branch coverage. Here, a branch is defined to be any transfer of control flow from one program statement to another. In statement-oriented languages, there is an implicit branch from one program statement to the next unless the first statement is a "go to" statement which causes an explicit branch to some statement out of sequence. The explicit branches are those that are commonly caused by conditional statements. There are

also implicit branches in structured statements such as repetitive statements which are based on conditions related to these statements.

Path coverage criterion requires every path in a flow model be covered as least once by some test path. However, a program containing a loop may have infinite number of paths, making it impractical in reality.

Most of the other control flow oriented path selection crieria fill the gap between the branch coverage and path coverage. These criteria include structured path testing [Howd 77], boundary-interior path testing [Howd 75], and linear code sequence and jump (LCSAJ) based criteria [HeWo 76, WoHe 80]. The original **LCSAJ criterion** [HeWo 76] requires each LCSAJ be exercised at least once by the selected set of test paths. An LCSAJ is a sequence of consecutive statements in a given program, starting at an entry point or after a jump and terminating with a jump or an exit point. Thus, an LCSAJ is represented by a triple (a start point, an end point, and a jump-to point). This criterion is later extended into a family of test path selection criteria [WoHe 80] by defining a class of test effectiveness ratios (TER_n) which corresponds to a test path selection criterion that requires $TER_n = 1$. Accordingly, TER_1 and TER_2 are equal to 1 if statement and branch coverage are achieved, respectively. $TER_{n+2} = 1$ implies that all subpaths containing combinations of up to n LCSAJs are covered.

In **structured path testing** [Howd 77], a limited number of paths are selected by restricting the number of iterations of each loop traversed by a path. In **boundary-interior path testing** [Howd 75], the number of paths are limited by grouping together paths that differ only in the number of times that they iterate loops and then selecting only a few representative paths from each group. Paths in each group are categorized, with respect to each loop, in two classes; those that enter the loop but do not iterate it (i.e., boundary test paths) and those that iterate the loop at least once (i.e., interior test paths). Among the

boundary test paths, those that follow different subpaths in the loop are selected. Among the interior test paths, those that follow different subpaths during the first iteration of the loop are selected. Note that both structured path testing and boundary-interior testing are equal to path coverage for a program that does not contain any loops.

Unfortunately, even if coverage is limited to the testing of all paths which cause only one or two iterations of a loop, there will often still be an enormous number of paths to test for some programs. On the other hand, even when a program is simple enough to have a small number of paths which do not traverse loops more than once or twice, it is often not enough just to test all such paths. There are always errors which are associated with more complex patterns of loop iterations such as " traverse the outer loop three times and the inner loop twice. On the first traversal of the outer loop follow the true-true path through the inner loop conditions on the first traversal of the inner loop, and the false-false path on the second traversal of the inner loop. On the second traversal of the outer loop reverse the traversal pattern on the inner loop and on the last traversal of the outer loop go back to the original traversal pattern for the inner loop." [Howd 87]. Patterns of loop traversal like this may seem to correspond to various functional cases in the data, but there are so many such patterns that the idea of cataloging them and using them to specify which paths to test is impractical.

A major problem with path selection criteria is the existence of infeasible paths. An **infeasible path** is a path through a program which is never traversed for any input data. This is because the boolean conditions that would have to be satisfied to cause this path to be followed are contradictory. Such paths are not necessarily caused by program errors.

If the percentage of infeasible paths in a program is very high, and if there were some way of automatically eliminating them, then path coverage might become practical. In a study of 6 of the NAG Algol 68 library of numerical analysis routines [WoHe 80], it was

found that an average of only 18 out of the first 1000 shortest paths through the programs were feasible. These statistics are very promising, indicating that not only path coverage but other testing and analysis methods which rely on looking at program paths may be able to avoid the path explosion problem. Techniques for symbolic evaluation [Clar 76] and heuristic approaches [KrSm 73, Oste 77, FrWe 88] may be used effectively to eliminate many of the infeasible paths.

A related problem is measuring the coverage achieved by a set of selected test cases. This problem is overcome by program instrumentation. For example, branch coverage tools have been used for a long time and have been implemented for variety of languages. Such a tool involves several components: the branch analyzer which analyzes the source program to produce a branch model and an instrumented program [Huan 75]. The instrumented program contains probes which count the number of times program branches are traversed. Several optimal probe insertion algorithms, e.g., [Prob 81], have been proposed to minimize the number of probes required. The tester runs the instrumented program against a set of test cases, and contains the branch-traversal table which is updated by the probes. The report generator uses the information in the branch-traversal table, along with the branch model, to construct a complete picture of branch coverage.

2.3.2 Data Flow Oriented Path Selection

Control flow oriented path selection criteria have not been entirely successful in attempting to extend branch coverage to larger program components which are closer to the program functionality. Another class of test coverage criteria, based on data flow coverage, is more promising [Howd 87]. The basic idea is that statements which have a data flow relationship should be tested together by the same test paths.

The functional interpretation of data flow coverage testing is that statements which have a data flow relationship are probably part of the same embedded program function and

should be tested together at least once. In this interpretation, data flow coverage testing schemes correspond to primitive attempts to extract functional patterns from the program code. Three types of data flow coverage criteria will be described in the subsequent chapters.

Chapter 3

Intra-Procedural Data Flow Testing

In this chapter, first we review some terminology necessary for our discussion, and describe the def-use graph model which can be used to model both control flow and intra-procedural data flow in a procedure. Then, as an example, we show how essential Pascal programming constructs can be modelled as components of the def-use graph. Finally, a literature survey of the well-known data flow oriented path selection criteria is conducted and possible improvements of these criteria, which are among the motivations for this thesis, are also pointed out.

It should be noted that all of the discussions in this and the following two chapters are in the context of procedure-level testing. Testing of a higher level program components will be discussed in Chapter 6.

3.1 Background

Procedure-level data flow testing is based on intra-procedural data flow analysis [Hech 77] which focuses on where variables are bound to values, where and how these variables are referenced.

In data flow testing, occurrences of variables in a procedure are classified as definitions and uses. A variable occurrence is called a **definition** (def) if it the value stored in its corresponding memory unit gets updated, otherwise it is called a **use**. A use of a variable can be further classified as a computational use (c-use) or a predicate use (p-use) [RaWe 85]. A **c-use** of variable x is a use of x which directly affects the computation being performed (e.g., an occurrence of x on the RHS of an assignment statement) or allows one to see the result of some earlier defs (e.g., an occurrence of x in the variable list of an output statement). A **p-use** of x is a use of x which directly affects the control flow in the procedure (e.g., an occurrence of x in the predicate portion of a conditional statement).

The basis of data flow testing is to regard each statement that contains a def of a variable as a potential source of erroneous values which could reach subsequent uses of the variable. That is, errors in a procedure may lead to incorrect values and, ultimately, as the result of propagation through associations between defs and uses , an erroneous value may show up at the procedure's output. Rather than selecting test paths solely based on the control flow in a procedure, data flow testing focuses on the flow of data by tracking variables through a procedure, and following them as they are modified, until they are ultimately used to compute values for other variables. Thus, data flow testing attempts to be more selective in its choice of paths, and at the same time provides a more diverse test of the procedure than control flow testing. In fact, from the viewpoint of testing the implemented functions in a procedure, data flow oriented path selection is perhaps more

suitable than selection solely based on the control flow because the former identifies data dependencies and thus implicitly requires testing of functional segments [Howd 87].

A number of data flow oriented path selection criteria have been proposed in recent years [Herm 76, Lask 82, Lako 83, Ntaf 84, RaWe 85, UrYa 88, UrYa 91]. The complexity of some of these criteria have been studied in [Weyu 84, Ntaf 88, Yang 88]. Comparisons among these and other criteria have been made in [WeGa 85, ClPo 85, Ntaf 88, Yang 88, ClPo 89, YaUr 89].

The existing data flow oriented path selection criteria differ from each other in terms of the degree that du-pairs in a procedure are covered by the selected test paths. Informally, a du-pair is a def of a variable v and a use of v such that the def of v reaches the use of v through a path without encountering another def of v. In [Herm 76], each du-pair is required be covered at least once. However, since this criterion does not differentiate between c-uses and p-uses of variables, it does not guarantee the branch coverage, which is considered as the minimum requirement for structural coverage. Rapps and Weyuker [RaWe 85] have proposed a family of criteria based on the classification of variable uses as c-uses and p-uses. The strongest (and also the often cited) two criteria in the family explicitly require each du-pair be covered at least once and implicitly guarantee the branch coverage.

The above mentioned data flow oriented path selection criteria view individual du-pairs as test units. In contrast, [Lask 82, Lako 83, Ntaf 84, UrYa 88, UrYa 91] take more meaningful combinations of du-pairs into consideration. In [LaKo 83], du-pairs whose uses occur in the same statement are combined as a test unit. In [Ntaf 84], a tester-determined number of du-pairs whose alternating defs and uses form a path is considered as a test unit. In [UrYa 88, UrYa 91], du-pairs whose alternating defs and uses form a path from a procedure input to a procedure output is considered as a test unit.

In order to precisely describe the data flow oriented path selection criteria and compare their relative strength, it is necessary to have a common data flow model. Such a model will be defined in the next section. It allows description of data flow testing methods and their implementations be independent of the language in which the procedure under test is written. It also provides a unified view of the existing methods, making comparisons among these methods easier and more accurate.

3.2 The Def-Use Graph Model

Without loss of generality, we consider procedures with a single entry and single exit. For our purpose, a procedure is viewed as a digraph *G(V, E)* called **def-use graph**, where *V* is a set of nodes, each representing the data flow (i.e., defs and c-uses) within a simple statement and *E* is a set of edges, each representing the control flow between a pair of nodes.

In *G*, the procedure entry and exit are represented by two special nodes *s* and *t*, called **entry node** and **exit node**, respectively. *s* has no predecessor whereas *t* has no successor. All other nodes have one or more predecessors and successors. Each outgoing edge of a node with multiple successors is associated with p-uses which correspond to a branching condition (i.e., Boolean expression) that complements the branching condition whose p-uses are associated with the other outgoing edge(s) of the node. An example def-use graph is given in Fig. 3.1, where d(x,y) stands for defs of variables x and y, c(x,y) for c-uses of x and y, p(x,y) for p-uses of x and y, and c(x); d(x) for c-use of x followed by a def of x, and so on.

```
BEGIN
READ (x, y)
WHILE r(x, y) DO
  z := f(x, y)
  IF s(z)
    THEN x := g(x)
    ELSE x := h(x)
END_WHILE
WRITE (z)
END;
```

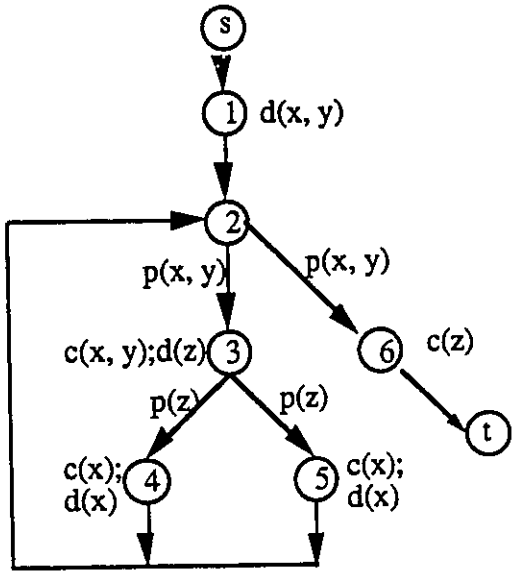


Figure 3.1. Procedure A and Its Def-Use Graph

Defs and c-uses in a node corresponding to a simple statement are ordered in the sequence of their occurrence. For example, in node 3 of Figure 3.1, c-uses and def are ordered as c(x,y) followed by d(z).

As a notational convenience, one can obtain a compact version of a def-use graph by collapsing a sequence of nodes into a single node which represents a **statement block** having the following two properties: a) whenever the first statement is executed, the rest is subsequently executed once in the given order; b) the first statement is the only statement which may be executed directly after the execution of the last statement in a block. In most of the figures in this thesis, we represent a given procedure by a compact def-use graph.

In the next section, we detail how defs and uses of variables are identified in a Pascal procedure and how the procedure is represented by a def-use graph. The set of translation rules is an enhanced version of that of Frankl and Weyuker given in [FrWe 88].

3.3 Representing Pascal Procedures with Def-Use Graphs

In the previous section, we have presented the def-use graph model which abstracts both the control flow and the data flow in a procedure. In this section, we illustrate how procedures written in a large subset of ISO Standard Pascal [ISO 82] can be translated into def-use graphs. Pascal is chosen because it is one of the computer programming languages used throughout the world [JeWi 85].

For simplicity a Pascal procedure, which can be a main program, a procedure, or a function, is assumed to have no:

a) with statement;

b) variant records;

c) functions having var parameters;

d) procedural or functional parameters;

e) conformant arrays;

f) goto statement.

It should be noted that it is not difficult to relax these assumptions. a), b), d), and e) are used to write shorter procedures which are, therefore, not essential constructs of Pascal language. For example, with statements and variant records can be eliminated by replacing them with using complete field identifiers and defining several record types different in only the variant fields, respectively. In case of c), such a function can be implemented by a procedure having var parameters. We do not encourage the use of goto statements, because they tend to make the procedure difficult to understand and analyze, and the resulting def-use graph ill-structured.

A variable in Pascal may be a simple variable (including set variables), a structured variable (i.e., array variable, record variable, or file variable), or a pointer variable. As

illustrated in Section 3.2, it is straightforward to interpret an occurrence of a simple variable as a def, c-use, or p-use. In this thesis, unless otherwise stated, the interpretation rules of defs and uses of non-simple variables is based on the convention shown in Table 3.1a which interprets each occurrence of non-simple variables in a Pascal procedure. In Tables 3.1 - 3.4, d(x), c(x), p(x) denote def, c-use, and p-use of a variable x, respectively. Note that u(x) stands for either c(x) or p(x), and e_i stands for an expression.

Table 3.1a. Defs & Uses of Non-simple Variables

VARIABLE TYPE	INTERPRETATION OF ITS DEF	INTERPRETATION OF ITS USE
Array: $a[e_1, ..., e_n]$	c(variables in $e_1, ..., e_n$); d(a)	u(variables in $e_1, ..., e_n$); u(a)
File: f	d(f↑)	u(f↑)
Record: r	if r is qualified d(field) if r is unqualified d(all fields)[1]	if r is qualified u(field) if r is unqualified u(all fields)[1]
Pointer: v	c(v), d(v↑)	u(v); u(v↑)

Let e and e_i, $1 \le i \le n$, be expressions. Function $f(e_1, ..., e_n)$ is interpreted as c-uses or p-uses of each variable occurring in the parameter list $(e_1, ..., e_n)$ depending on the expression e in which the function is called. It is assumed that no variables in $(e_1, ..., e_n)$ are updated in f.

Moreover, an occurrence of a predefined procedure is interpreted according to Table 3.1b.

[1] When a record name appears in a procedure or fuction call, then only those fields of the record that are referred to in the definition of the called procedure or function are classified as uses & defs.

Table 3.1b. Interpretation of Predefined Procedures

PROCEDURE	INTERPRETATION
Pointer Type: new(P, c_1, ..., c_n)	c(P, c_1, ..., c_n); d(P↑) We assume that the call is not in a loop.
File: get(f) put(f) reset(f) rewrite(f)	 c(f); d(f↑); d(eof) c(f↑); d(f); c(f↑); d(eof) d(eof, f↑) d(f, eof)
Input/Output: read(f, v_1, ..., v_n) write(f, v_1,..., v_n)	 c(f↑); d(v_1, ..., v_n); c(f); d(f↑); d(eof) c(v_1, ..., v_n); d(f↑); c(f↑); d(f); c(f↑); d(eof)

In Pascal, a statement is either a simple statement (i.e., empty statement, assignment statement, procedure statement, goto statement) or a structured statement (i.e., compound statement, conditional statement, repetitive statement). The translation rules from simple statements, conditional statements, and repetitive statements to subgraphs of the def-use graph are given in Tables 3.2 - 3.4, respectively. Note that in Table 3.3 and Table 3.4, statements S, S1, S2, ..., Sn are subject to the same rules as given in Tables 3.1-3.4.

Table 3.2. Translation Rules for Simple Statements

STATEMENT	DEF-USE GRAPH	ASSOCIATED DEFS & USES
Procedure Entry: begin		defs of all non-local variables, defs of all input parameters, def of input buffer input↑
Procedure Exit: end;		c-uses of all non-local variables, c-uses of all VAR parameters, c-use of input buffer input↑
Empty Statement: ;		None
Assignment Statement: v := e; 1. e contains no variables; 2. e contains variables x_1, ..., x_m;		1. d(v) 2. c(x_1, ..., x_m); d(v) Refer to Table 4.1 in cases that non-simple variables are involved.
Procedure Statement: P(e_1, ..., e_n);		c-uses of each variable occurring in the parameter list (e_1, ..., e_n), followed by defs of each variable that corresponds to a var formal parameter.

Table 3.3. Translation Rules for Conditional Statements

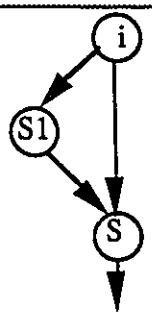
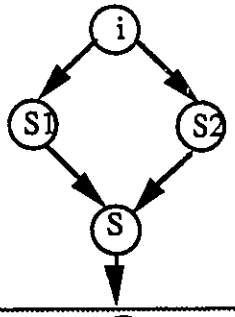
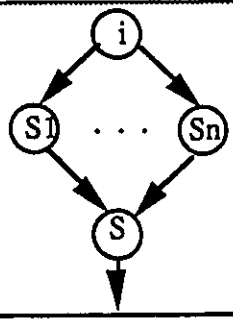
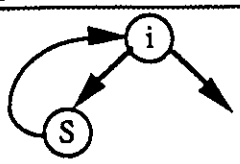
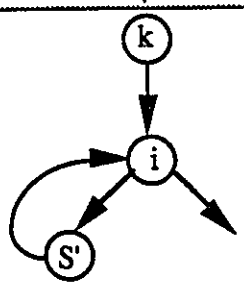
STATEMENT	DEF-USE GRAPH	ASSOCIATED DEFS & USES
If Statement: 1. if B then S1; S;		Let j, k, and m be the entry nodes of S, S1, and S2, respectively. Then, 1. edges (i, j) & (i, k) are associated with p-uses of each variable in the Boolean expression B.
2. if B then S1 else S2; S;		2. edges (i, k) & (i, m) are associated with p-uses of each variable in the Boolean expression B.
Case Statement: case e of constant_list1: S1; . . . constant_listn: Sn end; S;		Let v1, ..., vn be the entry node of S1, ..., Sn, respectively. Then, edges (i, v1), ..., (i, vn) are associated with p-uses of each variable in expression e.

Table 3.4. Translation Rules for Repetitive Statements

STATEMENT	DEF-USE GRAPH	ASSOCIATED DEFS & USES
While Statement: while B do S;		Let in and out be the loop entry (i.e., the entry node of S) and loop exit, respectively. Then, edges (i, in) & (i, out) are associated with p-uses of each variable in the Boolean expression B.
Repeat Statement: repeat S1; ...; Sn until B;		Let in and out be the loop entry (i.e., the entry node of S1) and loop exit, respectively. Then, edges (i, in) & (i, out) are associated with p-uses of each variable in the Boolean expression B.

For Statement:
for v:=e1 to e2 do S;
for v:=e1 downto e2 do S;
		Let S' contain S and c-use of v followed by a def of v, and in and out be the loop entry (i.e., the entry node of S) and loop exit, respectively. Then, edges (i, in) & (i, out) are associated with a p-use of v. And node k consists of c-uses of each variable in the ordinal-type expressions e1 and e2, followed by a def of v.

In the following section, we use the translation rules and the def-use graph model, both given in this section, to describe the well-known data flow oriented path selection criteria.

3.4 Literature Review and Critique

In this section, the well-known data flow oriented test path selection criteria, namely Rapps and Weyuker's ail-uses and all-du-paths criteria [RaWe 85], Laski and Korel's data context coverage criteria [LaKo 83], and Ntafos's required k-tuples criteria [Ntaf 84], are defined and evaluated. In their definitions the following terminology that is based on the def-use graph model are used.

Given a def-use graph, a **path** is a sequence of nodes $(n_1, n_2, ..., n_k)$, such that there is an edge from n_i to n_{i+1} for i = 1,2, ..., k-1. A path is a **simple path** if all nodes, except possibly the first and last, are distinct. A path is a **loop-free path** if all nodes are distinct. A **complete path** is a path from the entry node to the exit node.

Let P be a set of complete paths for a def-use graph of a given procedure, we say that **a node i is included** in (or covered by) P if P contains a complete path $(n_1, ..., n_m)$ such that $i = n_j$ for some j, $1 \leq j \leq m$. Similarly, an **edge** (i_1, i_2) **is included** in (or covered by) P if P contains a complete path $(n_1, ..., n_m)$ such that $i_1=n_j$ and $i_2=n_{j+1}$ for some j, $1 \leq j \leq m-1$. A **path** $(i_1, ..., i_k)$ **is included** in (or covered by) P if P contains a complete path $(n_1, ..., n_m)$ and $i_1=n_j$ and $i_2=n_{j+1}$, ..., $i_k=n_{j+k-1}$ for some j, $1 \leq j \leq m-k+1$. Equivalently, we say that a path p1 is covered by a path p2 if p1 is a subpath of p2.

As stated earlier, the existing data flow oriented path selection criteria differ from each other in terms of the degree that du-pairs in a procedure are covered by the selected test paths. Formally, a **du-pair** is a tuple (d(v), u(v)), where v is a variable, d(v) is a def of v, and u(v) is a use of v such that d(v) is **live** at u(v), meaning that d(v) reaches u(v) through a path without encountering another definition of v. A path $(i, n_1, ..., n_m, j)$, $m \geq 0$, containing no definitions of x in nodes $n_1, ..., n_m$ is called a **def-clear path** with respect to x from node i to node j or from node i to edge (n_m,j).

In [RaWe 85], a family of path selection criteria was proposed based on the classification of variable uses as c-uses and p-uses as well as the representation of p-uses along edges in a def-use graph. Among them, **all-uses** criterion requires that every du-pair in a given procedure be covered at least once. This criterion assumes that the program under test has at least one p-use and there are no syntactic undefined p-uses. Formally, a set of complete paths P satisfies the all-uses criterion if P includes an activating path for each du-pair in a def-use graph. An activating path for

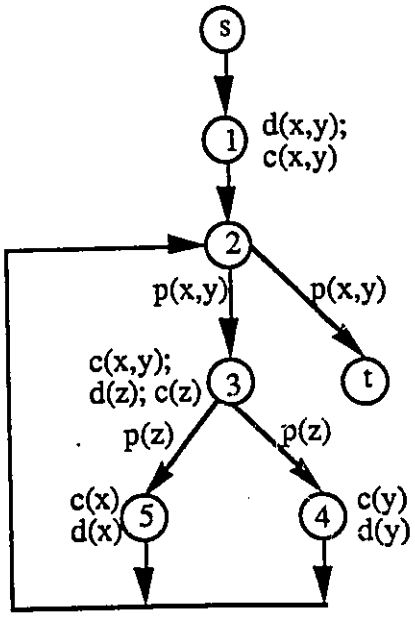
a) du-pair $(d^x_{n_1}, c^x_{n_m})$, a shorthand for d(x) at node n_1 and c(x) at node n_m, is a def-clear path wrt x from n_1 to n_m,

b) du-pair $(d^x_{n_1}, p^x_{n_{m-1},n_m})$, a shorthand for d(x) at node n_1 and p(x) on edge (n_{m-1},n_m), is a def-clear path wrt x from n_1 to (n_{m-1},n_m).

The strongest criterion in the family is **all-du-paths** criterion which requires all the du-paths for each du-pair in the def-use graph be covered at least once. A path $(n_1, n_2,..., n_j, n_k)$, is a **du-path** wrt a variable x if n_1 has a def of x and either 1) n_k has a c-use of x, such that $(d^x_{n_1}, c^x_{n_k})$ is a du-pair and $(n_1, ..., n_k)$ is a def-clear simple path wrt x, or 2) (n_j, n_k) has a p-use of x, such that $(d^x_{n_1}, p^x_{n_j,n_k})$ is a du-pair and $(n_1, ..., n_k)$ is a def-clear loop-free path wrt x. Formally, a set of complete paths P satisfies the all-du-paths criterion if P includes every du-path for each du-pair in a def-use graph.

Both the all-uses and the all-du-paths criteria guarantee the branch coverage. Moreover, since each du-path is either a simple path or a loop-free path, all-du-paths criterion sets a bound on the number of iterations for each loop. All-du-paths criterion differs from all-uses criterion when there exists more than one activating path for a du-pair: all-du-paths criterion requires that all those that are simple or loop-free be covered whereas

all-uses criterion requires only one of these paths be covered. For example, in the procedure given in Fig. 3.2, a path that starts and ends at node 5 containing possibly any number of occurrences of the subpath (2,3,4) but not containing any other occurrence of node 5 is an activating path for the du-pair (d_5^x, c_5^x). On the other hand, the path (5,2,3,5) is a du-path for the same du-pair.




```
begin
read(x ,y);
write(x,y);
while p1(x,y) do
  z := f(x,y);
  write(z);
  if p2(z)
    then x:= g(x)
     else y:= h(y)
end_while
end;
```

Figure 3.2. Procedure B and Its Def-Use Graph

Other criteria in the family are *all-p-uses/some-c-uses, all-c-uses/some-p-uses, all-defs,* and *all-p-uses.* They are "weaker" variations of all-uses criterion and less costly to apply, because less test paths are required. A problem with the criteria proposed in [RaWe 85] is that the associations between definitions and uses of variables are viewed as individual du-pairs. Since no relationship among du-pairs is explored, these criteria provide very little (if any) information about any meaningful computation, not to mention the intended functionality of the procedure under test.

Data context coverage criteria [LaKo 83] group du-pairs whose uses occur in the same node. The du-pairs in each group that can be formed by following a path which reaches the definitions in the group in some order are called an **elementary data context**. Data context coverage then requires that all elementary data contexts in a procedure be tested at least once. An **elementary data context** in node i that uses a vector of variables $X(i) = [x_1, x_2, ..., x_n]$ is a set of definitions $ec(i) = [d(x_1), d(x_2), ..., d(x_n)]$ of all variables from X(i) such that there exists a path from the entry node to node i and all defs from ec(i) to node i through this path without redefining the variables in X(i) [LaKo 83].

Formally, a set of complete paths P satisfies the data context coverage criterion, if P includes an activating path for every elementary data context in each node in the def-use graph of a procedure. A path $(n_1, p_1, n_2, p_2, ..., p_m, n_m)$ is an activating path for an elementary data context in node i, $[d_{n_1}^{x_1}, d_{n_2}^{x_2}, ..., d_{n_m}^{x_m}]$, if path $(n_i, p_i, n_{i+1}, ..., p_m, n_m)$ is a def-clear path wrt variable x_i, where $1 \leq i \leq m-1$.

As an extension to data context coverage criterion, **ordered context coverage** criterion imposes an ordering on data contexts by which defs in a data context are reached by its activating path, and further asks that all data contexts with distinct order be tested at least once. The major shortcoming of these criteria is that although being grouped (by uses at each statement), du-pairs are not "linked" as chains in order to relate to the implemented computations in a procedure. Moreover, in (ordered) context coverage criterion, du-pairs in each group may be overlapped in the sense that there exists a du-pair in the group such that coverage of this du-pair guarantees coverage of any other du-pair in the group. Furthermore, context and ordered context coverage do not guarantee branch coverage [ClPo 89].

Required k-tuples criteria [Ntaf 84] ask that all chains of k-1 (or less) related du-pairs be covered at least once. Each such chain is called a **k-dr interaction.** A chain of du-pairs is said to be **related** if the use in each but the last du-pair is a direct use in defining the variable associated with the succeeding du-pair. Thus, a **k-dr interaction** ($k \geq 2$) is a chain $[d_1^{x1}, u_2^{x1}d_2^{x2}, u_3^{x2}..., ...d_{k-1}^{xk-1}, u_k^{xk-1}]$ which is associated with k distinct nodes 1, 2, 3, ..., k, where for all i, $1 \leq i \leq k$, the ith def $d\backslash S(x_i,i)$ at node n_i reaches the ith use u_{i+1}^{xi} at node i+1 through a def-clear path wrt x_i [Ntaf 84]. If the last use is a p-use, the k-dr interaction is covered twice, once for each outcome of a predicate in which p-use occurs, to ensure branch coverage. If the first def or the last use of a k-dr interaction occurs in a loop, two test paths are generated for that interaction, reflecting two iteration counts for the loop, one contains the minimum iteration of the loop [Ntaf 84], the other contains a larger number of iterations [Ntaf 84].

Formally, a set of complete paths P satisfies the required k-tuples criterion for a given k, if P includes an activating path for every k-dr interaction in the def-use graph of a procedure. An activating path for the k-dr interaction $[d_1^{x1}, u_2^{x1}d_2^{x2}, u_3^{x2}..., ...d_{k-1}^{xk-1}, u_k^{xk-1}]$ is a path $(n_1, p_1, n_2, p_2, n_3,..., n_{k-1}, p_{k-1}, n_k)$ in which (n_i, p_i, n_{i+1}) is def-clear path wrt variable x_i, $1 \leq i \leq k-1$.

A major shortcoming of the required k-tuples criteria is that although they group related du-pairs into chains of "length" k (or less), it lacks justification of the meaningfulness of k. Suppose that a chain of 99 related du-pairs (a 100-dr interaction) is important and therefore need to be tested, then all chains which consist of related du-pairs of length 1 to 99 must be identified and tested. i.e., the required 100-tuples is a must. Another problem is that the first def and the last variable use in each chain are treated the same way as other defs or uses in the chain.

It can be argued that none of the criteria mention above makes an explicit effort to construct structural units that relate to functional segments implemented in a given procedure. In the following chapter, IP/O-chains criteria are introduced. These criteria are intended to identify def-use chains in a procedure which begin with procedure inputs and terminate with either procedure outputs or predicates for the purpose of exposing and subsequently covering implemented procedure functionality with test paths.

Chapter 4

IP/O-Chains Criteria

In this chapter, we introduce the IP/O-chains criteria. These criteria are based on input-predicate and input-output relations. The design objectives of the criteria are to select test paths using the two relations while, at the same time, guarantee what we consider the minimum control flow and data flow coverage.

The relative strength of the IP/O-chains criteria is determined by comparing them to the well-known data flow oriented path selection criteria. The results show that the IP/O-chains criteria guarantee the all-uses criterion. Furthermore, we prove that for any given constant k, there exists an IP/O_n-chains criterion which is stronger than the required k-tuples criterion. However, it is shown that IP/O-chains criteria are incomparable with either the all-du-paths criterion or the context coverage criteria.

4.1 Motivation

Before defining the IP/O-chains criteria, we shall detail the motivation behind the criteria by introducing the notion of *affect*.

It is observed that du-pairs in a procedure may be linked to form sequences of du-pairs (i.e., du-chains) in which the def in a du-pair is defined in terms of the use in the preceding du-pair. Each such chain may be viewed as a sequence of computations which captures particular control and data dependencies [Howd 87, Kore 87] in a procedure. This gives rise to the following notion.

A use of a variable y, u(y), which may be either a c-use or p-use, is **affected** by a def of a variable x, d(x), if either

a) x = y and the d(x) is live at the u(y); or

b) a d(z) is given in terms of a u(x) at which d(x) is live and the u(y) is affected by the d(z).

```
begin
read (x)
while p(x) do
     z:=f(x)
     if q(z)
       then x := g(x)
       else  x := h(x)
     write(z)
end_while
end;
```

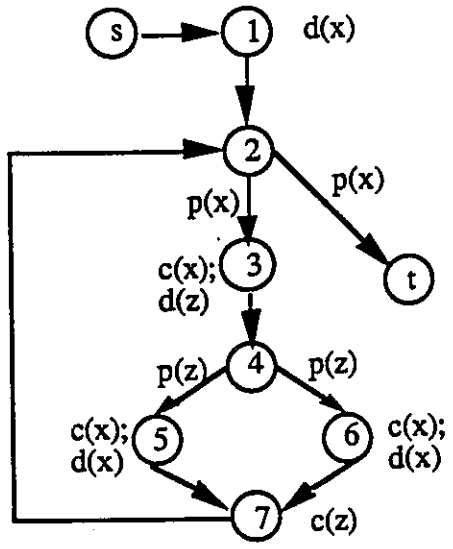


Figure 4.1 Procedure C and Its Def-Use Graph

Therefore, c_3^x and $p_{2,3}^x$ in Fig.4.1 are both affected by d_1^x. Recall that, except in figures, d_n^x, c_n^x or $p_{m,n}^x$ denote def, c-use and p-use of variable x in node n or edge (m,n), respectively. For ease of reference, u_q^x denotes either c_q^x, if q represents a node, or p_q^x, if q represents an edge.

It is important to note that we will use a more accurate interpretation of variable occurrences in the parameter list of a procedure statement than the one used in the def-use graph model. Recall that in the def-use graph model, it is assumed that a procedure statement contains c-uses of variables occurring in the parameter list followed by defs of var (i.e., pass-by-reference) parameters. For our purposes, we assume the existence of an abstract definition of each procedure called within the procedure represented by a def-use graph. In the abstract definition of a procedure, the manner each var parameter takes a value within the procedure can be determined. That is, for each var parameter v, there exists an assignment statement "v := expression" where expression is an n-ary function defined over the variables in the parameter list. The same applies to each nonlocal variable that is updated within the procedure. Thus, the variable occurrences in a procedure statement in a def-use graph are interpreted by first substituting the actual parameters for the formal parameters in the assignment statements in the abstract definition of the called procedure, and then applying the interpretation rules used for an assignment statement in the def-use graph model.

With the above definitions and interpretations, we can define a **du-chain** to be a sequence $[d_{n_1}^{x1}, u_{n_2}^{x1} d_{n_2}^{x2}, u_{n_3}^{x2}..., ... d_{n_m}^{xm}, u_{n_{m+1}}^{xm}]$ of du-pairs $(d_{n_1}^{x1}, u_{n_2}^{x1})$, $(d_{n_2}^{x2}, u_{n_3}^{x2})$,..., $(d_{n_m}^{xm}, u_{n_{m+1}}^{xm})$, where $m \geq 1$ and nodes $n_1, n_2, ..., n_{m+1}$ are not necessarily distinct, such that $d_{n_1}^{x1}$ affects $u_{n_{m+1}}^{xm}$. For convenience, when $u_{n_2}^{x1} d_{n_2}^{x2}$, $u_{n_3}^{x2}$...,... $d_{n_m}^{xm}$ need not be given explicitly, we denote a du-chain from a def d_p^x to a use u_q^x as **duc**(d_p^x, u_q^x).

It follows from the definitions of affect and du-chain that many du-chains may be contained in a du-chain duc ($d_{n_1}^{x1}$, $u_{n_{m+1}}^{xm}$):

e.g., duc($d_{n_1}^{x1}$, $u_{n_2}^{x1}$), duc($d_{n_1}^{x1}$, $u_{n_3}^{x2}$), and duc($d_{n_2}^{x2}$, $u_{n_{m+1}}^{xm}$).

We are interested in du-chains which begin with procedure inputs. An **input** of a procedure is the def of a variable which is not given in terms of any other variables. The motivation here is to trace the effects of procedure inputs on p-uses and c-uses through du-chains. Obviously, the variables referred to in input statements are considered as inputs. In addition, variable initializations through assignment statements (whose RHS is an expression consisting of only constants) are also considered as inputs since a) such initializations may be important for capturing the procedure functionality and b) automatic evaluation of the importance of a variable initialization is in general impractical.

Furthermore, we focus on du-chains that start with procedure inputs and terminate with either p-uses or with procedure outputs. The motivation here is to identify the procedure functionality in terms of the effects of procedure inputs on predicates or procedure outputs. In general, an **output** of a procedure is a c-use of a variable in an output statement of the procedure. We call such an output a **variable output**. A variable output O is said to be affected by a procedure input I if there is a du-chain that starts with I and terminates with O. On the other hand, in most programming languages, an output statement may contain not only variables but also constants. In particular, some output statements may contain only constants. We call such an output statement a **constant output**. In some cases, constant outputs in a procedure are non-trivial (e.g., the constant output in Fig. 4.2), and therefore the effects of inputs on these outputs should also be examined for capturing the procedure functionality. Apparently a constant output can not directly be affected by any procedure input since there is no du-chain that terminates with a constant output. However it is obvious that a constant output in a procedure will be tested

by following a du-chain from an input I to a last p-use which leads the control flow to the constant output. For example, in Fig. 4.2, duc(d_1^a, $p_{1,2}^a$) leads to and also implicitly facilitates the testing of the dependency of the constant output "EQUILATERAL TRIANGLE" on the input d_1^a.

```
...
read(a, b, c)
if (a = b) and (b = c)
   then write("EQUILATERAL TRIANGLE")
...
```

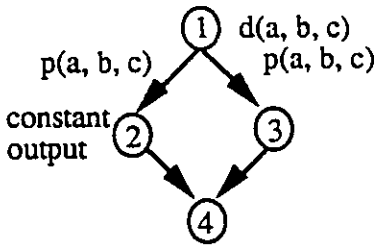


Figure 4.2. A Segment of the Triangle Classification Procedure & Its Def-Use Graph

The identification of the procedure functionality in terms of the effects of procedure inputs on predicates and procedure outputs (variable and/or constant outputs) can be accomplished by input-predicate relation analysis and input-output relation analysis, respectively. Let INPUTS, OUTPUTS, and P-USES be the sets of all the inputs, all the variable outputs, and all the p-uses in a procedure, respectively. Then, two binary relations wrt the notion of affect; namely, input-output relation and input-predicate relation can be defined as follows. **Input-output relation** is defined as R_{IO}={(i, o) | i ∈ INPUTS, o ∈ OUTPUTS, and o is affected by i}. **Input-predicate relation** is defined as R_{IP}={(i, p) | i ∈ INPUTS, p ∈ P-USES, and p is affected by i}. Let a ∈ INPUTS, b ∈ OUTPUTS, and c ∈ P-USES, then (a,b) ∈ R_{IO} or (a,c) ∈ R_{IP} if and only if there exists a du-chain that starts with a and terminates with b or c, respectively. We refer to such a du-chain as an **IP/O-chain** (Input-Predicate/Output Chain). From the definitions of R_{IO} or R_{IP}, there exists at least one IP/O-chain for every element (i,o)∈ R_{IO} or (i,p)∈ R_{IP}.

It is important to note that R_{IO} and R_{IP} are both finite sets, because $R_{IO} \subseteq$ INPUTS × OUTPUTS and $R_{IP} \subseteq$ INPUTS × P-USES, and for any given procedure, INPUTS, OUTPUTS, and P-USES are finite sets.

4.2 Definitions of the IP/O-Chains Criteria

Our objective is to design path selection criteria such that:

1. For every (i,o) $\in R_{IO}$ and (i,p) $\in R_{IP}$, IP/O-chains for (i,o) and (i,p) are covered with paths through which o and p is affected by i, respectively.
2. Minimum control flow requirements are satisfied (i.e., every edge is covered at least once).
3. Minimum data flow requirements are satisfied (i.e., every du-pair is covered at least once).

In general, there may be numerous IP/O-chains corresponding to different occurrences of the identical variable defs and c-uses which are associated with the nodes representing a loop. For example, in Fig. 4.3, there are n+1 IP/O-chains $duc(d_1^x, u_6^x)$, where n is the number of occurrences of "$u_5^x d_5^x$" (u_5^x defines d_5^x). In order to precisely describe the proposed criteria, we need to further classify the IP/O-chains. An **IP/O_n-chain** is an IP/O-chain such that there are m ($1 \le m \le n$) occurrences of "$u_j^* d_j^*$" (use of a variable defines a variable in node j), such that any d_j^* and its succeeding u_j^* form a du-pair, and for at least one node k, there are n occurrences of "$u_k^* d_k^*$". It is important to note that, as a special case, each du-pair whose def is an input and whose use is an output or a p-use is also considered as an IP/O_1-chain. Thus, in Fig. 4.3, $[d_1^x, p_{2,3}^x]$, $[d_1^x, c_6^x]$, and $[d_1^x, c_5^x d_5^x, c_6^x]$ are IP/O_1-chains, and $[d_1^x, c_5^x d_5^x, c_5^x d_5^x, c_6^x]$ is an IP/O_2-chain.

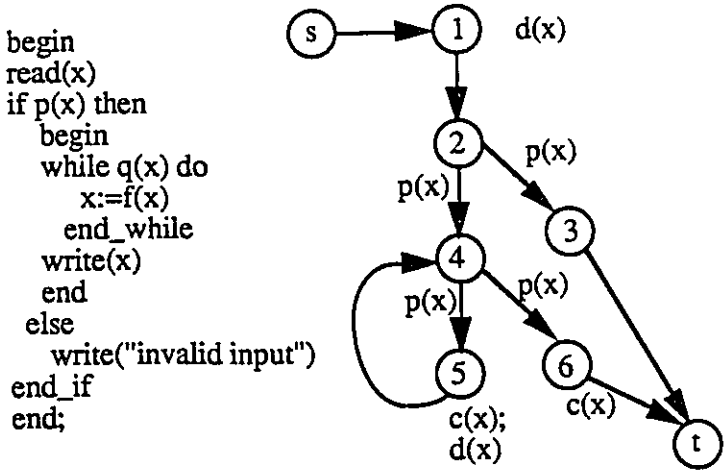


Figure 4.3. Procedure D and Its Def-Use Graph

This classification is also useful in determining what type of IP/O-chains relates to which types of associations between defs and uses in a given def-use graph. For example, a set of complete paths P that includes an activating path for each IP/O_1- and IP/O_2-chain in a given procedure, also includes an activating path for each du-pair in the procedure. Recall that an activating path for a du-pair $(d^x_{n_1}, c^x_{n_m})$ is a def-clear path wrt x from n_1 to n_m, and an activating path for a du-pair $(d^x_{n_1}, p^x_{n_{m-1},n_m})$ is a def-clear path wrt x from n_1 to (n_{m-1},n_m). An **activating path for a du-chain** $[d^{x1}_{n_1}, u^{x1}_{n2}d^{x2}_{n2}, u^{x2}_{n3}..., ... d^{xm}_{n_m}, u^{xm}_{n_{m+1}}]$ is a path $(n_1, p_1, n_2, p_2, n_3,..., n_m, p_m, n_{m+1})$ in which (n_i, p_i, n_{i+1}) is def-clear path wrt variable x_i for $1 \le i \le m$. We say that a **du-pair is covered by a du-chain** if an activating path for the du-pair is a subpath of an activating path for the du-chain.

As an example, consider the def-use graph in Fig. 4.3 which has ten du-pairs:
(d^x_1, c^x_5), (d^x_1, c^x_6), (d^x_5, c^x_5), (d^x_5, c^x_6),

$(d^x_1, p^x_{2,3})$, $(d^x_1, p^x_{2,4})$, $(d^x_1, p^x_{4,5})$, $(d^x_1, p^x_{4,6})$, $(d^x_5, p^x_{4,5})$, $(d^x_5, p^x_{4,6})$

Consider the IP/O_n-chain (d_1^x, $c_5^x d_5^x$, $c_5^x d_5^x$, ..., $c_5^x d_5^x$, c_6^x) which contains k consecutive occurrences of "$c_5^x d_5^x$". It's easy to see that seven of the 10 du-pairs, namely (d_1^x, c_5^x), (d_5^x, c_5^x), (d_5^x, c_6^x), (d_1^x, $p_{2,4}^x$), (d_1^x, $p_{4,5}^x$), (d_5^x, $p_{4,5}^x$), (d_5^x, $p_{4,6}^x$), are covered by the IP/O_n-chain. It is important to note that the IP/O_2-chain (d_1^x, $c_5^x d_5^x$, $c_5^x d_5^x$, c_6^x) also covers the same seven du-pairs, which means that we may reduce the number of occurrences of $c_5^x d_5^x$ in the IP/O_n-chain to two without affecting its coverage of du-pairs i.e., the resulting IP/O_2-chain covers the same du-pairs as the IP/O_n-chain.

The three du-pairs that can not be covered by the forementioned IP/O_n-chain are (d_1^x, c_6^x), (d_1^x, $p_{2,3}^x$), and (d_1^x, $p_{4,6}^x$). These three du-pairs can be covered by IP/O_1-chains which do not contain any occurrence of $c_5^x d_5^x$. [d_1^x, $p_{2,3}^x$] and [d_1^x, c_6^x] are such IP/O-chains. Therefore, it is sufficient to cover [d_1^x, $p_{2,3}^x$], [d_1^x, c_6^x], and [d_1^x, $c_5^x d_5^x$, $c_5^x d_5^x$, c_6^x] in order to cover all the du-pairs in the procedure. This relationship will be generalized as a theorem in section 4.3.

Now let's define the proposed data flow oriented test path selection criteria:

IP/O_n-chains coverage ($n \geq 1$) criterion requires that all the IP/O_k-chains ($1 \leq k \leq n$) for every element in R_{IO}, and all the IP/O_1-chains for every element in be R_{IP} be covered at least once.

Formally, a set of complete paths P satisfies the IP/O_n-chains coverage ($n \geq 1$) criterion if

a) for every $(i,o) \in R_{IO}$, an activating path for every IP/O_k-chain ($1 \leq k \leq n$) duc(i,o) is covered at least once by P, and

b) for every $(i,p) \in R_{IP}$, an activating path for every IP/O_1-chain duc(i,p) that is not covered in a) is covered at least once by P.

Henceforth, we refer to the entire class or any (in case that n in the IP/O_n-chains coverage criterion needs not to be explicitly given) of the proposed criteria as **IP/O-chains coverage** criteria/criterion.

IP/O-chains coverage criteria have certain advantages over the other data flow oriented test path selection criteria appeared in the literature. Specifically, the proposed criteria are based on a more general notion of what it means for a def to *affect* a use than are the all-du-paths or the required k-tuples criteria, and take into account longer du-chains than do the all-du-paths, the required k-tuples, or the data context coverage criteria. Moreover, the identification of the IP/O-chains in a procedure are very informative in the sense that they are helpful in providing better understanding of the functionality of the procedure and in checking the consistency of the procedure with its functional requirements. We shall see the strengths of the IP/O-chains coverage criteria in the subsequent section where we make some formal comparisons.

4.3 Comparisons

For the purpose of comparing the IP/O-chains coverage criteria with the existing criteria, a formal model of procedures based on the def-use graph is needed as the basis, because the existing criteria were defined using various terminology and assumptions [ClPo 89].

A **well-formed def-use graph** is a def-use graph in which a) every def of a variable reaches some use of the variable via some def-clear path; and b) every use of a variable is reached by some def of the variable which precedes the use through a def-clear path. Procedures with well-formed def-use graphs have the following property:

Lemma 4.1:

In a well-formed def-use graph, every use is affected by some input, and every def affects some output or p-use.

Proof:

Since the def-use graph is well-formed, for any variable x and node i, u_i^x is reached by some def of x, d_j^x (which precedes u_i^x) through a def-clear path wrt x.

1. If d_j^x is an input, u_i^x is affected by d_j^x;

2. Otherwise, there must exist a variable y, such that d_j^x is defined in terms of u_j^y.

The above arguments are now applied wrt y. Since for each use we always look at its def preceding it, the above procedure terminates at step 1. Therefore, every use in a well-formed def-use graph is affected by some input. By similar arguments, every def in a well-formed def-use graph affects some output or p-use.

The relative strength of test path selection criteria is commonly determined by investigating the inclusion relation [Weyu 84, RaWe 85, FrWe 88, ClPo 89]. For two test path selection criteria *A* and *B*, *A* includes *B* iff for any given flow model, any set of

complete paths that satisfies A also satisfies B. A **strictly includes** B iff A includes B but B does not include A. A and B are **incomparable** iff neither A includes B nor B includes A.

From the definitions of the IP/O-chains criteria, it follows that:

Theorem 4.1:

If $n>m\geq 1$, then IP/O_n-chains coverage criterion strictly includes IP/O_m-chains coverage criterion.

Theorem 4.2:

Every du-pair in a well-formed def-use graph is covered by an IP/O_1-chain or an IP/O_2-chain.

Proof:

First, we prove every du-pair in a well-formed def-use graph is covered by some IP/O_n-chain.

Without loss of generality, let the du-pair be (d_m^x, u_n^x).

If u_n^x is a p-use, it must be affected by some input i (Lemma 1). Thus, $(i, u_n^x) \in R_{IP}$, and (i, u_n^x) is covered by some IP/O_n-chain. Therefore, (d_m^x, u_n^x) is also covered by the IP/O_n-chain.

Otherwise, if u_n^x is a c-use, it must be affected by some input i (Lemma 1), and u_n^x defines some variable y (i.e., $u_n^x d_n^y$). Also from Lemma 1, d_n^y must affect some p-use or output u_q^z. Thus, $(i, u_q^z) \in R_{IO}$. Thus, (i, u_q^z) as well as (d_m^x, u_n^x) is covered by some IP/O_n-chain.

In [Huan 79], it is shown that the identification of all du-pairs in a given procedure requires at most two iterations of any loop in the procedure. Hence, if a du-pair is contained in a du-chain which requires a loop be iterated zero or n times, then the du-pair

is also contained in a du-chain which requires the loop be iterated zero times or twice. Therefore, every du-pair in a well-formed def-use graph is covered by some IP/O_1-chain or IP/O_2-chain.

4.3.1 IP/O-Chains vs. Some Control Flow-Oriented Criteria

Theorem 4.3:

IP/O_2-chains coverage criterion strictly includes branch coverage criterion.

Proof: From the fact that all-p-uses criterion strictly includes branch coverage criterion [RaWe 85], it is sufficient to prove that IP/O_2-chains coverage criterion includes all-p-uses criterion. All p-uses criterion requires that for any p-use p, every du-pair (d, p) be covered at least once [RaWe 85]. Formally, a set of complete paths P satisfies the all-p-uses criterion if P includes an activating path (i.e., a def-clear path wrt x from n_1 to (n_{m-1},n_m) for each du-pair $(d^x_{n1}, p^x_{n_{m-1},n_m})$.

If d is an input, then the du-pair is also an element in R_{IP}. From the definition of IP/O_2-chains coverage criterion, this du-pair is covered. Otherwise, d must be defined in terms of a c-use(s) of another variable(s). From Lemma 4.1, this c-use, and therefore p, is affected by some input i and thus, (i,p) is an element of R_{IP}. From the definition of IP/O_2-chains coverage criterion and Theorem 4.2, this element is covered by some IP/O_1-chain or IP/O_2-chain.

Theorem 4.4:

Path coverage criterion strictly includes IP/O_n-chains coverage criteria.

Proof: It is obvious that path coverage criterion includes IP/O_n-chains coverage criteria. The strictness can be seen from the fact that IP/O_n-chains coverage criteria require only finite iterations of loops, therefore the number of paths required is also finite, which may not be the case for path coverage criterion.

In the following subsections, we compare IP/O-chains coverage criteria with the often cited data flow-oriented criteria, including Rapps & Weyuker's all-uses and all-du-

paths criteria, Ntafos's required k-tuples criteria, and Laski & Korel's context coverage and ordered context coverage criteria. For convenience, the formal definitions of these criteria are repeated in the corresponding subsections.

4.3.2 IP/O-Chains vs. the Rapps-Weyuker Criteria

Rapps and Weyuker have proposed a family of six criteria [RaWe 85]. We now compare our criteria with only two of these criteria, namely all-uses or all-du-paths criteria. The inclusion relationships between IP/O_n-chains coverage criteria and the other four criteria of Rapps & Weyuker follow from the transitivity of the inclusion relationship.

Recall that **all-uses** criterion requires that every du-pair in a given procedure be covered at least once. Formally, set of complete paths P satisfies the all-uses criterion if P includes an activating path for each du-pair in a def-use graph. An activating path for du-pair $(d^x_{n_1}, c^x_{n_m})$ is a def-clear path wrt x from n_1 to n_m and for du-pair $(d^x_{n_1}, p^x_{n_{m-1},n_m})$ is a def-clear path wrt x from n_1 to (n_{m-1},n_m).

Theorem 4.5:

IP/O_2-chains coverage criterion strictly includes all-uses criterion.

Proof: Let P be a set paths that satisfies the IP/O_2-chains coverage criterion. From Theorems 4.2 and 4.3, every du-pair is covered by some path in P. Therefore, IP/O_2-chains coverage criterion includes all-uses criterion.

The strictness of the inclusion can be proved by considering the procedure and its def-use graph given in Fig. 4.4. There are four complete paths in the example (see TABLE 4.1). Path set P = {p1, p4} satisfies the all-uses criterion but P does not satisfy IP/O_2-chains coverage criterion which requires all four paths (i.e., p1, p2, p3, p4) be covered.

```
begin
read(x)
if even(x) then
      L := x / 2
  else
      L := (x-1) / 2
 z := f(L)
 if not integer(z) then
      z:=round(z)
write ("z =", z)
end;
```

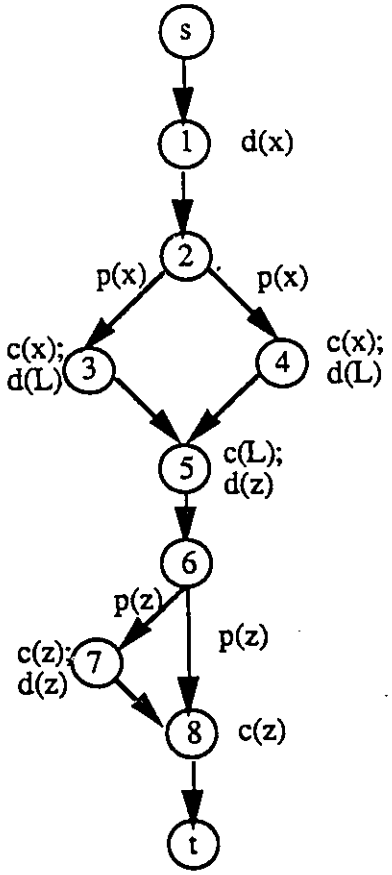


Figure 4.4. Procedure E and Its Def-Use Graph

TABLE 4.1
A Set of Paths Covering IP/O_1-Chains in Procedure E

p1 = (s, 1, 2, 3, 5, 6, 7, 8, t),
p2 = (s, 1, 2, 3, 5, 6, 8, t),
p3 = (s, 1, 2, 4, 5, 6, 7, 8, t),
p4 = (s, 1, 2, 4, 5, 6, 8, t).

Recall that **all-du-paths** criterion requires all the du-paths in a def-use graph be covered at least once. A path $(n_1, n_2, ..., n_j, n_k)$, is a **du-path** wrt a variable x if n_1 has a def of x and either 1) n_k has a c-use of x and $(n_1, ..., n_k)$ is a def-clear simple path wrt x, or 2) (n_j, n_k) has a p-use of x and $(n_1, ..., n_j)$ is a def-clear loop-free path wrt x. Formally,

a set of complete paths P satisfies the all-du-paths criterion if P includes every du-path for each du-pair in a def-use graph.

Theorem 4.6:

IP/O-chains coverage criterion is incomparable with all-du-paths criterion.

Proof: All-du-paths criterion does not even include IP/O_1-chains coverage criterion. For Procedure E, the path set P = {p1, p4} also satisfies the all-du-paths criterion but P does not satisfy the IP/O_1-chains coverage criterion.

IP/O-chains coverage criterion does not include all-du-paths criterion, because when there exists more than one du-path that cover the same du-pair, the former only requires one such path be covered. This can be seen from the example in Fig. 4.5 and tables 4.2 to 4.5, where du-path (1, 3, 4, 5, 6, 7) wrt x is not covered by any path in the Table 4.5.

```
begin
read(X,Y)
if Y≥0 then
    write("Y is positive")
  else
    write("Y is negative")
POW:=abs(Y)
Z:=1
while POW<>0 do
  Z:=Z*X
  POW:=POW-1
end_while
if Y<0 then
    Z:=1/Z
write(Z)
end;
```

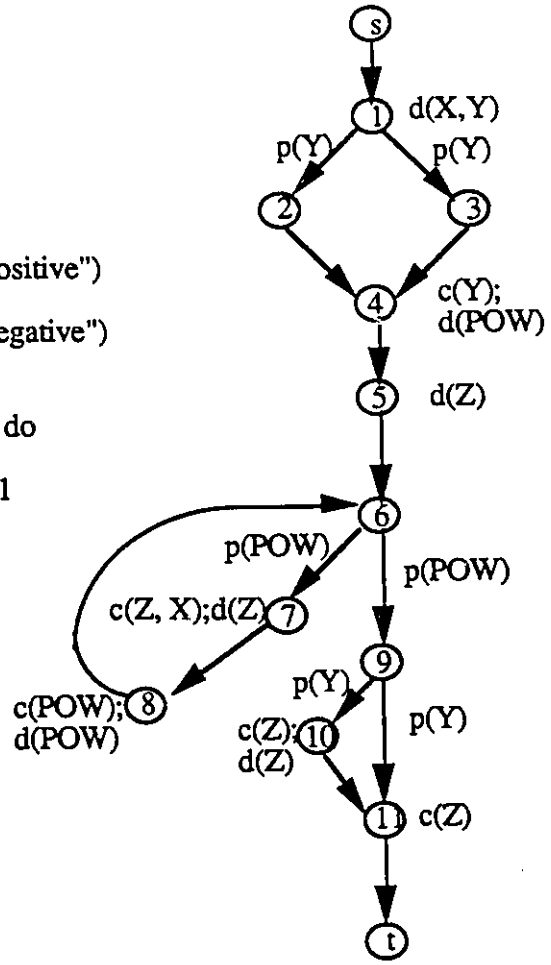


Figure 4.5. Procedure F and Its Def-Use Graph

TABLE 4.2
DU-Pairs and DU-Paths in Procedure F

DU-PAIRS	CORRESPONDING DU-PATHS
$[d_1^X, u_7^X]$	(1,2,4.5,6,7), (1,3,4,5,6,7)
$[d_1^Y, u_4^Y]$	(1,2,4), (1,3,4)
$[d_1^Y, u_{1,2}^Y]$	(1,2)
$[d_1^Y, u_{1,3}^Y]$	(1,3)
$[d_1^Y, u_{9,10}^Y]$	(1,2,4,5,6,9,10), (1,3,4,5,6,9,10)
$[d_1^Y, u_{9,11}^Y]$	(1,2,4,5,6,9,11), (1,3,4,5,6,9,11)
$[d_4^{POW}, u_8^{POW}]$	(4,5,6,7,8)
$[d_4^{POW}, u_{6,7}^{POW}]$	(4,5,6,7)
$[d_4^{POW}, u_{6,9}^{POW}]$	(4,5,6,9)
$[d_8^{POW}, u_8^{POW}]$	(8,6,7,8)
$[d_8^{POW}, u_{6,7}^{POW}]$	(8,6,7)
$[d_8^{POW}, u_{6,9}^{POW}]$	(8,6,9)
$[d_5^Z, u_7^Z]$	(5,6,7)
$[d_5^Z, u_{10}^Z]$	(5,6,9,10)
$[d_5^Z, u_{11}^Z]$	(5,6,9,11)
$[d_7^Z, u_7^Z]$	(7,8,6,7)
$[d_7^Z, u_{10}^Z]$	(7,8,6,9,10)
$[d_7^Z, u_{11}^Z]$	(7,8,6,9,11)
$[d_{10}^Z, u_{11}^Z]$	(10,11)
Total 19	23

TABLE 4.3
IP/O_1-Chains in Procedure F

$[d_1^X, u_7^X d_7^Z, u_{11}^Z]$	
$[d_1^X, u_7^X d_7^Z, u_{10}^Z d_{10}^Z, u_{11}^Z]$	
$[d_1^Y, u_{1,2}^Y]$	
$[d_1^Y, u_{1,3}^Y]$	
$[d_1^Y, u_{9,10}^Y]$	
$[d_1^Y, u_{9,11}^Y]$	
$[d_1^Y, u_4^Y d_4^{POW}, u_{6,7}^{POW}]$	
$[d_1^Y, u_4^Y d_4^{POW}, u_{6,9}^{POW}]$	
$[d_1^Y, u_4^Y d_4^{POW}, u_8^{POW} d_8^{POW}, u_{6,7}^{POW}]$	
$[d_1^Y, u_4^Y d_4^{POW}, u_8^{POW} d_8^{POW}, u_{6,9}^{POW}]$	
$[d_5^Z, u_{11}^Z]$	
$[d_5^Z, u_{10}^Z d_{10}^Z, u_{11}^Z]$	
$[d_5^Z, u_7^Z d_7^Z, u_{11}^Z]$	
$[d_5^Z, u_7^Z d_7^Z, u_{10}^Z d_{10}^Z, u_{11}^Z]$	
Total	14

TABLE 4.4
IP/O_2-Chains in Procedure F

$[d_1^X, u_7^X d_7^Z, u_7^Z d_7^Z, u_{11}^Z]$	
$[d_1^X, u_7^X d_7^Z, u_7^Z d_7^Z, u_{10}^Z d_{10}^Z, u_{11}^Z]$	
$[d_1^Y, u_4^Y d_4^{POW}, u_8^{POW} d_8^{POW}, u_8^{POW} d_8^{POW}, u_{6,7}^{POW}]$	
$[d_1^Y, u_4^Y d_4^{POW}, u_8^{POW} d_8^{POW}, u_8^{POW} d_8^{POW}, u_{6,9}^{POW}]$	
$[d_5^Z, u_7^Z d_7^Z, u_7^Z d_7^Z, u_{11}^Z]$	
$[d_5^Z, u_7^Z d_7^Z, u_7^Z d_7^Z, u_{10}^Z d_{10}^Z, u_{11}^Z]$	
Total	6

TABLE 4.5
A Set of Paths Satisfying IP/O_2-Chains Coverage in Procedure F

p1 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)
p2 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 10, 11, t)
p3 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 9, 11, t)
p4 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 9, 10, 11, t)
p5 = (s, 1, 3, 4, 5, 6, 9, 11, t)
p6 = (s, 1, 3, 4, 5, 6, 9, 10, 11, t)
p7 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)

4.3.3 IP/O-Chains vs. Required k-Tuples Criteria

Recall that **required k-tuples** criteria [Ntaf 84] ask that all chains of k-1 (or less) related du-pairs be covered at least once. Formally, a set of complete paths P satisfies the required k-tuples criterion for a given k, if P includes an activating path for every k-dr interaction $[d_1^{x1}, u_2^{x1} d_2^{x2}, u_3^{x2}..., ...d_{k-1}^{xk-1}, u_k^{xk-1}]$ is a path $(n_1, p_1, n_2, p_2, n_3,..., n_{k-1}, p_{k-1}, n_k)$ in which (n_i, p_i, n_{i+1}) is def-clear path wrt variable x_i, $1 \le i \le k-1$. In a newer version of these

criteria [Ntaf 88] which is referred to as required k-tuples$^+$ criteria, nodes in a k-dr interaction need not be distinct.

Theorem 4.7:

IP/O_2-chains coverage criterion and required k-tuples$^+$ criterion are incomparable.

Proof: First, we note that required k-tuples$^+$ criterion does not even include IP/O_1-chains coverage criterion, because for a given integer k, the procedure being tested may have IP/O_1-chains which consist of more than k-1 related du-pairs.

On the other hand, IP/O_2-chains coverage criterion does not include required k-tuples$^+$ criterion, since the latter may require higher (i.e., greater than two) iterations of loops.

However, it is obvious that IP/O_2-chains coverage criterion is inherently "stronger" than required k-tuples$^+$ criterion since for any k, every k-dr interaction is covered by some IP/O_2-chain(s) i.e., an activating path of the k-dr interaction is a subpath of an activating path of the IP/O_2-chain. Moreover, while generating a set of complete paths if we request that each loop is entered at least twice, corresponding to the boundary conditions of the loop, then the resulting IP/O-chains will not be IP/O_2-chains but IP/O_n-chains (n>2). This corresponds to IP/O_n-chains coverage criterion and gives rise of the following theorem.

Theorem 4.8:

For a given k, there exists n such that IP/O_{n-1}-chains coverage criterion strictly includes required k-tuples$^+$ criterion.

Proof: It follows from their definitions that every k-dr interaction is covered by some IP/O_m-chains (m≥1). Let n be the maximum of "some larger iteration count" [Ntaf 84] necessary for every loop in the procedure by required k-tuples$^+$ criterion. Then, IP/O_{n-1}-chains coverage criterion includes required k-tuples$^+$ criterion. The strictness of the

inclusion can be seen from Theorem 4.1 and that required k-tuples$^+$ criterion does not even include IP/O_1-chains coverage criterion.

4.3.4 IP/O-Chains vs. Context Coverage Criteria

Recall that **context coverage** criterion requires that each elementary data context be covered at least once, whereas **ordered context coverage** criterion requires that each ordered elementary data context be covered at least once. An **(ordered) elementary data context** in node i that uses a vector of variables X(i) = [x_1, x_2, ..., x_n] is a (ordered) set of definitions ec(i) = [d(x_1), d(x_2), ..., d(x_n)] of all variables from X(i) such that there exists a path from the entry node to node i and all defs from ec(i) to node i through this path (in the given order) without redefining the variables in X(i) [LaKo 83].

Formally, a set of complete paths P satisfies the (ordered) context coverage criterion, respectively, if P includes an activating path for every (ordered) elementary data context in each node in the def-use graph of a procedure. A path (n_1, p_1, n_2, p_2, ..., p_m, n_m) is an activating path for an elementary data context in node i, [$d_{n_1}^{x_1}$, $d_{n_2}^{x_2}$, ..., $d_{n_m}^{x_m}$], if path (n_i, p_i, n_{i+1}, ..., p_m, n_m) is a def-clear path wrt variable x_i, where $1 \le i \le m-1$.

IP/O-chains coverage criterion is incomparable with either context coverage or ordered context coverage criteria. The example in Fig. 4.6 shows that ordered context coverage (and therefore context coverage) criterion does not include IP/O_1-chains coverage criterion: The only ordered elementary data context associated with the def-use graph (shown in Fig. 4.6) is ODC(2)=[def(x) at in node 1]. Thus path P_1= (s, 1, 2, 3, t) satisfies ordered context coverage criterion but does not satisfy IP/O_1-chains coverage criterion which requires both P_1 and P_2= (s, 1, 2, 4, t) be tested. It is obvious that the IP/O-chains coverage criteria do not include context coverage criterion (and therefore

ordered context coverage criterion) either, because the former does not consider vectors of variables. Thus, we have the following theorem.

```
begin
input (x);
if odd(x) then
     output (1)
  else
     output (0);
end;
```

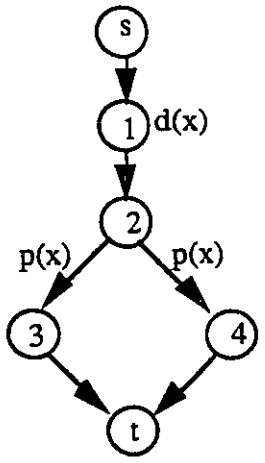


Figure 4.6. Procedure G and Its Def-Use Graph

Theorem 4.9:

IP/O-chains coverage criteria are incomparable with either context coverage or ordered context coverage criteria.

The results obtained in this section and the related results from [ClPo 89, Ntaf 88, RaWe 85] are summarized in Fig. 4.7.

Theorems 4.6 and 4.9 show that the IP/O-chains coverage criteria are incomparable with either the all-du-paths criterion and the data context coverage criteria. The following chapter examines the reasons in further detail and the possibility of enhancing the IP/O-chains coverage criteria in accordance with the all-du-paths criterion and the data context coverage criteria.

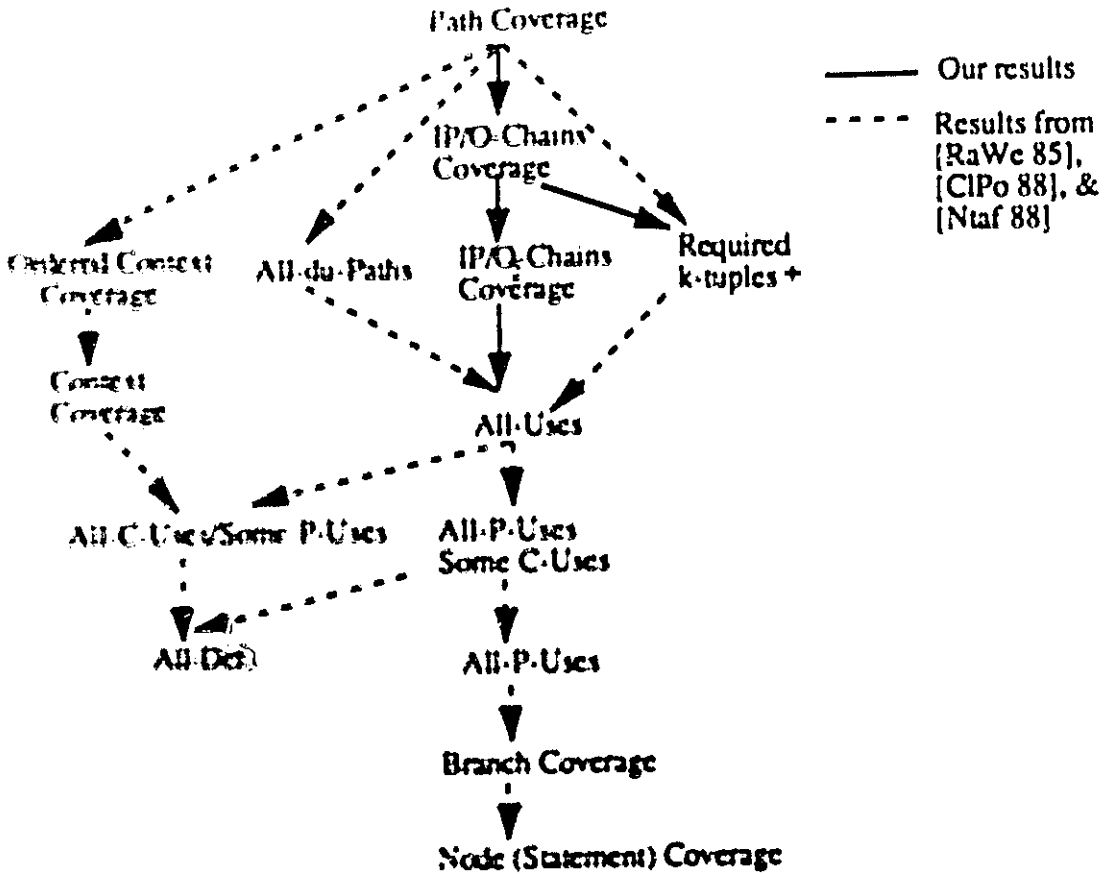


Figure 4.7. The Partial Inclusion Hierarchy

Chapter 5

Variations of the IP/O-Chains Criteria

In the previous chapter, we have shown that the IP/O-chains criteria are incomparable with the context coverage criteria which consider du-pairs for different variables being used in the same node. IP/O-chains criteria are also shown to be incomparable with the all-du-paths criterion which requires all the du-paths for every du-pair be covered.

In this chapter, we introduce two natural extensions to the IP/O_2-chains criterion, which suffice to strictly include the context coverage criterion and the all-du-paths criterion, respectively.

5.1 Definition of the IP/O_2-Chains+ Criterion

Let's look at a motivating example by Laski and Korel [LaKo 83]. The procedure and its corresponding def-use graph are given in Fig. 5.1. Table 5.1, 5.2, and 5.3 list the IP/O_1-chains, IP/O_2-chains, and data contexts in the procedure, respectively.

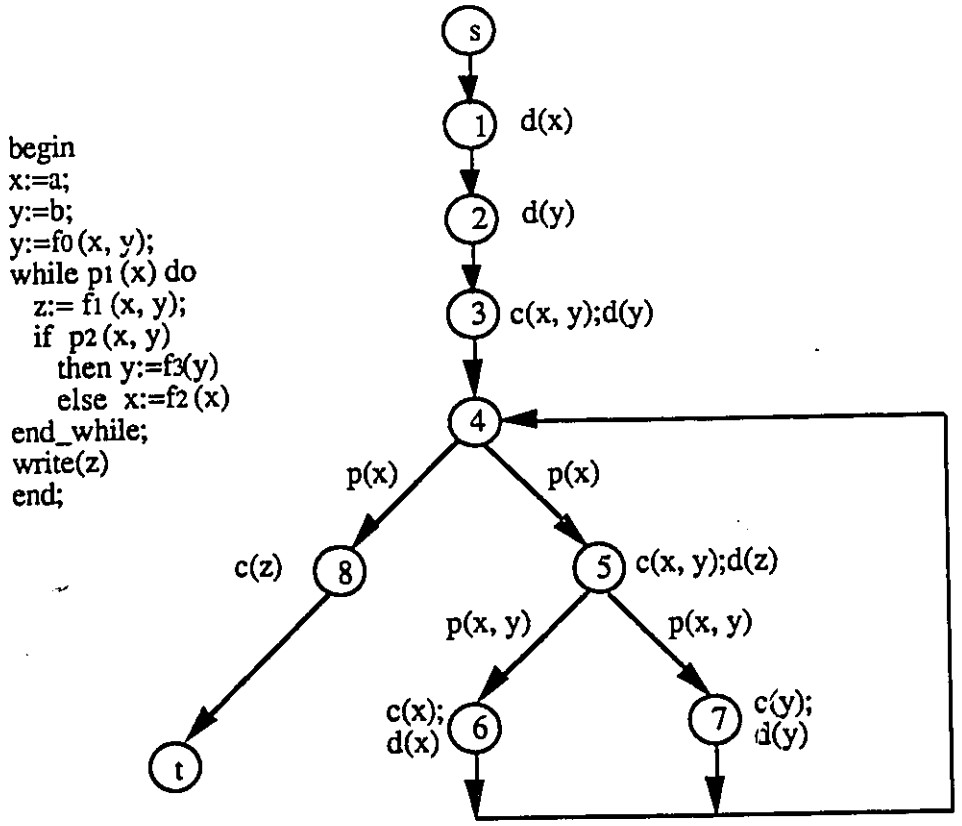


Figure 5.1. Procedure H and Its Def-Use Graph

TABLE 5.1. IP/O_1-Chains in Procedure H

$[d_1^x, c_3^x d_3^y, c_5^y d_5^z, c_8^z]$	$[d_2^y, c_3^y d_3^y, c_5^y d_5^z, c_8^z]$
$[d_1^x, c_3^x d_3^y, c_7^y d_7^y, c_5^y d_5^z, c_8^z]$	$[d_2^y, c_3^y d_3^y, c_7^y d_7^y, c_5^y d_5^z, c_8^z]$
$[d_1^x, c_5^x d_5^z, c_8^z]$	$[d_1^x, c_6^x d_6^x, c_5^x d_5^z, c_8^z]$
$[d_1^x, p_{4,5}^x]$	$[d_1^x, c_6^x d_6^x, p_{4,5}^x]$
$[d_1^x, p_{4,8}^x]$	$[d_1^x, c_6^x d_6^x, p_{4,8}^x]$

$[d_1^x, p_{5,6}^x]$	$[d_1^x, c_6^x d_6^x, p_{5,6}^x]$
$[d_1^x, c_3^x d_3^y, p_{5,6}^y]$	$[d_1^x, c_3^x d_3^y, c_7^y d_7^y, p_{5,6}^y]$
$[d_2^y, c_3^y d_3^y, p_{5,6}^y]$	$[d_2^y, c_3^y d_3^y, c_7^y d_7^y, p_{5,6}^y]$
$[d_1^x, p_{5,7}^x]$	$[d_1^x, c_6^x d_6^x, p_{5,7}^x]$
$[d_1^x, c_3^x d_3^y, p_{5,7}^y]$	$[d_1^x, c_3^x d_3^y, c_7^y d_7^y, p_{5,7}^y]$
$[d_2^y, c_3^y d_3^y, p_{5,7}^y]$	$[d_2^y, c_3^y d_3^y, c_7^y d_7^y, p_{5,7}^y]$

TABLE 5.2. IP/O_2-Chains in Procedure H

$[d_1^x, c_3^x d_3^y, c_7^y d_7^y, c_7^y d_7^y, c_5^y d_5^z, c_8^z]$	$[d_1^x, c_6^x d_6^x, c_6^x d_6^x, c_5^x d_5^z, c_8^z]$
$[d_2^y, c_3^y d_3^y, c_7^y d_7^y, c_7^y d_7^y, c_5^y d_5^z, c_8^z]$	
$[d_1^x, c_6^x d_6^x, c_6^x d_6^x, p_{4,5}^x]$	$[d_1^x, c_6^x d_6^x, c_6^x d_6^x, p_{4,8}^x]$
$[d_1^x, c_3^x d_3^y, c_7^y d_7^y, c_7^y d_7^y, p_{5,6}^y]$	$[d_1^x, c_6^x d_6^x, c_6^x d_6^x, p_{5,6}^x]$
$[d_2^y, c_3^y d_3^y, c_7^y d_7^y, c_7^y d_7^y, p_{5,6}^y]$	$[d_1^x, c_3^x d_3^y, c_7^y d_7^y, c_7^y d_7^y, p_{5,7}^y]$
$[d_1^x, c_6^x d_6^x, c_6^x d_6^x, p_{5,7}^x]$	$[d_2^y, c_3^y d_3^y, c_7^y d_7^y, c_7^y d_7^y, p_{5,7}^y]$

TABLE 5.3. Data Context in Procedure H

$DC(3) = \{(d_1^x, d_2^y)\}$

$DC(4) = \{(d_1^x)\ (d_6^x)\}$

$DC(5) = \{(d_1^x, d_3^y)\ (d_1^x, d_7^y)\ (d_3^y, d_6^x)\ (d_6^x, d_7^y)\}$

$DC(6) = \{(d_1^x)\ (d_6^x)\}$

$DC(7) = \{(d_3^y)\ (d_7^y)\}$

$DC(8) = \{(d_5^z)\}$

From these tables, it is easy to see that none of the activating paths for the elementary data contexts (d_6^x, d_7^y) in node 5, i.e., (6,4,5,7,4,5) and (7,4,5,6,4,5), is necessarily included in a complete path set satisfying the IP/O_2-chains coverage criterion. Thus, the IP/O_2-chains coverage criterion does not include context coverage criterion. However, (d_6^x, d_7^y) can be covered by merging two IP/O_1-chains or IP/O_2-chains, e.g.,$[d_1^x, c_6^x d_6^x, c_5^x d_5^z, c_8^z]$ and $[d_2^y, c_3^y d_3^y, c_7^y d_7^y, c_5^y d_5^z, c_8^z]$, such that the first chain contains du-pair (d_6^x, c_5^x) and the other chain contains du-pair (d_7^y, c_5^y). The resulting chains are $[d_1^x, d_2^y, c_3^y d_3^y, c_6^x d_6^x, c_7^y d_7^y, c_5^{x,y} d_5^z, c_8^z]$ and $[d_1^x, d_2^y, c_3^y d_3^y, c_7^y d_7^y, c_6^x d_6^x, c_5^{x,y} d_5^z, c_8^z]$, where $c_5^{x,y}$stands for c(x,y) in node 5. Either chain guarantees an activating path for (d_6^x, d_7^y) is included in the complete path set. This observation can be generalized as follows.

r IP/O_k-chains ($r \geq 2$, $1 \leq k \leq 2$) are **mergeable** wrt an elementary data context $(d_{n_1}^{x_1}, d_{n_2}^{x_2}, \ldots d_{n_m}^{x_m})$ at node n_u if each du-pair $(d_{n_j}^{x_j}, u_{n_u}^{x_j})$, $1 \leq j \leq m$, is contained in at least one of the IP/O_k-chains, and the portion of an IP/O_k-chain succeeding at least one $(d_{n_j}^{x_j}, u_{n_u}^{x_j})$ which is contained in the IP/O_k-chain is identical to that of the other IP/O_k-chains.

By definition of elementary data context, r mergeable IP/O_k-chains can be interleaved in a way (which may not be unique), such that there exists a complete path which includes an activating path for each elementary data context. We refer to the resulting chain as a **merged IP/O_2-chain** of the elementary data context ($d_{n_1}^{x_1}$, $d_{n_2}^{x_2}$, ... $d_{n_m}^{x_m}$) at node n_u. Note that uses $u_{n_u}^{x_1}$, $u_{n_u}^{x_2}$,...,$u_{n_u}^{x_m}$ in node n_u are represented in the merged IP/O_2-chain by $u_{n_u}^{x_1,x_2,\ldots,x_m}$. Note also that in the above definiton, if ordered elementary data contexts are used instead of using elementary data contexts, then, by similar arguments in [LaKo 83], a stronger criterion can be obtained which requires more paths.

IP/O_2-chains$^+$ coverage criterion requires the IP/O_2-chains coverage criterion be satisfied and a merged IP/O_2-chain of each elementary data context be covered at least once.

Formally, a set of complete paths P satisfies the IP/O_2-chains$^+$ coverage criterion if

a) for every (i,o) $\in$ R_{IO}, an activating path for every IP/O_k-chain ($1 \le k \le 2$) duc(i,o) is covered at least once by P, and

b) for every (i,p) $\in$ R_{IP}, an activating path for every IP/O_1-chain duc(i,p) that is not covered in a) is covered at least once by P.

c) P includes at least one activating path for a merged IP/O_2-chain for every elementary data context.

Let [$d_{n_a}^{v_a}$, $d_{n_b}^{v_b}$, ..., $d_{n_k}^{v_k}$, α, $c_{n_{d1}}^{y_1} d_{n_{d1}}^{x_{d1}}$, α_1, $c_{n_{d2}}^{y_2} d_{n_{d2}}^{x_{d2}}$, α_2, ..., $c_{n_{dm}}^{y_m} d_{n_{dm}}^{x_{dm}}$, α_m, $c_{n_u}^{x_1,x_2,\ldots,x_m} d_{n_u}^{x_u}$, α_u, $c_{n_o}^{x_o}$] be a merged IP/O_2-chain for elementary data context ($d_{n_1}^{x_1}$, $d_{n_2}^{x_2}$, ... $d_{n_m}^{x_m}$), where (d1, d2, ..., dm) $\in$ {permutation(1, 2, ..., m)} and each of $c_{n_{d1}}^{y_1}$, $c_{n_{d2}}^{y_2}$, ..., or $c_{n_{dm}}^{y_m}$ is affected by one of the inputs $d_{n_a}^{v_a}$, $d_{n_b}^{v_b}$, ..., or $d_{n_k}^{v_k}$. Then, an **activating path for the merged IP/O_2-chain** at node n_u is a path (n_a, α_a, n_b, α_b, ..., n_k, α, n_{d1}, α_1, n_{d2}, α_2,

..., n_{dm}, α_m, n_u, α_u, n_o) such that path (α_1, ..., α_m) is a def-clear path wrt variable x_{dl}, $1 \leq l \leq m$.

From the above definition, we have:

Theorem 5.1:

Path coverage criterion strictly includes IP/O_2-chains$^+$ coverage criterion, and IP/O_2-chains$^+$ coverage criterion strictly includes IP/O_2-chains coverage criterion.

Theorem 5.2:

IP/O_2-chains$^+$ coverage criterion strictly includes the context coverage criterion.

Proof: For any elementary data context, if it only involves a single variable then covering the elementary data context is the same as covering the corresponding du-pair. Since IP/O_2-chains$^+$ coverage criterion strictly includes IP/O_2-chains coverage criterion (Theorems 5.1) and IP/O_2-chains coverage criterion strictly includes the all-uses coverage criterion (Theorem 4.5), the elementary data context (the du-pair) is required to be covered by IP/O_2-chains$^+$ coverage criterion.

If an elementary data context involves two or more variables, then from the definition of IP/O_2-chains$^+$ coverage criterion, at least one activating path of a merged IP/O_2-chain for the elementary data context is required by IP/O_2-chains$^+$ coverage criterion.

Therefore, IP/O_2-chains$^+$ coverage criterion includes the context coverage criterion. The strictness can be seen from Theorems 4.9 & 5.1.

In this section, we have further examined the differences between the IP/O_2-chains coverage criteria and the context coverage criterion. The former takes into account the fact that a procedure input should affect some predicate or output in the procedure. The latter,

on the other hand, considers all the defs that are used together. To take advantage of the two families of criteria, the IP/O_2-chains$^+$ coverage criterion is proposed as a useful variation of the IP/O_2-chains coverage criterion. The relative strength of the IP/O_2-chains$^+$ coverage criterion is given in the form of two theorems (Theorems 5.1 & 5.2).

In the following section, we will further examine the reason why the IP/O_2-chains coverage criterion and the all-du-paths criterion are incomparable.

5.2 Definition of the IP/O_2-Chains* Criterion

It is shown in Chapter 4 that the IP/O-chains coverage criteria are incomparable with the all-du-paths criterion (Theorem 4.6). Our intention there was to design a useful test path selection criterion which is not too demanding and covers the input-output and input-predicate relations while also guarantees the minimum data flow (all uses) and control flow (branch) coverage. In this section we propose a more demanding criterion, called the IP/O_2-chains* coverage criterion, which is derived from the IP/O_2-chains coverage criterion. The relative strength of the IP/O_2-chains* coverage criterion is also studied against the all-du-paths criterion.

Recall that a du-pair may have more than one activating path. The all-du-paths criterion requires all the simple and loop-free activating paths for every du-pair related to a c-use and p-use, respectively, be covered. Although, on the other hand, every du-pair is contained in some IP/O_1-chain or IP/O_2-chain, only one activating path for the IP/O_1-chain or IP/O_2-chain is required by the IP/O_2-chains coverage criterion.

IP/O_2-chains* coverage criterion requires that for each IP/O_1-chain or IP/O_2-chain required by the IP/O_2-chains coverage criterion, the paths formed from the concatenation of du-paths for the du-pairs in the IP/O_1-chain or IP/O_2-chain are covered at least once.

Formally, a set of complete paths P satisfies the IP/O_2-chains* coverage criterion if

a) for every $(i,o) \in R_{IO}$, all the activating paths for every IP/O_k-chain ($1 \le k \le n$) duc(i,o), formed by concatenating du-paths for the du-pairs in the IP/O_k-chain, are included in P , and

b) for every $(i,p) \in R_{IP}$, all the activating paths for every IP/O_1-chain duc(i,p) that is not covered in a), formed by concatenating du-paths for the du-pairs in the IP/O_1-chain, are included in P.

From the related definitions it is easy to see:

Theorem 5.3:

Path coverage criterion strictly includes IP/O_2-chains* coverage criterion, and IP/O_2-chains* coverage criterion strictly includes IP/O_2-chains coverage criterion.

Theorem 5.4:

IP/O_2-chains* coverage criterion strictly includes the all-du-paths criterion.

Proof: IP/O_2-chains* coverage criterion includes all-du-paths criterion. This is because the du-pair corresponding to any du-path is contained in some IP/O_1-chain or IP/O_2-chain, and thus, from the definition of IP/O_2-chains* coverage criterion, the du-path is covered.

The strictness can be seen from Theorems 4.6 and 5.3.

The set of complete paths in Procedure F required by IP/O_2-chains* coverage criterion is given in Table 5.4. By comparing Table 5.4 to Table 4.5, we see that IP/O_2-chains* coverage criterion imposes more extensive coverage than the all du-paths coverage criterion.

TABLE 5.4. Paths Required by IP/O_2-Chains* for Procedure F

(s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)
(s, 1, 3, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)
(s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 10, 11, t)
(s, 1, 3, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 10, 11, t)
(s, 1, 2, 4, 5, 6, 7, 8, 6, 9, 11, t)
(s, 1, 3, 4, 5, 6, 7, 8, 6, 9, 11, t)
(s, 1, 2, 4, 5, 6, 7, 8, 6, 9, 10, 11, t)
(s, 1, 3, 4, 5, 6, 7, 8, 6, 9, 10, 11, t)

(s, 1, 2, 4, 5, 6, 9, 11, t)
(s, 1, 3, 4, 5, 6, 9, 11, t)
(s, 1, 2, 4, 5, 6, 9, 10, 11, t)
(s, 1, 3, 4, 5, 6, 9, 10, 11, t)
(s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)
(s, 1, 3, 4, 5, 6, 7, 8, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)

Theorem 5.5:

IP/O_2-chains$^+$ coverage criterion is incomparable with IP/O_2-chains* coverage criterion.

Proof: From Tables 4.3, 4.4 & 5.5, we obtain two merged IP/O_2-chains :

$[d_1^X, d_5^Z, u_7^{X,Z} d_7^Z, u_{11}^Z]$ and $[d_1^X, d_5^Z, u_7^{X,Z} d_7^Z, u_7^Z d_7^Z, u_{11}^Z]$

The set of paths in Procedure F required by IP/O_2-chains$^+$ coverage criterion is given in Table 5.6. By comparing Table 5.6 to Table 5.4, we see that IP/O_2-chains$^+$ coverage criterion does not include IP/O_2-chains* coverage criterion.

TABLE 5.5. Data Context in Procedure F

$DC(1) = \{(d_1^Y)\}$

$DC(4) = \{(d_1^Y)\}$

$DC(6) = \{(d_4^{POW})\ (d_8^{POW})\}$

$DC(7) = \{(d_1^X, d_5^Z)\ (d_1^X, d_7^Z)\}$

$DC(8) = \{(d_4^{POW})\ (d_8^{POW})\}$

$DC(9) = \{(d_1^Y)\}$

$DC(10) = \{(d_5^Z)\ (d_7^Z)\}$

$DC(11) = \{(d_5^Z)\ (d_7^Z)\ (d_{10}^Z)\}$

TABLE 5.6. Paths Required by IP/O_2-Chains$^{\pm}$ for Procedure F

p1 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)

p2 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 10, 11, t)

p3 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 9, 11, t)

p4 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 9, 10, 11, t)

p5 = (s, 1, 3, 4, 5, 6, 9, 11, t)

p6 = (s, 1, 3, 4, 5, 6, 9, 10, 11, t)

p7 = (s, 1, 2, 4, 5, 6, 7, 8, 6, 7, 8, 6, 7, 8, 6, 9, 11, t)

On the other hand, from Table 5.7 we can see that none of the du-paths in the Procedure H covers the elementary data contexts (d_6^x, d_7^y) at node 5. Thus, IP/O_2-chains* coverage criterion does not include IP/O_2-chains$^+$ coverage criterion either. We complete this proof.

TABLE 5.7 DU-Pairs and DU-Paths in Procedure H

DU-PAIRS	CORRESPONDING DU-PATH(S)
$[d_1^x, u_3^x]$	(1,2,3)
$[d_1^x, u_5^x]$	(1,2,3,4,5)
$[d_1^x, u_6^x]$	(1,2,3,4,5,6)
$[d_6^x, u_5^x]$	(6,4,5)
$[d_6^x, u_6^x]$	(6,4,5,6)
$[d_1^x, u_{4,5}^x]$	(1,2,3,4,5)
$[d_1^x, u_{4,8}^x]$	(1,2,3,4,8)
$[d_1^x, u_{5,6}^x]$	(1,2,3,4,5,6)
$[d_1^x, u_{5,7}^x]$	(1,2,3,4,5,7)
$[d_6^x, u_{4,5}^x]$	(6,4,5)
$[d_6^x, u_{4,8}^x]$	(6,4,8)
$[d_6^x, u_{5,6}^x]$	(6,4,5,6)
$[d_6^x, u_{5,7}^x]$	(6,4,5,7)
$[d_2^y, u_3^y]$	(2,3)
$[d_3^y, u_5^y]$	(3,4,5)
$[d_3^y, u_7^y]$	(3,4,5,7)
$[d_7^y, u_5^y]$	(7,4,5)
$[d_7^y, u_7^y]$	(7,4,5,7)
$[d_3^y, u_{5,6}^y]$	(3,4,5,6)
$[d_3^y, u_{5,7}^y]$	(3,4,5,7)
$[d_7^y, u_{5,6}^y]$	(7,4,5,6)

$[d_7^y, u_{5,7}^y]$	(7,4,5,7)
$[d_5^z, u_8^z]$	(5,6,4,8), (5,7,4,8)

The complete results obtained in sections 5.2 and 5.3 and the related results from [RaWe 85, Ntaf 88, ClPo 89] are summarized in Fig. 5.2.

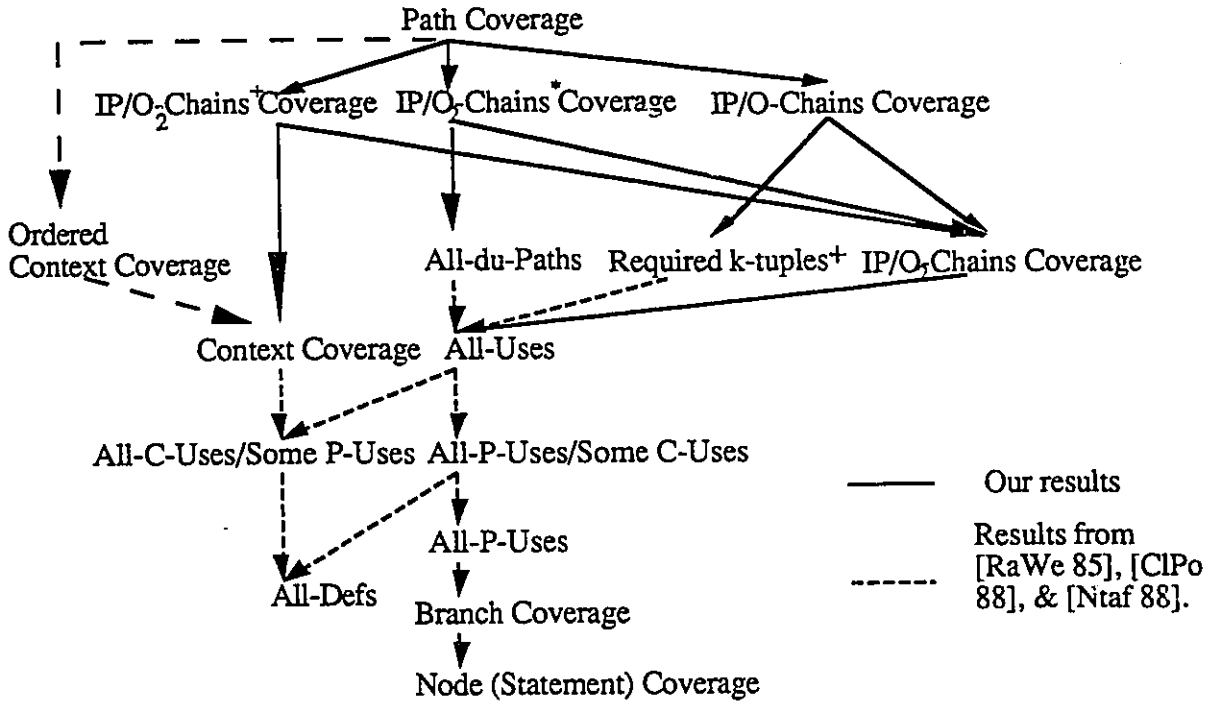


Figure 5.2. The Complete Inclusion Hierarchy.

Chapter 6

Inter-Procedural Data Flow Testing

In previous chapters, testing is assumed to be performed at the procedure level and the discussion is facilitated by using the def-use graph model which is only capable of modeling data dependencies within procedures (i.e., intra-procedural data flow). Data dependencies across procedure boundaries (i.e., inter-procedural data flow) have not been taken into account. As procedures in a subprogram are closely interrelated through inter-procedual control and data dependencies, it is believed that, while still being manageable, it is more meaningful to perform data flow testing at the subprogram level. This is expected to affect the accuracy and effectiveness of a particular data flow oriented path selection criterion being used.

The rest of this chapter is organized as follows. In Section 6.1, we point out the necessity of modeling and testing inter-procedural data flow and address some of the problems that are inherent to them. Section 6.2 presents an existing inter-procedural data flow model. In this section we argue that, for the purpose of data flow testing, intra-procedural and inter-procedural data dependencies are not separable and therefore should be considered together. In Section 6.3, we introduce a flow model to represent intra-procedural as well as inter-procedural control and data dependencies. As an application of the model, we show how it can be used to represent Pascal programs. In Section 6.4, the

data flow testing techniques described in Chapters 3 and 4 are utilized and applied to the model. A Pascal example is used throughout the chapter for illustrations and detailed discussions.

6.1 Background and Motivation

Procedural programming languages are well-structured and therefore most widely used for implementing large software systems. The smallest compilable software unit (henceforth referred to as "subprogram") of such a system in a procedural language usually consists of a main procedure and a relatively small number of component procedures.

In the literature, data flow testing is usually targeted at procedure level due to its complexity. Since procedures in a subprogram are very closely related in terms of inter-procedural control and data dependencies, we believe that, while still being manageable, it is more meaningful and, likely, more effective to perform data flow testing at subprogram level. To do so, we need to identify defs and uses in a subprogram, and to represent their dependencies in the control flow exhibited by the subprogram. This in turn necessitates a set of translation rules and a program model. The translation rules are language dependent and specify how defs and uses are identified in a particular programming construct of the language used to write the subprogram, and the program model facilitates analysis and testing.

Translation rules and program models are given at procedure level in [RaWe 85, FrWe 88, ClPo 89] for intra-procedural data flow dependencies. Nevertheless, they allow precise descriptions of data flow testing methods and their implementations be independent of the language in which the program under test is written. They also provide a unified view of the existing methods, making comparisons among these methods easier and more accurate.

As a preparation for a program model at the subprogram level, which models control flow as well as both intra-procedural and inter-procedural data flow, we first define the DUG model which is based on the def-use graph [FrWe 88]. Using this model, we

show that program models at procedure level are incapable of accurately representing inter-procedural data flow and in fact, leading to misinterpretations of the actual behavior of several program constructs.

A **DUG** is a digraph *G(V, E)*, where *V* is a set of nodes, each representing a data flow expression within a simple statement and *E* is a set of edges, each representing the control flow between a pair of nodes. Each outgoing edge of a node with multiple successors is associated with a predicate expression.

A **data flow expression** corresponds to a statement and can be defined using the Backus-Naur Form (BNF) as follows.

<data flow expression> ::= <factor>;

<factor> ::= <actions> | <actions> , <factor> | (<factor>)

<actions> ::= d(<variable_list>) | c(<variable_list>) | <term>

<term> ::= <update> | <update>; <term>

<update> ::= c(<variable_list>) ; d(<variable_list>)

<variable_list> ::= <variable> | <variable>; <variable_list>

where <variable> is a legal variable name.

A **predicate expression** corresponds to a branching condition (Boolean expression) and can be defined as:

<predicate expression> ::= p(<variable_list>);

In the above defnitions, terminal ";" means *followed by*, and terminal "," means *occur concurrently*. Thus $A(v_1; v_2; ...; v_n)$ represents $A(v_1); A(v_2); ...; A(v_n)$, where A is either d or c, and $v_1, v_2, ..., v_n$ are variable names. For convenience, we also write $A(v_1, v_2, ..., v_n)$ to represent $A(v_1), A(v_2), ..., A(v_n)$. Accordingly, c(L;S) means that c-use of L followed by c-use of S, and (c(L;S), c(H, S), c(MX)); d(MX) means that c(L;S), c(H,

S) and c(MX) may occur independent to each other and are used together to define variable MX.

```
1|  procedure Main(input, output);
2|     const  N=1000;
3|     type   elements=array[1..N] of real;
4|     var    S:elements;
5|            I, MAX: integer;
6|     procedure GetMax(L, H: integer; var MX: integer);
7|        var M,M1,M2:integer;
8|        procedure PairMax(I,J:integer; var K:integer);
9|        begin
10|       if I>J then K:=I else K:=J
11|       end;
12|    begin
13|    if L+1=H then PairMax(S[L], S[H], MX)
14|               else  begin
15|                     M:=(L+H) div 2;
16|                     GetMax(L, M, M1);
17|                     GetMax(M+1 ,H, M2);
18|                     PairMax(M1, M2, MX)
19|                     end
20|    end;
21| begin
22| for I:=1 to N do read(input, S[I]);
23| GetMax(1,N,Max);
24| writeln(output, Max)
25| end;
```

Figure 6.1. A Pascal Subprogram Listing.

Without loss of generality, we consider subprograms with a single entry and single exit. The subprogram entry and exit are represented by two special nodes *s* and *t*, called **entry node** and **exit node**, respectively. *s* has no predecessor whereas *t* has no successor. All other nodes have one or more predecessors and successors. Let's look at Fig.6.1 which is a Pascal implementation of the subprogram due to Harrold and Soffa [HaSo 89]. It takes N elements S[1], ..., S[N] of array S, computes the maximum element and prints it out. The subprogram consists of a main procedure and two nested procedures: GetMax and PairMax.The DUG for the subprogram is given in Fig.6.2. Note that the rules for translation from Pascal constructs to subgraphs given in chapter 3 are assumed. Thus,

for example, conditional statements such as the FOR statement in Fig.6.1, are decomposed into two or more statement blocks, each represented by a subgraph in which at least one of its nodes has multiple successors. Each outgoing edge of a node with multiple successors is associated with a predicate expression representing p-uses in the corresponding Boolean expression.

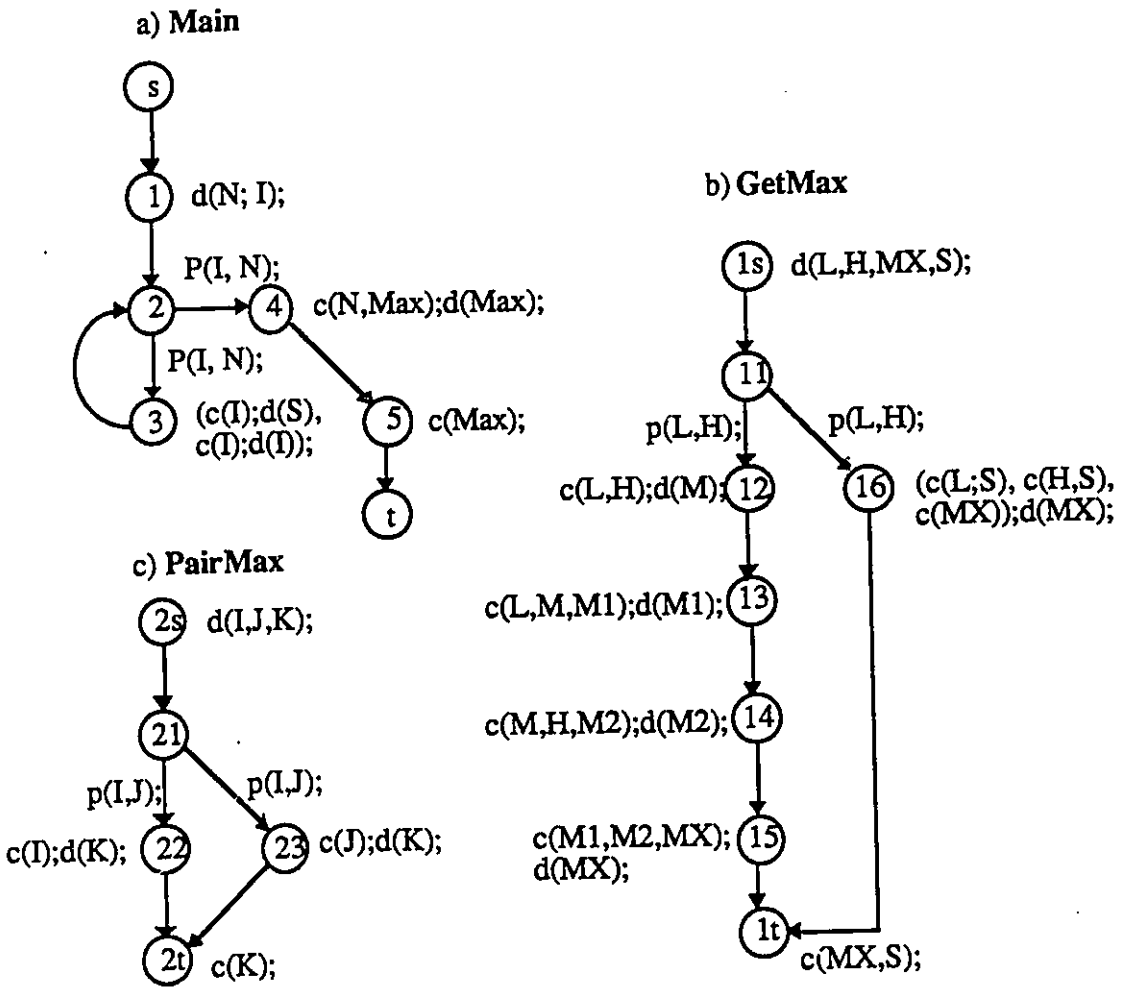


Figure 6.2. DUGs for Main, GetMax, and PairMax

Clearly, data dependencies exist not only within procedures (intra-procedural data flow), but also across procedure boundaries (inter-procedural data flow). As they are

closely related to each other, both should be tested. However, the DUG model as well as other models in the data flow testing literature fail to accurately reveal their relationships. This is because a called procedure is treated as a black-box, where each call to the procedure is considered as c-uses of all parameters followed by defs of all pass-by-reference parameters. Such handling is inaccurate and could be very misleading. By doing so, a test selection criterion simply ignores the existence of the inter-procedural data dependencies and capability of utilizing such information in designing useful tests. Lack of information about the called procedures may also lead to misinterpretation of the actual behavior of these procedures. These points will become clear with the illustration of the intra-procedural data dependencies in each procedure in the simple but, nonetheless, real Pascal subprogram of Fig.6.1 in Fig.6.2.

In Main, there is only one procedure call: GetMax(1, N, Max). Because the procedure is treated as a black-box, the call is interpreted as "c(N,Max);d(Max);" in Fig.6.2. Global (to GetMax and PairMax) variable S is not even handled in the DUG for Main. Clearly this interpretation is far from the intended behavior : "c(S, N); d(Max)", therefore it does not make the coverage of these data dependencies very meaningful.

Another problem with the black-box treatment of procedures is that some important paths in the called procedures may not be tested because the accurate data flow information about these procedures are not available to the calling procedure which is under test. For example, as pointed out in [HaSo 89], test data 3,5,1,6 for S when N is 4 satisfies the all-uses criterion [RaWe 85] for procedure GetMax, but it only covers one of the defs of variable K in PairMax. This is due to the fact that information about the intra-procedural data dependencies in PairMax is not available to the calling procedure GetMax. This problem can be overcome only when the inter-procedural data dependencies are identified and used for the test selection.

In addition to procedure calls and global variables, recursion, aliasing as well as procedure entries and exits are not well-handled, if at all, by the existing data flow coverage criteria using DUG or models similar to DUG. This is due to the fact that inter-procedural data dependencies have not been taken into account.

In GetMax, direct recursions occur: there are two calls to the procedure itself. This further complicates the problem of getting accurate information about the procedure. In particular, data flow information (i.e., data dependencies) is collected through static analysis. Dynamic properties remain unknown. Recursive calls to GetMax are handled in the same way as for other procedures (i.e., in this case, Main and PairMax). That is, they are considered as a black-box, and each call to a procedure consists of c-uses of all actual parameters followed by defs of the actual pass-by-reference parameters. Therefore, a call GetMax(P1, P2, P3) is interpreted as c-uses of actual parameters P1, P2 and P3, followed by a def of the actual pass-by-reference parameter P3.

Having shown the limitations of the DUG and other similar program models in the data flow testing literature, the following sections describe how the fore-mentioned problems can be resolved.

6.2 Inter-Procedural Data Flow Modeling and Analysis

In [HaSo 89], Harrold and Soffa introduce a model called inter-procedural flow graph (IFG) and describe an algorithm for computing the inter-procedural data dependencies. This information is then used in the all-uses criterion for guiding the selection of paths. It is shown that by doing so data flow testing can be used to test individual procedures in a subprogram more accurately since data flow information about called procedures as well as the interactions of these procedures as indicated by the inter-procedural data dependencies are known. In the following we will review their approach and point out some of its problems.

6.2.1 The Inter-Procedural Flow Graph Model

A straightforward way to represent a subprogram is by in-line substitution of procedures at call sites. The obvious problem is its memory requirements. Also, both scoping of local variables in procedures and binding of formal and actual parameters are difficult because the entire subprogram is viewed as one procedure. Another problem is its incapability for representing recursive procedures. A call graph could also be used as a model. As defined in Chapter 2, the nodes in such a graph represent procedures and edges represent procedure calls. However, the call graph is insufficient for computing the inter-procedural data dependencies because it has no return information and provides no information about the control and data flow in the individual procedures.

Before describing the IFG model by [HaSo 89], we define some terminology related to inter-procedural data flow analysis: In the literature, a procedure call is often referred to as a **call site** and a procedure return is often referred to as a **return site**. One writes $P \xrightarrow{(a1,a2,\ldots,ak)} Q(f_1, f_2, \ldots, f_k)$ if there is a call-site in procedure P calling procedure Q with actual parameters $a_1, a_2, \ldots, a_k$ where a_i corresponds to formal parameter f_i. One writes $Q(f_1, f_2, \ldots, f_k) \xrightarrow{(a1,a2,\ldots,ak)} P$ if there is a return-site in procedure Q

called by procedure P with actual parameters a_1, a_2, ..., a_k where a_i corresponds to formal parameter f_i. As indicated in [HaSo 89], there are two types of inter-procedural data dependencies in a subprogram: direct data dependencies and indirect data dependencies. A **direct data dependency** is a du-pair whose def occurs in procedure P and use occurs in a directly called procedure Q of P. Data dependencies exist when (1) a def of an actual parameter in one procedure reaches a use of the corresponding formal parameter at call site (i.e., a procedure call); or (2) a def of a formal parameter in a called procedure reaches a use of the corresponding actual parameter at return site (i.e., a procedure return); or (3) a def of a global variable reaches a call or return site. An **indirect data dependency** is a du-pair whose def occurs in procedure P and use occurs in an indirectly called procedure Q of P. Conditions for indirect data dependencies are similar to those for direct data dependencies, except that multiple levels of procedure calls and returns are considered. Determining the indirect data dependencies can be accomplished by considering the possible uses of defs along the calling sequences. When a formal parameter is passed as an actual parameter at a call site, an indirect data dependency may exist. In this case, a def of an actual parameter in one procedure may have uses in procedures more than one level away in the calling sequence, considering both calls and returns. Examples of direct data dependencies in the Pascal subprogram given in Fig.1 are:(d_1^N, c_{12}^H) and (d_{16}^{MX}, c_5^{Max}), and an example of indirect data dependencies is: (d_{22}^K, c_5^{Max}). Here d_n^v or c_n^v should be read as def or c-use of variable v in node n, respectively.

The **inter-procedural flow graph** (IFG) [HaSo 89] is based on the program summary graph (PSG) proposed by Callahan [Call 88]. Similar to the PSG model, an IFG has four types of nodes: entry, exit, call and return nodes and three types of edges: binding edges, reaching edges and interreaching edges.

Entry nodes and **exit nodes** represent procedure entry and procedure exit, respectively. Both entry and exit nodes are created for every formal reference parameter of every procedure.

Call nodes and **return nodes** represent procedure invocations and procedure returns, respectively. Both call and return nodes are created for every actual reference parameter of every procedure.

Binding edges are from call nodes to entry nodes and exit nodes to return nodes. These edges correspond to the binding of formal and actual parameters.

Reaching edges from entry and return nodes to exit and call nodes summarize the control information in the procedure by indicating that a def that reaches the source of an edge also reaches the sink of the edge. A reaching edge from an entry node to an exit node represents that the variable is not redefined in the procedure. A reaching edge from an entry node to a call node represents that the variable is not redefined prior to the call in the procedure. A reaching edge from a return node to an exit node represents that the variable is not redefined after the call in the procedure. A reaching edge from a return node of one procedure to a call node of a succeeding procedure represents that the variable is not redefined prior to the succeeding call.

These edges are strictly intra-procedural since they are computed without incorporating the control structure of called procedures by using "best case" assumptions at call sites. The best case is to assume that there is no definition or use of the variable in the called procedure and that the variable is not preserved over the call [Lome 77].

Inter-reaching edges from call nodes to return nodes that abstract the intra-procedural data flow information of the called procedure allow this information to be incorporated into the calling procedure. This edge is added to the IFG if the variable is

preserved over a call to the procedure. The inter-reaching edges allow the calling context of the called procedures to be preserved during propagation.

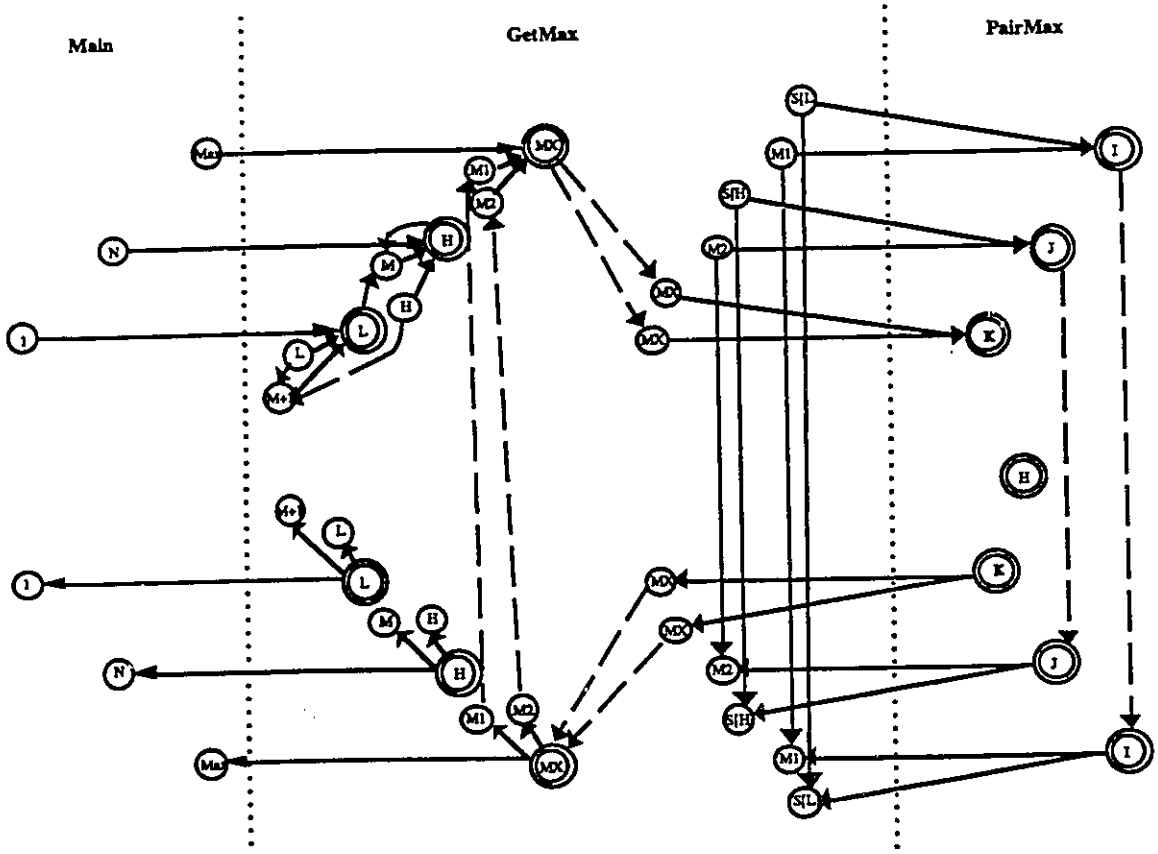


Figure 6.3 The IFG for the Pascal Example in Fig. 6.1

Fig. 6.3 gives the IFG for the program in Fig. 6.1. In the graph, circles represent call and return nodes, double circles represent entry and exit nodes, solid lines correspond to binding edges, dashed lines to reaching edges and bold lines to inter-reaching edges.

6.2.2 Computing Inter-Procedural Data Dependencies

The algorithm given in [HaSo 89] for computing the inter-procedural data dependencies consists of four steps. This algorithm summarizes the procedure information at call sites and then propagates this information throughout the program to obtain inter-procedural def-use information for both global variables and reference parameters. Recursion is also handled to allow data flow analysis of recursive procedures.

Step 1: Construction of IFG subgraphs to abstract control flow information for each procedure in the program. Nodes in a subgraph for a procedure represent regions of code associated with points that are of interest inter-procedurally, and edges represent the control flow in the procedure. Local information is computed for non-local variables and is attached to appropriate nodes in the subgraph.

Step 2: Construction of an IFG to represent the inter-procedural control flow in the program. The subgraphs of the procedures obtained in step 1 are combined to create this IFG. Information attached to nodes is computed for each procedure using the IFG, and edges that represent this information are added to the graph.

Step 3: Propagation throughout the IFG to obtain global information. The local information at each node is propagated in two phases throughout the graph resulting in the inter-procedural defs that reach, and the inter-procedural uses that can be reached from, the parts of the program represented by the node in the graph.

Step 4: Computation of the inter-procedural def-use and use-def chains [HaSo 89] using both the local information and the propagated global information.

As procedures are processed one at a time in some order, the results of the intra-procedural data flow analysis are used to construct the IFG. In step 1, def and use information are recorded by set DEF and set UPEXP, respectively. DEF and UPEXP are

attached to the nodes in the IFG. These sets contain data flow information about a procedure.

The inter-procedural data flow analysis algorithm is described in detail in [HaSo 89]. It enables the efficient computation of information dealing with the location of defs and uses needed in inter-procedural data flow testing. A technique utilizing this information is presented in [HaSo 89] which takes into account the various associations of names with defs and uses across procedures.

Accordingly, Table 6.1 shows inter-procedural data dependencies identified when the algorithm [HaSo 89] is applied to the program in Fig. 6.1.

Table 6.1. Inter-procedural Data Dependencies among Main, GetMax, and PairMax

Variable	Def	Use(s)
N	d_1^N	$p_{11,12}^H, p_{11,16}^H, c_{12}^H,$ $p_{21,22}^J, p_{21,23}^J, c_{23}^J$
Max	d_{22}^K d_{23}^K	c_5^{Max} c_5^{Max}
S	d_3^S	$p_{21,22}^I, p_{21,23}^I, c_{22}^I,$ $p_{21,22}^J, p_{21,23}^J, c_{23}^J$

Once the inter-procedural data flow information is computed and the required data flow units are identified, test paths that cover these data flow units can be obtained in the same way as for intra-procedural data flow testing [HaSo 89].

6.3 Extended Def-Use Graph: The Proposed Flow Model

The IFG model described in the previous section is inspired by a similar approach [Call 88] for inter-procedural data flow analysis. Unfortunately, this model is not suitable for testing purposes. It is incapable of capturing data flow chains which span across procedures. The IFG model does not distinguish inputs from other defs and outputs from other uses. Furthermore, the model can not be used directly to guide the selection of test paths due to the fact that paths in IFG are not execution paths. In addition, only inter-procedural data dependencies are computed. From the viewpoint of data flow testing, test paths should be selected to exercise both intra-procedural and inter-procedural data dependencies which are interrelated, since it is not very meaningful to select test paths to cover only intra-procedural or inter-procedural data dependencies, or to cover one first with some paths and then additional paths are selected to cover the other.

In this section, we propose a model to overcome the above problems. The new model which we call the extended DUG (EDUG) model is based on the DUG model and the super graph model.

6.3.1 The Super Graph Model

Super graph [Myer 81] is based on the flow graph model by incorporating inter-procedural flow information. A super graph is capable of modeling not only the control flow within procedures in a program, but also the effects induced by inter-procedural mechanisms such as pass-by-reference, recursion, and procedure nesting.

In a super graph, each procedure in a program, including the main program, is represented by a flow graph with unique entry and exit, and a procedure call in the procedure is represented by a separate node N in the flow graph. The **super graph** for the program consists of all such flow graphs by splitting each node N into two nodes: a **call**

node C and a **return node** C', such that the incoming edges of C are those incoming edges of N, and the outgoing edges of C' are those outgoing edges of N. A **call edge** is added from the call node C to the entry node of the called procedure, and a **return edge** is added from the exit node of the called procedure to the return node C'. Note that the actual parameters in a procedure call are associated with the corresponding call node.

The super graph for the Pascal example in Fig. 6.1 is given in Fig. 6.4. Clearly, the supergraph model is not suitable for representing data flow in terms of explicit defs and uses of each variable and do not handle procedure calls and returns in a manner suitable for data flow testing. The super graph model does not explicitly represent data flow information. It is language dependent and would need translation rules.

In the following subsection, we extend the DUG model by incorporating the concepts from supergraph model into a new model, called extended DUG (or EDUG in short) to represent the control flow as well as both intra-procedural and inter-procedural data flow in a given subprogram.

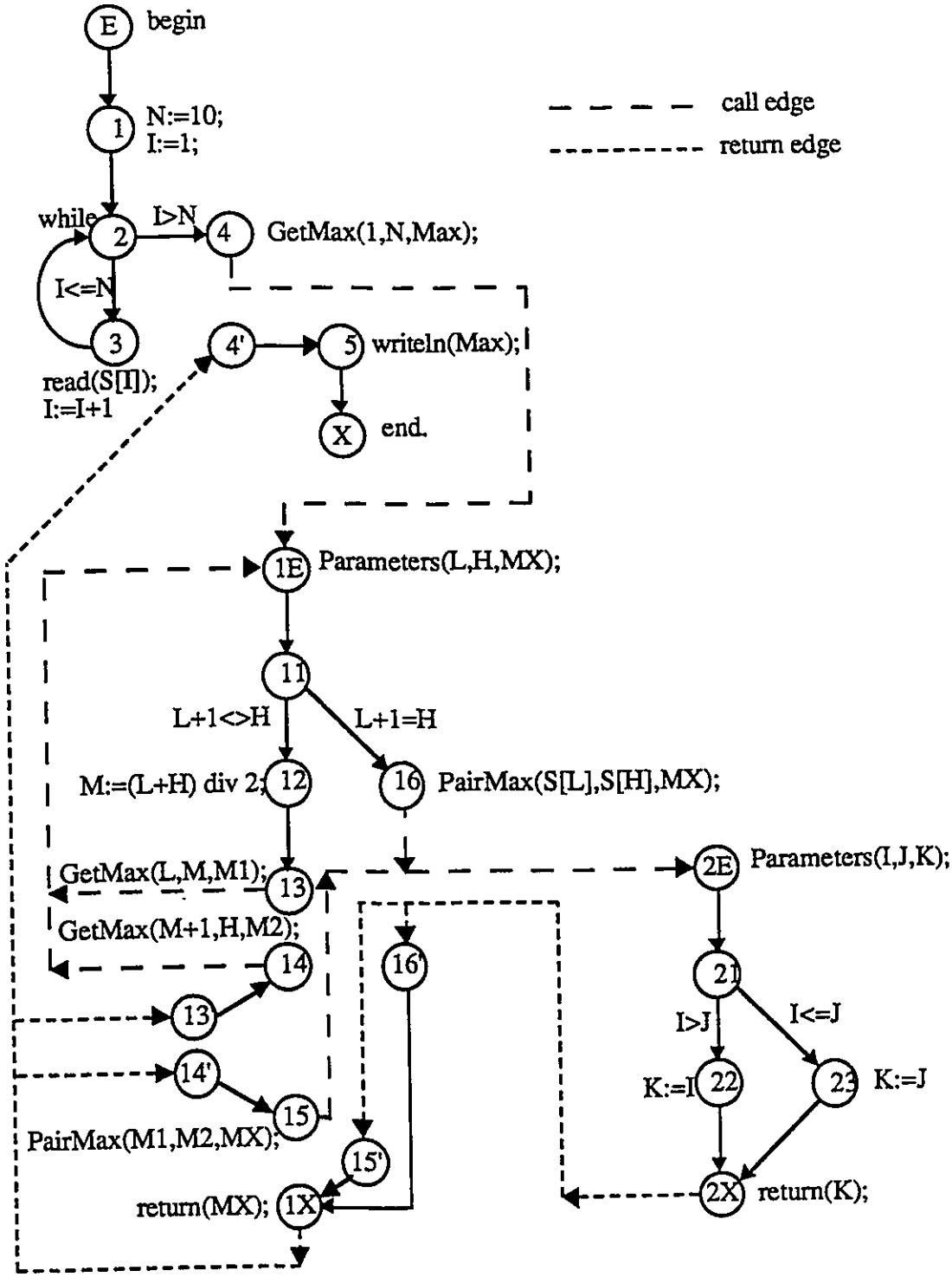


Figure 6.4. The Super Graph for the Pascal Example in Fig. 6.1

6.3.2 The EDUG Model

Here, we propose a program model which is based on the DUG model and super graph [Myer 81]. We refer to the new model as extended DU graph (EDUG). An EDUG consists of a DUG for each procedure, including the main procedure. Similar to the super graph model, each call site is represented by a separate node N in the DUG that contains the call. The EDUG for a subprogram consists of all the DUGs for procedures in the subprogram by splitting each node N into two nodes: a call node C and a return node C', such that the incoming edges of C are those incoming edges of N, and the outgoing edges of C' are those outgoing edges of N. A call edge is added from the call node C to the entry node of the called procedure, and a return edge is added from the exit node of the called procedure to the return node C'. The call edge form C represents the call to the procedure and the return edge from C' represents the return from the called procedure. Note that the actual parameters in a procedure call are associated with the corresponding call node.

6.3.3 Representing Pascal Programs with EDUG

In Chapter 3, we have shown how a procedure written in a large subset of ISO Standard Pascal can be represented by the def-use graph model. The drawback of such an approach is that this model can not represent inter-procedural data dependencies accurately, even with the assumptions made in Section 3.3 for the explicit relationship between parameters. The proposed EDUG model is well suited for fulfilling the task. It is capable of modeling not only the control and data flow within procedures in a subprogram, but also the effects induced by inter-procedural mechanisms such as pass-by-reference, recursion, and procedure nesting as well as the use of global variables.

The translation rules from Pascal programming constructs to EDUG constructs are similar to those for the def-use graph model. When the EDUG model is used, we have a more precise interpretation of procedure statements than that of [FrWe 88, ClPo 89]. Recall

that, as shown in Table 3.2, procedure statement $P(e_1, ..., e_n)$ is, when the def-use model is used, interpreted as c-uses of all variables occurring in the parameter list $(e_1, ..., e_n)$, followed by defs of all variables corresponding to var formal parameters.

On the other hand, in the EDUG model, the called procedure is associated with the calling procedure in three steps: First of all, each call statement is represented by a call node C and a return node C', a call edge from C to the entry node P_s of procedure P is added, and a return edge from the exit node P_t of procedure P to C' is also added. Second, the call node C is associated with c-uses of all variables occurring in the actual input parameter list $(e_1, ..., e_n)$, followed by defs of their corresponding formal parameters $(f_1, ..., f_n)$ in the definition of procedure P. Finally, the exit node P_t is associated with c-uses of formal pass-by-reference parameters followed by defs of their corresponding actual parameters.

In this way, the aliasing (pass-by-reference) and procedure nesting problems are resolved due to the handling of procedure calls and returns. As result, global variables are made local to the called procedures. The EDUG for the Pascal subprogram of Fig.6.1 is given in Fig.6.5. Recursive calls need to be interpreted when the EDUG model for a subprogram is used. As shown in Fig.6.5, only the last def of M in node 12 can reach its use in node 14. This must be interpreted for recursive calls to procedure GetMax in the implementation of a data flow oriented path selection criterion by using different names (e.g., using an array to index each recursive call).

It is important to note that the EDUG model introduces more infeasible paths: a path containing the call edge of a procedure may be followed in the path by the return edge of a different procedure. e.g., path (s,1,2,4,1s,11,16,2s,21,22,2t,15',1t,4',5,t). However, selection of these paths can be easily and naturally avoided by using a stack to keep track of the sequence of procedure calls. For the above calling sequence, the stack contents would

be [4-1s,16-2s]. Because the top of the stack is call edge 16-2s, the next return edge should be 2t-16' instead of 2t-15'.

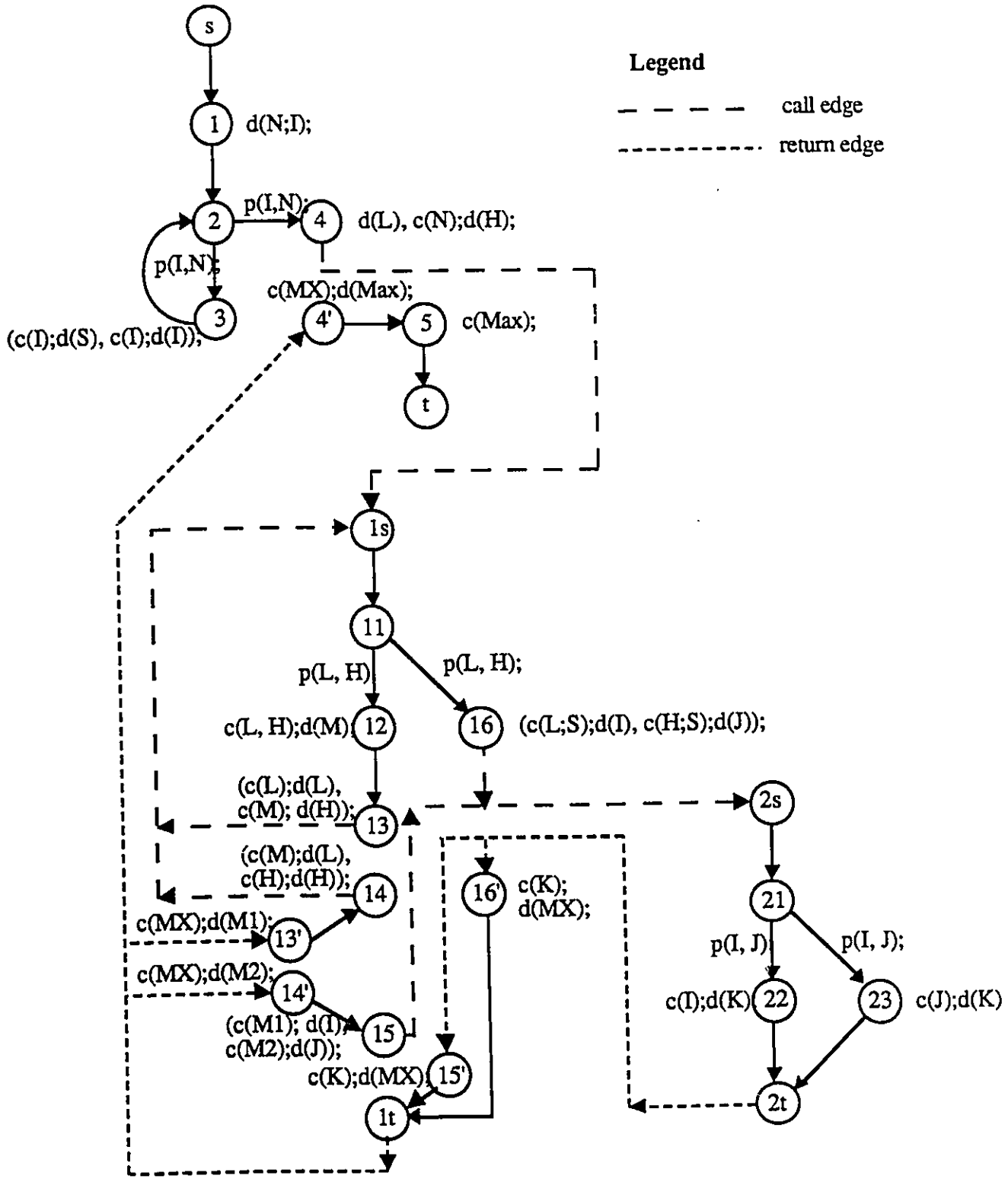


Figure 6.5. The EDUG for the Pascal Example in Fig. 6.1

6.4 Data Flow Testing Using the EDUG Model

In this section, we demonstrate the usefulness of the EDUG model by showing how it improves the existing data flow oriented path coverage criteria in terms of their data flow coverage. As examples, we compare the coverage of two data flow oriented path selection criteria in the DUG and the EDUG models, namely the all-uses criterion [RaWe 85] and the IP/O_2-chains criterion. The Pascal subprogram given in Fig.6.1 is used for illustration.

Using the terminology defined in the previous sections, the all-uses criterion can be rephrased as requiring that every du-pair in a DUG for a procedure is covered at least once by the selected test paths. From the DUGs in Fig. 6.2, the du-pairs in Main, GetMax, PairMax are identified and given in Table 6.2, 6.3 and 6.4, respectively.

Table 6.2. du-pairs in Procedure Main of Fig. 6.2

Variable	Def(s)	Use(s)
N	d_1^N	c_4^N, $p_{2,3}^N$, $p_{2,4}^N$
Max	d_4^{Max}	u_5^{Max}
I	d_1^I	c_3^I, $p_{2,3}^I$, $p_{2,4}^I$
	d_3^I	c_3^I, $p_{2,3}^I$, $p_{2,4}^I$

Table 6.3. du-pairs in Procedure GetMax of Fig. 6.2

Variable	Def(s)	Use(s)
L	d_{1s}^L	$p_{11,12}^L$,$p_{11,16}^L$, c_{12}^L,c_{13}^L,c_{16}^L
H	d_{1s}^H	$p_{11,12}^H$,$p_{11,16}^H$, c_{12}^H,c_{14}^H,c_{16}^H

MX	d_{1s}^{MX} d_{15}^{MX} d_{16}^{MX}	c_{15}^{MX}, c_{16}^{MX} c_{1t}^{MX} c_{1t}^{MX}
M	d_{12}^{M}	c_{13}^{M}, c_{14}^{M}
M1	d_{13}^{M1}	c_{15}^{M1}
M2	d_{14}^{M2}	c_{15}^{M1}
S	d_{1s}^{S}	c_{16}^{S}, c_{1t}^{S}

Table 6.4. du-pairs in Procedure PairMax of Fig. 6.2

Variable	Def(s)	Use(s)
I	d_{2s}^{I}	$p_{21,22}^{I}, p_{21,23}^{I}, c_{22}^{I}$
J	d_{2s}^{J}	$p_{21,22}^{J}, p_{21,23}^{J}, c_{23}^{J}$
K	d_{22}^{K} d_{23}^{K}	c_{2t}^{K} c_{2t}^{K}

As shown in Table 6.5, du-pairs in the EDUG model (Fig. 6.5) are more accurate. Thus, coverage of these du-pairs yields more meaningful and, likely, more effective test paths.

Table 6.5. DU-Pairs in Fig. 6.3

Variable	Def(s)	Use(s)
N	d_1^N	$p_{2,3}^N, p_{2,4}^N, c_4^N$
I	d_1^I d_3^I	$p_{2,3}^I, p_{2,4}^I, c_3^I$ $p_{2,3}^I, p_{2,4}^I, c_3^I$
S	d_3^S	c_{16}^S

L	d_4^L d_{13}^L d_{14}^L	$p_{11,12}^L, p_{11,16}^L, c_{12}^L, c_{13}^L, c_{16}^L$ $p_{11,12}^L, p_{11,16}^L, c_{12}^L, c_{13}^L, c_{16}^L$ $p_{11,12}^L, p_{11,16}^L, c_{12}^L, c_{13}^L, c_{16}^L$
H	d_4^H d_{13}^H d_{14}^H	$p_{11,12}^H, p_{11,16}^H, c_{12}^H, c_{14}^H, c_{16}^H$ $p_{11,12}^H, p_{11,16}^H, c_{12}^H, c_{14}^H, c_{16}^H$ $p_{11,12}^H, p_{11,16}^H, c_{12}^H, c_{14}^H, c_{16}^H$
Max	$d_{4'}^{Max}$	c_5^{Max}
M	d_{12}^M	c_{13}^M, c_{14}^M
M1	$d_{13'}^{M1}$	c_{15}^{M1}
M2	$d_{14'}^{M2}$	c_{15}^{M2}
I	d_{15}^I d_{16}^I	$p_{21,22}^I, p_{21,23}^I, c_{22}^I$ $p_{21,22}^I, p_{21,23}^I, c_{22}^I$
J	d_{15}^J d_{16}^J	$p_{21,22}^J, p_{21,23}^J, c_{23}^J$ $p_{21,22}^J, p_{21,23}^J, c_{23}^J$
MX	$d_{15'}^{MX}$ $d_{16'}^{MX}$	$c_{4'}^{MX}, c_{13'}^{MX}, c_{14'}^{MX}$ $c_{4'}^{MX}, c_{13'}^{MX}, c_{14'}^{MX}$
K	d_{22}^K d_{23}^K	$c_{15''}^K, c_{16'}^K$ $c_{15''}^K, c_{16'}^K$

Recall that a set of complete paths P satisfies the IP/O_2-chains coverage criterion if in a DUG for a procedure or an EDUG for a subprogram:

a) for every $(i,o) \in R_{IO}$, an activating path for every IP/O_k-chain $(1 \le k \le 2)$ duc(i,o) is covered at least once by P, and

b) for every (i,p) ∈ R_{IP}, an activating path for every IP/O_1-chain duc(i,p) that is not covered in a) is covered at least once by P.

From the DUGs in Fig.6.2, IP/O_1 and IP/O_2-chains in Main, GetMax, PairMax are identified and given in Table 6.6, 6.7 and 6.8, respectively. As shown in Table 6.9, IP/O_1 and IP/O_2-chains in the EDUG model (Fig. 6.3) are more accurate.

Table 6.6. IP/O-Chains in Main

$[d_1^N, p_{2,3}^N]$
$[d_1^N, p_{2,4}^N]$
$[d_1^N, c_4^N d_4^{Max}, c_5^{Max}]$
$[d_1^I, p_{2,3}^I]$
$[d_1^I, p_{2,4}^I]$
$[d_1^I, c_3^I d_3^I, p_{2,3}^I]$
$[d_1^I, c_3^I d_3^I, p_{2,4}^I]$
$[d_1^I, c_3^I d_3^I, c_3^I d_3^I, p_{2,3}^I]$
$[d_1^I, c_3^I d_3^I, c_3^I d_3^I, p_{2,4}^I]$

Table 6.7 IP/O-Chains in GetMax

$[d^{L}_{1s}, p^{L}_{11,12}]$

$[d^{L}_{1s}, p^{L}_{11,16}]$

$[d^{H}_{1s}, p^{H}_{11,12}]$

$[d^{H}_{1s}, p^{H}_{11,16}]$

$[d^{L}_{1s}, c^{L}_{12}d^{M}_{12}, c^{M}_{13}d^{M1}_{13}, c^{M1}_{15}d^{MX}_{15}, c^{MX}_{1t}]$

$[d^{L}_{1s}, c^{L}_{12}d^{M}_{12}, c^{M}_{14}d^{M2}_{14}, c^{M2}_{15}d^{MX}_{15}, c^{MX}_{1t}]$

$[d^{L}_{1s}, c^{L}_{13}d^{M1}_{13}, c^{M1}_{15}d^{MX}_{15}, c^{MX}_{1t}]$

$[d^{L}_{1s}, c^{L}_{16}d^{MX}_{16}, c^{MX}_{1t}]$

$[d^{H}_{1s}, c^{H}_{12}d^{M}_{12}, c^{M}_{13}d^{M1}_{13}, c^{M1}_{15}d^{MX}_{15}, c^{MX}_{1t}]$

$[d^{H}_{1s}, c^{H}_{12}d^{M}_{12}, c^{M}_{14}d^{M2}_{14}, c^{M2}_{15}d^{MX}_{15}, c^{MX}_{1t}]$

$[d^{H}_{1s}, c^{H}_{14}d^{M2}_{14}, c^{M2}_{15}d^{MX}_{15}, c^{MX}_{1t}]$

$[d^{H}_{1s}, c^{H}_{16}d^{MX}_{16}, c^{MX}_{1t}]$

$[d^{MX}_{1s}, c^{MX}_{16}d^{MX}_{16}, c^{MX}_{1t}]$

$[d^{MX}_{1s}, c^{MX}_{15}d^{MX}_{15}, c^{MX}_{1t}]$

$[d^{S}_{1s}, c^{S}_{16}d^{MX}_{16}, c^{MX}_{1t}]$

$[d^{S}_{1s}, c^{S}_{1t}]$

Table 6.8. IP/O-Chains in PairMax

$[d^{I}_{2s}, p^{I}_{21,22}]$
$[d^{I}_{2s}, p^{I}_{21,23}]$
$[d^{I}_{2s}, c^{I}_{22}d^{K}_{22}, c^{K}_{2t}]$
$[d^{J}_{2s}, p^{J}_{21,22}]$
$[d^{J}_{2s}, p^{J}_{21,23}]$
$[d^{J}_{2s}, c^{J}_{23}d^{K}_{23}, c^{K}_{2t}]$

Table 6.9. IP/O-Chains in Fig. 6.3

$[d_1^N, p_{2,3}^N]$

$[d_1^N, p_{2,4}^N]$

$[d_1^N, c_4^N d_4^H, p_{11,12}^H]$

$[d_1^N, c_4^N d_4^H, p_{11,16}^H]$

$[d_1^N, c_4^N d_4^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_1^N, c_4^N d_4^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_1^N, c_4^N d_4^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,12}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,16}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX},$
$c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,12}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,16}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H c_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I,$
$p_{21,22}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I,$

$p_{21,23}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I,$

$c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, p_{11,12}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, p_{11,16}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, c_{22}^I d_{22}^K,$

$c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,12}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,16}^H]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1},$

$c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1},$

$c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1},$

$c_{15}^{M1} d_{15}^I, c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{14}^M d_{14}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^J, p_{21,22}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{14}^M d_{14}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^J, p_{21,23}^J]$

$[d_1^N, c_4^N d_4^H, c_{12}^H d_{12}^M, c_{14}^M d_{14}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^J, c_{23}^J d_{23}^K, c_{15'}^K d_{15'}^{MX},$

$c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^I, p_{2,3}^I]$

$[d_1^I, p_{2,4}^I]$

$[d_1^I, c_3^I d_3^I, p_{2,3}^I]$

$[d_1^I, c_3^I d_3^I, p_{2,4}^I]$

$[d_1^I, c_3^I d_3^I, c_3^I d_3^I, p_{2,3}^I]$

$[d_1^I, c_3^I d_3^I, c_3^I d_3^I, p_{2,4}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, p_{21,22}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, p_{21,23}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^I, p_{21,22}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^I, p_{21,23}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^I, c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, p_{21,22}^J]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, p_{21,23}^J]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^I, p_{21,22}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^I, p_{21,23}^I]$

$[d_1^I, c_3^I d_3^S, c_{16}^S d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^I, c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_4^L, p_{11,12}^L]$

$[d_4^L, p_{11,16}^L]$

$[d_4^L, c_{16}^L d_{16}^I, p_{21,22}^I]$

$[d_4^L, c_{16}^L d_{16}^I, p_{21,23}^I]$

$[d_4^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, p_{11,12}^H]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, p_{11,16}^H]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,12}^H]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, p_{11,16}^H]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, c_{22}^I d_{22}^K,$

$c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, p_{11,12}^H]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, p_{11,16}^H]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, p_{21,22}^I]$

$[d_4^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, p_{21,23}^I]$

$[d_4^L, c_{12}^L d_{12}^M, c_{14}^M d_{14}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^J, p_{21,22}^J]$

$[d_4^L, c_{12}^L d_{12}^M, c_{14}^M d_{14}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^J, p_{21,23}^J]$

$[d_4^L, c_{12}^L d_{12}^M, c_{14}^M d_{14}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{14'}^{MX} d_{14'}^{M2}, c_{15}^{M2} d_{15}^J, c_{23}^J d_{23}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_4^L, c_{13}^L d_{13}^L, p_{11,12}^L]$

$[d_4^L, c_{13}^L d_{13}^L, p_{11,16}^L]$

$[d_4^L, c_{13}^L d_{13}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, p_{21,22}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, p_{21,23}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, p_{11,12}^H]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, p_{11,16}^H]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I,$

$c_{22}^I d_{22}^K, c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, p_{11,12}^H]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, p_{11,16}^H]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,22}^J]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, p_{21,23}^J]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,22}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I, p_{21,23}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{12}^H d_{12}^M, c_{13}^M d_{13}^H, c_{16}^H d_{16}^J, c_{23}^J d_{23}^K, c_{16'}^K d_{16'}^{MX}, c_{13'}^{MX} d_{13'}^{M1}, c_{15}^{M1} d_{15}^I,$
$c_{22}^I d_{22}^K c_{15'}^K d_{15'}^{MX}, c_{4'}^{MX} d_{4'}^{Max} c_5^{Max}]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, p_{11,12}^H]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, p_{11,16}^H]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, p_{21,22}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{13}^M d_{13}^H, c_{14}^H d_{14}^H, c_{16}^H d_{16}^J, p_{21,23}^I]$

$[d_4^L, c_{13}^L d_{13}^L, c_{12}^L d_{12}^M, c_{14}^M d_{14}^L, c_{16}^L d_{16}^I, c_{22}^I d_{22}^K, c_{16'}^K d_{16'}^{MX}, c_{4'}^{MX} d_{4'}^{Max}, c_5^{Max}]$

As indicated by the above table, more accurate data flow testing is achieved if the proposed IP/O_2-chains criterion is used. It is important to note that the subprogram shown in Fig. 6.1 contains a typical boundary error: the condition in line 13 should be L+1>=H instead of L+1=H, otherwise the subprogram may get into infinite recursion. The necessary and sufficient condition for the error to be detected is that the size of the array is set to an odd number. As indicated in Table 6.9, the IP/O_2-chains criterion guarantee that this condition is met. This is due to the fact that IP/O_1-chain $[d_1^I, c_3^I d_3^I, p_{2,4}^I]$ is required to be covered and its coverage requires the value of N be set to 1. As for the well-known data flow oriented path selection criteria, only required 3-tuple criterion also guarantees that this condition is satisfied. Both the all-uses and the all-du-paths criteria do not guarantee the error be detected as they consider only individual du-pairs. Because only single variables

are involved in node 3, the context coverage criteria also fail to guarantee the detection of the error.

Chapter 7

Conclusions and Future Research

This chapter summarizes the previous chapters and indicates the points where further research is needed.

Software testing is one of the most widely used quality assurance methodologies. A large software system usually has a hierarchical structure: for example, system, subsystems (programs), subprograms, and procedures. Testing of the system can be done at different levels with different approaches and emphases. Testing at the two lower levels is the focus of this thesis. A class of data flow oriented path selection criteria is proposed. Among them, the most useful criterion is the IP/O_2-chains criterion.

7.1 Summary of the Thesis

In this thesis, first the state-of-the-art flow model based testing techniques are reviewed. Although our focus is on selecting test paths using the flow models which capture the control as well as data flow in a piece of code, a survey of the existing FSM-based testing techniques is also included for the sake of completeness.

Two flow models are described, namely the def-use graph (DUG) for a procedure and the extended def-use graph (EDUG) for a subprogram. With a particular set of translation rules, the former can be used to represent the control flow and data flow within a procedure, whereas the latter is capable of doing the same among procedures in a subprogram.

Based on the def-use graph model, we have proposed the IP/O-chains criteria and their variants. These criteria are shown to have certain advantages over the existing data flow oriented path selection criteria.

In the data flow testing literature, testing is targeted at the procedure level. We believe that it is more meaningful, while still manageable, to perform data flow testing at subprogram level. To do so, it is necessary to have a model which captures both intra-procedural and inter-procedural control and data flow. The EDUG model is shown to fulfil such a need. With this model, both the existing and the proposed criteria can be used more accurately for selecting test paths at subprogram level.

The major contributions of the thesis are:

1) Conception and formalization of the notion "a variable definition affects a variable use". With this notion, we are able to clearly and precisely define data flow chains.

2) Emphasis of the importance of identifying how inputs affect outputs or predicates in a program. Definition of input-output and input-predicate relations are given using the notion of "affect". We argue that it is necessary to make use of these two relations as the basis of test path selection.

3) Introduction of IP/O-chains criteria. These criteria are designed to select paths such that a) every element in the input-output and input-predicate relations is covered at least once; b) minimum data flow requirements are satisfied (i.e., every definition-use pair is covered at least once); and c) minimum control flow requirements are satisfied (i.e., every branch in a flow model is covered at least once).

4) Comparisons of IP/O-chains criteria to the well-known data flow oriented path selection criteria. The IP/O-chains criteria are shown to strictly include the all-uses criterion, but they are incomparable with the all-du-paths criterion and the context coverage criteria. It is shown that, for any k, there exists an IP/O-chains criterion which strictly includes the required k-tuples criterion.

5) Introduction of some useful variations of the IP/O-chains criteria, namely, IP/O_2-chains$^+$ criterion and IP/O_2-chains* criterion, both meet the fore-mentioned design objectives. The IP/O_2-chains$^+$ criterion is shown to strictly include the context coverage criterion, and the IP/O_2-chains* criterion is shown to strictly include the all-du-paths criterion

6) Introduction of the extended def-use graph model for testing at subprogram level. We believe that it is more meaningful, while still manageable, to perform data flow testing at subprogram level. The extended def-use graph model is shown to be suitable for modelling and testing both intra-procedural and inter-procedural control and data flow.

Existing data flow oriented test path selection criteria may use this model to yield more meaningful and more accurate test paths.

7.2 Future Research

In this thesis, we have focused on test path selection based on data flow information. We have proposed a class of test path selection criteria by analyzing the data flow relationship between program inputs and program outputs or predicates. An unfortunate characteristic of the proposed criteria as well as any other white-box testing method is the existence of infeasible paths. Selecting these paths will definetely affect the accuracy and applicability of a path selection criterion. On the other hand, it's well-known that the problem of determining the feasibility of a given path in a given program is unsolvable, being equivalent to the Halting Problem. Some effort has been made to tackle this problem by using program semantics to suppress the generation of infeasible paths. In some cases, symbolic evaluation has been shown to be an effective means to solve path constraints for selecting only feasible paths. Some heuristic procedures have also been proposed to deal with this problem. In a approach, all pairs of conflicting conditions, known as impossible pairs, are identified. Test paths are then selected in a way so that no impossible pair is covered by any selected test path.

A recent paper by Frankl and Weyuker [FrWe 88] studied the applicability of the well-known data flow oriented path selection criteria. It is shown that for a given program, there may be no feasible paths which cover test units (e.g., du-pair, du-path, etc.) required by a particular criterion. Thus, they suggest that a test path selection criterion requires only test units that can be covered by feasible paths. Such an approach may change the inclusion hierarchy among test path selection criteria.

The inclusion relation used to compare the relative strength of test path selection criteria may not reflect the actual effectiveness of these criteria in detection of errors [Haml 89]. As some studies have shown [Ntaf 88, Weyu 90], empirical comparisons of path selection criteria may be more acurate in revealing the actual effectiveness as well as the actual cost in terms of how many test cases are required by a criterion. An empirical study of the proposed criteria seems to be necessary.

Undoubtedly, effective error-detection rely on not only the extent of code coverage, but also the test data chosen for traversing the selected paths. In fact, many errors along a selected path can only be detected if the path is executed with particular values. However, determination of these values needs semantic information, therefore, is outside the scope of white-box testing.

References

[AhDa 88] A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours", in Protocol Specification, Testing and Verification, 8, North-Holland, eds.: S. Aggarwal and K. Sabnani, 1988, pp. 75-86.

[Boch 78] G.v. Bochmann, "Finite state description of communication protocols", Computer Communications, Vol. 2, No. 4-5, Sep.-Oct. 1978, pp. 361-372.

[BoSu 80] G.v. Bochmann and C.A. Sunshine, "Use of formal methods in communication protocol design", IEEE Trans. Comm., Vol. 28, No. 4, Apr. 1980, pp. 624-631.

[BoSu 83] G.v. Bochmann and C.A. Sunshine, "A survey of formal methods", in Computer Network Architectures and Protocols, Plenum Press, New York. Ed.: P. E. Green, 1983, pp. 561-578.

[BoUr 90] S. Boyd and H. Ural, "On FSM-based optimal test sequence generation" summitted for publication, 1990.

[Call 88] D. Callahan, "The program summary graph and flow-sensitive inter-procedural data flow analysis", Proc. of SIGPLAN'88 Conf. on Programming Language Design and Implementation, Atlanta, Ga, June 1988, pp. 47-56.

[Chen 90] M.-S. Chen, Y. Choi, and A. Kershenbaum, "Approaches utilizing segment overlap to minimize test sequences", in Protocol Specification, Testing, and Verification, 10, North-Holland, ed. L. Logrippo, R.L. Probert, and H. Ural, 1990, pp.85-98.

[ChVu 89] W.Y.L. Chan, S. Vuong, and M.R. Ito, "An improved protocol test generation procedure based on UIOS", in Proc. ACM SIGCOMM'89, Sep. 19-22, Austin, Texas, 1989, pp.283-294.

[Chow 78] T.S. Chow, "Testing software design modeled by finite-state machines", IEEE Trans. Software Eng., Vol. 4, No. 3, May 1978, pp. 178-187.

[Clar 76] L.A. Clarke, "A System to generate test data and symbolically execute programs", IEEE Trans. Software Eng., Vol. 2, No. 3, Sep. 1976, pp. 215-222.

[ClPo 85] L.A. Clarke, A. Podgurski, D. Richardson, and S.J. Zeil, "A comparison of data flow path selection criteria", in Proc. 8th Int'l Conf. on Software Eng., Aug. 1985, pp. 244-251.

[ClPo 89] L.A. Clarke, A. Podgurski, D. Richardson, and S.J. Zeil, "A formal evaluation of data flow path selection criteria", IEEE Trans. Software Eng., Vol. 15, No. 11, Nov. 1989, pp. 1318-1332.

[DaSh 79] S.R. Das, C.L. Sheng, Z. Chen , and W.J. HSU, " Transition matrices in the measurement and control of synchronous sequential machines", Inf. Sc., 1979, 18, pp. 47-65.

[EdJo 73] J. Edmonds and E.L. Johnson, "Matching, Euler tours and the Chinese postman", Mathematical Programming, Vol. 5, 1973, pp. 88-124.

[EdKa 72] J. Edmonds and R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems", J. ACM, Vol. 19, No. 2, 1972, pp. 248-264.

[FeOt 87] J. Ferrante, K.J. Ottenstein and J.D. Warren, "The program dependence graph and its use in optimization", ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July 1987, pp. 319-349.

[Fran 90] P.G. Frankl, "Partial symbolic evaluation of path expressions", Technical Report PUCS-105-90, Dept. of Computer Science, Polytechnic University, Brooklyn, NewYork, 1990.

[FrWe 88] P.G. Frankl and E.J. Weyuker, "An applicable family of data flow testing criteria", IEEE Trans. Software Eng. Vol.14, No.10, Oct.1988, pp. 1483-1498.

[Gill 62] A. Gill, Introduction to the theory of finite-state machines, McGraw-Hill, New York, 1962.

[Gone 70] G. Gonenc, "A method for the design of fault detection experiments", IEEE Trans. Computers, Vol. 19, No. 6, Jun. 1970, pp. 551-558.

[Haml 89] R. Hamlet, "Theoretical comparison of testing methods", in: Proc. ACM SIGSOFT 3rd Symposium on Software Testing, Analysis, and Verification, Dec. 1989, pp. 28-37.

[HaSo 89] M.J. Harrold and M.L. Soffa, "Interprocedural data flow testing", in: Proc. of 3rd Symposium on Testing, Analysis, and Verification, Key West, Florida, Dec. 1989, pp. 158-167.

[Hech 77] M.S. Hecht, Flow analysis of computer programs, North-Holland, New York, 1977.

[Henn 64] F.C. Hennie, "Fault-detecting experiments for sequential circuits", in Proc. 5th Annual Symp. on Switching Circuit Theory and Logical Design, Nov. 1964, pp. 95-110.

[Herm 76] P.M. Herman, "A data flow analysis approach to program testing", Australian Computer J., Vol. 8, No. 3, Nov. 1976, pp. 92-96.

[HeWo 76] M.A. Hennell, M.R. Woodward, and D. Hedley, "On program analysis", Information Processing Letters, Vol.5, 1976, pp. 136-140.

[Howd 75] W.E. Howden, "Methodology for the generation of program test data", IEEE Trans. Computers, Vol. 24, No. 5, May 1975, pp. 554-559.

[Howd 76] W.E. Howden, "Reliability of the path analysis testing strategy", IEEE Trans. Software Eng., Vol. 2, No. 3, Sep. 1976, pp. 208-215.

[Howd 77] W.E. Howden, "Symbolic testing-design techniques, costs, and effectiveness", NTIS PB-268518, May 1977.

[Howd 87] W.E. Howden, Functional program testing and analysis, McGraw Hill, New York, 1987.

[Huan 79] J.C. Huang, "Detection of data flow anomalies through program instrumentation", IEEE Trans. Software Eng., Vol. SE-5, No.3, May, 1979, pp. 226-236.

[ISO 82] International Organization for Standardization, Specification for computer programming language Pascal, ISO 7185-1982, 1982.

[JeWi 85] K. Jensen and N. Wirth, Pascal user manual and report, Springer-Verlag, New York, 3rd edition, 1985.

[KaLo 86] B. Kanungo, L. Lamout, R.L. Probert, and H. Ural, "A useful FSM representation for test suite design and development", in: Protocol Specification, Testing and Verification, 6, North-Holland, eds.: B. Sarikaya and G.v. Bochmann, 1986. pp. 163-176.

[Koha 78] Z. Kohavi, Switching and finite automata theory, McGraw-Hill, New York, 1978.

[Kore 87] B. Korel, "The program dependence graph in static program testing", Information Processing Letters, Vol. 24, Jan. 1987, pp. 103-108.

[KrSm 73] K.A. Kranse, R.W. Smith and M.A. Goodwin, "Optimal software test planning through automated network analysis", in: Proc. IEEE Symp. Computer Software Reliability, New York, 1973, pp. 18-22.

[Kuan 62] M.-K. Kuan, "Graphic programming using odd or even points", Chinese Math., Vol. 1, 1962, pp. 273-277.

[Lask 82] J. Laski, "On data flow guided program testing", ACM SIGPLAN Notices, Vol. 17, Sep., 1982, pp. 62-71.

[LaKo 83] J.W. Laski and B. Korel, "A data flow oriented program testing strategy", IEEE Trans. Software Eng., Vol. 9, No. 3, May 1983, pp. 347-354.

[Lome 77] D. Lomet, "Data flow analysis in the presence of procedure calls", IBM J. Research & Development, Vol. 21, No. 6, Nov. 1977. pp. 559-571.

[McRi 77] J. McCall, P. Richards and G. Walters, "Factors in software quality", NTIS AD-A 049-o14, 015, 055, Nov. 1977.

[MiHo 81] E. Miller and W.E. Howden, Software testing and validation techniques, IEEE Tutorial, 2nd ed., IEEE Computer Society Press, Los Alamitos, 1981.

[MiPa 90] R.E. Miller and S. Paul, "Generating minimal length test sequences for conformance testing of communication protocols", Proc. of 3rd. Int. Workshop on Protocol Test Systems, 1990.

[Myer 79] G. J. Myers, The art of software testing, John Wiley & Sons, New York, 1979.

[Myer 81] E.W. Myers, "A precise inter-procedural data flow algorithm", 8th Ann. ACM Symp. on Principle of Programming Languages, 1981, pp.219-230.

[NaTs 81] S. Naito and M. Tsunoyama, "Fault detection for sequential machines by transition tours", In Proc. 11th IEEE Fault Tolerant Computing Symp., IEEE Comput. Soc. Press, 1981, pp. 238-243.

[Ntaf 84] S.C. Ntafos, "On required element testing", IEEE Trans. Software Eng., Vol. 10, No.6, Nov. 1984, pp. 795-803.

[Ntaf 88] S.C. Ntafos, "A comparison of some structural testing strategies", IEEE Trans. Software Eng., Vol. 14, No. 6, Jun. 1988, pp. 868-874.

[Oste 77] L.J. Osterweil, "The detection of unexecutable program paths through static data flow analysis", in Proc. COMPSAC'77, 1977, pp. 406-413.

[Pete 77] J.L. Peterson, "Petri nets", ACM Computing Surveys, Vol. 9, No. 3, Sep. 1977, pp. 223-252.

[Pete 81] J.L. Peterson, Petri nets theory and modeling of systems, Prentice-Hall, New York, 1981.

[Pres 87] R.S.Pressman, Software Engineering: a Practioner's Approach, 2nd ed., McGraw-Hill, 1987.

[PrMy 87] R.E. Prather and J.P. Myers, Jr., "The path prefix software testing strategy", IEEE Trans. Software Eng., Vol. 13, No. 7, Jul. 1987, pp.761-766.

[Prob 81] R.L. Probert, "Optimal insertion of software probes in well-delimited programs", IEEE Trans.Software Eng., Vol. SE-8, No. 1, Jan. 1981. pp. 34-42.

[Prob 82] R.L. Probert, "Grey-box (design-based) testing techniques", In Proc. of 11th Hawaii Int'l Conference on System Science, 1982, pp. 94-102.

[RaWe 85] S. Rapps and E. Weyuker, "Selecting software test data using data flow information", IEEE Trans. Software Eng., Vol. 11, No. 4, Apr. 1985, pp. 367-375.

[SaBo 84] B. Sarikaya and G.v. Bochmann, "Synchronization and specification issues in protocol testing", IEEE Trans. Communications, Vol. 32, No. 4, 1984, pp. 389-395.

[SaBo 87] B. Sarikaya, G.v. Bochmann, and E. Cerny, "A test design methodology for protocol testing", IEEE Trans. Software Eng., Vol. 13, No. 5, May 1987, pp. 518-531.

[SaDa 85] K.K. Sabnani and A.T. Dahbura, "A new procedure for generating protocol tests", in Proc. 9th Data Communications Symp., Sep. 1985, pp. 36-43.

[SaDa 88] K.K. Sabnani and A.T. Dahbura, "A protocol test generation procedure", Computer Networks and ISDN Systems, Vol. 15, No. 4, 1988, pp. 285-297.

[ShLo 89] Y.N. Shen, F. Lombardi, and A.T. Dahbura, "Protocol conformance testing using multiple UIO sequences", in Protocol Specification, Testing and Verification, 9, North-Holland, Jun. 1989, pp. 2-13.

[SiLe 89] D.P. Sidhu and T.K. Leung, "Formal methods for protocol testing: a detailed study", IEEE Trans. Software Eng., Vol. 15, No. 4, Apr. 1989, pp. 413-426.

[Stuc 73] L. G. Stucki, "Automatic generation of self-metric software", in: Proc. IEEE Symposium on Computer Reliability, New York, Apr. 1973, pp. 94-100.

[Ural 87] H. Ural, "Test sequence selection based on static data flow analysis", Computer Communications, Vol. 10, No. 5, 1987, pp. 234-242.

[UrYa 88] H. Ural and B. Yang, "A structural test selection criterion", Information Processing Letters, Vol. 28, No. 3, Jul. 1988, pp. 157-163.

[UrYa 91] H. Ural and B. Yang, "A protocol conformance test generation method", to appear in: IEEE Trans. on Communications, Apr. 1991.

[UyDa 86] M.U. Uyar and A.T. Dahbura, "Optimal test sequence generation for protocols: the Chinese postman algorithm applied to Q.931", in Proc. IEEE Global Telecomm. Conf., 1986, pp. 68-72.

[WeGa 85] M.D. Weiser, J.D. Gannon, and P.R. McMullin, "Comparison of structural test coverage metrics", IEEE Software, Vol. 2, Mar. 1985, pp. 80-85.

[West 78] C. West, "General technique for communication protocol validation", IBM J. Res. & Develop., Vol. 22, No. 4, July 1978, pp. 393-404.

[Weyu 84] E. J. Weyuker, "The complexity of data flow criteria for test data selection", Information Processing Letters, Vol. 19, Aug. 1984, pp. 103-109.

[Weyu 90] E. J. Weyuker, "The cost of data flow testing: an empirical study", IEEE Trans. Software Eng., Vol. 16, No. 2, Feb. 1990, pp. 121-128.

[WoHe 80] M.R. Woodward, D. Hedley, and M.A. Hennell, "Experience with path analysis and testing of programs", IEEE Trans. Software Eng., Vol. 6, No. 3, May. 1980, pp. 278-286.

[Yang 88] B. Yang, "A test selection strategy based on input-output relation analysis", M.C.S. Thesis, Computer Science Dept., Univ. of Ottawa, Ottawa, Canada, Mar. 1988.

[YaUr 89] B. Yang and H. Ural, "On data flow oriented program testing", in: Proc. of BISYCP'90, Aug. 1989, pp. 349-356.

[YaUr 90a] B. Yang and H. Ural, "Protocol conformance test generation using multiple UIO sequences with overlapping ", in: Proc. of ACM SIGCOMM'90, Computer Communications Review, Vol. 20, No. 4, Sep. 1990, pp. 118-125.

[YaUr 90b] B. Yang and H. Ural, "Test path selection using input-output & input-predicate relations", in: Proc. of InfoJapan'90, Oct. 1990, pp. 193-200.

[YaUr 90c] B. Yang and H. Ural, "On FSM-based optimum test sequence generation", in: Proc. of ICCS'90, Nov. 1990, pp. 2.2.1-2.2.5.

[Zafi 78] P. Zafiropulo, "Protocol validation by duologue matrix analysis", IEEE Trans. on Communications, Vol. 26, 1978, pp. 1187-1194.