Structural-based Testing Methodologies for Visual Dataflow Languages

by

Marcel R. Karam

Submitted in partial fulfilment of the requirements for the degree of

DOCTOR of PHILOSOPHY

Major Subject: Computer Science

at

DALHOUSIE UNIVERSITY

Halifax, Nova Scotia

December, 2001.

©by Marcel R. Karam, 2001

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-77596-8

Dalhousie University

Faculty of Computer Science

The undersigned hereby certify that they have examined, and recommended to the Faculty of Graduate studies for acceptance, the Thesis entitled "Structural-based Testing Methodologies for Visual Dataflow Languages" by Marcel R. Karam in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dated: May 22/02

Supervisor: Dr. Trevor Smedley

External Examiner: Dr. Allen Ambler
University of Kansas, USA

Examiners: Dr. Phil Cox

Dr. William Philips

DALHOUSIE UNIVERSITY

Faculty of Computer Science

"AUTHORITY TO DISTRIBUTE MANUSCRIPT THESIS"

Date: December, 07, 2001.

AUTHOR:	Marcel R. Karam
TITLE:	Structural-based Testing Methodologies for Visual Dataflow Languages
MAJOR SUBJECT:	Computer Science
DEGREE:	Doctor of Philosophy
CONVOCATION:	May, 2002

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above thesis upon the request of individuals or institutions.

Signature of Author

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in this thesis (other than brief excerpts requiring only proper acknowledgment in scholarly writing), and that all such use is clearly acknowledged.

To my beloved parents Ragheb & Helene,

my brother Issam & his family,

and

my brother Jalal.

ALSO

To the precious soul of my uncle Emile.

Table of Contents

LIST OF FIGURES viii

LIST OF TABLES x

LIST OF ABBREVIATIONS xi

ACKNOWLEDGMENTS xii

ABSTRACT xiii

CHAPTER 1 Basic Definitions and Thesis Objectives 1

1.1 Introduction 1
1.2 Fundamentals of Software Testing 2
1.3 What is a Test Adequacy Criterion? 3
1.3.1 Classes of Test Adequacy Criteria 4
1.4 Program-based Structural Testing 6
1.4.1 The Control Flow Graph Model of a Program: An Informal Introduction 6
1.4.2 Control-flow Test Adequacy Criteria. 7
1.4.2.1 The Subsume Relationship 9
1.4.3 Data-flow Test Adequacy Criteria 10
1.5 Programming Paradigm 10
1.6 Thesis Objectives 11

CHAPTER 2 Program-based Structural Testing 13

2.1 Introduction 13
2.2 The Control Flow Graph Model: A Formal Definition 14
2.2.1 Representing The Control Flow Graph 17
2.2.2 Dealing With Loops 17
2.3 Control-flow Adequacy Criteria 18
2.3.1 Statement Coverage 19
2.3.2 Branch Coverage 19
2.3.3 Path Coverage 20
2.3.4 Feasibility 20
2.4 Data-flow Adequacy Criteria 21
2.4.1 Data-flow Information in a CFG 21
2.4.2 Intraprocedural Data-flow Adequacy Criteria 25
2.4.3 The Rapps and Weyuker Criteria Family 26
2.4.3.1 The All-definitions 26
2.4.3.2 The All-uses 27
2.4.3.3 The All-definition-use Paths 28
2.4.3.4 The All-c-uses/Some-p-uses and The All-p-uses/Some-c-uses 29
2.4.4 The Ntafos Criteria 30
2.4.4.1 A k-dr Interaction 30

2.4.4.2 The Required k-tuples Criteria . . . 31
2.4.5 The Laski and Korel Criteria . . . 32
2.4.5.1 The Reach Coverage Criterion . . . 32
2.4.5.2 The Context Coverage Criterion . . . 32
2.4.5.3 The Ordered Context Coverage Criterion . . . 34
2.4.6 Data-flow Testing for Structured and Dynamic Data . . . 35
2.5 The Structural Testing Subsume Hierarchy . . . 36
2.6 Integration Testing . . . 37
2.7 Interprocedural Data-flow Testing . . . 38
2.7.1 Issues in Computing Interprocedural Definition-use Chains . . . 39
2.8 Visual Programming . . . 45
2.9 Prograph; A Brief Overview . . . 47
2.9.1 The Dataflow Computational Model . . . 48
2.9.2 Visual Syntax . . . 48
2.9.3 Editing Environment . . . 49

CHAPTER 3 A Control-flow Testing Methodology for Prograph . . . 50

3.1 Introduction . . . 50
3.2 Dataflow Languages in The context of Prograph . . . 51
3.2.1 The Order of Execution in Prograph . . . 54
3.2.1.1 The Data Dependency . . . 54
3.2.1.2 The Control Dependencies . . . 55
3.2.1.3 The Control Annotations . . . 55
3.2.2 Restricted Prograph . . . 59
3.3 Testing Form-based Languages: Related Work . . . 60
3.4 Testing Visual Dataflow Languages . . . 61
3.4.1 An Abstract Model for Prograph . . . 62
3.4.2 Building Data and Control Dependencies in OCGs . . . 63
3.4.3 Implementation . . . 64
3.5 An Example . . . 67
3.6 Findings Summary and New Directions . . . 69

CHAPTER 4 A Data-flow Testing Methodology for Visual Dataflow Languages . 71

4.1 Introduction . . . 71
4.2 The Procedural Aspect of Dataflow Languages . . . 72
4.2.1 The Method Structure in Prograph . . . 73
4.2.2 Definition-use Association for Dataflow Languages . . . 76
4.3 Implementation . . . 78
4.3.1 An Example . . . 81
4.4 Interprocedural Data-flow Testing for Dataflow Languages . . . 85
4.4.1 Issues in Collecting The Interprocedural Dataflow Analysis in Dataflow Languages . . . 86
4.4.2 Constructing The Interprocedural Operation Case Graph . . . 90
4.4.3 Dealing With Aliases . . . 103
4.5 An Example . . . 105
4.6 Findings Summary and New Directions . . . 113

CHAPTER 5 Testing Visual Object-flow Languages 114

5.1 Introduction 114
5.2 Data-flow Testing for Text-based Object Oriented Languages 116
5.2.1 Other Issues in Data-flow Testing for Text-based OOP Languages. 121
5.3 The Object-flow in Prograph 124
5.4 Testing Variable Interactions in Visual Object-flow Languages 124
5.4.1 Definition-use Association for Object-flow Languages 129
5.5 Data-flow Testing of Classes in Dataflow Languages 131
5.5.1 Intramethod Data-flow Testing 131
5.5.1.1 Collecting the Intramethod Static du-associations 132
5.5.1.2 Visually Representing Executed du-associations. 132
5.5.2 Intermethod Testing 133
5.5.3 Intraclass Testing 134
5.5.4 Polymorphic Testing 135
5.6 Summary and New Directions 135

CHAPTER 6 Concluding Remarks 137

6.1 Chapters Summary 137
6.2 Visual-based Testing of Imperative Languages 140
6.2.1 Integrated Testing and Visual Validating Environment. 141
6.2.2 Proposed System and Method. 142
6.2.3 Reflecting The Testedness 148
6.2.4 Locating the Error in the Source Code. 148
6.2.5 Benefits 149
6.3 Conclusion and Future Work 149

Bibliography 151

List of Figures

CHAPTER 1 Basic Definitions and Thesis Objectives 1

Figure 1-1. The structural testing hierarchy of test adequacy criteria. 5
Figure 1-2. Data-flow and control-flow program-based structural testing. 6
Figure 1-3. The control flow graph or flow graph of a program. 7
Figure 1-4. An example illustrating both statement and branch coverage criteria. 9

CHAPTER 2 Program-based Structural Testing 13

Figure 2-1. The rules for generating a control flow graph model. 15
Figure 2-2. A CFG illustrating a loop. 18
Figure 2-3. An example of an infeasible code. 20
Figure 2-4. A sample C program and its control flow graph (CFG). 23
Figure 2-5. A definition-clear subpath wrt. x to a use reached by that definition. 27
Figure 2-6. A CFG illustrating the All-defs criterion. 27
Figure 2-7. A definition-clear subpaths wrt. x illustrating the All-uses criterion. 27
Figure 2-8. An example illustrating the All-uses criterion. 28
Figure 2-9. Definition-clear subpaths wrt. x illustrating the All-du-paths criterion. 29
Figure 2-10. Ntafos 1-dr and 2-dr interaction paths. 30
Figure 2-11. An example illustrating the required k-tuple criteria. 31
Figure 2-12. An example illustrating the concept of the context coverage criterion. 33
Figure 2-13. An example of a Context Coverage. 33
Figure 2-14. Illustrating the concept of the ordered context coverage criterion. 34
Figure 2-15. An example of an Ordered Context Coverage. 35
Figure 2-16. A code fragment illustrating du-path issues in arrays. 35
Figure 2-17. The subsume hierarchy for structural-based testing. 37
Figure 2-18. An example illustrating the interprocedural dataflow analysis. 40
Figure 2-19. An example illustrating the presence of aliases in imperative languages. 43

CHAPTER 3 A Control-flow Testing Methodology for Prograph 50

Figure 3-1. A Prograph program for Quicksort. 51
Figure 3-2. An example illustrating a Next Case on success. 56
Figure 3-3. A Next Case applied on success to an operation that cannot fail. 56
Figure 3-4. An operation that is control annotated with finish of failure in a local. 57
Figure 3-5. An operation that is control annotated with finish of failure in a looped case.. 57
Figure 3-6. An operation annotated with terminate on failure in a looped case. 58
Figure 3-7. An iterative Factorial method containing one error (right), and its OCG (left). 67
Figure 3-8. Modified Factorial Local. 69

CHAPTER 4 A Data-flow Testing Methodology for Visual Dataflow Languages.. 71

Figure 4-1. An example of a c-use and a p-use in Prograph. 73
Figure 4-2. Method input and output in Prograph cases. 74
Figure 4-3. The binding an actual parameter to a reference parameter at a call site. 74
Figure 4-4. Variable interactions in the structure of a local operation in Prograph. 75
Figure 4-5. An example of a looped annotated local operation in Progragh. 75
Figure 4-6. The iterative Factorial method containing an error (right), and its OCG (left)..82
Figure 4-7. The inspected iterative procedure. 84

Figure 4-8. The corrected factorial looped local. 85
Figure 4-9. A Prograph example illustrating both direct and indirect data dependencies. 86
Figure 4-10. Procedures Main, A, and B and its corresponding OCG sub-graphs. 88
Figure 4-11. An example of an alias introduced at a call site. 90
Figure 4-12. Creating the Entry and Exit Nodes for procedures A and B. 93
Figure 4-13. The algorithm to construct the IOCG(P) of a visual dataflow program P. 94
Figure 4-14. Creating the Call and Return Nodes for procedures A and B. 95
Figure 4-15. Reaching edges for procedures Main, A, and B. 96
Figure 4-16. Interreaching edges for procedures Main, A, and B in Figure 4-10. 97
Figure 4-17. The connected IOCG sub-graphs for procedures Main, A, and B in. 99
Figure 4-18. The procedure used to propagate the interprocedural uses information. 101
Figure 4-19. An example of an alias at a call site. 104
Figure 4-20. The unaliased IOCG for the program depicted in Figure 4-19. 105
Figure 4-21. The OCGs (left) for methods iterative and factorials (right). 106
Figure 4-22. Interprocedural du-chains with looped universals. 108
Figure 4-23. The connected IOCG for the Factorial program. 110
Figure 4-24. A visual communication of the testedness of the factorial method. 112

CHAPTER 5 Testing Visual Object-flow Languages 114

Figure 5-1. The class and its call graph representation. 118
Figure 5-2. A class A (left) and its Class Call Graph (right). 120
Figure 5-3. The representation of a Class Call Graph enclosed in a frame. 121
Figure 5-4. An example of a message and a polymorphic server. 122
Figure 5-5. A pictorial representation of the Section icon in Prograph. 124
Figure 5-6. An example of get and set methods. 125
Figure 5-7. The example of Figure 5-6 with a synchro enforced between n5 and n8. 126
Figure 5-8. The alternative way of visually coding the example of Figure 5-7. 128
Figure 5-9. Modified labels after the scan and replace technique. 129
Figure 5-10. A p-use of attributes in visual object-flow languages such as Prograph. 130
Figure 5-11. Intermethod du-associations in Prograph. 133
Figure 5-12. The visual representation of a Class Calling Graph. 134

CHAPTER 6 Concluding Remarks 137

Figure 6-1. The integrated testing and validating environment in the proposed IDE. 142
Figure 6-2. A C example containing an error (left) and its CFG (right). 143
Figure 6-3. The control and data-flow information of the example in Figure 6-2. 145
Figure 6-4. The collection of blocks inside the loop labeled n3 of Figure 6-3. 147

List of Tables

CHAPTER 2 Program-based Structural Testing 13

Table 2-1. Tabulated p-uses and their associated edges in the CFG of Figure 2-4. 24
Table 2-2. Tabulated definitions, and their c-uses in each node in the CFG of Figure 2-4. ... 24
Table 2-3. The DCU and DPU of the program depicted in Figure 2-4 25

CHAPTER 3 A Control-flow Testing Methodology for Prograph 50

Table 3-1. A test suite for the dataflow program shown in Figure 3-2. 68

CHAPTER 4 A Data-flow Testing Methodology for Visual Dataflow Languages .. 71

Table 4-1. Exercised du-associations and edges of the program in Figure 4-6. 83
Table 4-2. The DEF and UPCON sets for the IOCG sub-graphs nodes of Figure 4-16. 98
Table 4-3. The DEF and UPCON sets for the program in Figure 4-21 109
Table 4-4. Intraprocedural testedness of factorial and iterative methods. 111
Table 4-5. Interprocedural testedness of the Factorial program in Figure 4-21. 111

CHAPTER 6 Concluding Remarks 137

Table 6-1. Node and edges test suite for the C program shown in Figure 6-2. 144
Table 6-2. All-du-paths test suite for the C program shown in Figure 6-2. 144

List of Abbreviations

AV	Administrative View
CFG	Control Flow Graph
CRG	Cell Relation Graph
CCG	Class Call Graph
DFTT	Dataflow Testing Tool
GUI	Graphical User Interface
ICFG	Interprocedural Control Flow Graph
IDE	Integrated Development Environment
IOCG	Interprocedural Operation Case Graph
ITVVE	Integrated Testing and Visual Validating Environment
OCG	Operation Case Graph
OOP	Object Oriented Programming
VDPLs	Visual Dataflow Programming Languages
VP	Visual Programming
VPEs	Visual Programming Environments
VPLs	Visual Programming Languages
VV	Validation View
WYSIWYT	What You See Is What You Test

Acknowledgments

First and foremost, I would like to express my sincere and most profound gratitude to my supervisor, Dr. Trevor Smedley, for his extensive guidance, unparalleled support, and infinite wisdom. His confidence in my ability to complete the work presented in this thesis, has always given me the strength to overcome numerous obstacles.

I would also like to express my deepest gratitude to Dr. Phil Cox for his guidance and encouragement throughout the course of my graduate work.

I would also like to express my sincere gratitude to Dr. Abdel-Aziz Farrag for his unmeasured support during my undergraduate studies.

To my longtime friend and Ph.D. role model, Dr. Camille Habib, I would like to say thank you for your constant support and motivation.

To my brother, Dr. Jalal Karam, I would like to say thank you for always being there for me.

Last but not least, I would like to express my love, appreciation, and gratitude to my wonderful parents. Although words can hardly express how I feel about them; however, I trust they know that I treasure both their love and unmatched support for me.

Abstract

Visual dataflow programming languages have become an important topic of research in recent years, yielding a variety of research systems and commercial applications [9][10][21][80]. As with any programming language, be it visual or textual, programs written in visual dataflow languages like Prograph may contain faults. Thus, to ensure the correct functioning of visual dataflow programs, testing is required. Despite this valid observation, we find no discussion in the literature that addressed a specific testing methodology for visual dataflow programs. We did find, however, adequate discussions related to testing methodologies for imperative, declarative, and form-based languages.

In this thesis, we investigated, from structural testing perspectives, differences between imperative and visual dataflow languages. Our investigations revealed opportunities to adapt code-based structural testing to test visual dataflow languages. Based on that adaptation, we have developed an integrated Dataflow Testing Tool (DFTT) and used it to visually and empirically validate our testing results. Those results showed that our structural-based testing methodology, in particular the "All-du paths" testing, provide an important error detection ability in visual dataflow languages. To communicate the testedness of a visual dataflow program, we provided a visual testing and validating environment in the *DFTT* by using visual annotations to reflect, based on a certain testing criterion, the testedness of an operation, a predicate operation, or a datalink.

We also investigated from a data-flow testing perspective, differences between code-based object oriented languages and visual object-flow languages in the context of Prograph. Our findings revealed that, analogous to code-based object oriented languages, there are three levels of testing instance variable data-flow interactions in visual object-flow languages. In each level we showed how code-based data-flow testing techniques can be adapted to collect that level's appropriate du-chains. As for our new research directions, we have proposed an Integrated Testing and Visual Validating Environment (ITVVE) for imperative languages that allows users/testers to visualize both tested and untested du-associations. We also showed that the *ITVVE* provides a visual testing environment that facilitates the task of approximating the location of errors in the code.

1 Basic Definitions and Thesis Objectives

1.1 Introduction

Visual programming languages (VPLs) have become an important topic of research in recent years. The dataflow paradigm is one of the most popular computational models for *VPLs* [49]. The visual dataflow paradigm for programming is represented as a directed graph where nodes represent user-defined or system-defined operations, and the data flows through edges or links between nodes. Links going into a node represent the operation's data input, links going out of a node represent the operation's output or result. Therefore, programming in this paradigm involves assembling operation elements that send and receive data. Once the data flows into an operation, it is evaluated accordingly, and the resulting data flows out of the operation and on towards other operations along datalinks for further evaluation. Unlike imperative programming languages where the order of execution of programming statements is rigidly structured by the programmer, a dataflow language simply specifies data dependencies, and an operation is executed when all of its input data become available. This model of execution is known as the dataflow computational model. In a pure dataflow model, control constructs are not explicitly specified by the programmer; rather, the order of execution is implied by the operations' data interdependencies. To allow the user to explicitly add control constructs such as those found in imperative languages, dataflow languages [9][10][21][80] extended the pure dataflow model to include the necessary control constructs. This extension is necessary in order for a dataflow language to have any practical use. Thus, a dataflow program can be characterized by both its data and control dependencies.

Visual dataflow programming languages (VDPLs) provide meaningful visual representations for the creation, modification, execution and examination of programs. Users of *VDPLs* write

programs by creating icons that are connected via datalinks. We refer to this activity in this thesis as visual coding. Like any other programming language, be it textual or visual, dataflow programs are prone to faults that could originate at the visual coding level. In spite of this valid observation, we find no discussion in the research literature of techniques for testing or assessing the reliability of visual dataflow programs. In contrast, we find that most test adequacy criteria were researched for imperative and declarative languages [8][17][29][56][59][76], and a number of both commercial and experimental testing packages have been developed based on these strategies [33][46]. Some recent work focused on testing form-based languages [16][17]. Although the visual dataflow paradigm is similar to the visual form-based paradigm in that they are both visual, several characteristics of the form-based paradigm such as the dependency-driven nature of its evaluation engine, suggest a different approach when testing *VDPLs* More on testing form-based languages can be found in Section 3.3 of Chapter 3.

In this Chapter, we seek to illustrate the basic notions – and not the precise definition – underlying the followings terms: *software testing*; *test adequacy criteria*; *control flow graph*; *program-based structural testing*; *programming paradigms*; and *dataflow visual programming languages.* This will make further discussions more readily understandable, and lay the groundwork for the introduction of our research objectives. In Section 1.2 and Section 1.3, we briefly cover the fundamentals of software testing and test adequacy criteria, respectively. In the latter, we explain what it means to adequately test a program and how measurements are obtained from a particular criterion. In Section 1.4, we informally introduce program-based structural testing, briefly cover the basic notion of a control flow graph, and introduce two testing criteria of program-based structural testing: *control-flow* and *data-flow.* In Section 1.5, we give the basic definition of a programming paradigm, and give special attention to the visual dataflow programming paradigm. Finally, in Section 1.6, we define our research objectives and discuss the organization of the remainder of this thesis.

1.2 Fundamentals of Software Testing

Testing and debugging are certainly related but not synonymous. Formal definitions of the word "testing" are: "testing is the process of demonstrating that errors are not present; the pur-

pose of testing is to show that a program performs its intended functions correctly; and testing is the process of establishing confidence that a program does what it is supposed to do" [35]. These definitions have misguided both readers and testers, and are considered by some to be the primary cause of poor program testing [64]. In general, program testing is a costly activity, and that cost should be compensated by increasing the reliability of a program. Since one can make the assumption that any program is almost certain to contain errors, one should not test a program to show that it works; rather, to find as many errors as possible. Thus, a more appropriate definition for testing is: "testing is the process of executing a program with the intent of finding errors" [64]. Given this definition, it is only natural to ask: can we test a program to find "all" of its errors? The answer to this question, as will be shown, is negative. Debugging, on the other hand, is a two-part process; it begins with some indication of the existence of an error and it is the activity of (1) determining the exact nature and location of the suspected error within the program, and (2) fixing or repairing the error.

Having defined the objective of testing, the only technique to guarantee a program's correctness is to execute it on all possible inputs. While testing all possible input values would provide the most complete picture of a program's behavior, the input domain is usually too large for exhaustive testing to be practical. Rather, the usual procedure is to select a relatively small subset that is in some sense a representative of the entire input domain. When evaluating large programs however, satisfying these requirements becomes impractical due to the infinite input tests that have to be generated. Thus, other, less "stern" techniques had to be developed. Goodenough and Gerharts [37] pioneering work in software testing focused on testing techniques that are based on some selected criteria. These criteria are best known as *test adequacy criteria*, and will be discussed briefly next.

1.3 What is a Test Adequacy Criterion?

Test adequacy criteria are testing objectives that are quantified, reasonable, and achievable [48]. This means that, under any one particular test adequacy criterion, the tester is expected to: (1) define which parts of the program need be tested, and measure how well those parts have been exercised; and (2) determine whether testing of a particular program is sufficient.

A number of test adequacy criteria have been proposed and investigated in the testing literature [93][94][95]. The most frequently quoted classes of test adequacy criteria are: *structural-based* testing; *fault-based* testing; and *error-based* testing. Each class will be briefly introduced next.

1.3.1 Classes of Test Adequacy Criteria

One way to classify test adequacy criteria is by the underlying testing approach [95]. There are three basic approaches to software testing:

- **structural-based testing:** this approach specifies testing requirements in terms of the coverage of a particular set of elements in the structure of a program or its specification. In the structure of a program, properties such as logical decisions are considered the set of elements that need to be exercised or covered. In the specification of a program, properties such as algebraic specifications are considered the set of elements that need to be exercised.

- **fault-based testing:** this approach focuses on using test data to detect defects in the program or its specifications. For example, program-fault-based testing attempts to uncover faults that have been intentionally planted in the program. Planted faults are known as mutants. Specification-fault-based testing attempts to detect faults in the implementation that are derived from misinterpreting the specification of a program.

- **error-based testing:** this approach focus on using test data that mimic the kinds of mistakes that programmers make when constructing programs. Deriving these test data requires partitioning the input domain of a program according to either the program or the specification. When partitioning the input domain according to the program, we say that two input data belong to the same sub-domain if they cause the same "computation" or execution path to be traversed. Thus, sub-domains correspond to subsets of the program's execution paths. When partitioning the input domain according to the specification, a subset of data is considered a sub-domain if the specification requires the same function on the data.

As depicted in Figure 1-1, each approach can be applied based on the specification of the software (specifications-based) or the program code (program-based or code-based). Specification-based structural testing requires testing in terms of the requirements of the software so that a test set is adequate if all the identified features have been exercised. Program-based testing specifies testing requirements in terms of the program under test and decides whether the program has been thoroughly exercised. As depicted in Figure 1-2, structural program-based testing can be based on either the control-flow or the data-flow of a program. When applied to the control-flow of a program, program-based testing is known as the control-flow testing criteria; however, when applied to the data-flow of a program, it is known as data-flow testing criteria. In this research work, we focus on investigating program-based testing for visual dataflow languages.

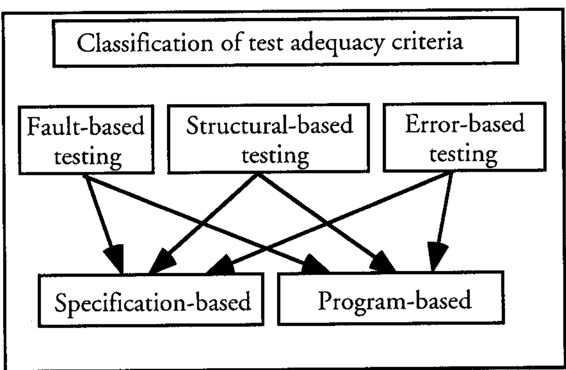

Figure 1-1. The structural testing hierarchy of test adequacy criteria.

The testing literature often refers to white-box testing and black-box testing as the two major categories in software testing. In white-box testing, testers have access to the internal structure of a program. Test cases derived from the program's structure attempt to exercise program elements such as all logical decisions on both their true or false sides. Thus, any program-based testing belongs to the white-box testing category. Black-box testing, on the other hand, treats the program under test as a "black box" where no knowledge about the implementation of the program is assumed. It is performed during later stages of testing and attempts to find errors such as: incorrect or missing functions; interface errors; performance errors; and errors in data structures or external database access. Thus, any specification-based testing belongs to the black-box testing category.

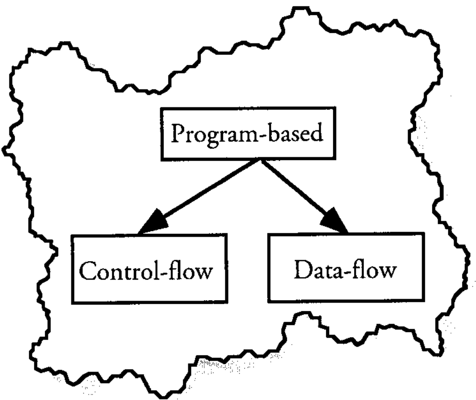

Figure 1-2. Data-flow and control-flow program-based structural testing.

1.4 Program-based Structural Testing

Program-based testing is a special method for obtaining code coverage. When using program-based testing, we try to cover the program's code with as many test cases as possible. Test cases are derived from the code logic and are independent of the functional specifications. As mentioned in Section 1.3.1, program-based structural test adequacy criteria can be applied to the control-flow or the data-flow of a program. Both types are based on the control flow graph model of a program structure. Before introducing the ideas behind both control-flow and data-flow testing, it is essential that we briefly introduce the control flow graph model of a program under test.

1.4.1 The Control Flow Graph Model of a Program: An Informal Introduction

There are a number of conventions for control flow graph (CFG) models with subtle differences; however, most adequacy criteria can be defined independently of such conventions [95]. In general, a *CFG* is a directed graph that consists of a set N of nodes and a set of $E \subset N \times N$ directed edges between nodes. Each node represents a linear sequence of computation. Each edge representing a transfer of control is an ordered pair (*n1, n2*) of nodes, and is associated with a predicate that represents the condition of control transfer from node *n1* to node *n2.*

In a *CFG*, there is an entry node n_e and an exit node n_x. Every node in a *CFG* must be on a path from n_e to n_x. An example of a *CFG* is depicted in Figure 1-3.

1.4.2 Control-flow Test Adequacy Criteria.

One way to classify control-flow test adequacy criteria is by the underlying coverage approach. There are three basic coverage criteria associated with control-flow testing: *path coverage*; *statement coverage*; and *branch coverage*. Each coverage criterion will be informally introduced next.

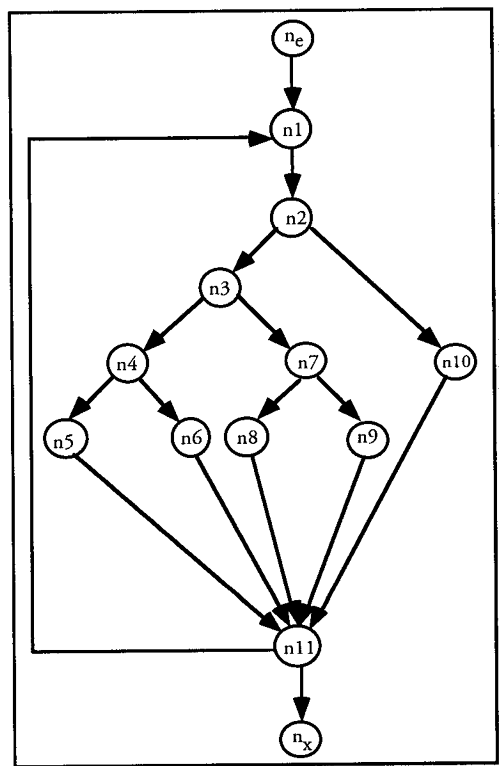

Figure 1-3. The control flow graph or flow graph of a program.

- **Path Coverage.** The path coverage criterion requires that all execution paths from a program's entry to its exit are executed during testing. This is accomplished through an exhaustive path testing that tests all possible paths of control flow through the program. In practice however, the number of logical paths through a program can be astronomical. For example, consider the trivial program represented with a *CFG* in Figure 1-3. Each node or *circle* in the graph of Figure 1-3 represents a segment of statements that executes sequen-

tially, possibly terminating with a branching statement. Each edge or *arc* represents a transfer of control or branch between segments. The graph of Figure 1-3 depicts a 10 to 20 program statements consisting of a "for" loop in node *n1* that iterates 20 times, and contains nested "if" statements, then determining the number of unique logic paths is the same as determining the total number of unique ways of moving from node *n1* to node *n11.* If we assume that all decisions in the program are independent from one another, the number of paths from *n1* to *n11* is approximately $5^{20} + 5^{19} + ... + 5^{1}$. One might however argue that, in actual programs, every decision is not independent from every other decision, meaning that the number of possible paths is less. This argument can be easily defeated since actual programs are much larger than the simple program represented by the *CFG* in Figure 1-3.

- **Statement Coverage.** In software testing practices, testers are often required to generate test cases to exercise every statement in the program under test. Exercising every statement means traversing every node in the *CFG* representing the program under test. The percentage of traversed nodes during testing is a measurement of test adequacy. Thus, a test set that results in the traversal or coverage of all nodes is considered adequate according to the statement coverage criterion. Statement coverage is also known as the *All-statements* or *All-nodes* criterion. Consider the example in Figure 1-4 (top right) and its *CFG* (bottom). In this example, it is possible to execute every statement by writing a single test case that traverses the following paths: *n1*, *n2*, *n3*, *n4*, and *n5*. That is, by setting the following assignments to the variables: $a = 2$, $b = 0$, and x to 4. This criterion is considered weak since, some edges, such as (*n1*, *n2*) that represent the control flow, are not exercised. To test all the edges in a *CFG* of a program under test, a branch coverage criterion is needed, and will be briefly presented next.

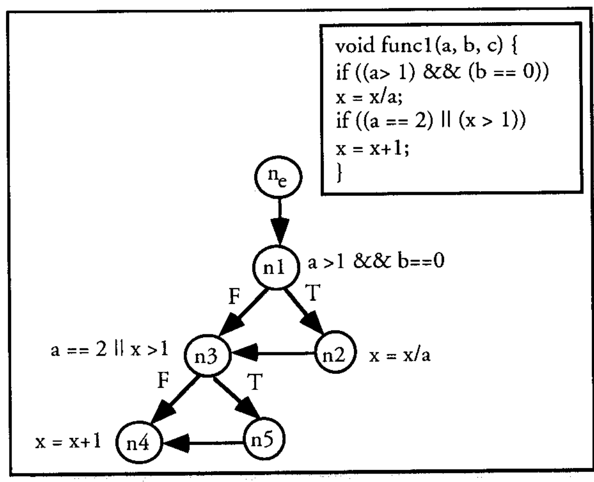

Figure 1-4. An example illustrating both statement and branch coverage criteria.

- **Branch Coverage.** This criterion requires that test cases exercise both the true and false outcome of each logical decision in a program. In other word, each branch direction in the *CFG* must be traversed at least once. Consider the example shown in Figure 1-4. In this example, edges (*n1, n2*) and (*n1, n3*) cover the true and false branches of the predicate or conditional statement in node *n1*. The percentage of the traversed branches during testing is a measurement of test adequacy. The branch coverage criterion is also known as the *All-branches* criterion.

1.4.2.1 The Subsume Relationship

We say that a test adequacy criterion *A subsumes* some other testing criterion *B* if all the elements that *B* exercises are also exercised by *A*. For example, in the branch coverage criterion, every statement must be executed if every branch direction is executed because every statement is on some subpath emanating from either a branch statement or from the entry point of the program. Thus, branch coverage subsumes statement coverage.

1.4.3 Data-flow Test Adequacy Criteria

Given the definition of a variable *x* in a program, it is frequently useful to know what uses might be affected by the particular definition. The inverse is also true; that is, for a given use the definitions of data items which can potentially supply values to it are of interest. Such data-flow relationships are called definition-use or *def-use* and use-definition or *use-def*, respectively. Data-flow testing focuses on how variables are bound to values, and how these variables are to be used. So rather than selecting program paths based on the control structure of a program, data-flow testing is based on selecting paths that trace input variables through a program until they are ultimately used to produce output values. Accounting for the data-flow paths within a procedure or unit is known as *intraprocedural* data-flow testing. Accounting for data-flow paths in the program as whole is known as *interprocedural* data-flow testing. More on data-flow test adequacy criteria can be found in the data-flow literature survey of Chapter 2.

1.5 Programming Paradigm

A paradigm is the various systems of ideas that have been used to guide the design of programming languages. It can be thought of as a collection of programming "styles" or abstract features that categorize a group of languages which are accepted and used by a group of programmers. The most prominent paradigms discussed in the literature are: imperative; declarative; object oriented; and dataflow. The imperative paradigm is well known, and represents languages such as C and Pascal where programming is procedure-oriented. The declarative paradigm focuses on predicate logic in which the basic concept is a relation. An example of a declarative language is Prolog. The object oriented paradigm covers languages that support, among other features, classes, methods, objects, and message passing. Some of these languages are: C++; SmallTalk; and SIMULA. The dataflow paradigm covers languages in which the ordering of operations is not explicitly specified by the programmer, but is that implied by the data interdependencies [81]. Visual languages that adopt the dataflow paradigm, are known as visual dataflow programming languages or *VDPLs*. In *VDPLs*, icons are the language constructs, and data flows between these icons via datalinks.

VDPLs can be either domain specific or general purpose. In a domain specific *DVPLs*, users cannot map a variety of application domains using the language's constructs. Thus, the language becomes restricted to one particular domain. LabView® [10] for example, is a domain specific (scientific) *VDPL* that is designed for creating virtual instrumentations. Prograph® [80] on the other hand, is a general purpose *VDPL* that it is not particular to any one specific domain of programming. It is a representative of both commercial and research *VDPLs*, and was used to develop a number of commercial software packages.

1.6 Thesis Objectives

Our objective is to provide *VDPL* users with a structural-based testing methodology to test their visual dataflow programs. In pursuit of this objective, we investigate, from a structural testing perspective, differences between imperative and visual dataflow languages. Our findings, as we shall see, reveal opportunities to adapt code-based testing for visual dataflow languages. Based on those findings, we introduce new testing methodologies that make it possible for visual dataflow language users to visually and empirically inspect, based on a particular structural-based testing criterion, the testedness of a visual dataflow program.

The remainder of this thesis is organized as follows: in Chapter 2, we formally introduce code-based control-flow and code-based data-flow test adequacy criteria for imperative languages. We also discuss some issues related to data-flow analysis of both intraprocedural and interprocedural data-flow testing. Finally, we discuss the main characteristics of visual programming, followed by an overview of the visual dataflow paradigm in the context of Prograph. In Chapter 3, we illustrate some of Prograph's main functionalities and discuss the subset of the language considered for this work. We then investigate, from a structural-based testing perspective, differences between dataflow and imperative languages. The results reveal opportunities for adapting code-based control-flow testing criteria to dataflow languages. We show that our proposed testing methodology is well suited for visual dataflow programs. In particular, the All-branches criterion provides important error detection ability, and can be applied to any dataflow program. Empirical results obtained using the Data Flow Testing Tool (DFTT), a tool we have developed, will reveal that analogous to imperative languages, the All-branches criterion cannot

detect all the errors in a visual dataflow program. Thus, to help catch those undetected errors, a more rigorous testing should be applied. This is indeed the focus of Chapter 4. Related work pertaining to the use of visual annotation in reflecting the testedness of form-based languages is also discussed in Chapter 3. In Chapter 4, we discuss, in the context of Prograph, differences in the language structure, between imperative and the procedural aspect of dataflow languages. The findings reveal an opportunity to adapt code-based definition-use testing or du-testing for visual dataflow languages. The adapted du-testing is subsequently used to achieve both intra-procedural and interprocedural data-flow test adequacy criteria for visual dataflow languages, in particular the *All-du-paths*[1] criterion. Examples and empirical results will show that, analogous to imperative languages [32], the All-du-paths criterion that we have adapted for visual dataflow languages subsumes the All-branches criterion we have introduced in Chapter 3. In Chapter 5, we take the code-based structural testing methodology introduced in Chapters 3 and 4 for testing the procedural aspect of visual dataflow languages, and extend it to test the object oriented features of visual dataflow languages. Analogous to text-based object oriented languages, the object oriented features of visual dataflow languages introduce new bug hazards and challenges for code-based testing. We examine these challenges and hazards for object oriented dataflow languages in the context of Prograph, and subsequently introduce a new methodology to allow visual object-flow program testers to visually test for the presence of these hazards. In Chapter 6, we present a summary of issues and ideas pertaining to the originality of the work presented in this thesis. This summary is then followed by a discussion on our new research directions. This involves a proposal of a method and system for an Integrated Development Environment (IDE)-specific compiler technique that deals with the use of visual elements to interactively communicate to the user/tester, according to a certain structural-based testing criterion, the testedness of imperative and textual-based object oriented programs. Finally we conclude with some future work which may involve: (1) Establishing a hierarchy of structural testing criteria for visual dataflow languages and comparing that with the hierarchy of structural testing criteria for imperative languages, and (2) generalizing our proposed method and system so that is could be, with moderate changes, applied to any existing *IDE*.

---

1. The All-du-paths is a member of the Rapps and Weyuker [76] family of data-flow test adequacy criteria, and will be explained in detail in Section 2.4.3 of Chapter 2.

2 Program-based Structural Testing

2.1 Introduction

As the collection of high level languages increased over time, so did the paradigms that represent them [4]. A paradigm is the various systems of ideas that have been used to guide the design of programming languages. It can be thought of as a collection of abstract features that categorizes a group of languages which are accepted and used by a group of programmers. The most prominent paradigms discussed in the literature are: imperative, declarative, functional, object oriented, and dataflow. Imperative languages such as C provide facilities for assigning and retrieving values to and from memory locations. The structural-based testing techniques discussed in this chapter are all applicable to imperative programs. Declarative languages such as Prolog and FormsIII [16] provide facilities for defining relations or functions among the entities of the program. A technique that adapts code-based testing was introduced in [16][17] to test, in the context of FormsIII, declarative languages. Functional languages such as LISP allow the user to express computations as the evaluation of mathematical functions. We are not aware of any structural-based testing technique for functional languages. Object oriented languages such as C++ allow the user to associate behavior with data-structures called "objects" – that belong to classes – which are usually structured into a hierarchy. Structural-based testing for object oriented methods within a class is analogous to structural testing for imperative languages. However, in the object oriented paradigm, polymorphism[1] for example, introduces a new set of bug hazards [12] that cannot occur in imperative languages. For those types of bug hazards, various techniques have been proposed [12][40]. Chapter 5 offers an in depth look at how structural-based testing is appropriately extended to test bugs introduced by the unique

1. Polymorphism is a mechanism found in object oriented languages that allows the same message to be bound to different methods in a class hierarchy.

features of object oriented languages. The visual dataflow paradigm allows users to associate program constructs with links through which data travels or flows. To the best of our knowledge, the testing literature does not contain any work on testing visual dataflow languages. Thus, a primary goal of this research work is to provide visual dataflow language users with a structural-based testing methodology to test their visual dataflow programs.

The rest of this Chapter is organized as follows: In Section 2.2, we formally define the control flow graph model. In Section 2.3, we formally define the following test adequacy criteria: statement coverage; branch coverage; and path coverage. In Section 2.4, we formally define dataflow test adequacy criteria and the concept of intraprocedural data-flow testing. In Section 2.5, we briefly discuss the subsume relationship among the control-flow and data-flow adequacy criteria discussed in this Chapter. In Section 2.6, we briefly discuss the concept and approaches to integration testing. The latter introduces new challenges for structural testing. For example, when applying data-flow testing to two or more integrated units or procedures, the data-flow analysis of these components has to account for definitions whose uses, due to procedure calls, extend beyond procedure boundaries. This testing is known as interprocedural data-flow testing, and is indeed the focus of Section 2.7. Finally, in Section 2.8, we conclude with a general discussion on visual languages, and the visual dataflow paradigm in the context of Prograph.

2.2 The Control Flow Graph Model: A Formal Definition

The use of graphs is widespread. Society uses graph-based notations such as organizational charts, circuit diagrams, and flow charts [13]. The control flow graph (*CFG*) model has been used as a mapping model for program structures. It is widely used in the static analysis of software, such as defining and studying program-based test adequacy criteria. A *CFG* is a directed graph that consists of a set N of nodes and a set $E \subset N \times N$ of directed edges between nodes. Each node represents a linear sequence of computation. Each edge representing a transfer of control is an ordered pair ($n1$, $n2$) of nodes, and is associated with a predicate that represents the condition of control transfer from node $n1$ to node $n2$. In a *CFG* for a procedure p, there is an entry node n_e and an exit node n_x. Every node in a *CFG* must be on a path from n_e to n_x. It should be noted here that there are a number of conventions for *CFG* models with subtle

differences; however, most adequacy criteria can be independently defined of such conventions. For programs written in procedural programming languages, a *CFG* can be automatically generated using the rules depicted in Figure 2-1. These rules represent the most common textual imperative programming constructs.

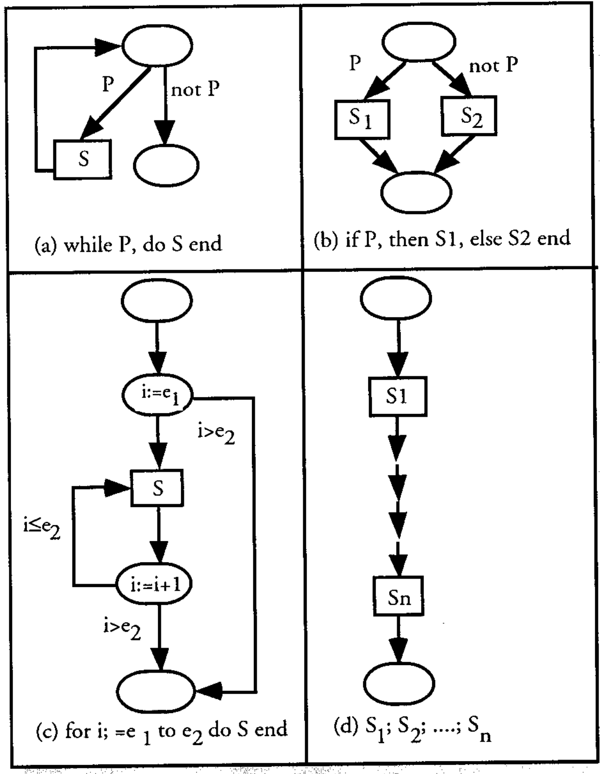

Figure 2-1. The rules for generating a control flow graph model.

In general, to construct a *CFG*, a program code is decomposed into a set of disjoint blocks of linear sequences of statements. A block is a sequence of consecutive statements in which the flow of control enters at the beginning of the block and exists at the end without halt or the possibility of branching except at the end; that is, whenever the first statement of a block is executed, the other statements in that block are executed in the given order. The first statement of a block is the only statement that may be executed directly after the execution of a statement in another block. Usually, each block corresponds to a node in the *CFG*. Thus, we define a block as a maximal set of ordered statements $S = \{s_1, s_2, s_3, \ldots, s_n\}$ such that if $n > 1$, for $i = 2, \ldots, n$, s_i is the unique executional successor of s_{i-1}, and s_{i-1} is the unique executional predecessor of s_i. Thus, the first statement of a block is the only one that may have an executional predecessor

outside the block, and the last statement is the only one that may have an executional successor outside the block. Furthermore, since conditional transfer cannot have unique executional successors, every conditional transfer must be the last statement of a block. A control transfer from one block to another is represented by a directed edge between the nodes such that the condition of the control transfer is associated with it.

Let G be a *CFG* representing a program Q where a node n_i corresponds to a block b_i of Q. An edge from node n_j to node n_k exists in G and is denoted (n_j, n_k), iff either the last statement of b_j is a transfer whose target is the first statement of b_k, or the last statement of b_j is not an unconditional transfer and it semantically precedes the first statement of b_k. Given an edge (n_j, n_k), we say that n_j is a predecessor of n_k, and n_k is a successor of n_j. Informally, the presence of an edge (n_j, n_k) in G indicates that it is possible for execution to flow from the last statement in b_j to the first statement in b_k. The following is graph theory related terminologies that are applicable to G:

- A *complete computation path* is a path that starts with n_e and ends with n_x in G. It is also known as a computation path or execution path. It is important to note here that there might not be input data that will cause the sequence of statements of a complete path to be executed. There is no algorithm in the literature that can predict the excecutability of a certain complete path; however, testing does not require that all complete paths be executed, rather a subset;
- An *empty path* is a path of length 1. In this case, the path contains no edges; rather a node, and it is written as (n_1);
- A *cycle* is a path $p = (n_1, n_2,\ldots, n_t, n_1)$;
- A *cycle-free path* is a path that does not contain cycles as subpaths;
- A *finite path* is a sequence of nodes $(n_1,\ldots, n_k)$, $k \geq 2$, such that there is an edge from n_i to n_{i+1} for $i = 1, 2,\ldots, k\text{-}1$;
- A *simple path* is a path whose nodes, except possibly the first and the last, are distinct;
- A *loop-free path* is a path whose nodes are all distinct; and

- A *syntactically endless loop* is a path $(n_1, \ldots, n_k)$, $k > 1$, $n_1 = n_k$, such that none of the blocks represented by the nodes on the path contains a conditional transfer statement whose target is either in a block that is not on the path or is a halt statement. Such a path contains no possible escape and can be detected and eliminated from a program. Thus, it is assumed that programs under test do not contain syntactically endless loops.

2.2.1 Representing The Control Flow Graph

Representing a *CFG* can be accomplished using standard methods that include an adjacency matrix and a path matrix. An adjacency matrix may be formed from the control flow graph such that an edge (i, j) is one, iff there is an arc in the graph from node i to node j. A path matrix may be formed from the adjacency matrix in which an edge (i, j) is one, iff there is some path (along any number of arcs) from node i to node j. For each basic block, two boolean vectors are defined such that one vector indicates which variables the block defines, while the other gives the variables referenced by the block. These two matrices and vectors provide sufficient information to test the program without further reference to the program source code [47]. Inspecting a graph can be done using standard algorithms such as breadth-first or depth-first search.

2.2.2 Dealing With Loops

A loop is a special language construct that allows for the iteration of one or more statements. A loop that contains no other loops is called an inner loop. In a *CFG*, a loop can be represented as a collection of nodes such that: (1) the collection of nodes has a unique entry called the *header*; (2) there must be at least one way to iterate the loop; that is, there should be at least one path back to the header; and (3) all nodes in the collection are strongly connected; that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop.

A header is a node in a loop such that the only way to reach a node of the loop from a node outside the loop is first to go through the header. The entry point into a loop must dominate

all other nodes in that loop; otherwise, it would not be the sole entry to the loop. A node n_i of a flow graph dominates a node n_j, if every path from the header to n_j goes through n_i. Under this definition, every node dominates itself, and the entry of the loop dominates all other nodes in the loop. For example, in the *CFG* depicted in Figure 2-2, *n1* dominates every other node; *n2* dominates itself since control can reach any other node along the edge (*n1*, *n3*); *n3* dominates all but *n1* and *n2*; *n4* dominates all but *n1*, *n2*, and *n3* since all paths must go through either (*n1*, *n2*, *n3*, *n4*) or (*n1*, *n3*, *n4*); *n5* and *n6* dominate only themselves because the flow of control can skip around either by going through the other; *n7* dominates *n7*, *n8*, *n9*, and *n10*; *n8* dominates *n8*, n9, and *n10*; *n9* and *n10* dominate only themselves.

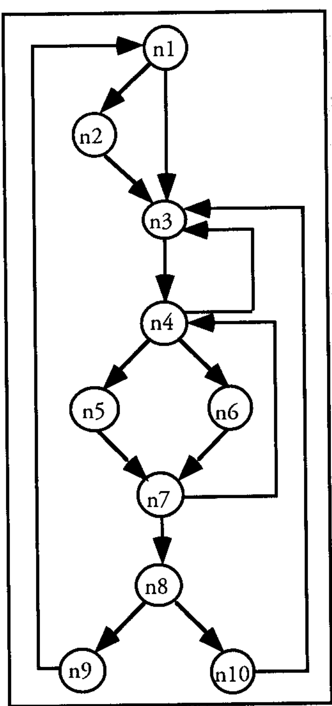

Figure 2-2. A *CFG* illustrating a loop.

2.3 Control-flow Adequacy Criteria

Control-flow testing is modeled as a traverse in the *CFG* representing a program *Q*. Every execution corresponds to a path in the *CFG* from the entry node to the exit node. Such a path is called an execution path. An effective criterion requires paths with a high probability of revealing faults; that is, when the program is run with test data that cause the selected paths to be

executed, there is a high probability that faults, if they exist, are exposed by those test runs. Essentially, the effectiveness of such criteria depends not only on the selected paths but also on the test data for those paths. Test cases used in control-flow testing are derived from the logic of the source code, and are independent of functional specifications [48]. There are three well known control-flow testing criteria that are dependent on path traversal in a *CFG*. These are statement coverage; branch coverage; and path coverage. Each of these coverage criteria will be formally defined next.

2.3.1 Statement Coverage

Statement coverage requires executing all the statements in the program under test [48]. Since statements in *Q* are organized as blocks in *G*, and those blocks correspond to nodes in the *CFG*, covering all the statements means covering all the nodes in *G*. Given a set of execution paths $P = \{p_1, p_2, \ldots, p_m\}$ and the set of all nodes $N = \{n_1, n_2, \ldots n_m\}$ in a control flow graph *G*, *P* satisfies the statement coverage criterion iff *N* is on *P*. A test set that satisfies this requirement is considered to be adequate according to the statement coverage criterion. The percentage of executed statements is a measurement of the statement coverage.

2.3.2 Branch Coverage

Branch coverage is similar to statement coverage, except that in branch coverage, all control transfers in a program under test are exercised during testing. The percentage of the control transfers executed during testing is a measurement of test adequacy. Given a set of execution paths $P = \{p_1, p_2, \ldots, p_m\}$ and the set of all edges $E = \{e_1, e_2, \ldots, e_m\}$ in a control flow graph *G*, *P* satisfies the branch coverage criterion iff for any edge $e \in E$, some path in *P* contains *e*.

It is important to observe that if all edges in a *CFG* are covered, all statements are necessarily covered. Thus, a test set that satisfies branch coverage, must also satisfy statement coverage. This relationship between test coverage criteria is called the subsume[1] relationship. Hence branch coverage subsumes statement coverage. If all branches are exercised, it does not mean

1. The subsume hierarchy of structural-based test adequacy criteria discussed in Chapter 2 is depicted in Figure 2-17.

that all combinations of control transfers are traversed. This requirement is usually covered by applying the path coverage criterion which is explained next.

2.3.3 Path Coverage

The path coverage criterion requires that all the execution paths from the program's entry node n_e to its exit node n_x are traversed during testing. Given a set of execution paths $P = \{p_1, p_2, \ldots, p_m\}$ in a control flow graph *G*, *P* satisfies the path coverage criterion iff *P* contains all execution paths from n_e to n_x in *G*. The path coverage criterion is too strong to be practically useful, because there exist an infinite number of different paths in programs with loops. This would result in infinite test sets which means that testing cannot finish in a finite period of time. Practically speaking, however, a test has to be completed in a finite period. The requirement that a test adequacy criterion can always be satisfied by a finite set is called finite applicability [94].

2.3.4 Feasibility

Statement and branch coverage, in some cases, cannot be fully achieved because of the possibility of the existence of infeasible or dead code. This problem occurs simply when none of the test input data can exercise a specific statement or a branch. To illustrate, consider the fragment of code and its *CFG* in the example of Figure 2-3. In this example, the programmer has omitted the minus sign before "3" in the statement "if(k >3)". If the value of *k* is less than 2, then *k* cannot possibly be greater than 3. Thus, neither "x = x + k" can be reached, nor edges (*n2, n4*) and (*n2, n5*) can be traversed.

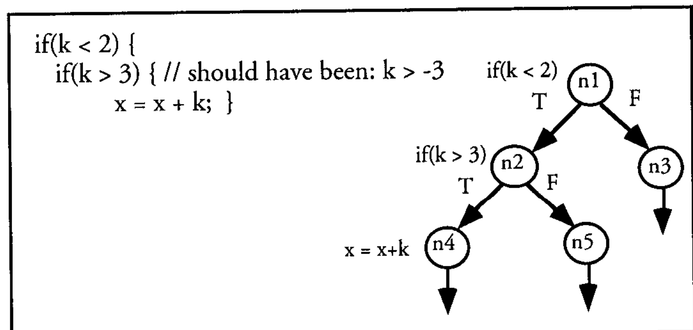

Figure 2-3. An example of an infeasible code.

For both statement and branch coverage criteria, the possible presence of infeasible code in a program makes these criteria not finitely applicable. For these criteria, a finitely applicable version called the feasible version of the testing criterion was subsequently defined [94]. The feasible version recognizes, due to the presence of dead code, the possibility of not fully satisfying a certain coverage criterion. The introduction of a feasible criterion, as argued by Weyuker [87], might cause undecidability in testing, because (1) it may not be decided whether a test set satisfies a given adequacy criterion, and (2) the question whether a statement, branch, or path is feasible, is undecidable. In general, one of the major weaknesses of all these aforementioned coverage criteria is that they are solely based on syntactic information and do not consider semantic issues such as infeasible paths [20].

2.4 Data-flow Adequacy Criteria

In the previous section, we described how a program's control-flow information is represented and used in a *CFG* to achieve control-flow test adequacy criteria such as the All-statements and All-branches. In this section, we discuss how a programs's data-flow information is incorporated into a *CFG*, and how this information is used to achieve data-flow test adequacy criteria. Then, three intraprocedural (unit) dataflow adequacy criteria are reviewed: Rapps and Weyuker [29][76]; Ntafos [66] [67]; and Laski and Korel [56].

2.4.1 Data-flow Information in a CFG

Although data-flow analysis is traditionally used in the last two phases of a compiler, namely, code optimization and code generation, it has also become an integral part of other language processing tools such as anomaly checkers, and testers. In the latter, paths from nodes containing definitions to nodes where those definitions are used (definition-use chains) are of interest. This data-flow information; in particular definition-use chains (du-chains) are required to apply data-flow testing. Data-flow analysis techniques for computing definition-use chains for individual procedures Aho [1] (pp. 632-633) are well known and have been used in various tools, including data-flow testers introduced by: Frankl and Weyuker [33][34]; Harrold and Soffa [44]; and Korel and Laski [57]. In general, data-flow analysis refers to the part of a com-

piler that examines the *CFG* of a program's source code, and collect desired information pertaining to the definitions and uses of variables in $n \in CFG$. Once definitions and uses in each node n have been identified and represented, collecting the du-chains in the program can be accomplished by using well known techniques such as the one proposed by Aho [1].

As discussed previously, occurrences of variables at each node $n \in CFG$ are classified as either a definition occurrence or a use occurrence. A node $n \in CFG$ contains a definition occurrence of a variable if there is a statement in *n*'s corresponding block *b* that binds a value to that variable. For example, in Figure 2-4, the node labeled *n4* contains a definition of *z*. A use occurrence of a variable is where the value of the variable is referenced. Each use occurrence is further classified as being a computational use or a predicate use [76].

A node $n \in CFG$ contains a computational use or c-use of a variable if there is a statement in *n*'s corresponding block *b* that references the value of that variable. For example, in Figure 2-4, the node labeled *n2* contains a computational use or c-use of *y*.

Since data-flow analysis for the purpose of data-flow testing is concerned with tracing definitions whose uses do not occur in the same node, we make a distinction between a *local* c-use and *global* c-use. A c-use of a variable *x* is said to be global at a node *n* if no definition of *x* semantically precedes the c-use in *n*. That is, the value of *x* must have been bound to it in some other node $n \in CFG$.

A node $n \in CFG$ contains a predicate use or *p-use* of a variable if the last statement in the corresponding block *b* contains a predicate statement where the value of that variable is used to decide whether a predicate is true for selecting execution paths. A conditional transfer statement in a block *b* corresponding to a node $n \in CFG$ has two executional successors: n_i and n_j, such that $i \# j$. Since the value of the variable occurring in the predicate statement determines whether n_i or n_j will be executed next, a *p-use* is associated with edges rather than with the node in which the predicate statement occurs. For example, in Figure 2-4, edges (*n1*, *n2*) and (*n1*, *n3*) contain a *p-use* of *y*. Since p-*uses* are associated with edges, no distinction need be made between local *p-uses* and global *p-uses*.

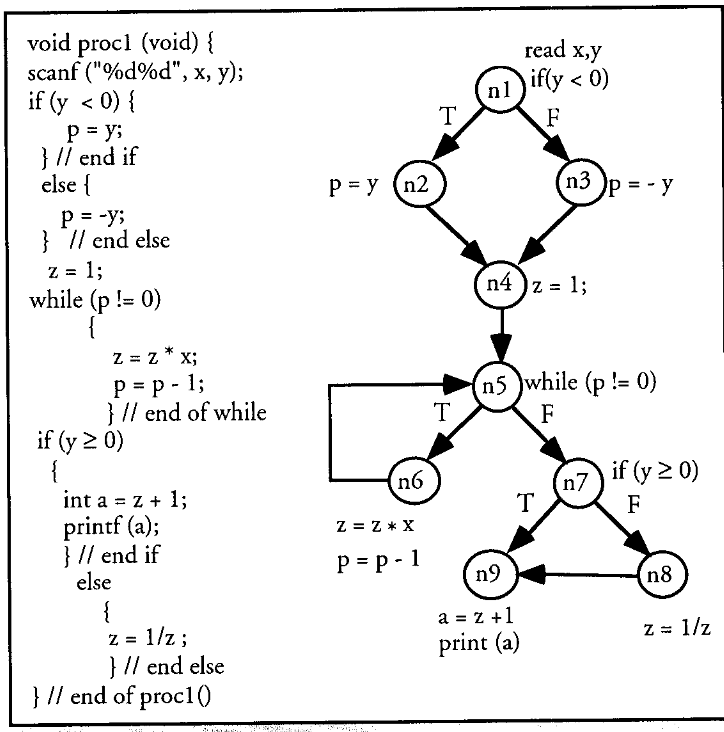

Figure 2-4. A sample C program and its control flow graph (*CFG*).

Data-flow test adequacy deals with subpaths from definitions to nodes where those definitions are used or definition-use chains. Let $P = \{p_1, p_2, p_3,..., p_n\}$ be the set of all subpaths from definitions to nodes where those definitions are used. We say that a definition-clear path with respect to a variable x is a path p_i $(1 < i < n)$ such that for all nodes N in p_i there is no definition occurrence of x. A definition of x at a node $u \in CFG$ reaches a c-use of x at node $v \in CFG$, iff there is a path p_i from u to v such that $p_i = (u, w_1, w_2,..., w_m, v)$ and $(w_1, w_2,...,w_m)$ is definition-clear with respect to x, and the occurrence of x at v is a global c-use. Similarly, a definition of x at u reaches a p-use in w_m, if there is a path $p_i = (u, w_1, w_2,..., w_m, v)$ from u to v, and $(w_1, w_2,..., w_m)$ is definition-clear with respect to x, and there is a predicate occurrence of x associated with the edge (w_m, v). We say that a definition feasibly reaches a use on a path $p \in P$, iff there exists at least one input datum that can cause the execution or traversal of p in *CFG*.

Table 2-1. Tabulated p-uses and their associated edges in the *CFG* of Figure 2-4.

P-USE	edges
{y}	(n1, n2)
{y}	(n1, n3)
{p}	(n5, n6)
{p}	(n5, n7)
{y}	(n7, n8)
{y}	(n7, n9)

To incorporate definition and use information into each node $n \in$ *CFG,* two sets, DEF[n] and C-USE[n], are associated with each node n, and one set, P-USE[n_i, n_j], is associated with each edge containing a predicate use. These sets are defined as follows: DEF[n] is the set of variables for which n contains a global definition; C-USE[n] is the set of variables for which n contains a global c-use; and P-USE[n_i, n_j] is the set of variables for which edge (n_i, n_j) contains a predicate use. Table 2-1 illustrates the p-uses and their associated edges in the *CFG* of Figure 2-4. Similarly, Table 2-2 illustrates the definitions and their c-uses for each node in the *CFG* of Figure 2-4.

Table 2-2. Tabulated definitions, and their c-uses in each node in the *CFG* of Figure 2-4.

node	DEF	C-USE
1	{x, y}	{ø}
2	{p}	{y}
3	{p}	{y}
4	{z}	{ø}
5	ø	{ø}
6	{z, p}	{x, z, p}
7	{ø}	{ø}
8	{z}	{z}
9	{a}	{z, a}

After associating the DEF, C-USE, and P-USE sets with the appropriate nodes and edges in the *CFG,* the definition predicate-use (DPU) and the definition computation-use (DCU) sets can be defined and associated to each node $n \in$ *CFG* as follows: Let n_i be any node in a *CFG*

and x any variable $\subset$ DEF[n_i], we say that DCU[x, n_i] is the set of all nodes n_j such that $x \in$ C-USE[n_j] for which there is a definition-clear path with respect to x from n_i to n_j. A DPU[x, n_i] is the set of all edges (n_j, n_k) such that $x \in$ P-USE[n_j, n_k] for which there is a definition-clear path with regards to x from n_i to n_j. Table 2-3 illustrates the DPU and DCU sets for the example in Figure 2-4. We say that a path (n_1,...., n_j, n_k) is a definition-use path or *du-path* with respect to a variable x if n_1 has a global definition of x and either n_k has a *c-use* of x and (n_1,...., n_j, n_k) is a definition-clear simple path with respect to x, or (n_j, n_k) has a *p-use* of x and (n_1,...., n_j) is a definition-clear loop-free path with respect to x.

Table 2-3. The DCU and DPU of the program depicted in Figure 2-4

DCU (x, n1) = {n6}	DPU (x, n1) = {ø}
DCU (y, n1) = {n2, n3}	DPU (y, n1) = {(n1, n2), (n1, n3), (n7, n8), (n7, n9)}
DCU (p, n2) = {n6}	DPU (p, n2) = {(n5, n6), (n5, n7)}
DCU (p, n3) = {n6}	DPU (p, n3) = {(n5, n6), (n5, n7)}
DCU (p, n6) = {n6}	DPU (p, n6) = {(n5, n6), (n5, n7)}
DCU (z, n4) = {n6, n8, n9}	DPU (z, n4) = {ø}
DCU (z, n6) = {n6, n8, n9}	DPU (z, n6) = {ø}
DCU (z, n8) = {6}	DPU (z, n8) = {ø}

2.4.2 Intraprocedural Data-flow Adequacy Criteria

Data-flow testing of a procedure or intraprocedural data-flow testing, considers the flow of data within a procedure, while assuming some approximation about definitions and about uses of reference parameters and global variables at call sites. A call site in imperative languages is a statement S that invokes a procedure. For example, S_1: x= $B(y)$ is a statement that contains a call site that invokes a procedure B, such that y, an actual parameter in procedure A, is passed into the argument list of A. Once y is bound in the called procedure to its corresponding reference parameter, say z, it is assumed that z is neither defined nor used inside A. This assumption is based on the fact that intraprocedural data-flow testing is concerned with testing one procedure at a time. Thus, in the data-flow analysis of procedure A we say that y is used at S_1. There are three prominent groups of intraprocedural data-flow adequacy criteria in the testing literature. These criteria will be reviewed next.

2.4.3 The Rapps and Weyuker Criteria Family

Rapps and Weyuker [76] proposed a family of test adequacy criteria based on the data-flow information of a program. The foundation of this family is to compute definition-clear sub-paths from each definition to all or some use(s). This family of criteria is mainly concerned with the simplest type of data-flow paths which start with a definition of a variable and end with a use of the same variable. This family includes: *All-definitions*; *All-uses*; *All-cuses/Some-puses*; *All-puses/Some-cuses*; *All-puses*; and *All-du paths*. In [29], Frankl and Weyuker redefined the original dataflow criteria [76] because it did not take into account the feasibility issue. The feasibility issue says that for a given program *Q* and a data-flow criterion *C*, it may be the case that no test data for *Q* satisfies *C*. This scenario occurs when none of the paths that cover a particular du-association required by *C* is executable. In such a scenario, *Q* cannot be adequately tested according to *C*. The new family of adequacy criteria was called the feasible data flow criteria, and it required that test data exercise only those du-associations that are executable. Next we discuss each member of the feasible family of test adequacy criteria.

2.4.3.1 The All-definitions

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of all possible executable paths in a control flow graph *G* representing a program *Q*. As illustrated in Figure 2-5, the All-definitions criterion or All-defs requires tracing some definition-clear subpath of a variable *x* at a node n_i to some use reached by that definition at a node n_j. An adequate test should cover all definition occurrences in the sense that for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition. In general, we say that a definition *d* of a variable *x* reaches a use *u* at a point *pnt* in the program if there is a path *p* from *d* to *u*, such that *x* is not redefined or killed along *p*.

Formally, a set *L* of execution paths, $\subset P$, satisfies this criterion, iff for all definition occurrences of a variable *x* such that there is a use of *x* which is feasibly reachable from the definition, there is at least one path $p \subset L$ that includes a subpath through which the definition of *x* reaches some use occurrence of *x*.

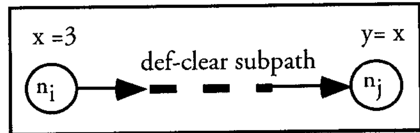

Figure 2-5. A definition-clear subpath wrt. *x* to a use reached by that definition.

For example, in Figure 2-6, a satisfying All-defs with regards to variable *x* is: (*n1*, *n3*). The reader should note that, in this criterion, errors that occur in un-executed nodes or untraversed edges would go undetected.

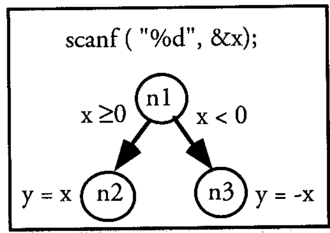

Figure 2-6. A *CFG* illustrating the All-defs criterion.

2.4.3.2 The All-uses

This criterion is based on the fact that a definition occurrence of a variable may reach more than one use occurrence. As depicted in Figure 2-7, the definition of *x* in n_i reaches, via definition-clear subpaths with regards to *x*, uses of *x* in n_j, n_k, and n_l, repectively. Thus, this criterion is concerned with tracing definition-clear subpaths with regards to a variable *x* at a node n_i to each use (n_j, n_k, n_l) reached by that definition.

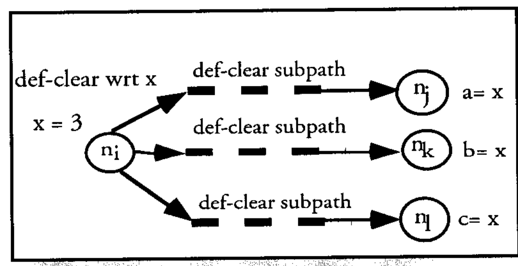

Figure 2-7. A definition-clear subpaths wrt. *x* illustrating the All-uses criterion.

Formally, a set L of execution paths, $L \subset P$, satisfies this criterion iff for all definition occurrences of a variable x, and all the use occurrences of x that the definition feasibly reaches, there is at least one path p in L such that p includes a subpath through which that definition reaches the use. For example, in Figure 2-8, the All-uses paths with regards to x are: ($n1$, $n2$, $n4$, $n5$, $n7$), and ($n1$, $n3$, $n4$, $n6$, $n7$). The All-uses criterion was also proposed by Herman [1976], but was called the *reach coverage* criterion.

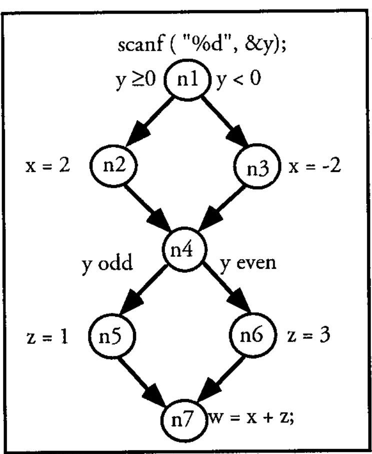

Figure 2-8. An example illustrating the All-uses criterion.

2.4.3.3 The All-definition-use Paths

This criterion is concerned with tracing all definition-clear subpaths that are cycle-free or simple-cycles from each definition of a variable x to each use reached by that definition and each successor node of the use. For example, in Figure 2-9, the definition of x at node n_i reaches, via three definition-clear subpaths the use of x at n_j. Given a definition occurrence of a variable x and a use of x that is reachable from that definition, there may exist many paths through which the definition reaches the use. Since there may exist an infinite number of infeasible paths, the applicability of this criterion restricts these paths to be cycle-free or simple cycles.

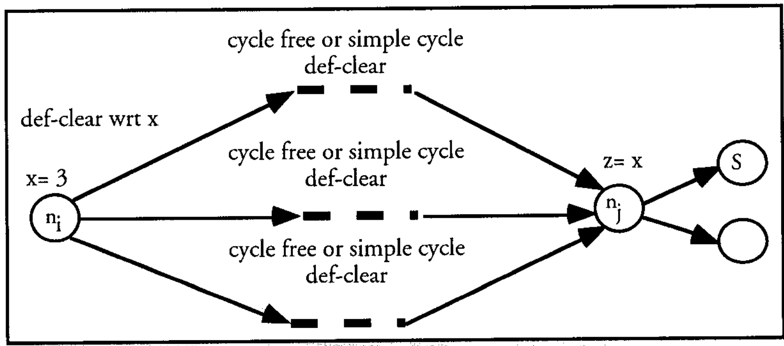

Figure 2-9. Definition-clear subpaths wrt. *x* illustrating the All-du-paths criterion.

Formally, a set L of execution paths, $L \subset P$ satisfies this criterion iff for all definitions of a variable x and all paths $M \subset L$ through which that definition reaches a use of x, there is at least one path p in L such that M is a subpath of L, and M is cycle-free or contains only simple cycles. For example, in Figure 2-8, the All-du-paths with regards to x are: (*n1, n2, n4, n5, n7*), (*n1, n3, n4, n6, n7*), and (*n1, n2, n4, n6, n7*). The the All-du-paths subsumes the All-uses criterion because the All-du-paths requires that test data be included which cause certain combination of path segments to be traversed, whereas the All-uses criterion does not.

2.4.3.4 The All-c-uses/Some-p-uses and The All-p-uses/Some-c-uses

Emphasis in these criteria is placed on either computational uses or predicate uses. The All-c-uses/Some-p-uses criterion with regards to a variable *x*, requires that all *x*'s computational uses be exercised; however, if no c-uses exist for *x* one p-use, at least, should be exercised. In contrast, the All-p-uses/Some-c-uses places emphasis on p-uses. It requires paths that exercise all p-uses, and exercise at least one c-use, when there is no p-use. Two more weaker criteria were also defined. These criteria are: All-p-uses and All-c-uses. The All-p-uses ignores all the computational uses, whereas the All-c-uses ignores all the predicate uses.

2.4.4 The Ntafos Criteria

Ntafos [66] proposed a family of test adequacy criteria the foundation of which is the observation of how the values of different variables interact. The family of adequacy criteria is called the *required k-tuples*, where *k* is a natural number >1. The required k-tuples require that a path set *P* covers chains of alternating definitions and uses, or definition-reference interactions called *k-dr* interaction (*k* > 1). Each definition in a k-dr interaction reaches the next use in the chain, which occurs at the same node as the next definition in the chain. For example, in Figure 2-10, a 1-dr interaction for *k* = 1, is a definition-clear path with regards to *x* from its definition at a node n_1 to its use at a node n_2. Also a 2-dr interaction for *k* = 2, is a definition-clear path with respect to *x* from its definition in n_3 to its use in the definition of *y* in n_4, and a definition-clear subpath with regards to *y* from its definition in n_4 to its use in n_5. We formally define a k-dr interaction next.

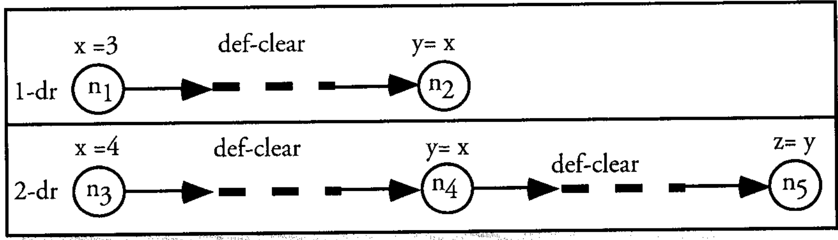

Figure 2-10. Ntafos 1-dr and 2-dr interaction paths.

2.4.4.1 A k-dr Interaction

In a k-dr interaction, the definition of a variable x_1 in a statement s_1 (i.e, the first definition) and the reference x_{L-1} in s_l (the last reference) are treated as the focal points of the interaction "or as a path that starts at the beginning and ends at the end". The rest of the interaction may be viewed as a sequence of events leading up to the last reference or a sequence of actions that follows the first definition. In general, there will be a set of k-dr interactions that have the same last reference, each representing one of a variety of paths along which that reference can be reached. Similarly, there will be a set of k-dr interactions that have the same first definitions, each describing one of a variety of sequences of that definition. A k-dr interaction specifies that the reference to x_i in s_{i+1} is used directly (in the same statement) to define x_{i+1} in s_{i+1}. Given

that $1 \leq i \leq k$, a k-dr interaction can be formally defined as a sequence $S = [d_1(x_1), u_1(x_1), d_2(x_2), u_2(x_2), ..., d_k(x_k), u_k(x_k)]$ where: $d_i(x_i)$ is a definition of variable x_i; $u_i(x_i)$ is a use of variable x_i; both $u_i(x_i)$ and $d_{i+1}(x_i)$ are associated with the same node n_{i+1}; and for all i, the *ith* definition reaches the *ith* use. Given that $i \leq 1 \leq k$, an interaction path for a k-dr interaction is a path $P = (n_1) \bullet p_1 \bullet (n_2) \bullet p_2 \bullet (n_3) \bullet ... \bullet (n_{l-1}) \bullet P_{l-1} \bullet (n_l)$ such that, $d_i(x_i)$ reaches $u_i(x_i)$ through p_i. The required k-tuples criterion which we formally define next, thus requires that all k-dr interactions be tested.

2.4.4.2 The Required k-tuples Criteria

The *required k-tuples* requires some subpath propagating each k-dr interaction such that (1) if the last use is a predicate the propagation should consider both branches, and (2) if the first definition or the last use is in a loop, the propagation should consider either a minimal or some larger number of loop iterations. Thus, a required k-tuples is a class of strategies obtained by varying k. This means that higher values of k result in a more complex data flow interaction. A set P of execution paths satisfies the required k-tuples iff for all j-dr interactions L, $1 < j \leq k$, there is at least one path p in P such that p includes a subpath which is an interaction path for L. The *CFG* in Figure 2-11 illustrates a 1-dr interaction and its satisfying paths.

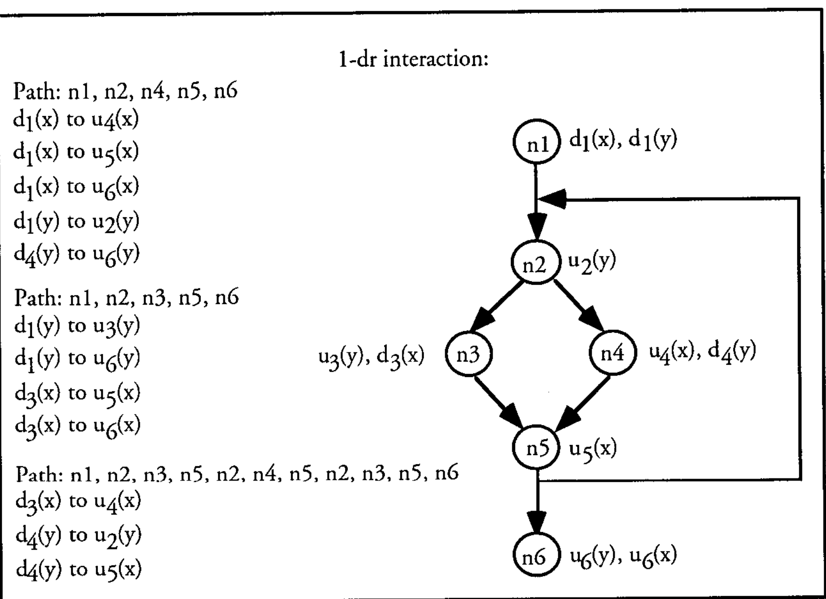

Figure 2-11. An example illustrating the required k-tuple criteria.

2.4.5 The Laski and Korel Criteria

Laski and Korel [56] defined a set of adequacy criteria that require a combination of definitions that reach uses at some node via a subpath. These criteria emphasize the fact that a given node may contain uses of several different variables, where each use may be reached by several definitions occurring at different nodes. Thus, these criteria are concerned with selecting subpaths along which the various combinations of definitions reach the desired node. These criteria are: the *reach coverage*; *context coverage;* and the *ordered context coverage.*

2.4.5.1 The Reach Coverage Criterion

The reach coverage is analogous to the All-uses criterion introduced by Weyker [76]. It requires some definition-clear subpath from each definition to all uses reached by that definition. This criterion is satisfied when a path set contains at least one subpath between each definition and each use reached by that definition. Thus a set P satisfies this criterion iff for all definitions $d_m(x)$ and all uses $u_n(x)$ reached by $d_m(x)$, P contains at least one subpath $(m) \bullet p \bullet (n)$, such that p is a definition-clear with respect to variable x.

2.4.5.2 The Context Coverage Criterion

The Context Coverage Criterion requires some subpath along which each set of definitions reaches uses at each node. For example, in Figure 2-12, the definition of x in *n1*, reaches the definition of y in *n2* via a definition-clear path with respect to x, then the definition of y in *n2*, reaches the definition of z in *n3* via a definition-clear path with respect to x and y, and finally the definition of z in *n3*, reaches the use of x, y, and z in node *n4* via a definition-clear path with respect to x, y, and z. Before this criterion can be formally defined, the following definition is needed.

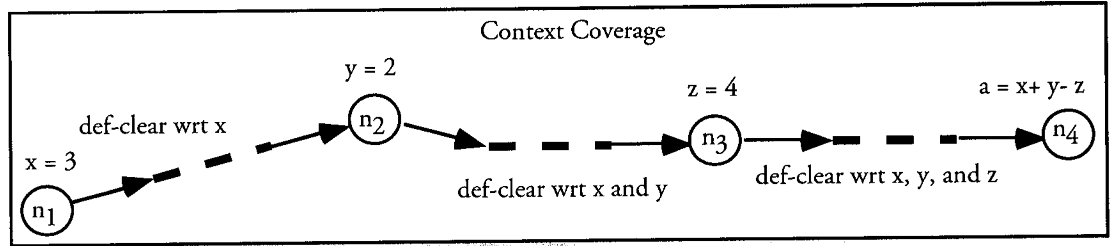

Figure 2-12. An example illustrating the concept of the context coverage criterion.

Definition Context (*DC*): A definition context of a node *n* is the set of definitions $DC(n)$ = $[d_1(x_1), d_2(x_2),..., d_k(x_k)]$, some permutation of which is an ordered definition context of *n*. In general, there is a many-to-one relationship between the ordered definition contexts of *n*, which are sets of definitions, and the definition contexts of *n*, which are sequences of definitions. Having defined the *DC*, the context coverage criterion can be formally defined as follows: a set *P* of execution paths satisfies the Context Coverage criteria iff for all nodes *n* and for all $DC(n)$, there is at least one path *p* in *P* such that *p* is a context subpath for $DC(n)$. The example in Figure 2-13 illustrates what we have described formally for the Context Coverage criterion.

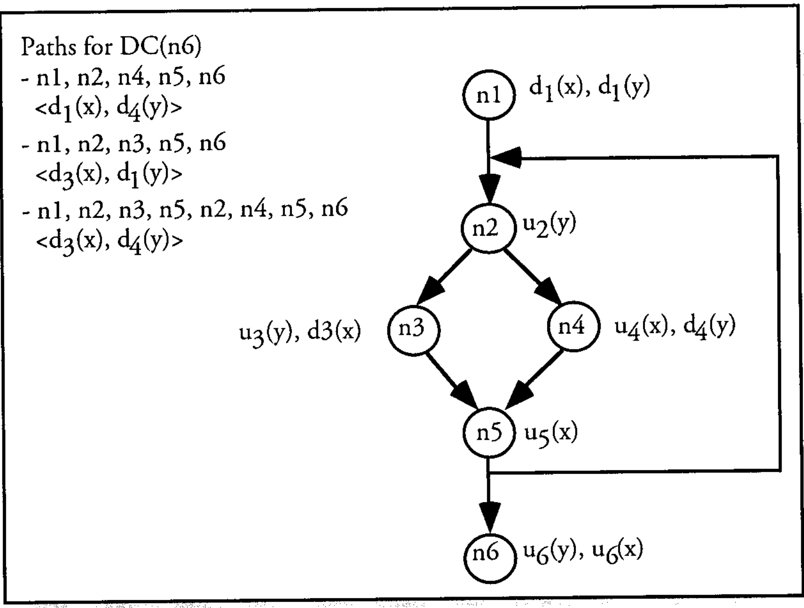

Figure 2-13. An example of a Context Coverage.

2.4.5.3 The Ordered Context Coverage Criterion

To illustrate the concept of this criterion, consider the trivial example in Figure 2-14. The sequence of definitions <*n1, n2, n4*>, <*n1, n2, n3, n4*>, and <*n1, n2, n3, n2, n3, n4*> in Figure 2-14 is an ordered context coverage with respect to *x*, *y* and *z*. Thus, the ordered context coverage criterion requires that some subpath along which each sequence of definitions reach uses at each node. Before this criterion can be formally defined, the following definition is needed.

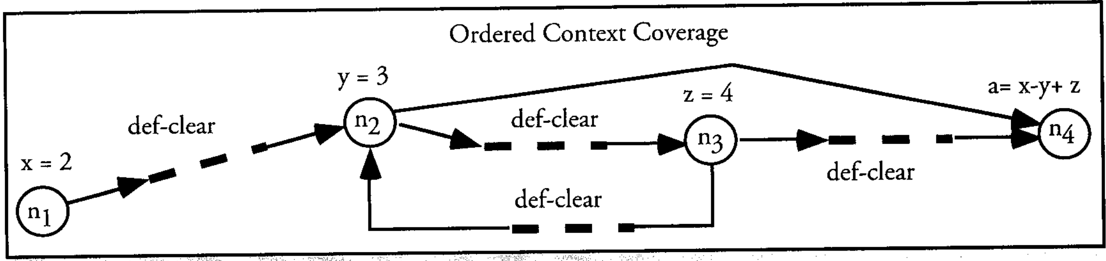

Figure 2-14. Illustrating the concept of the ordered context coverage criterion.

Ordered Definition Context (*ODC*): an ordered definition context is sequence of definitions that occur along the same subpath and that reaches uses at that node via the subpath. The order of the definitions in the sequence is the same as their order in the subpath. An ordered context subpath for an ordered definition context is a subpath along which the ordered definition context occurs. Let n be a node in a control flow graph G, and let $\{x_1, x_2,\ldots, x_k\}$ be the (non-empty) subset of variable use occurrences in n. An ordered definition context of node n or $ODC(n)$ is a sequence of definitions $[d_1(x_1), d_2(x_2),\ldots, d_k(x_k)]$ associated with nodes $n_1, n_2,\ldots, n_k$ such that there exists a path $P = p_1 \bullet (n_1) \bullet p_2 \bullet (n_2) \ldots \bullet p_k \bullet (n_k) \bullet p_{k+1} \bullet (n)$ called an ordered context path for the node n with respect to the sequence $[n_1, n_2,\ldots, n_k]$, iff for all $i = 2,3,\ldots, k$, the subpath $p_i \bullet (n_i) \bullet p_{i+1} \bullet (n_{i+1}) \bullet \ldots \bullet p_{k+1}$ is definition-clear with respect to x_{i-1}. Having defined the ODC, the ordered context coverage criterion can be formally defined as follows: Given a set P of execution paths, this criterion is satisfied iff for all nodes n and all ordered definition contexts $ODC(n)$, there is at least one path p in P such that p is a context subpath for $ODC(n)$. The example in Figure 2-15 illustrates what we have described formally for the ordered context coverage criterion.

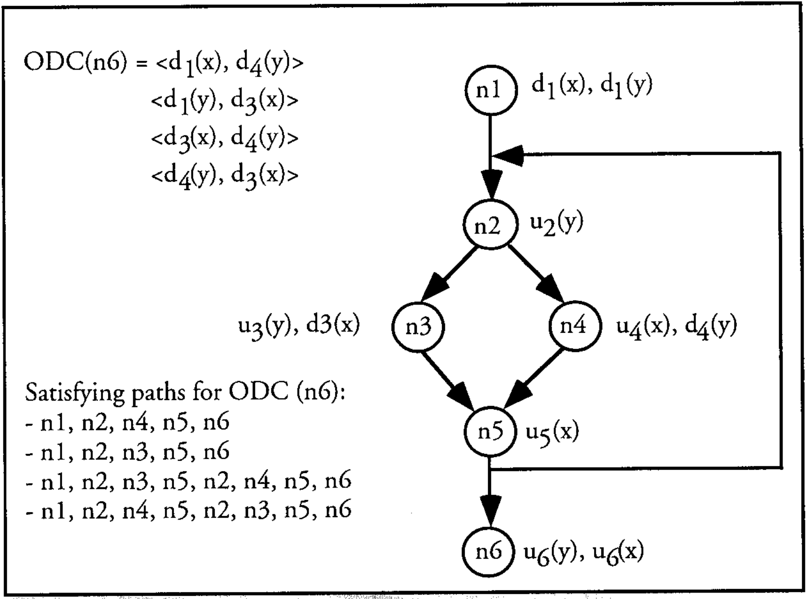

Figure 2-15. An example of an Ordered Context Coverage.

2.4.6 Data-flow Testing for Structured and Dynamic Data

The data flow testing strategies discussed thus far have made no distinction between atomic data such as integers and structured data such as arrays. Hamlet et al. [39] argue that treating structured data as aggregate values may lead to two problems. The first problem occurs when a du-path is identified but it is not present for an array element. For example, consider the code fragment that exchanges two distinct elements in array *M*, using *M*[0] as temporary in Figure 2-16.

```
M[0] = M[i];
M[i] = M[j];
M[j] = M[0];
```

Figure 2-16. A code fragment illustrating du-path issues in arrays.

Treating *M* as a single value would results in a du-path from each statement to the next. However, a closer inspection of the code reveals that the only actual du-path involving *M*[0] is in the first and last statement. The second problem occurs when a path is missed because of a false

intermediate assignment. For example, in Figure 2-16, the actual du-path is the only one that will not be identified since it appears to be "falsely" interrupted by the middle statement. Those two problems are known as the commission and omission mistakes, repectively. One solution to circumvent these mistakes is to treat each element in a structured data as an independent atomic data. For example, array access can be treated in a similar fashion to simple variables. An access can be either a definition $M[x] = m$, or reference $n = M[y]$. A definition $M[x] = m$, may occur in some statement s_1, and $m = M[x]$ can be referenced in some other statement s_n (many statements later), iff $M[a]$ has not been redefined anywhere between s_1 and s_n. The presence of subscripts also complicates conditions under which a test case is satisfied, and can give rise to non-satisfiable test cases [94]. For example, there is no easy way to tell if two subscript expressions can ever be the same, or to tell if they are necessarily different. A partial solution to this problem was proposed by Hamlet et al. [39]. The partial solution uses a symbolic equation solver to determine whether two occurrences of an array element are indeed the same element.

The dataflow testing strategies discussed thus far also did not take into account the effect of dynamic data such as pointers. The presence of pointers in imperative languages makes dataflow analysis more complex, for they cause uncertainty regarding what is defined and used. One safe assumption that can be made with regards to what a pointer *p* points to given that we know nothing about *p* is to assume that it is a potentially going to change or define any variable. By the same token we assume that any use of the data pointed to by *p* can potentially use any variable. This assumption can result in inconclusive testing results especially in programs with more complex pointers interactions. A technique in [68] proposes a solution to deal with one level-pointers in the C language.

2.5 The Structural Testing Subsume Hierarchy

As mentioned in Section 1.4.2, a test strategy *A* is said to subsume some other strategy *B* if all the elements that *B* exercises are also exercised by *A*. The subsume hierarchy is an analytical ranking of coverages. Many researchers have compared the test strategies that make up the subsume hierarchy; however, no generalizable results about relative bug-finding effectiveness have been established that correlate with this ranking.

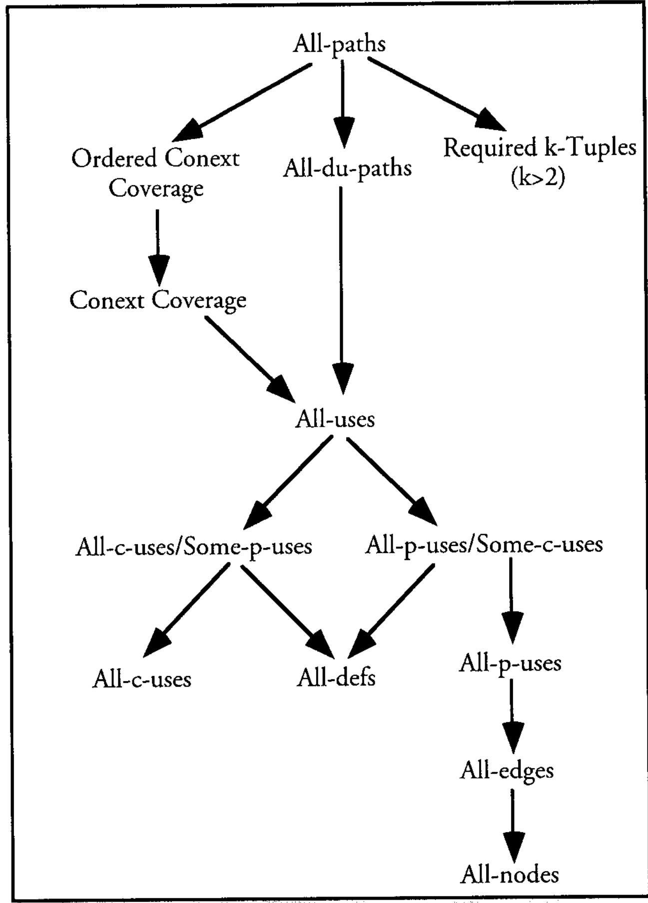

Figure 2-17. The subsume hierarchy for structural-based testing.

Since nothing conclusive can be inferred about the number and kind of bugs that remain, reaching a coverage goal does not indicate that a higher criterion is necessarily better at finding bugs for a particular application and vise versa [12]. Figure 2-17 depicts, based on the result of the work done by [94], the subsume hierarchy of the test adequacy criteria presented thus far.

2.6 Integration Testing

Unit testing focuses on individual components. Once faults in each component or unit have been removed and test cases do not reveal any new faults, units are ready to be integrated into a larger subsystem. At this point, components are still likely to contain faults. These faults stem from the interactions of integrated procedures. To test these faults interprocedural testing is

required. Interprocedural data-flow testing is the topic of discussion in Section 2.7. Integration testing allows the testing of increasingly more complex parts of the system while keeping the location of potential faults relatively small. This means that the most recently added component is usually the one that triggers the most recent fault discovery. Several approaches have been devised to accomplish integration testing and they are: big bang testing; bottom-up; top-down; and sandwich testing. Each approach will be briefly discussed next.

- The big bang strategy assumes that all components are first tested individually and then tested together as a single system. Although it sounds simple, big bang testing is expensive. This is mainly due to the fact that if a test case uncovers a fault, it is sometimes difficult to accurately pinpoint the specific area in the code responsible for the error.
- The bottom-up approach strategy first tests each component of the bottom layer and then integrates them with components of the next layer up. A layer in this context indicates the place of a component in the calling graph of a program. When two components are tested together, we call this a double test. Testing three components together is a triple test. Likewise, testing four components is called quadruple test [35] [79]. This process is repeated until all layers are combined together.
- The top-down testing strategy unit tests the components of the top layer first and then integrates the components of the next layer down. Once all components of a new layer have been tested together, the next layer in the layer hierarchy is chosen
- The sandwich testing strategy combines both the top-down and the bottom-up strategies in an effort to make use of the best of both strategies.

2.7 Interprocedural Data-flow Testing

The data-flow testing methods discussed thus far have been restricted to testing data dependencies that exist within a program unit or intraprocedural testing. In this section we will discuss the concept of testing a program as a whole or interprocedural testing. In general, given a program P consisting of a set of procedures $P = \{p_1, p_2, \ldots, p_n\}$ each p_i in P $(2 \leq i \leq n)$ is represented by its own *CFG* where local information about formal and actual parameters at call and return sites are collected. The call site is the point before we enter a called procedure or a sub-

routine, and the return site is the point after we "come back" or return from the called procedure. Information about the interaction among procedures is then used to connect the subgraphs to construct the interprocedural or "global" *CFG*. Using structural testing criteria such as data-flow testing can then be applied to test the program as a whole. Issues relating to the identification and computation of interprocedural data-flow analysis for imperative languages will be discussed next.

2.7.1 Issues in Computing Interprocedural Definition-use Chains

The basic idea of interprocedural data-flow testing is to test the data dependencies that stretch across procedure boundaries. With imperative languages, there are two types of interprocedural data dependencies: direct data dependencies; and indirect data dependencies. A direct data dependency is a du-association whose definition occurs in a directly called procedure *p2* of *p1*. This dependency exists when: (1) a definition occurrence of an actual parameter in one procedure reaches a use occurrence of the corresponding actual parameter at a call site, (i.e. procedure call); (2) a definition occurrence of a formal parameter in a called procedure reaches the use of the corresponding actual parameter at the return site, (i.e. call return); and (3) a definition of a global variable reaches a call or return site. A formal parameter exists in the input parameter list of a called procedure, whereas an actual parameter is a variable in the calling procedure that reaches the call site. At call sites, actual and formal parameter are bound.

An indirect data dependency is a du-association whose definition occurs in procedure *P* and use occurs in an indirectly called procedure *Q* of *P*. The conditions governing an indirect data dependencies are similar to those for direct data dependency, except that multiple levels of procedure calls and return are considered. Indirect data dependencies can be determined by considering the possible uses and definitions along the calling sequence. Given the aformentioned conditions under which either a direct or indirect data dependency may occur, data-flow adequacy criteria designed for unit testing can be extended to cover variable definitions and their uses which extend beyond a program unit's boundary. This type of dependency between a definition in one procedure and its uses, via its corresponding formal parameter in another procedure, is known as interprocedural data dependency. Thus, interprocedural data-flow testing is

concerned with testing these dependencies. In this section we discuss issues concerning the identification of interprocedural dependencies.

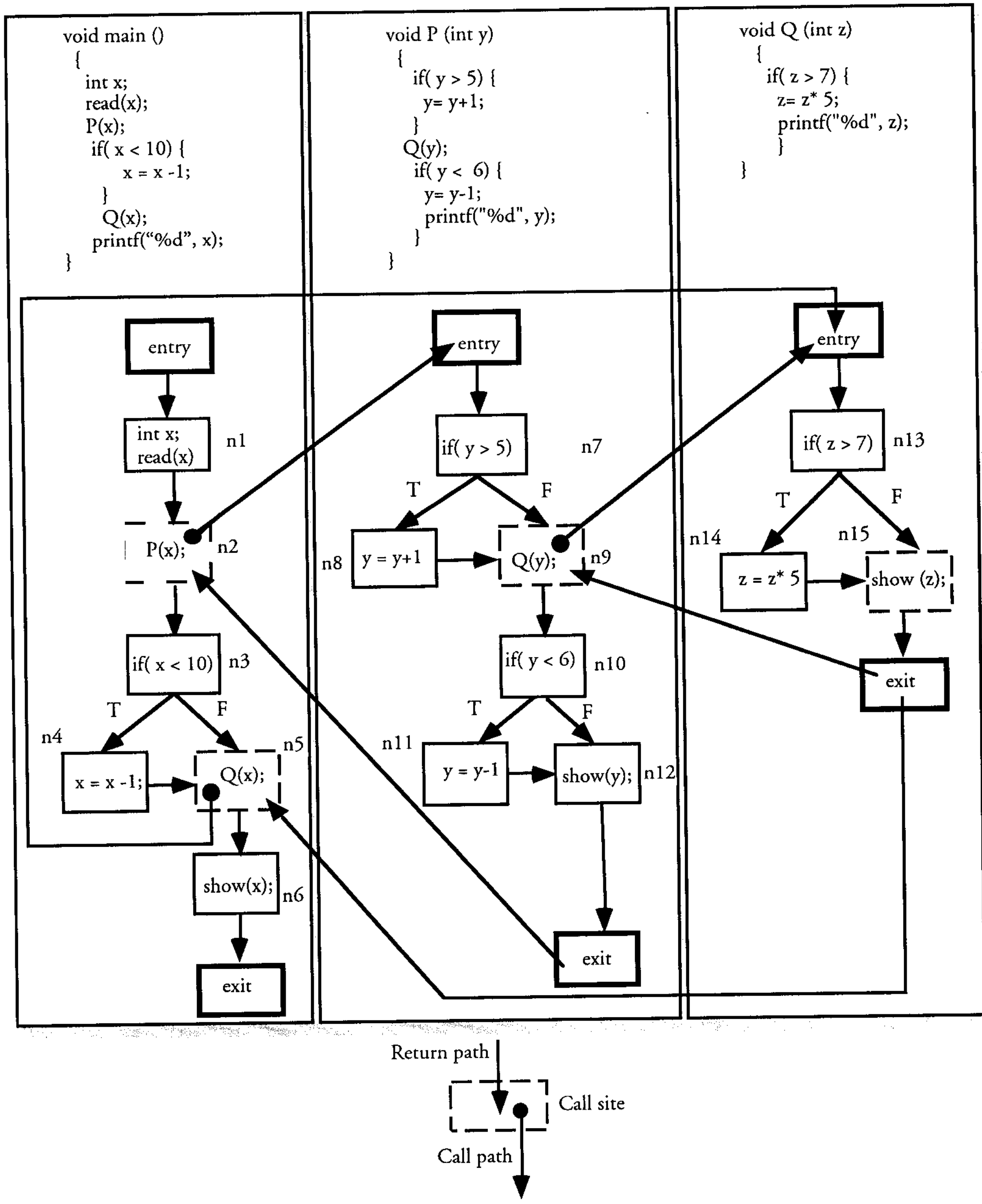

Figure 2-18. An example illustrating the interprocedural dataflow analysis.

Consider the example in Figure 2-18. In that example, there is a three procedure program: *main*; *P*; and *Q*. Each procedure's *CFG* is depicted beneath its code. To simplify the interprocedural dependency discussion, we represent a call site by a single basic block, shown as a dashed box. Arrows whose source are a filled circle ●⟶ represent a call path, whereas

simple arrows represent a return path. We also associate a node number with each node or block in the *CFG*(s). There are three main issues related to interprocedural du-chain analysis in imperative languages: preserving a variable over a procedural call, preserving the calling context, and dealing with aliases. Each issue will be discussed next.

• **Preserving a variable over procedure call.** In Figure 2-18, the definition of *x* in *n1* may be preserved over the call to *P* in *n2.* This means that when *x* is bound to *y* in *P*, *y* can reach, unchanged, the call to *Q* in *n9*, which causes *y* to be bound to *z* in *Q*. In the latter, there is a path on which *z* can reach, unchanged, the end of *Q* and subsequently back into *P* via the return path from the exit node of *Q* to the dashed box *n9* in *P*. In the latter, *y* can reach, unchanged, the end *P* and back into *main* via the return path from the exit node of *P* to the dashed box *n2* in *main.*

Thus, information about the interprocedural data-flow in *P* and *Q* is required to determine that the definition of *x* in *n1* may be preserved over the call to *P* in *n2* [58], and subsequently reaches the use of *x* in *n4 n5*, and *n6*. Moreover, tracking the definition of *x* in *n1* over procedure calls and returns is required to determine that this definition of *x* (in *n1*) reaches the uses of *y* in *n7*, *n8*, *n10*, *n11*, and *n12* in *P*, and the uses of *z* in *n13*, *n14*, *and n15* in *Q*.

Gathering this information requires that either (1) the information about called procedures be incorporated at call sites during the analysis of the calling procedure or (2) an estimate of the information about called procedures be used during initial analysis of the calling procedure and that this information be updated when more accurate data-flow information is determined. The problem with the first method is that, if procedures are processed in any order or are recursive, incomplete information may be available about called procedures at call sites. With the second method, each procedure can be individually processed to abstract the intraprocedural information, and an estimate about the definition and use information at call sites. This initial estimate can then be updated by propagating information about other procedures over the nodes and edges of a graph that represents the entire program. For the sake of simplicity, we refer to this graph as the "global" graph of a program.

• **Preserving the calling context of called procedures.** To preserve the calling context during the computation of interprocedural du-chains, only those paths through the program that agree with the call sequence for some possible control path should be traversed when tracking data-flow pairs over return paths from procedures. For example, consider the definition of *x* in *n4* that reaches the call to procedure *Q* in *n5* in Figure 2-18. Since there is a path through *Q* on which *z* is not re-defined, the definition of *x* in *n4* can reach the end of *Q*. Since there are two calls to *Q*, there are two return paths from *Q*: one that returns directly to *main* and the other that returns indirectly through *P*. Ignoring the call sequence suggests that the definition of *x* in *n4* has uses in *n3, n4, and n6*. However, a closer inspection of control paths through the program reveals that this definition in *n4* reaches the end of *Q*, and subsequently back into *main*, only when it is called directly from *main*. Thus, this definition can only reach the use of *x* in *n6*.

• **Dealing with aliases.** The presence of aliases complicates the identification of precise interprocedural du-chains. An alias may be introduced at the call site, if the variable reaching the call site is passed in two different locations in the parameter list of the call. To illustrate, we refer now to the example in Figure 2-19. In that figure, variable *x* in *n1* is passed in two locations in the parameter list of the call to *P*. At the call site of *P*, formal parameters *y* and *i* are aliases of each other. Then, in *n6*, this alias is propagated to procedure *B* because *y* and *i* are passed as parameters, thus causing *z* and *j* to be aliased. If we ignore the effects of aliasing, the only definition in *B* that reaches the use of *y* in *n7* is the definition of *z* in *n9*. Acknowledging the presence of aliases reveals that, when *y* and *i* are aliased, the definition of *j* in *n10* also reaches the use of *y* in *n7*.

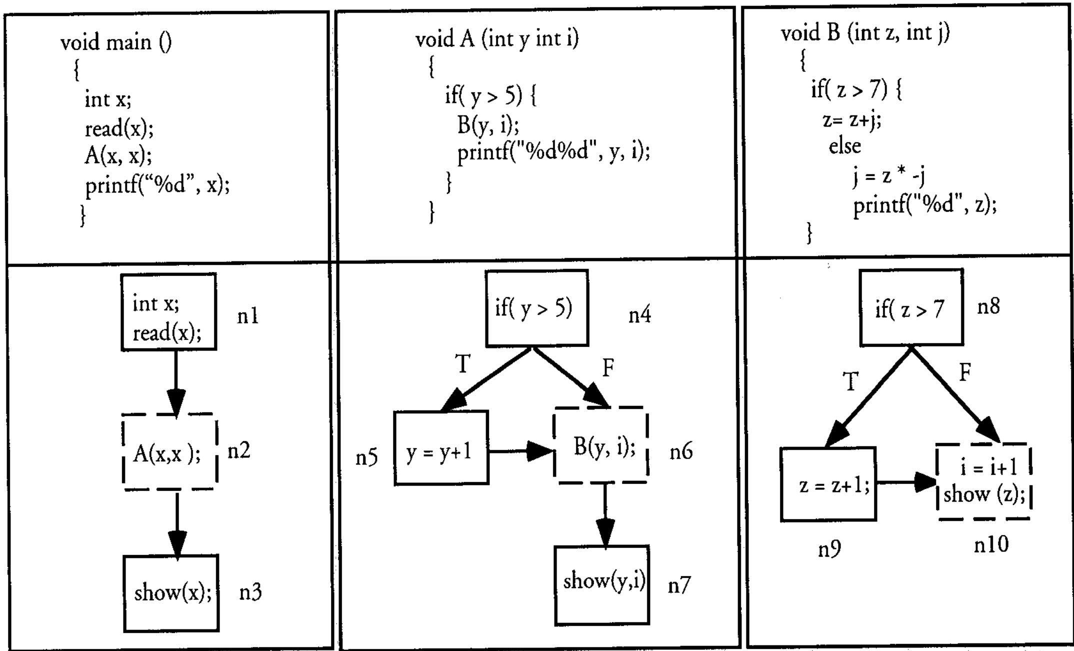

Figure 2-19. An example illustrating the presence of aliases in imperative languages.

A number of data-flow analysis techniques have been developed to compute interprocedural dependency information. Some existing flow-insensitive[1] data-flow analysis techniques such as Banning [7], Barth [6], Cooper and Kennedy [22], and Lomet [58] provide summary data-flow information for determining the local effects of called procedures at call sites. These techniques do not provide information about the locations of interprocedural definitions and uses in other procedures in the program. A flow sensitive technique that processes non-recursive procedures in reverse invocation order Allen et al. [3] incorporates the abstracted information about called procedures at call sites to obtain the local reaching information. This technique requires that a procedure be processed only after those that it calls have been processed, which imposes an ordering on the procedure processing. This order restriction results in a penalty when changes are made in a procedure, for it causes the reanalysis of those procedures directly or indirectly dependent on the changed procedure. Also, this technique does not compute the

1. Interprocedural data-flow information is flow-insensitive if the control flow of called procedures is not used in the computation of definitions and uses in the program

locations of the definition-use chains across procedure boundaries and cannot handle recursive procedures with the ordering restriction.

The program summary graph developed by Callahan [19] provides flow-sensitive[1] interprocedural data-flow information that solves the interprocedural *kill*, *mod*, and *use* problems. For example, in the program summary graph, the kill of each formal parameter in a procedure is represented with a boolean variable that indicates whether the variable is redefined along all paths by a call to the procedure. An iterative technique uses the paths through the program summary graph to compute the kill for the formal parameters in each procedure. The structure of the program summary graph does not allow for the preservation of the calling context of called procedures.

A technique introduced by [65], uses the super graph or in-line substitution to compute the interprocedural du-chains. The super graph structure is prohibitive for large programs, and cannot provide efficient data-flow analysis in the presence of recursive procedures. In addition, separate compilation cannot be supported since either the code or the control flow graph of each procedure must be available during the analysis.

Another technique was introduced by Sofa and Rapp [44] This technique computes definition-use and use-definition chains that extend across procedure boundaries at call and return sites. Intraprocedural data-flow analysis for each procedure is performed separately and does not require information from other procedures, thus supporting separate compilation. When a procedure is processed, information about definition and use information is abstracted and used to construct its interprocedural control flow graph (ICFG). Once all the procedures under test are processed, their intraprocedural data-flow information is then propagated throughout the program via the *ICFG* to obtain sets of reaching definitions and/or reachable uses for each interprocedural control point. The control points include procedure entry, exit, call, and return. The propagation algorithm proposed in this approach uses a two phase propagation

1. Interprocedural data-flow information is flow-sensitive if the control flow of called procedures is used in the computation of definitions and uses in the program

strategy to preserve the calling context of called procedures. The algorithm also handles recursive procedures and alias pairs.

2.8 Visual Programming

Visual Programming (VP) is a process by which meaningful graphical representations are used to create programs. An important factor in determining whether a given programming system can be considered as being a *VP* system, lies in the conceptual correctness of the first phrase. *VP* can be categorized into: Visual Programming Environments (VPEs) and Visual Programming Languages (VPLs). Each category will be briefly discussed next.

- **The Visual Programming Environment:** A *VPE* usually consists of a set of tools that interact to accomplish parts of a programming task. A programming language is considered to be a visual environment if its tools are graphical and allow the programmer to use graphical techniques for manipulating visual artifacts, and displaying the structure of the program. VisualAge® is an example of a visual environment. It provides graphical tools such as class browsers, wizards, and syntax coloring techniques to assist or aid the programmer. One difficulty with large programs in a *VPE* can be attributed to the overwhelming number of "visual objects/pictures". Such overwhelming number of objects makes it difficult for a user to visualize the system as a whole. Incorporating some filtering mechanism can aid the programmer in managing the amount of visualized data.

- **Visual Programming Languages:** *VPLs* allow programmers to develop programs using visual constructs such as charts, diagrams, icons, or tables. *VPLs* are not entirely "text-free", text can be part of the visual syntax providing services such as labels for the graphical elements and comments. *VPLs* are used to improve the programmers ability to express program logic and to understand how a program works. The graphical nature of a *VPL* allows for a cleaner, more explicit display of program relationships than is possible with textual languages. This is achieved through the closeness of mapping provided between the problem domain and the solution. In general, *VPLs* share the following characteristics: conceptual simplicity; concrete

programming process; explicit relational depiction; and visual feedback [63]. Each of these four characteristics will be briefly explained next.

- **Conceptual Simplicity**: *VPLs* attempt to reduce the complexity of programming by reducing the number of concepts needed to create and comprehend a program. Some *VPLs* eliminate the concept of explicit variable declarations and operator types. Others shield the user from more difficult concepts such as memory management. *VPLs* that target novice end-users tend to be simple; however, they still provide enough computing flexibility to perform any task in the domain specific area. Conceptual simplicity cannot be considered weak. General purpose *VPLs* balance conceptual simplicity while ensuring enough programming flexibility to express any computation that can be accomplished in a textual language. Prograph is a general purpose dataflow *VPL* that is more conceptually simple than most textual languages, yet it can still perform any computation. Domain specific *VPLs* on the other hand, can be generally conceptually simple since they focus on the application's domain. LabView [10] is a domain specific *VPL* that is used for scientific instrumentations.

- **Concrete Programming Process**: Concreteness is a shared characteristic of a *VPL* and its encompassing *VPE* [63]. This characteristic is manifested in the programmer's ability to use a specific value for a computation rather then a description of all possible values. For example, the number 0 is a concrete value that can be used for a computational task. On the other hand, declaring a variable of type integer which may hold the value "0" is not concrete. Concreteness can also be used to provide feedback or to specify a program. For example, in a spreadsheet, when entering a formula for a cell instead of specifying a cell to be used in computation by its row and column index, a user can point to the desired cell and the system will fill in the appropriate values.

- **Explicit Relational Depiction**: The relationship between program constructs in a *VPL* is visually explicit. For example, dataflow diagrams show the relationships between operations while flowcharts show the control flow of the program.

- **Feedback**: *VPEs* and *VPLs* provide two types of feedbacks: visual feedback, and immediate visual feedback [25]. A visual feedback is a process by which the environment visually alerts the user about the presence of "bugs" as a result of running a program. Immediate visual feedback is a process by which a user is directly informed about a certain action. For example, the auto recalculation feature of spreadsheet languages such as FormsIII, is an immediate knowledge of the effect of a program edit. Immediate feedback does not hinder the programmer from creating the program, nor does it interrupt his/her thought process; rather, it acts as a helpful guide. Visual immediacy is important for program creation and debugging. There are three kinds of visual immediacy that the *VPL* and *VPE* can provide temporal, spatial, and semantic [84]. Temporal immediacy, as previously described, is the level of feedback or liveliness of a system. Spatial immediacy reduces the spatial gap between the locations of the error and that of the object used to indicate the presence of the error. There are two kinds of spatial gaps, low spatial gap, and high spatial gap. An example of a high spatial gap is the output from a compiler that informs the user of the line number of an incorrect statement. An example of a low spatial gap is the highlighting of an incorrectly spelled word by a spell checker. Semantic immediacy is the distance between semantically related bits and sources of information. For example, in Prograph the distance between a method call and the actual method body is narrow because the user can double-click on the method icon – where that method is used – to open its body for viewing or editing.

2.9 Prograph; A Brief Overview

Prograph is a visual dataflow, object oriented language developed by P.T. Cox and T. Pietrzykowski at the Technical University of Nova Scotia (TUNS) in 1984. We have chosen Prograph in this Thesis as a representative of visual dataflow languages because it is one of the few commercial visual dataflow languages that is available to us, and a number of commercial software packages have been created using it. Prograph exists on the Macintosh and the Windows platforms. It has a number of features that makes it a desirable visual dataflow language. These features are: visual syntax; dataflow computational model; editing environment; and object oriented capability. Each of these features will be explained next.

2.9.1 The Dataflow Computational Model

The dataflow is one of the most popular computational models for *VPLs* [49]. The most prominent feature that characterizes the power of a dataflow *VPL* and determines its acceptance, is the availability of the rich library of predefined methods that are used as the elementary building blocks. Prograph uses the dataflow paradigm for programming. It is represented as a directed graph where nodes represent user-defined or system-defined methods, and the data flows through edges or "links" between nodes. Links going into a node represent the method's data input, links going out of a node represent the method's output or result. A node is executed when all of its input data is available. This model of execution is called data-driven. In practice, a linear execution order for the operations in a method is predetermined by topologically sorting the directed acyclic graph of operations, subject to certain constraints. For example, an operation with a control should be executed as early as possible. In a pure data flow model where no control constructs are added, the sequence in which operations are executed is not explicitly specified. That is, the ordering of the operations is not specified by the programmer, but is implied by the data interdependencies. Many dataflow languages such as Prograph, have modified the pure dataflow model to allow control constructs such as iteration and sequential execution. Prograph easily allows the programmer to control the order of execution via a special mechanism called the *synchro.*

2.9.2 Visual Syntax

Prograph has an entirely graphical syntax based on the dataflow nature of the its computational model. The syntax is simple, but still allows for the expression of powerful language constructs without sacrificing efficiency. Prograph's syntax is almost completely iconic with text being used only for naming language elements and for commenting the semantic intent of the code. Prograph's syntax consists entirely of icons and associated visual annotations representing various kinds of operations, methods, data links, and groups of code. The icons are the language, they do not represent textual information that lies "hidden" behind or in them; rather, they are the executable code. This allows for intuitive debugging since the programmer debugs the actual source code and not a transformation of the source code.

2.9.3 Editing Environment

The environment for Prograph is an integrated program development system that includes: program editor, code interpreter, debugger, and Graphical User Interface (GUI) builder. The visual syntax of Prograph provides for a seamless integration of the Integrated Development Environment (IDE) and the language. In general, The visual syntax of Prograph shields the user from syntactical errors; however, semantic errors can still be present and will only be caught at run time if the code containing the "error" is executed. Prograph's interpreter facilitates progressive program evaluation since it allows bits and pieces of program code to be executed independently, or in isolation from other code. An interesting feature of Prograph is that, while a program is running, major changes such as adding or removing language constructs or even classes can be accomplished and the changes take effect immediately.

3 A Control-flow Testing Methodology for Prograph

3.1 Introduction

As with any programming language, visual or textual, a dataflow program may contain faults. To ensure the correct functioning of dataflow programs, and increase confidence in the quality of these programs, testing is required. In this Chapter, we investigate, from a testing perspective, differences between dataflow and imperative languages. We show that significant similarities exist between dataflow and imperative programs. For example, in imperative languages, tests are often run in three steps: first, inputs are initialized; next, the program is executed; and finally, results are validated [16]. A similar process is applicable to programs written using dataflow languages. Further similarities between the control dependencies in both imperative and dataflow languages reveal opportunities for adapting code-based control-flow testing criteria to test dataflow languages.

The remainder of this Chapter is organized as follows: in Section 3.2 we give a brief overview of Prograph and discuss its subset that we are considering for this work. In Section 3.3 we review related work involving the use of visual annotation to communicate the testedness of visual form-based languages. We show that the testing methodology developed for form-based languages is not suitable for dataflow languages. In Section 3.5, we propose an Operation Control Graph (*OCG*) model for testing dataflow languages. We show how both the data and control dependencies of dataflow languages can be represented in the *OCG*, and how the latter is suitable for applying control flow testing criteria to dataflow languages. We also show that our testing methodology is well suited for dataflow programs. In particular, the All-branches criterion provides important error detection ability, and can be applied to any dataflow program. We have implemented a testing system that allows users to visually and empirically investigate

the testedness of predicate edges and nodes in Prograph. Our testing tool can communicate testing results in the visual environment of dataflow languages in a way that requires no extensive formal knowledge of testing on the part of the user. This testing approach for dataflow languages is based on the empirical studies of WYSIWYT [16] which have shown that programmers can indeed use this approach to test software without extensive formal knowledge of testing. In this work we assume that the user of the *DFTT* serves as the *oracle* that validates the correctness of test case outputs. Our empirical results in Section 3.6 show that, analogous to imperative languages, the All-branches criterion cannot detect all the errors in a dataflow program. Thus, to help catch those undetected errors, a more rigorous testing should be applied. This is indeed the focus of Chapter Four. Finally, in Section 3.7, we conclude with a summary of our findings and brief discussion on directions for further work

3.2 Dataflow Languages in The context of Prograph

In visual dataflow languages, users code by creating icons and linking them together. The icons in dataflow languages are the source code and not some visual representation of a textual code that lies beneath the icons. The order of execution, when not explicitly defined by the user, is determined by the editing environment of the language implementation.

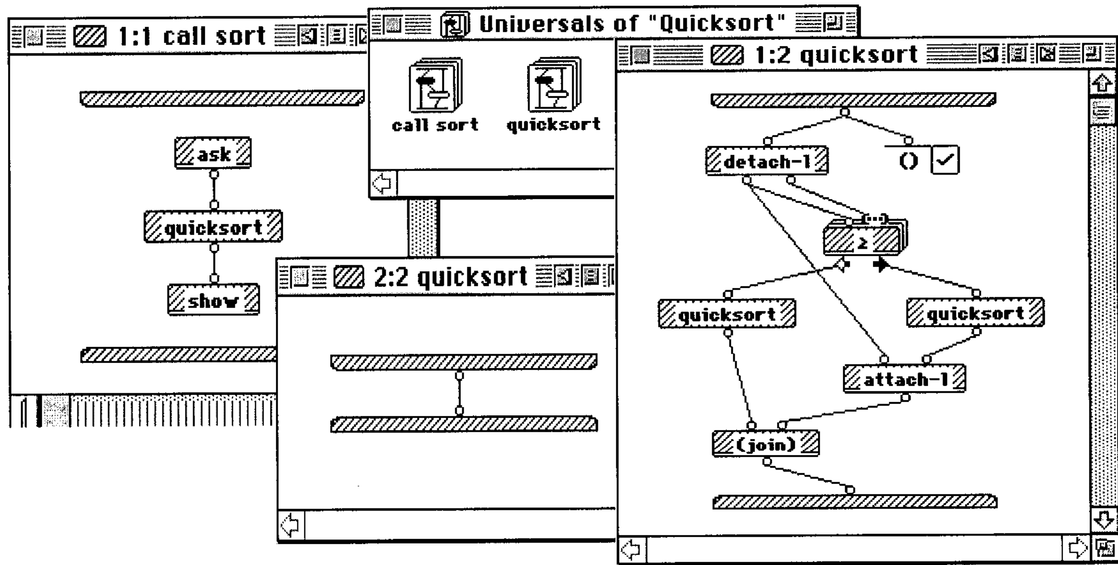

Figure 3-1. A Prograph program for Quicksort.

Prograph is an example of a dataflow visual programming language. Since much of our work is based on the visual programming environment of Prograph, in this section we will informally introduce its syntax and semantics using an example. Figure 3-1 shows a Prograph implementation of the well known algorithm "quicksort" for sorting a list into ascending order. The top centre window in this diagram depicts two icons for the methods **call sort** and **quicksort**. Note that Prograph is an object-oriented language, hence the term "method" is used to refer to entities known as procedures in standard programming languages.

The left side in Figure 3-1 shows the details of the method **call sort**, a dataflow diagram in which three operations are connected sequentially. The first operation in this diagram, **ask**, is a primitive that calls system-supplied code to produce a dialogue requesting input from the user. Once this has been executed, the data input by the user flows down the datalink to the operation **quicksort**, invoking the method **quicksort**. This method expects to receive a list, which it sorts as explained below, outputting the sorted list, which flows down the datalink to the **show** operation. The **show** produces a dialogue displaying the sorted list. The small circle icons on the top and bottom of an operation, representing inputs and outputs are called *terminals* and *roots* respectively.

The method **quicksort** consists of two *cases*, each represented by a dataflow diagram as shown in the lower right two windows of Figure 1. The first *case*, shown in the window entitled **1:2 quicksort**, implements the recursive *case* of the algorithm, while the second implements the base *case*. In general, a method consists of a sequence of *cases*. In the first *case* of **quicksort**, the first operation to be executed is the match operation, () [✓], which tests to see if the incoming data is the empty list. The icon attached to the right end of the match is a *next case on success* control, which is triggered by success of the match, immediately terminating the execution of the first *case* and initiating execution of the second. If this occurs, the empty list is simply passed through as the output of the second *case*, and execution of **quicksort** finishes, producing the empty list. The thin operation at the top of a *case* where parameters are copied into the *case* is called an *input bar*, while the one at the bottom where results are passed out is called an *output bar*.

If the input list is not empty, the control on the match operation in the first *case* is not triggered, and the first *case* is executed. Here, the primitive operation **detach-l** outputs the first element of the list and the remainder of the list on its left and right *roots* respectively. Next, the operation, , is executed. This operation is an example of a *multiplex* illustrating several features of the language. First, the three-dimensional nature of the operation indicates that the primitive operation ≥ will be applied repeatedly. Second, the *terminal* annotated as (..) is a *list terminal*, indicating that a list is expected as data, one element of which will be consumed by each execution of the operation. In this example, when the multiplex is executed, the first element of the list input to the *case* will be compared with each of the remaining elements. Finally the special *roots* ◇ and ➧ indicate that this particular multiplex is a *partition*, which divides the list of items arriving on the list annotated *terminal* into two lists, items for which the comparison is successful and those for which it is not. These two lists appear on the ◇ and ➧ *roots* respectively.

The lists produced by the partition multiplex are sorted by recursive calls to the **quicksort** method. The final sorted list is then assembled using the two primitive operations **attach-l**, which attaches an element to the left end of a list, and **(join)**, which concatenates two lists.

Clearly, the execution mechanism of Prograph is data-driven dataflow. That is, an operation executes when all its input data is available. In practice, a linear execution order for the operations in a *case* is predetermined by topologically sorting the directed acyclic graph of operations and subject to certain constraints. For example, an operation with a control should be executed as early as possible.

In our example the method **quicksort** has only one input and one output, and therefore does not illustrate the relationship between the *terminals* of an operation and the *roots* of the input bar in a *case* of the method it invokes. These *terminals* and *roots* must be of equal number, and are matched from left to right. A similar relationship exists between the *roots* of an operation and the *terminals* of the output bar in a *case* of a method invoked by the operation.

One important kind of operation not illustrated in the above example is the *local operation.* A local operation is one that does not call a separately defined method such as the quicksort method shown above. Instead it contains its own sequence of *cases*, called a local method. It is therefore analogous to a parametrized *begin-end* block in a standard procedural language.

The formal semantics of Prograph are defined by specifying an *execution function* for each operation in a program. Each execution function maps a list *X* to a pair (*Y*, *c*) where c is a control flag, *Y* is a list, and the lengths of the lists *X* and *Y* are respectively equal to the number of *terminals* and the number of *roots* of the operation, and the elements *X* and *Y* are from a domain Δ containing all values of simple types, and instances of classes. Execution functions may produce the special value **error**; for example, if a list *terminal* receives a value which is not a list. By defining execution functions for operations, the input/output behavior of a program is specified.

3.2.1 The Order of Execution in Prograph

The order of execution in Prograph is determined by the editor during the "visual coding" phase. The language's constructs or operations are topologically sorted in a way that preserves two kinds of dependencies among those different constructs: data dependency; and control dependency.

3.2.1.1 The Data Dependency

The data dependency is represented via data links from an operation's *root*(s) to another operation's *terminal*(s), and an operation executes only when all of its data are available on its *terminal*(s). For example, let *A* and *B* be two non-annotated operations, if *A* is data-dependent on *B,* in which case there exists a data link from *B's root*(s) to *A's terminal*(s), *A* executes after *B.* If *B* is data-dependent on *A*, in which case there exists a data link from *A's root*(s) to *B's terminal*(s), *B* executes after *A*. If *A* and *B* are not data-dependent; however, the data becomes first available on either *A* or *B*, the order of execution can be either *A*->*B* or *B*->*A*, where "->" denotes the order of execution from left to right.

3.2.1.2 The Control Dependencies

The control dependencies can be divided into two categories: *multiplex* annotations and control annotations. *Multiplex* operations are analogous to loop constructs in imperative languages. The Prograph editor permits the following multiplex annotations: *repeat*; *list*; *and loop.* The control annotations are similar to the predicate statements or controls found in imperative languages. Both types of controls will be briefly introduced next.

3.2.1.3 The Control Annotations

Every operation except the *input bar* can have a control annotation. The flow of execution is affected by the control annotations to operations. Prograph contains the following control annotations: *Continue*; *Next Case*; *Finish; Terminate*; and *Fail.* Each of these control annotations can be applied to an operation on either *success* or *failure.* Operations in Prograph are divided into two sets: Δ and Δ', such that Δ contains the set of operations that can fail and Δ' contain the set of operations that cannot fail. Let O be an operation $\in \Delta$, O' an operation $\in \Delta'$, we say that there are two signals associated with O (*fail* and *success*), and one signal (*success*) associated with O'. Let M be a user-defined method or *local*, and $\underline{M}$ a *multiplexed* user-defined method or *local* in a Prograph program, then the following flow of execution are possible:

- *Continue*: this is the default or implicit control for every operation in Prograph. When an operation with implicit *Continue* on *success* is encountered, the flow of execution simply continues to the next operation. When an operation is annotated with a *Continue* on *failure* ☒, the flow of execution also continues to the next operation.

- *Next Case*: this control decides whether execution continues in the present *case* or jumps to the next *case.* When O in M *or* $\underline{M}$ is annotated with a *Next Case* on either *success* ☑ or *failure* ☒, and the condition associated with the control succeeds, the flow of execution abandons the current *case* and jumps to the next *case*, otherwise the flow of execution is not interrupted and it simply moves to the next operation in the present *case.* For example, consider the *universal* U in Figure 3-2. When the value received from *ask* is **not** equal to

"3", the execution breaks out of "*1:2 U*" and start executing in "*2:2 U*". The *Next Case* control is analogous to the *then* part of an *if then* statement in imperative languages. When *O'* in *M or* <u>*M*</u> is annotated with a *Next Case* on *failure*, the flow of execution simply moves to the next operation in the current *case*; however, if *O'* is annotated with a *Next Case* on *success*, the flow of execution unconditionally abandons the current *case* and jumps to the next *case*. For example, consider the *universal U* in Figure 3-3. When the flow of execution reaches the *show* primitive, it is executed, and the execution breaks out of "*1:2 U*" and starts executing in "*2:2 U*". When applied to an operation *O'*, the *Next Case* on *success* behaves like a "*goto*" statement in imperative languages.

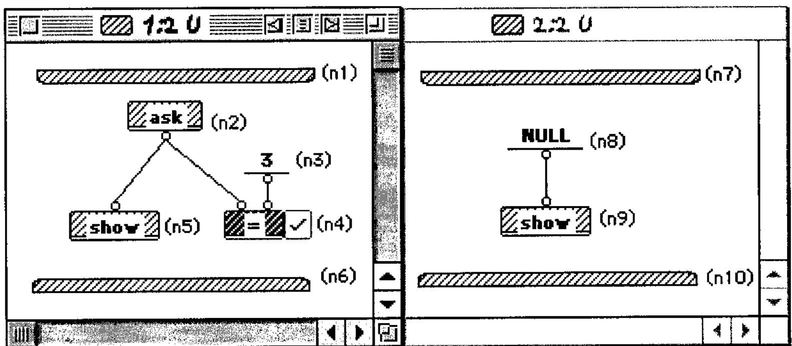

Figure 3-2. An example illustrating a *Next Case* on *success*.

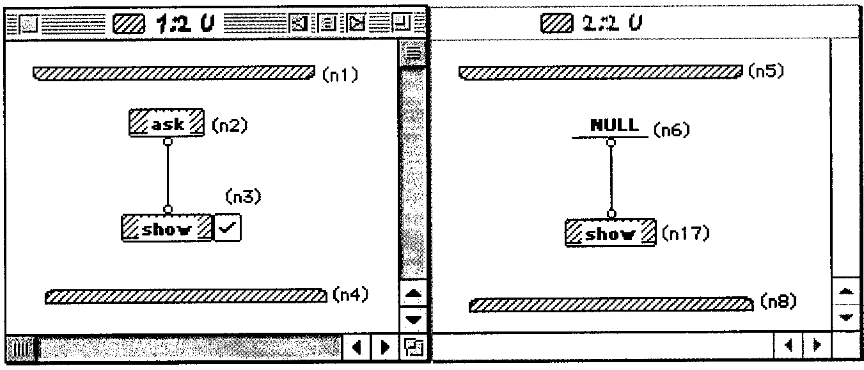

Figure 3-3. A *Next Case* applied on *success* to an operation that cannot fail.

- *Finish*: this control is only generally useful when applied to operations in ***multiplexed locals*** or user-defined methods. When *O* in *M* is annotated with a *Finish* on either *success* [☑] or *failure* [☒], and the condition associated with the control either succeeds or fails, the flow

of execution simply continues to the next operation. For example, consider the example in Figure 3-4. Regardless of the value that is received at the *match* operation, the flow of execution will continue to the next operation. This is also the case, had the match operation been annotated with a *Finish* on *success.* When *O* in $\underline{M}$ is annotated with a *Finish* on either *success* or *failure,* and the condition associated with the control succeeds, the flow of execution continues normally; however once the end of the *case* is reached, the execution breaks out of the *multiplexed* operation. For example, consider the example in Figure 3-5. When the value is received by the operation that is labeled *n8* is > "3", the flow of execution continues; however, once the Output Bar of "*1:1 local1*" (the operation labeled *n10*) is executed the loop breaks. When *O'* in either *M* or $\underline{M}$ is annotated with a *Finish* on *failure,* the flow of execution does not break (infinite loop) out of either *M* or $\underline{M}$ since *O'* cannot return a *fail* signal. When *O'* in either *M* or $\underline{M}$ is annotated with a *Finish* on *success,* and the condition associated with the control succeeds, the flow of execution continues normally; however once the end of the *case* is reached, the execution breaks out.

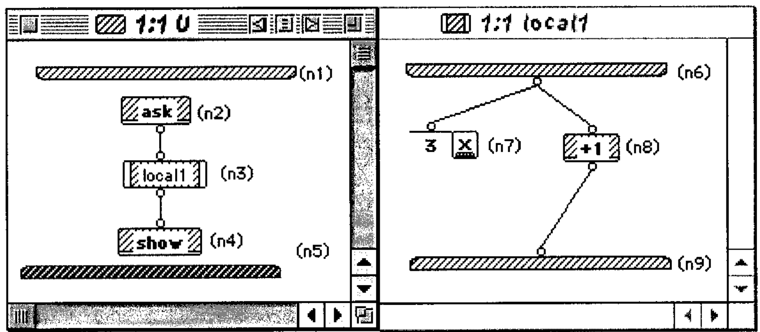

Figure 3-4. An operation that is control annotated with *finish* of *failure* in a *local.*

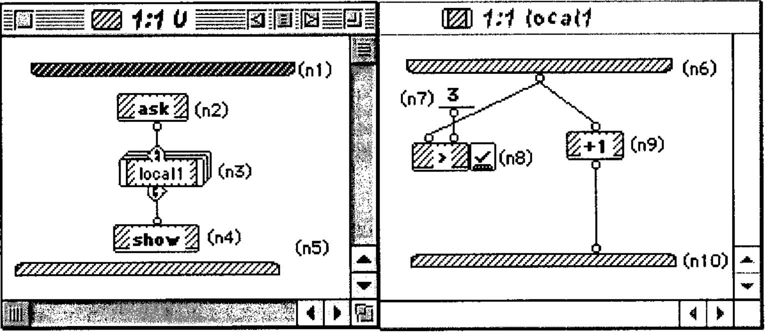

Figure 3-5. An operation that is control annotated with *finish* of *failure* in a looped *case.*

- *Terminate*: this control is only generally useful when applied to operations in both *multiplexed* and non-multiplexed *locals* or user-defined methods. When *O* in either *M* or $\underline{M}$ is annotated with *Terminate* on either *success* [☑] or *failure* [☒], and the condition associated with the control succeeds, the flow of execution simply breaks out of the current *case*. For example, consider the example in Figure 3-6. When the value that is received by operation that is labeled *n8* is > "3", the flow of execution breaks out of the looped *case*. When *O'* in either *M* or $\underline{M}$ is annotated with a *Terminate* on *failure*, the flow of execution, does not break out (infinite loop) of either *M* or $\underline{M}$ since *O'* cannot return a *fail* signal. When *O'* in either *M* or $\underline{M}$ is annotated with a *Terminate* on *success*, and the condition associated with its control succeeds, the flow of execution breaks out.

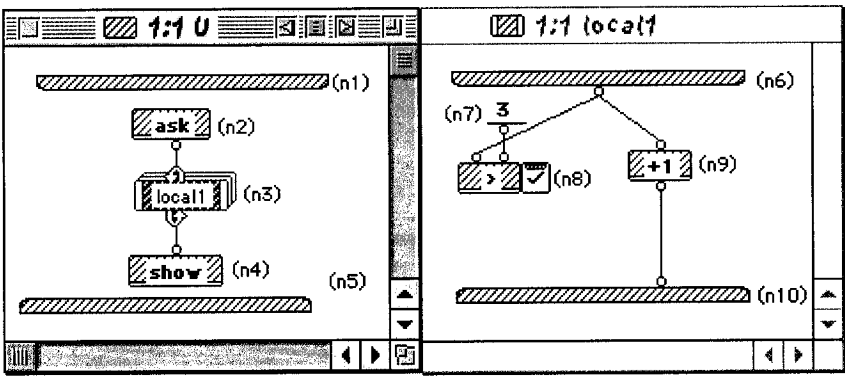

Figure 3-6. An operation annotated with *terminate* on *failure* in a looped *case*.

- *Fail*: the fail annotation plays the role of the *throw* construct of Java or C in exception handling. When *O* in *M* or $\underline{M}$ is annotated with a *Fail* on either *success* ⊘ or *failure* ⊗, and the condition associated with its control succeeds, the flow of execution simply breaks out of the current *case* and the *failure* is propagated to either *M* or $\underline{M}$, provided that *M* or $\underline{M}$ can catch that error by applying the appropriate control. If that appropriate control is not applied to either *M* or $\underline{M}$, then an error is generated during run-rime. When *O'* in either *M* or $\underline{M}$ is annotated with a *Fail* on *failure*, the flow of execution is not interrupted. When *O'* in either *M* or $\underline{M}$ is annotated with a *Fail* on *success*, it behaves like *O*.

3.2.2 Restricted Prograph

Prograph is a commercial visual dataflow object-oriented language. Prograph allows its users to develop methods that can be either part of a class or *universal* methods. The *universal* methods are analogous to global procedures in imperative languages. Features such as persistents or global variables are also available in Prograph.

Operations in Prograph can execute as soon as all of the data they need is available. The result of this is that operations in a method have no guaranteed order of execution [80]. Various features of Prograph allow for operations to have side-effects. With no fixed execution order, in the presence of side-effects, it would be necessary to test every possible order to ensure that a program functions correctly. Since this is not practical, for the purpose of our current work, we have chosen a subset of Prograph with no side effects. In particular, the object-oriented features and global variables are not considered. In future works we intend to address these issues.

Furthermore, there are a variety of features of Prograph which can be considered "syntactic sugar" (i.e. they can be reproduced from more basic elements, albeit with sometimes significantly more complicated code). Although they are extremely useful to Prograph programmers, there would be no value in including them in our discussions, and, in fact, they would significantly complicate our presentations. Thus, we further simplify the structure of the language by not considering the following operations: the partition multiplex; the continue control annotations; the fail control annotation; and the list annotation on *roots* or *terminals*.

Therefore, the subset of Prograph considered here contains the following operations: *input bar*; *output bar*; *local* and *universal methods*; *constants*; *match*; and *primitives* or system defined methods. These operations, may have the following control annotations: *next case*, *finish*, *terminate*, or *repeat* annotation. *Roots* and *terminals* may have *loop* annotations. An operation which is control annotated is referred to as a *predicate* operation otherwise it is a *non-predicate* operation.

3.3 Testing Form-based Languages: Related Work

Form-based visual programming languages provide a declarative approach to programming, characterized by a dependency-driven, direct-manipulation working model [17]. Users of form-based languages create cells, and define formulas for those cells. These formulas may or may not reference values contained in other cells. When a cell *X* references a cell *Y* in its formula, a dependency is created between *X* and *Y*, and we say that *X* is dependent on *Y*. If a newly created or edited cell references other cells, its value is calculated only after the values of the cells it references are recalculated. This is much the same as spreadsheet calculations work, although form based languages such as FormsIII are more general than spreadsheet applications [16]. Along with data dependencies such as these, another kind of dependency in form-based languages is the control dependencies that exist within each cell's formula. The use of predicate expressions within cell formulas create control dependencies that are comparable to the flow of control in imperative languages. Thus, from a testing perspective, a form-based language has two significant properties: the flow of control within each formula; and the data dependencies between cells.

To test form-based visual languages, [17] proposed to model these properties in Cell Relation Graph (CRG). In a *CRG*, each formula is represented by a formula graph. The formula graph is comparable to a control flow graph that represents a procedure in an imperative program. Cell dependencies between formula graphs are represented by edges between these graphs. For example, if cell *X* depends on cell *Y*, an edge would be added from *X*'s formula graph to *Y*'s formula graph. The cell dependency edges do not represent the flow of control between formula graphs, but rather the order in which *X* or *Y* or both must be executed for a given input, or whether the execution of *X* will immediately follow the execution of *Y*. Edges within the formula graphs, on the other hand, do represent the flow of control within the cell's formulas. The abstract model proposed by [17] made it possible to apply a variety of testing criteria. The work in [16] was extended in [16]. The latter proposed the use of visual elements to communicate to the user the testedness of cell formulas under a particular test adequacy criterion.

Our work parallels this, dealing with the specific problems related to testing and presenting test results in visual dataflow programs. In our work we adopt the approach proposed in [17], that is, we make use of the visual constructs of dataflow languages to communicate the testedness of a procedure or universal method under a certain testing criterion.

Although dataflow languages are similarly characterized by both data and control dependencies, several features of dataflow languages make the abstract model developed in [17] not fully applicable for testing dataflow languages. First, not all visual dataflow languages are responsive. This means that program execution is not accomplished automatically, and the user must explicitly run tests after making program modifications. Furthermore, the data dependencies in visual dataflow languages like Prograph are local to the procedure in which they exist. That, is, when P is modified, the new data dependencies (if any) that exist between P's visual constructs, including those involved in a call to another universal method Q, are re-evaluated by the editing environment only for P, and not for Q.

3.4 Testing Visual Dataflow Languages

In imperative languages, the strict ordering of execution of non-predicate statements allow these statements to be grouped into sets of disjoint blocks which can be treated as a single unit when constructing the control flow graph of the program.

The situation is different in visual dataflow languages, however. The order of execution of non-predicate operations or statements is not predetermined by the programmer, but is simply based on data dependencies. If we were to construct a control flow graph of a program, taking into consideration all possible execution orders, it would be extremely large and complex. It would also most likely be impossible to satisfy any criteria using this graph, since most dataflow language implementations choose a specific execution ordering for non-predicate operations and use that for every execution.

This is, in fact, the execution behavior of Prograph programs. The execution order is maintained by a topological sort that is performed on all operations during the phases of program-

ming and modification of the visual code. At this stage in our research, we are considering a subset of Prograph in which operations have no side-effects. As a result, any execution order of a sequence of non-predicate operations will yield the same result. Thus, we need only consider one order — the one determined by the editing environment of the language.

In trying to represent both the data and control dependencies of dataflow languages in a suitable graph, we find that the notion of a disjoint blocks in dataflow based languages is different from that of imperative languages. There, a disjoint block represents a group of non-predicate statements at the end of which a predicate statement exists. In dataflow languages each disjoint block is represented by a single operation. Thus, we represent every non-predicate operation with a node in the flow graph, linked together in the execution order chosen as described above. Predicate or control annotated operations in a dataflow language are analogous to their imperative counterparts, and they are represented with a node that has *true* and *false* edges corresponding to the flow of execution based on the outcome of the predicate. Adapting the control flow graph to accommodate the data flow computation model permits us to apply all-nodes and all-edges testing criteria to a dataflow program in a manner that is effective and efficient. Next we present our abstract model for testing dataflow languages.

3.4.1 An Abstract Model for Prograph

As was discussed in previous sections, test adequacy criteria for different paradigms are generally defined on abstract models of programs, rather than directly on the code itself. Here we use this approach by adapting the flow graph designed for imperative languages to accommodate visual dataflow languages.

Each procedure in Prograph consists of one or more *cases*. So given a Prograph procedure p in a program P, we can construct an Operation Case Graph (*OCG*) for each *case* in p. Each *OCG* has a unique entry and exit nodes n_e, and n_x respectively. The n_e node of a procedure p is linked to the n_e node of the first case of the method. The n_x node of each successive case in p is linked to the n_x node of p. Each *OCG* will be assigned the name and label of its corresponding *case* in

Prograph. For example, if a method *X* has two *cases* (1:2 *X* and 2:2 *X* denoting *X* "*case one* of *two*" and *X* "*case two* of *two*", respectively), the *OCG* graphs will be labeled *1:2 X*, and *2:2 X*.

Locals are also comprised of one or more cases. So when a *local* operation is encountered, the appropriate *OCG* will be constructed, and an edge is constructed to connect the node representing the predecessor of the local operation with the entry node n_e of the *OCG* representing the *local*.

3.4.2 Building Data and Control Dependencies in *OCGs*

Given a program *P*, we represent each procedure $p \in P$ with an *OCG*. The *OCG* represents two properties of *p*'s operations: data and control dependencies. For each procedure, there is a sequence of operations, $O = <o_1, o_2, o_3, \ldots, o_m>$, whose order corresponds to a valid execution ordering with respect to the data dependencies in the procedure (in particular, this will be the execution order chosen by the editing environment). Thus, to model the data dependencies in the *OCG*, and preserve the execution order, we represent each non-control operation $o_i \in O$, where $1 < i < m$, with a node n_i in *OCG*, linked together with an edge to n_{i+1} the node representing the next operation o_{i+1} in *O*.

To represent the control dependencies between operations in *O*, we represent each control-annotated operation $o_i \in O$ with a node n_i and two edges representing the (*true* and *false*) execution possibilities. The node of the control-annotated operations is analogous to those of the nodes used to represent predicate statements in imperative languages.

As was mentioned earlier, the subset of Prograph we are considering has four different control annotations: Next Case; Terminate; Finish; or Repeat, and one *root/terminal* annotated loop. Next we show how we represent these annotations in the *OCG*.

An operation o_i with a Next Case annotation will be represented by a node n_i with two edges coming from it (*true* and *false*), one connected to n_{i+1} which is the node representing the next sequential operation o_{i+1} in the current case, and the other to n_e of the next *case*.

An operation o_i with a Terminate annotation will be represented by a node n_i with two edges coming from it (*true* and *false*), one connected to n_{i+1} which is the node representing the next sequential operation o_{i+1} in the current case, and the other to n_x of the current *case*.

An operation o_i with a Finish annotation is unique in the sense that when it is activated in non-repeated or looped *case*, the flow of control does not change upon the outcome of the Finish control. This means that if the outcome is either true or false, the next node that gets traversed or executed is the same. The same situation occurs when the Finish annotated operation happens to be in a repeated or looped *case*; however, here the true or false outcome may set a flag that will indicate whether successive iterations will take place after finishing the current iteration. Thus, to handle a Finish annotated operation, be it in a repeated or non-repeated *case*, and represent both its true and false outcome, we represent o_i with a node n_i with two edges coming from it (*true* and *false*). One edge connected to n_{i+1}, the node representing the next sequential operation o_{i+1} in the current *case*, and the other edge to a dummy node *d*. From the dummy node *d* we also construct an edge to n_{i+1}. The reason for constructing this edge is to allow the flow of control to go to n_{i+1} when the outcome of the Finish control goes through the dummy node *d*. When *d* exists in a looped *local* and is traversed, it sets a flag value that indicates whether the loop edge can be traversed.

An operation o_i with a repeat or a loop annotation will be represented according to the type of the operation. For example, if o_i is a *local* operation, we construct an edge from the node representing the Output Bar in the *local's OCG* to the node representing the Input Bar in the *local's OCG*. On the other hand, if o_i is not a *local* operation we represent it with a node n_i, construct an edge that goes out of n_i and back into n_i. A Prograph program and its *OCG* are shown in Figure 3-7.

3.4.3 Implementation

The iconic nature of the visual dataflow paradigm makes it impossible to use conventional text scanners to construct the *OCG* that represents the visual code. To accommodate this funda-

mental difference between visual and textual languages, we have modified a Prograph-to-Java translator in a way that allowed us to provide the following functionalities:

- The ability to collect the static analysis of both nodes and edges by probing each operation of the set *O*.
- The ability to track the dynamic analysis of both nodes and edges.
- The ability to visually communicate the testedeness of operations, which in turn helps locate the faults in the visual code.

Testing and communicating results is an integrated process within the Prograph environment. We have developed a dataflow testing tool or *DFTT* that is integrated in Prograph. This tool allows the user to choose a testing criterion such as All-branches, and visually observe the testedness of each control annotated operation. For example, if a control annotated operation is only tested on its either true or false outcome, then our testing environment tracks that annotated operation in the Prograph code and colors half of it green and the other red, signalling that only one branch was tested. We next discuss our testing methodology which provides these functionalities described above.

Task1: Collecting static nodes and edges.

Several algorithms for collecting static branches and nodes have been developed for imperative languages [17]. All these algorithms rely on the textual nature of the program's source code to extract its static analysis. These algorithms are not applicable to dataflow languages. Collecting the static analysis in dataflow languages requires access to the internal data structure that host these operations. For each node $n_1 \in OCG$ representing operation $o_1 \in O$, we maintain two attributes: *Traversed*; and *@p*. The *Traversed* attribute is assigned to FALSE, and it indicates that this particular node that represents a Prograph operation has not been exercised. The *@p* is a pointer that points to o_1, and is used by the *DFTT* to locate operations in the Prograph code during the visual presentation of operation testedness. For each node $n_2 \in OCG$ representing a control annotated operation $o_2 \in O$, we maintain two attributes: *Tb* or True branch;

Fb or False branch. The *Tb* and *Fb* are boolean variables, and are initially assigned to FALSE. They indicate that the true and false branches of the predicate or annotated operation in Prograph have not been exercised.

Task 2: Tracking the Execution

To track the dynamic execution, we have simply modified the Prograph-to-Java translator in a way that allows us to probe each operation before it is translated. Once the textual code has been compiled, test cases are run on the Java code rather than on the Prograph visual code. The *DFTT* records every executed operation and maintains its appropriate attributes. For example, if a non-control annotated operation $o \in O$ has been executed, the testing tool changes its *Traversed* attribute value to TRUE. Likewise, as a control-annotated operation is executed on both its true and false outcome, the testing tool changes the operation attributes values of *Tb* and *Fb* to TRUE.

Task3: Communicating testing results

The data collected in Tasks 1 and 2 provide test adequacy information to the user in a way that requires no extensive formal understanding of testing notions. For example, if the user chooses the All-nodes criterion, the *DFTT*, after each test case, incrementally determines the executed nodes for each operation, and calculates the tested node percentage. Each executed operation will be colored green to indicate that it is tested and red otherwise. Moreover, the tested percentage result is inserted as a comment under each universal method. For control annotated operations, they are colored green if both edges are executed, half green and half red if one edge is tested, and red otherwise. This visual feedback helps users identify quickly what has and has not been tested. For those operations that remain untested, the visual feedback is twofold: first, it aids the user in developing the appropriate test cases, and second, it helps in locating any remaining faults in the visual code.

3.5 An Example

We will illustrate what we have described thus far with a simple example that uses an iterative approach to calculate the factorial of a number. The program that is shown in the right side of Figure 3-7 contains one error.

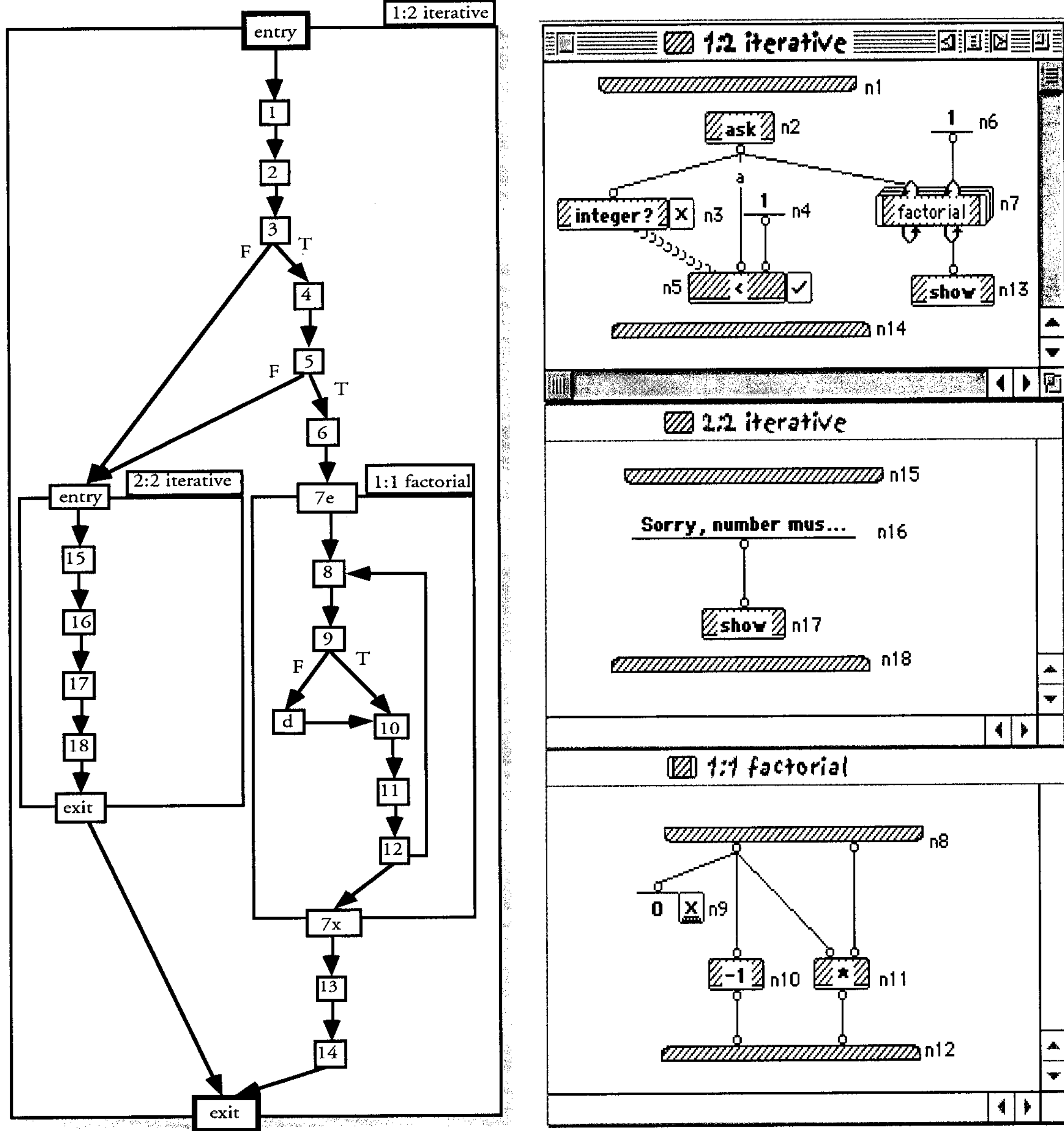

Figure 3-7. An iterative Factorial method containing one error (right), and its OCG (left).

The control annotated operation in *case* "factorial 1:1" should have been annotated with a Terminate on success rather then a Finish on failure. The program takes one input and cal-

culates its factorial. The label (*a*) on the *root* of the *ask* operation in Figure 3-7 represents the input to the *factorial* method and will be used throughout this discussion. The *OCG* for the *factorial* method is shown in the left side of Figure 3-7. Labels placed on the nodes in the *OCG* correspond to the operation labels in the program, as shown in Figure 3-7.

Consider the test suite shown in Table 3-1. The first three test cases in the test suite are node-adequate; however, they are not edge-adequate because edge (9, 10) is not traversed. After running the test suite under the all-branches criterion, the operation labelled *n9* will be colored half green and half red. No amount of further testing will result in 100% edge-coverage, and the red part of the operation labeled *n9* will persist. Aside from indicating that the program contains an error, the red part of the operation labeled *n9* helps locate the error in the visual code. The fact that the test was node-adequate and not edge adequate illustrates that, analogous to imperative languages, All-branches subsumes All-nodes coverage in dataflow languages.

Table 3-1. A test suite for the dataflow program shown in Figure 3-2.

a	output	nodes	%	edges	%
1.1	invalid domain	1, 2, 3,15, 16, 17, 18	38.8	(3, 15),	16.6
-1	invalid domain	1, 2, 3, 4, 5, 15, 16, 17, 18	50.0	(3,4), (5, 15),	50.0
2	2	1, 2, 3, 4, 5, 6, 7, 8, 9, d,10, 11, 12, 13, 14.	100.0	(3, 4), (5, 6), (9, d)	83.3
3	3"wrong result"	1, 2, 3, 4, 5, 6, 7, 8, 9, d,10, 11, 12, 13, 14	100.0	(3, 4), (5, 6), (9, d)	83.3

Now suppose we modify the factorial method so that the *factorial local* looks like Figure 3-8. The program now contains a different kind of error. Notice that although the operation labelled *n9* is now corrected, the data link connecting the *terminal* on the "-1" *primitive* is now connected to the second *root* on the Input Bar. This is indeed a fault.

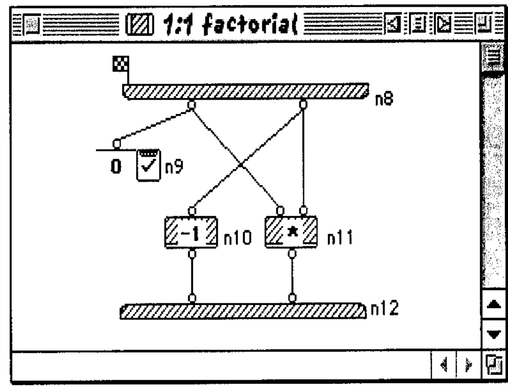

Figure 3-8. Modified Factorial Local.

Running the same test suite described above on the modified *factorial* program will result in the same output; however, after running the test suite, both edge and node testedness would be reported as 100%, but the fault would not have been detected. This is an example of a fault which may go undetected under all-nodes or all-branches testing. To help catch errors of this type, a more rigorous testing technique is required. More on this can be found in Chapter Four.

3.6 Findings Summary and New Directions

We have described issues in, and strategies for, the testing of visual dataflow programs, and presented test adequacy criteria that are based on the control flow of those programs. We have shown ways to apply both All-nodes and All-branches testing criteria to visual dataflow languages. The All-branches criterion can provide important error detection ability, and subsumes the All-nodes criterion, as is the case in imperative languages.

However, as we have seen in the example, there can still be errors which will not be detected by either the All-branches or All-nodes criteria. Thus, to help catch such common errors in Prograph, a more rigorous testing approach is needed. Data-flow testing criteria have proven more effective in uncovering errors than control-flow test adequacy criteria [27][28][31][32][67]. Dataflow testing criteria analyze the definition and use of data elements in a

program [26]. These definition and use associations are commonly called *def-use* relationships. Dataflow analysis focuses on how variables are bound to values, and how these variables are to be used. So rather than selecting program paths based on the control structure of a program, data-flow criteria select paths based on tracing input variables through a program, until they are ultimately used to produce output values.

In adapting data-flow test adequacy criteria to dataflow languages, we run into some difficulty that stems from the lack of explicit variable definitions in Prograph. One possible approach is to treat each *root* as a variable definition and each *terminal* as a variable use, but exactly how this can be used to apply data-flow testing criteria to dataflow languages is the topic of discussion of Chapter Four.

4 A Data-flow Testing Methodology for Visual Dataflow Languages

4.1 Introduction

Data-flow test adequacy criteria for imperative languages relate test coverage to interactions between occurrences of variables in the source code. Variable occurrences can be either definitions or uses, depending on whether those occurrences store values in, or fetch values from the memory, respectively. Intraprocedural data-flow analysis for imperative languages considers the flow of data within a procedure, while assuming some approximation about definitions and uses of reference parameters and global variables at call sites. Interprocedural data-flow analysis considers the flow of data in a program as whole.

In contrast to imperative languages, variable definitions in visual dataflow languages such as Prograph are not explicit. A *root* in Prograph implicitly defines a variable, whereas a *terminal* implicitly uses or references that variable, provided that there is a datalink between the *root* and the *terminal*. Thus, interactions between occurrences of variables in visual dataflow languages are modeled as datalinks from *roots* to *terminals*. Testing these datalinks or visual interactions is the purpose of this Chapter. We discuss differences between imperative languages and the dataflow aspect of Prograph. Our findings reveal an opportunity to adapt code-based data-flow testing to visual dataflow languages. The adapted code-based data-flow testing is subsequently used to achieve both intraprocedural and interprocedural data-flow test adequacy criteria for visual dataflow languages, in particular the "All-du-paths". The remainder of this Chapter is organized as follows: in Section 4.2, we give an overview of the visual dataflow aspect of Prograph, and discuss how code-based du-associations are adapted to visual dataflow languages. In Section 4.3, we present our data-flow testing methodology for testing intraproce-

dural du-associations in visual dataflow languages, followed by an example and its empirical results. The latter revealed that data-flow testing for visual dataflow languages provide a more inclusive coverage than those obtained by either All-nodes or All-branches criteria introduced for visual dataflow languages in Chapter 3. In Section 4.4, we discuss issues relating to the analysis and collection of interprocedural du-associations in Prograph. We then give an example to illustrate the usefulness of interprocedural data-flow testing for dataflow languages; in particular the detection of errors resulting from interfacing with loop annotated methods.

4.2 The Procedural Aspect of Dataflow Languages

The data interaction model in visual dataflow languages, although visual, is analogous to that of imperative languages. In imperative languages, data interaction between variables is made by explicitly defining and referencing the names of variables in a procedure. For example, the statement s_1: $x = 3$ explicitly defines x and writes to its memory location, whereas the statement s_2:$y = x+3$ explicitly uses or references x by reading from its memory location. In visual dataflow languages, variables cannot be explicitly defined.

Variable interactions are modeled as datalinks connecting operations' *roots* to other operations' *terminals*. In this interaction model, *roots* serve as variable definitions and *terminals* connected to those *roots* serve as variable uses. When a *root* r, on an operation o_i is connected to a *terminal* t on an operation o_j (i and $j > 1$), we say that t in o_j references r in o_i. In other word, r is used in o_j. As with imperative languages, we recognize two types of variable uses in visual dataflow languages: *c-use* or computational use; and *p-use* or predicate use. A c-use occurs when a datalink connects a *root* r in o_i to a *terminal* t that exists on a non-control annotated operation o_j. A p-use occurs when a datalink connects a *root* r on an operation o_k to a *terminal* t that exists on a control annotated operation o_l (k and $l > 1$). For example, in Figure 4-1, the *root* labeled *r1* is c-used in the *primitive* operation *show*, whereas the *root* labeled *r2* is p-used in the control annotated *match* "2".

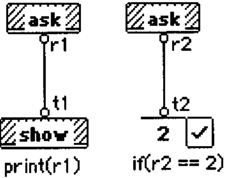

Figure 4-1. An example of a c-use and a p-use in Prograph.

Given a procedure p in a visual dataflow program P, let $O = \{o_1, o_2,..., o_n\}$ be the set of operations in p and $R = \{r_1, r_2,....,r_x\}$ be the set of *roots* on an operation $o_i \in O$, such that $1 \leq i \leq n$. Also, let $T = \{t_1, t_2,...,t_y\}$ be the set of *terminals* on an operation $o_j \in O$, such that $1 \leq j \leq n$. A *root* $r \in R$ in o_i is c-used in o_j, iff there is a datalink from r to a *terminal* $t \in T$ in o_j such that, $i < j < n$ and o_j is a non control-annotated operation. A *root* $r \in R$ in o_k where $(1 \leq k \leq n)$ is p-used in o_l where $(1 \leq l \leq n)$, iff there is a datalink from r to $t \in T$ in o_l, such that $k < l < n$ and o_l is a control-annotated operation.

4.2.1 The Method Structure in Prograph

The basic structure of a Prograph *universal* method is similar to a procedure in imperative languages. The *roots* on the Input Bar of a *universal* represent the method's input, and correspond to reference parameters in imperative languages. The *terminals* on the Output Bar of a *universal* represent the method's output, and correspond to variables "returned" in imperative languages. The reader should note that Prograph, unlike imperative languages, allows more than one *root* on the Output Bar. When a *universal* method has more than one *case*, *roots* on the Input Bar of each *case* are essentially the same "variables". A similar situation exist for *terminals* on the Output Bar. For example, in Figure 4-2, the *roots* on the Input Bars of *cases "1:2B"* and *"1:2B"* are the same. Likewise the *terminals* on the Output Bars are also the same.

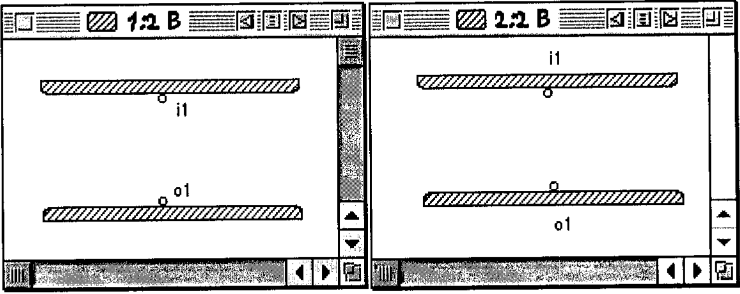

Figure 4-2. Method input and output in Prograph cases.

Operations in the body of a *universal* that are connected to *roots* on the *universal's* Input Bar, get their values from the *roots* which are connected to the *terminals* at the call site. For example, in Figure 4-3, the reference parameter labeled *i1* in "*1:1B*" gets its value from the *root* labeled *r1* in "*1:1A*". As with imperative languages, reference parameters and actual parameters in visual dataflow languages are bound at call sites. Thus, we say that the reference parameter or *i1* in "*1:1B*" is bound to the actual parameter or *r1* in "*1:1A*".

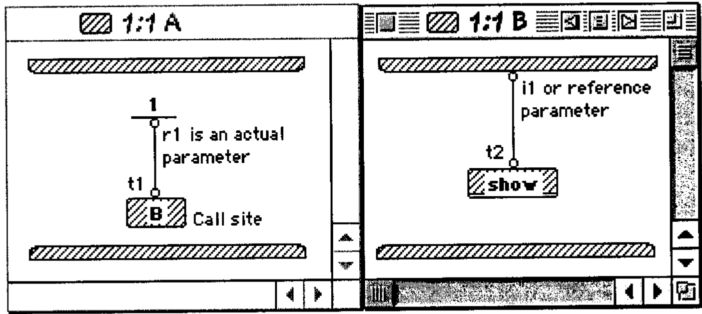

Figure 4-3. The binding an actual parameter to a reference parameter at a call site.

A *local* operation is similar in structure to a *universal* method; however, the *roots* on the *local*'s Input Bar are not considered reference parameters; rather, they correspond to the *roots* to which the *local* operation's *terminals* are connected. For example, in Figure 4-4, the *root* on the Input Bar of *case* "*1:2 aLocal*" corresponds to the *root* labeled *r1* on the *ask primitive* in method "*1:1B*". The *terminals* on a *local's* Output Bar carry the values of the *roots* to which they are connected, and through an assignment at the Output Bar assign those values to the *local* operations' *roots*. To illustrate, consider the *local* operation "*1:2 alocal*" in Figure 4-4. In the latter,

the *terminal* labeled *t5* carries the value of the *root* labeled *r4*, and through the *"r2 = r5"* assignment at the Output Bar, assigns the value of *r4* to *r2* in *"1:1B"*. A similar assignment takes place on the Output Bar of *case* "*2:2 aLocal*" operation.

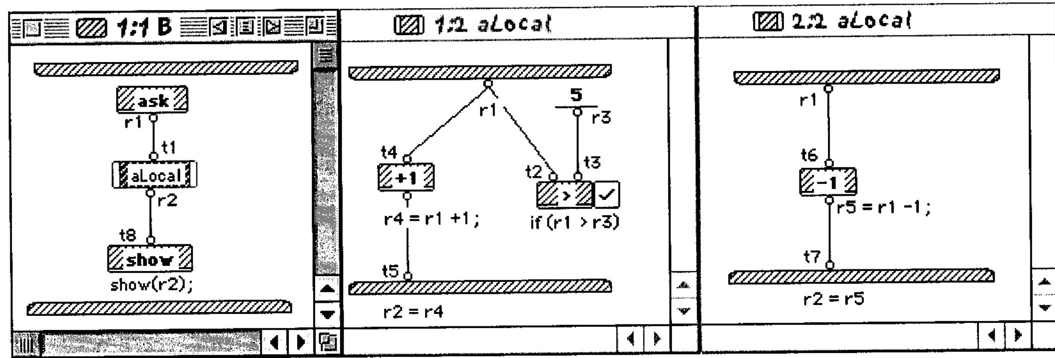

Figure 4-4. Variable interactions in the structure of a *local* operation in Prograph.

Roots corresponding to formal parameters cannot be redefined in the procedural aspect of Prograph. On the other hand, *roots* corresponding to actual parameters can be redefined only when they are used as loop *roots*. To illustrate, consider the looped *local* operation *"1:1 aLoop"* in Figure 4-5. In the latter, the assignments "*r2 = r1*" and *"r2 = r4"* occur at the at the loop *root* in "*1:1A*" and Output Bar of the *"1:1 aLoop"*, respectively. The first assignment *"r2 = r1"* is necessary to carry the flow of data in case the flow of execution breaks before it reaches the Output Bar of *"1:1 aLoop"*. The second assignment is necessary to increment the index of the loop or *r2*.

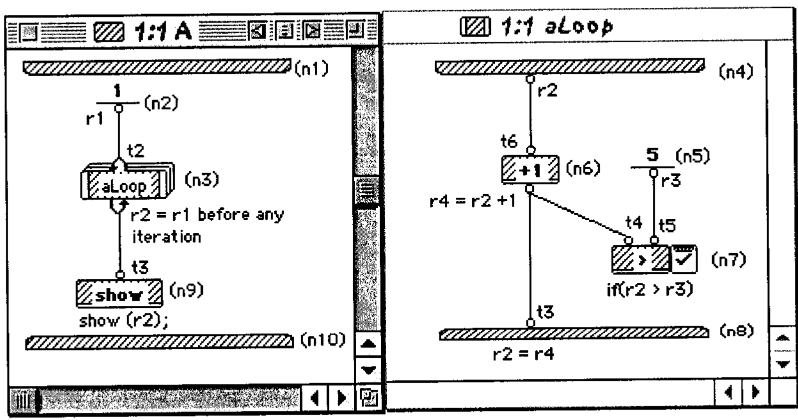

Figure 4-5. An example of a looped annotated local operation in Progragh.

Every *universal* method contains implicit variables that are maintained by the editing environment of Prograph. These implicit variables are associated with each *case* in the method and each control annotated operation present in the body of each *case*. The purpose of these implicit variables is to control the flow of execution of operations in each *case* of the method. For example, in Figure 4-5, when the control annotated operation ">" in the looped *local* "*1:1 aLoop*" executes, the implicit control variable associated with it is checked against the implicit control variable associated with the "*1:1 aLoop*" *case*. If the value received at *t4* in the control annotated operation ">" is greater then *5*, then the value of the implicit control variable associated with the control annotated operation ">" is changed, and is compared to the value of the implicit variable of the *case*. This comparison reveals that the iteration should stop. If the value received at t4 is less than 5, the value of the implicit control variable associated with the control annotated operation ">" remains unchanged, and thus once compared with the implicit control variable of the *case*, reveal that the iteration can continue. The values of implicit control variables are particularly useful during data-flow testing. For example, in an *OCG* representing a looped *local*, the value of an implicit control variable associated with an operation that is annotated with *Terminate* on *success* inside the loop, is required to determine whether the edge representing the iteration of the loop should be traversed. More on this can be found in Section 4.3.

4.2.2 Definition-use Association for Dataflow Languages

Adapting the approach in [17], a du-association in a visual dataflow program links a *root* with *terminals* that the *root* or definition can reach. We consider two types: a definition-c-use association; and definition-p-use association. Given a procedure p in a visual dataflow program P, let $O = \{o_1, o_1, \ldots, o_n\}$ be the set of operations in p, and N be the set of blocks or nodes in an *OCG* representing p. Let $R = \{r_1, r_2, \ldots, r_x\}$ be the set of *roots* on an operation o_i, where $1 < i < n$. Also, let $T = \{t_1, t_2, \ldots, t_y\}$ be the set of *terminals* on an operation o_j, where $1\ i < j < n$. A def-c-use association is a triple $((n_i, n_j, (r, t))$, such that, n_i and n_j are nodes or blocks in N representing operations $o_i \in O$ and $o_j \in O$ respectively, $r \in R$ in o_i, $t \in T$ in o_j, there is a datalink between r and t, o_j is a non-control annotated operation, and there exists an assignment of values to p's input, in which n_i reaches n_j. For example, the def-c-use with respect to *r4* at *n6*

and its c-use at the Output Bar in the *"1:1 aloop"* of Figure 4-5 is $((n6, n8, (r4, t3))$. A definition-p-use association is a triple $(n_i, (n_j, n_k), (r, t))$, such that, n_i, n_j, and n_k are nodes or blocks in *OCG* representing the subset of operations $\{o_i, o_j, o_k\} \in O$, $r \in R$ in o_i, $t \in T$ in o_j, there is a datalink between r and t, o_j is a control annotated operation, and there exists an assignment of values to p's input, in which n_i reaches n_j, and causes the predicate associated with n_j to be evaluated such that n_k is the next node to be reached. For example, the def-p-use with respect to $r4$ at $n6$ and its p-use at $n7$ in the *"1:1 aloop"* of Figure 4-5 is: $\{((n6, (n7, n8), (r4, t4)), ((n6, (n7, n3_x), (r4, t4))\}$, where $n3_x$ is the exit node of the *OCG* of *"1:1 aloop"*.

Visual dataflow languages are like their imperative counterpart in that they are both error prone to infeasible or dead code. Thus, we preserve the applicability of data-flow adequacy criteria by specifying only executable du-associations. That is, du-associations for which there exists some input that causes the definition to reach the use. Analogous to imperative languages, determining whether a du-association – in a visual dataflow program – is executable, is however, a difficult problem [17]. Algorithms for calculating du-associations that exist in an imperative program conservatively approximate them by collecting the du-associations that appear (statically) to exist in the code.

In this work we use the same approach, that is, we define a data-flow test adequacy criterion in terms of du-associations that appear to exist under static analysis. With imperative languages, we can require All/some definitions reach All/some of their associated c-uses/p-uses, via all/some of these paths over which the definitions can possibly reach those uses [76]. Many of these criteria can be adapted to visual dataflow languages; however, in this thesis we focus on the All-du-path criterion since it has been shown in imperative languages to subsume all other code-based testing criteria [32][93]. Let Q be the set of complete execution paths in an *OCG*. We say that a test suite Δ is All-du-adequate for a visual dataflow program p if and only if, for each *root* r in an operation o_i, all du-associations with respect to r are included in Q.

The presence of dynamically determined addressing such as dynamic array indexing or pointer access complicates the data-flow analysis for imperative languages. These complications, although they have been addressed [39], still result in some imprecision during the data-flow

analysis. With visual dataflow languages, such as the one we are considering for this work, pointers and dynamic arrays are not considered.

4.3 Implementation

To provide a truly visual testing and validating environment when testing du-associations related to datalinks in visual dataflow languages, we have developed a testing methodology that provides the following functionalities:

- The ability to collect the static analysis of du-associations.
- The ability to track the dynamic analysis of du-associations.
- The ability to visually communicate the testedness of each datalink, which in turn helps locate faults in the visual code.

Testing and communicating results is an integrated process within the Prograph environment. We extend the *DFTT* that was introduced in Chapter 3, to allow the user to chose a data-flow testing criterion such as All-du-paths. This testing methodology allows the tester to visually observe the testedness of du-associations and their associated datalinks. A datalink represents a def-c-use association iff the former links a *root* to a *terminal* on a non-control annotated operation. To reflect the testedness of a datalink that represent a def-c-use association, our testing environment tracks the datalink in the visual code and colors it green if it is traversed, or red otherwise. On the other hand, a datalink represents a def-p-use association iff the former links a *root* to a *terminal* on a control annotated operation. To reflect the testedness of a datalink that represent a def-p-use association, our testing environment tracks the datalink in the visual code and colors it green if both outcomes of the evaluated predicate are traversed, half green and half red if only one outcome of the evaluated predicate is traversed, and red otherwise.

Task1: Collecting static du-associations.

For imperative languages, a wide range of analysis techniques for computing du-chains for individual procedures are well known Aho [1] (pp. 632-633) are well known and have been

used in various tools, including data-flow testers introduced by: Frankl and Weyuker [33][34]; Harrold and Soffa [44]; and Korel and Laski [57]. These techniques propagate definitions along control flow paths to conservatively identify executable du-associations before they encounter a redefinition or a "kill". In visual dataflow languages, we could adapt the imperative approach to compute du-chains; however, there is one attribute of visual dataflow languages that requires a simpler approach. In general, a *root* in visual dataflow languages such as Prograph cannot be redefined. With the presence of loops in Prograph however, *roots* associated with a loop or *loop-roots* are first defined at the *local's* looped *root*, and then implicitly redefined at the Output Bar of each *case* in the looped *local*. For example, as depicted in Figure 4-5, the *loop-root r2* is first defined at the *local* operation, and subsequently redefined at the operation labeled *n8* with the implicit statement *"r2 = r4"*. Thus, the du-associations of a *root r* that are not associated with a *loop-root* is the set of all datalinks connecting *r* to a set of *terminals* $\{t_1, t_2,..., t_n\}$. For example, the definition of *r2* on the operation labeled *n4* in Figure 4-5 has a def-c-use on the operation labeled *n7* and a def-p-use on the operation labeled *n6*. On the other hand, the du-associations related to a *loop-root lr* are divided into two sets. One set that satisfies the du-associations with regards to the definition of *lr* on the looped *local*, and a second set that satisfies the du-associations with regards to the implicit redefinition of *lr* on the Output Bar of the looped *local*. The first set is collected by computing the du-associations with regards to the definition of *lr* that is connected or has uses, via wrap-around datalinks, to operations inside the looped *local*. For example, the definition of the *loop-root r2* on the operation labeled *n3* in Figure 4-5 has a c-use on the operation labeled *n6*. The second set is collected by computing the du-associations with regards to the implicit redefinition of *lr* (at the Output Bar) that has uses on operations inside the looped *local*. For example, the implicit redefinition of the *loop-root r2* at the operation labeled *n8* has a c-use on the operation labeled *n6*. Since the implicit redefinition of a *loop-root* always occur at the Output Bar of the looped *cases*, collecting the du-associations associated with a *loop-root* before the implicit redefinition, can be resolved statically by relying on the automatic collection of the dataflow information provided by the editing environment during the visual coding phase.

Given a procedure p in a visual dataflow program P, let $O = \{o_1, o_2,..., o_n\}$ be the set of operations in p, and $B = \{b_1, b_2,..., b_n\}$ be the set of blocks or nodes in the *OCG* representing p, such

that there is a one-to-one mapping between *O* and *B*. For each *p* in *P*, we scan the data structure that host *p*'s visual code representation, and attach to every block $b_i \in B$ $(1 < i < n)$ corresponding to every operation $o_i \in O$ the set of *terminals* *V(t)* where each element $t_i \in V(t)$ contains the set of connected *roots* *V(cr)*, such that each element $cr_i \in V(cr)$ contains information about the operation where t_i's connected *root* is defined. Using the information attached to each block $b_i \in B$, we linearly traverse *B* and determine for each *root r* defined in a block *b* its appropriate def-c-uses and def-p-uses. To efficiently keep track of each *root*'s traversed du-associations, we maintain one boolean variable for each *root's* def-c-use association, and two boolean variables for each *root's* def-p-use associations. The boolean variable associated with a *root's* def-c-use association is toggled to true when the path associated with that def-c-use association is traversed, and false otherwise. The boolean variables associated with each *root*'s def-p-use associations are both toggled to true when both outcome of the predicate are traversed, one toggled to true and the other to false when one of the predicate outcome is traversed, and both to false when none of the predicate outcome are traversed. For each datalink, we also maintain two pointers, one for the *root* and the other for the *terminal* connected to the *root*. These pointers are used during the validation phase to reflect the testedness of a datalink. More on this can be found in Task 3.

Task 2: Tracking execution traces.

The conventional way of probing the source code in imperative languages for testing purposes does not directly apply to visual dataflow languages. Thus, in this work, as mentioned in Chapter 3, we extend a PrographToJava translator in a way that allows us to probe a textual representation of the visual code. Whenever an operation in Prograph is executed, the probe records the execution of its corresponding textual Java representation. The information collected during the execution of the textual Java allow us to compare the static and executed du-associations so as to determine the testedness of the visual dataflow function.

Task 3: Visually representing executed DU association outputs

In imperative languages, testing results are often communicated back to the "tester" in the form of a log file. The visual environment and the lack of explicit variables in visual data-flow lan-

guages suggest a different approach when communicating the dataflow testing results back to the user. In visual dataflow languages, we use the information collected in Task 1 and Task 2 to allow the user to visually inspect whether a particular datalink that is related to a particular du-association has been executed. To communicate the testing result in a way that complements environment of visual dataflow languages, and the nature of its du-associations, we represent the validated results in two different ways. After a test suite is executed, the static and dynamic du-associations are compared. Based on this comparison, the percentage of traversed du-associations is inserted under the method name. The second method is similar to what we have introduced for node and branch coverage. That is, when a datalink is associated with a def-c-use association it is colored green when the boolean variable is toggled to true, and red otherwise. When a datalink is associated with a def-p-use association it is colored green when both boolean variables are toggled to true, half green and half red when one of the boolean variables is toggled to true, and red otherwise. For those wrap-around datalinks or datalinks associated with loop *roots*, we construct links from the loop *roots* to the *terminals* where they are used inside the loop *case*. The validation process of constructed links is similar to that of datalinks.

4.3.1 An Example

To illustrate what we have described thus far, consider the Factorial example and its *OCG* in Figure 4-6. In the latter, the iterative method reads an input, and that input is passed to the *factorial local* method which computes its factorial. Labels on the *roots*, *terminals*, and operations in the *factorial* example are also represented on the appropriate nodes in the *OCG*, and will be used throughout the rest of this discussion. The *local* looped *factorial* in the example of Figure 4-6 contains an error. The datalink between *r5* an *t9* should have been constructed between *r4* and *t9*.

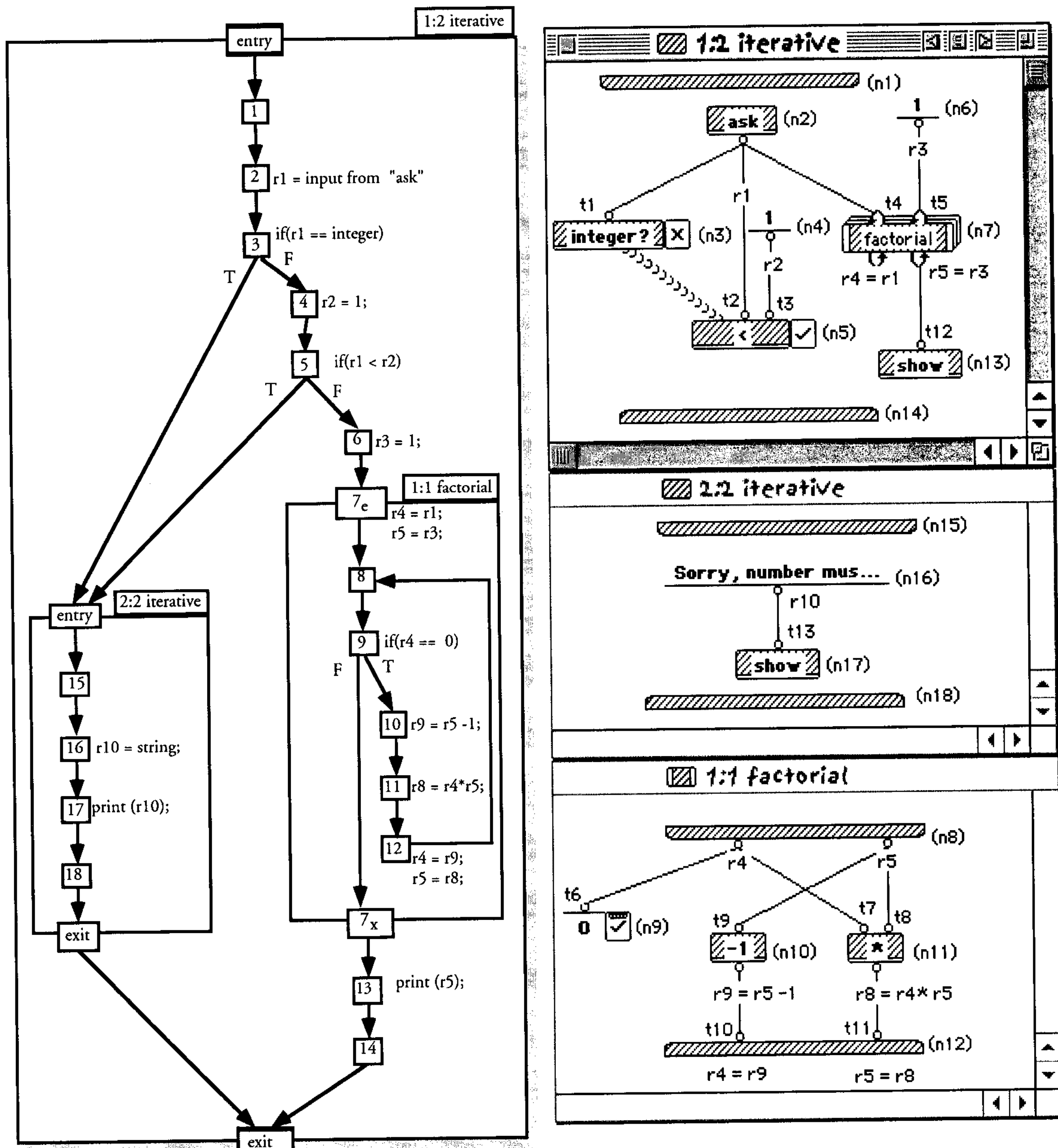

Figure 4-6. The iterative Factorial method containing an error (right), and its *OCG* (left).

Consider the test suite in Table 4-1. The first three test cases in the test suite are both All-nodes and All-edges adequate; however, all four test cases are not All-du-paths adequate because: (a) the implicit redefinition of *r4* in *n12* did not reach its uses at *terminals t7* and *t8*, in *n11*; (b) the implicit redefinition of *r5* in *n12* did not reach its uses at *terminals t9* in *n10;* and (c) the implicit redefinition of *r4* in $n7_e$ or the entry node of the looped *case n7* reached only one of

its uses on the predicate operation *n9*. In (a) and (b), *r4* and *r5* in *n12* did not reach their uses since there exists no input that would cause the path which includes the loop edge that starts at *n12* and subsequently reaches *n10* and *n11*. In (c) *r4* in $n7_e$ did not reach the true use in *n9* since *r4* in could never receive the value "0", and thus *r4* could never be tested against "0" or the true outcome of the predicate statement in *n9*. This is an example of an infeasible du-association.

Table 4-1. Exercised du-associations[1] and edges[2] of the program in Figure 4-6.

r1	output	Traversed def-use associations	All-du%	Traversed edges	All-edges%
1.1	invalid domain	(n2, (n3, n15), (r1, t1)); ((n16, n17), (r10, t13)).	9.92	(n3, n15),	16.6
-1	invalid domain	(n2, (n3, n4), (r1, t1)); (n4, (n5, n6), (r2, t3)); (n2, (n5, n15), (r1, t2)); ((n16, n17), (r10, t13)).	22.7	(n3, n4),(n5, n15),	50.0
2	2	(n2, (n3, n4), (r1, t1)); (n4, (n5, n6), (r2, t3)); (n2, (n5, n6), (r1, t2)); ((n6, $n7_e$), (r3, t5)); ((n2, $n7_e$), (r1, t4)); ($n7_e$,(n9, n10)(r4, t6)); (($n7_e$, n10), (r5, t9)); (($n7_e$, n11), (r4, t7)); (($n7_e$, n11), (r5, t8)); ((n10, n12), (r9, t10)); ((n11, n12), (r8, t11)); ($n7_e$, (n9, n12), (r4, t6)); ((n12, n13), (r5, t12)).	77.2	(n3, n4), (n5, n6), (n9, n10), (n9, nx)	100.0
3	3 "wrong result"	(n2, (n3, n4), (r1, t1)); (n4, (n5, n6), (r2, t3)); (n2, (n5, n6), (r1, t2)); ((n6, $n7_e$), (r3, t5)); ((n2, $n7_e$), (r1, t4)); ($n7_e$,(n9, n10)(r4, t6)); (($n7_e$, n10), (r5, t9)); (($n7_e$, n11), (r4, t7)); (($n7_e$, n11), (r5, t8)); ((n10, n12), (r9, t10)); ((n11, n12), (r8, t11)); ($n7_e$, (n9, n12), (r4, t6)); ((n12, n13), (r5, t12)).	77.2	(n3, n4), (n5, n6), (n9, n_x)	100.0

Since any input value > 2 will cause the value received at *n9* during the second iteration of the looped *local* to be "0", no further testing will result in traversing the du-associations of (a), (b),

1. The notation associated with the traversed du-associations can be divided into a def-c-use association notation and a def-p-use association notation.The notation of a def-c-use association is: (n_i, n_j, (r, t)) such that, n_i is the node where the root or variable r is defined; n_j is the node where r is c-used; and t is the terminal that exists on the operation represented by n_j, and to which r is connected.The notation of a def-p-use association is: ((n_i, n_j), n_k, (r, t)) such that, n_i is the node where the root or variable r is defined; n_j is the node where r is p-used; n_k is the node that is reached when the condition associated with n_j is evaluated, and t is the terminal that exists on the operation represented by n_j, and to which r is connected.

2. The notation associated with the traversed edges simply indicates which edges in the *CFG* of Figure 4-6 have been traversed.

and (c), and thus they are flagged as potential errors. To reflect the untested du-associations of (a) and (b), we construct, as depicted in Figure 4-7, three red links and one half red/half green link. The red links are constructed as follows: a red link between the loop *root r4* and *t7* in *n11*; a red link between the loop *root r5* and *t9* in *n10*; and a red link between *r5* and *t8* in *n11*. The half red/half green link is constructed between the *loop-root r4* and *t6* at *n9*. To reflect the untested du-association of (c) we color the datalink between *r4* at *n8* and *t6* at *n9* half green and half red.

Now suppose we fixed the error in the factorial method so that the *factorial local* looks like Figure 4-8. Running the same test suite described above on the modified *factorial* program will result in a correct output and only 95.45% All-du-paths testedness. Since the definition of *r4* in $n7_e$ could never be assigned a value that is less than 1, *r4* could never be tested against "0" on the predicate operation *n9*. Thus, the du-association ($n7_e$, ($n9$, $n7_x$), ($r4$, $t6$)) cannot be executed. Only when the value of the operation that is labeled *n4* is changed from 1 to 0 and a value of "0" is added to the test suite of Table 4-1, that the previously mentioned infeasible du-association can be traversed.

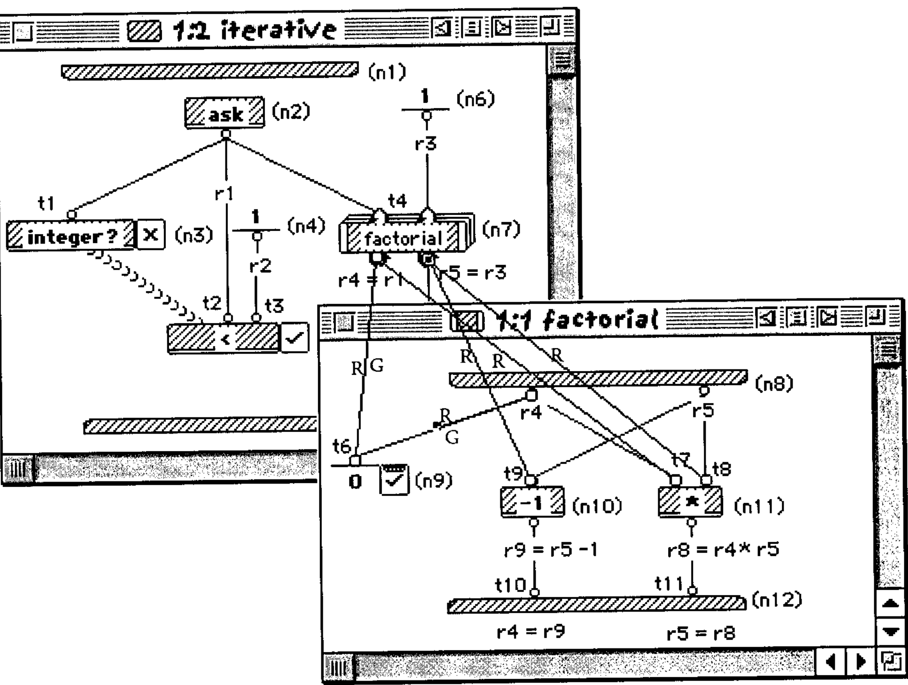

Figure 4-7. The inspected iterative procedure.

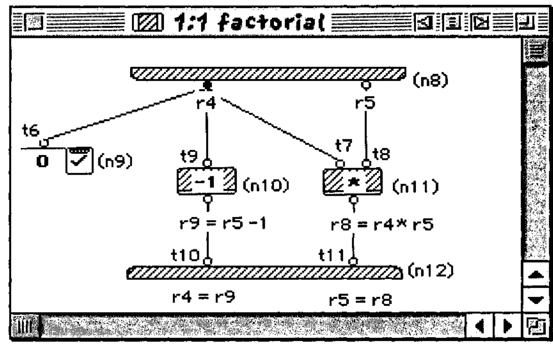

Figure 4-8. The corrected factorial looped local.

4.4 Interprocedural Data-flow Testing for Dataflow Languages

The dataflow testing methodology discussed thus far for visual dataflow languages, has been restricted to testing only du-associations that exist within a procedure or a *universal* method. As with imperative languages, visual dataflow languages encourage a high degree of modularity. This means that a visual dataflow program would normally consist of at least two or more interacting procedures. As mentioned in Section 4.2.1, *roots* on the Input Bar of a method *m* correspond to *m's* formal parameters, whereas *roots* defined anywhere else in *m* correspond *m's* actual parameters. Throughout the discussion of this Section, we will simply refer to a method's Input Bar *roots* as formal parameters, and *roots* defined anywhere else in a method as actual parameters. When testing du-chains that exist among these interacting visual dataflow procedures, actual parameters that are connected to call sites of calling procedures are bound to formal parameters in called procedures. A call site in visual dataflow languages is an operation that invokes a procedure. For example, in Figure 4-9, the operation that is labeled *n3* is a call site. Due to procedure calls, uses of a bound formal parameter in the called procedure become part of the interprocedural du-chains of its corresponding actual parameter in the calling procedure. To account for those interprocedural uses, interprocedural data-flow analysis is required. Next we discuss some of the issues related to interprocedural data-flow analysis and testing of visual dataflow languages.

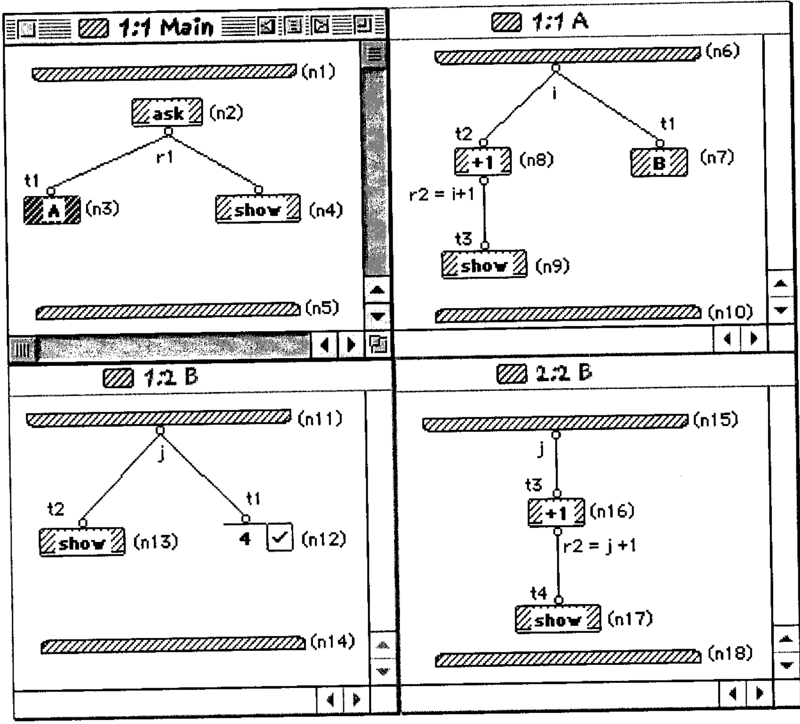

Figure 4-9. A Prograph example illustrating both direct and indirect data dependencies.

4.4.1 Issues in Collecting The Interprocedural Dataflow Analysis in Dataflow Languages

Given a visual dataflow program $P = \{p_1, p_2,..., p_n\}$, we say that interprocedural data-flow analysis in P is concerned with accounting for data dependencies that extend beyond procedure boundaries. We identify two types of interprocedural data dependencies in P: direct data dependency; and indirect data dependency. A direct data dependency is a du-association whose definition occurs in visual dataflow procedure A, and use occurs in a directly called visual dataflow procedure B of A. The condition for such a dependency exists only when: (1) a actual parameter in A is connected to a *terminal* t_1 on an operation o_1 representing a call to B; (2) i is a formal parameter in B; and (3) i is connected to *terminal* t_2 on an operation o_2 in B. For example, in Figure 4-9, the *root* labeled *r1* in *"1:1 Main"* has a direct uses in operations *n7* and *n8* in procedure *"1:1 A"*.

With imperative languages, as mentioned in Section 2.7.1, another condition that satisfies the existence of a direct data dependency states that a definition of a formal parameter in a called procedure reaches a use of the corresponding actual parameter at a return site. This condition does not hold in visual dataflow languages since a formal parameter cannot be redefined.

An indirect data dependency is a du-association whose *root* is created in procedure *A* and use occurs in an indirectly called procedure *B* of *A*. Such a dependency exists when a formal parameter is passed as an actual parameter at a call site. For example, in Figure 4-9, the *root* labeled *r1* in *Main* is bounded to formal parameter *i* in *A*, and *i* in turn is passed as an actual parameter to *B* in *A*. Thus, *r1* in *Main* has indirect uses in operations *n12, n13* in *"1:2 B"* and in operation labeled *n16* in *"2:2 B"*. Given the above conditions under which either a direct or indirect data dependency may occur, we can say that interprocedural data-flow testing for visual dataflow languages is concerned with testing both direct and indirect data dependencies.

In this Section, we discuss issues concerning the identification of interprocedural dependencies in a visual dataflow program. Consider the example in Figure 4-10 that depicts a Prograph program consisting of three universals: *Main*; *A*; and *B*. The *OCG* for each universal is also shown in Figure 4-10. To simplify the interprocedural dependency discussion, we: (1) represent a call site by a single basic block, shown as a dashed box; (2) represent a call path with an arrow whose source are a filled circle ●——▶; (3) represent a return path with a simple arrow; and (4) associate a node number with each node or block in the *OCG*(s).

There are two main issues related to interprocedural du-chain analysis in visual dataflow languages: preserving the calling context; and dealing with aliases. Each issue will be discussed next.

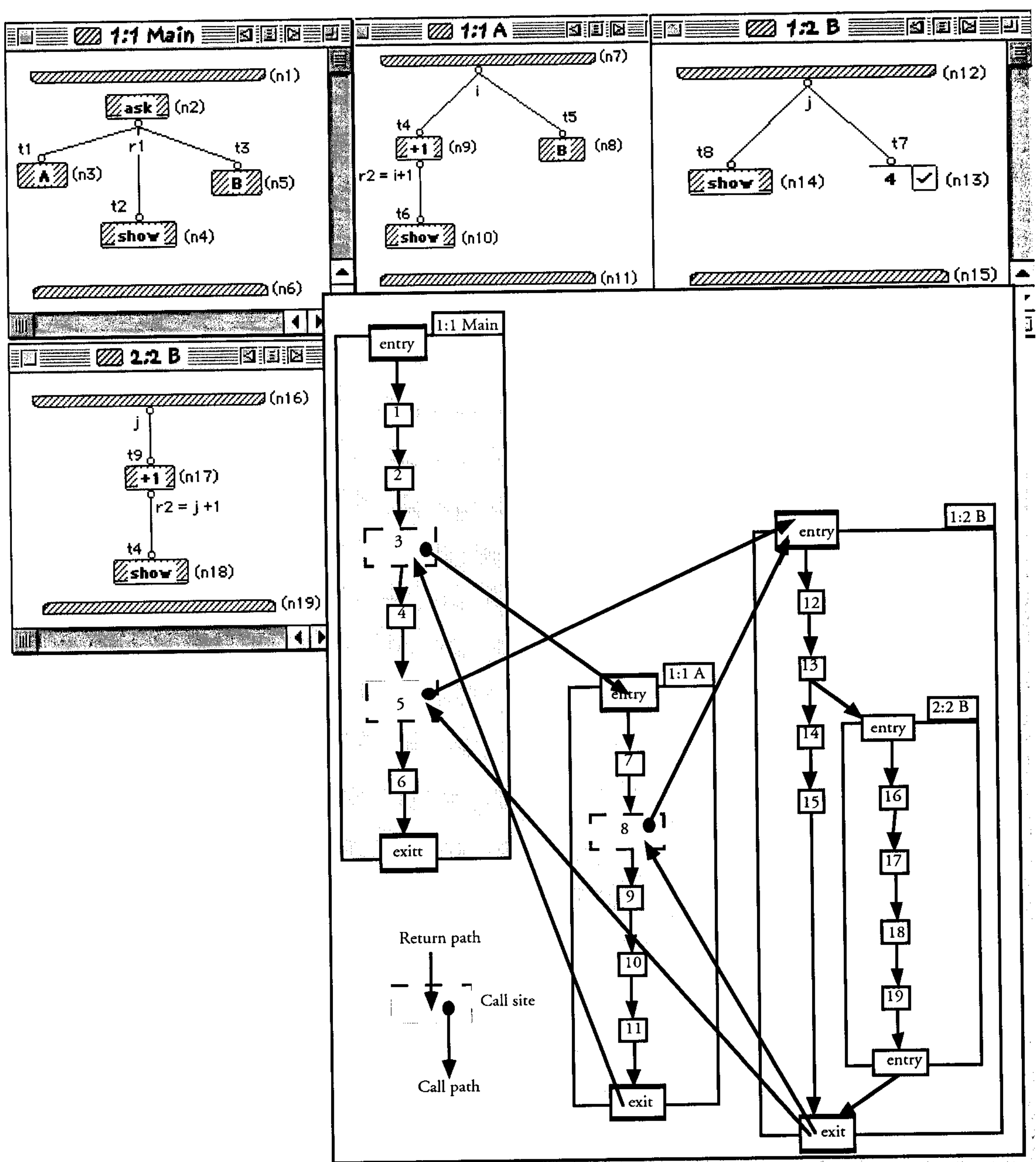

Figure 4-10. Procedures *Main*, A, and B and its corresponding *OCG* sub-graphs.

• **Preserving the calling context of called procedures.** Preserving the calling context of called procedures is important during the computation of interprocedural du-chains. To preserve the calling context, only those paths in the program that agree with the call and return sequences should be traversed. To illustrate, consider, in Figure 4-10 the definition of *r1* in *n2* that is con-

nected to the call site of procedure *B* in the operation labeled *n5*. Since there are two calls to *B*, there are two return paths from procedure *B*: one that returns directly to *Main;* and the other that returns indirectly to *Main* through *A*. These return paths are illustrated as two simple arrows starting at the exit node of *B* and into dashed boxes *n5* and *n8*. If we don't follow the correct path that agree with the call and return sequences of the call to *B* from *Main*, we could follow the return path from the exit node of *B* into dashed box *n8*. By so doing, we are suggesting that *r1,* on the call to *B* from *Main*, has interprocedural uses in the operation labeled *n9* in *A*; however, a closer inspection of control paths through the program reveals that *r1* reaches the end of *B* and subsequently back into *Main*, only when it is called directly from *Main*. Preserving the calling context requires a technique to account for the call and return sequences. More on this in Section 4.4.1.

• **Dealing with aliases.** Aliases can occur in visual dataflow languages when a *root* is connected to two or more *terminals* on an operation that represents a procedure call or call site. For example, in Figure 4-11, the actual parameter *r1* is connected to *t1* and *t2* on the operation labeled *n3*. On this call to *A* (the operation labeled *n3*), formal parameters *i* and *j* in *n5* are aliases of each other in *A*. Then, on the call to *B*, (the operation labeled *n6*) this alias is propagated to procedure *B* since *i* and *j* are passed as actual parameters, causing *m* and *n* to be aliased in *B*. The interprocedural du-chains of *r1* in *Main* is then the union of the uses of {*i*, *j*} in *A*, and {*m*, *n*} in *B*. Section 4.4.3 offers an in depth look at how our interprocedural data-flow analysis deals with aliases.

With imperative languages, as mentioned in Section 2.7.1, the issue of "maybe" preserving a variable over a procedure call has to be taken into consideration when dealing with interprocedural data-flow analysis. Given two imperative procedures *P* and *Q* such that, *x* is an actual parameter in *P*, *y* is a formal parameter in *Q*, *x* is passed to a call site that invokes *Q*, and *x* is bound to *y* in *P*. We say that *x* may be preserved over the procedure call to *Q* since we don't know if *y* can reach, unchanged, the end of *Q*. With visual dataflow languages however, this issue does not apply since a formal parameter cannot be redefined on any path in the program.

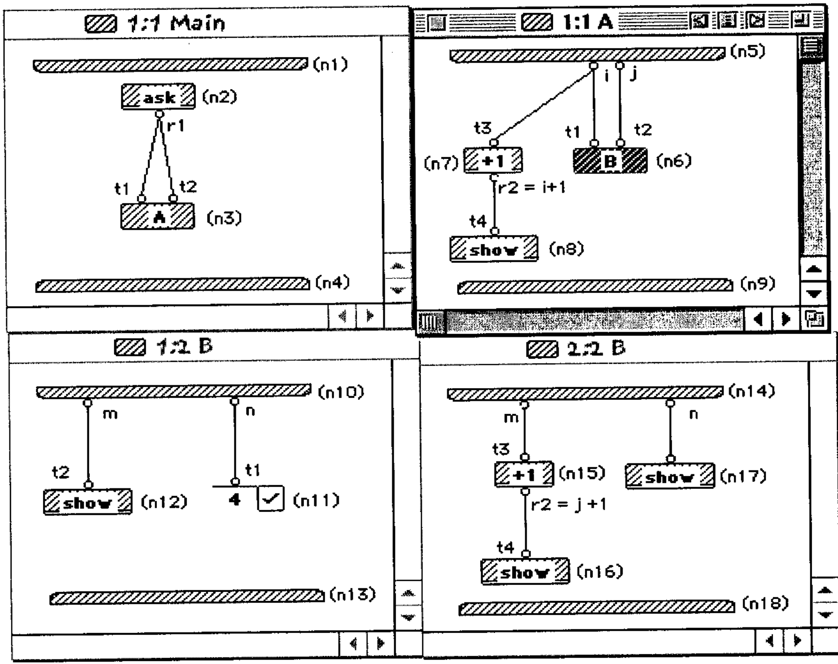

Figure 4-11. An example of an alias introduced at a call site.

4.4.2 Constructing The Interprocedural Operation Case Graph

Given a visual dataflow program $P = \{p_1, p_2, \ldots, p_n\}$, where each procedure $p_i \in P$, $(1 \leq i \leq n)$, is represented with an Interprocedural Operation Case Graph (IOCG) sub-graph, denoted $IOCG_{(pi)}$. The collection of all the *IOCG* sub-graphs, when properly connected together over call and return paths, is denoted $IOCG_{(P)}$. The $IOCG_{(P)}$ represents the control-flow of the program as whole. When constructing an $IOCG_{(pi)}$, we could collect interprocedural data-flow information by either (1) incorporating information about called procedures at call sites during the analysis of the calling procedure, or (2) estimating information about called procedures during the initial analysis of the calling procedure and update that information when more accurate data-flow information is determined about the called procedure. With the first approach, information about called procedures has to be available. This puts a restriction on the order in which procedures must be processed. Thus, in this work we use the second approach.

Adapting the approach in [19], our algorithm for constructing an $IOCG_{(P)}$ is divided into four Tasks. In the first Task, we process each procedure *p* in *P* in any order and build its $IOCG_{(p)}$. In each $IOCG_{(p)}$, we construct, when appropriate, the following nodes: *Entry*, *Exit*, *Call*, and *Return* nodes. An *Entry* node represents the point prior to the entry into the procedure, whereas an *Exit* node represents the point after the end of the procedure. An *Entry* node and an *Exit* node will be constructed for each *root* corresponding to a formal parameter. A *Call* node represents the point prior to the procedure call, whereas a *Return* node represents the point after the return from a procedure call. A *Call* node and a *Return* node will be constructed for each *root* (formal or actual parameter) that is connected to a call site. *Entry*, *Exit*, *Call*, and *Return* nodes represent the four interprocedural events or control points of: procedure entry; procedure exit; procedure call; and procedure return, respectively.

In each $IOCG_{(p)}$, we construct, when appropriate, *Interreaching* and *Reaching* edges. An *Interreaching* edge is constructed between *Call* and *Return* nodes that are associated with the same variable. The purpose of *Interreaching* edges will be explained during the propagation phase of Task 3. A *Reaching* edge is constructed as follows:

• A *Reaching* edge is constructed from an *Entry* node that is associated with of a formal parameter to a *Call* node that is associated with the same formal parameter. For example, a *Reaching* edge from an *Entry* node to a *Call* node indicates that the formal parameter is connected to a call site where it is used as an actual parameter.

• A *Reaching* edge is constructed from an *Entry* node that is associated with of a formal parameter to the *Exit* node that is associated with the same formal parameter, if the latter is not connected to any call sites.

• A *Reaching* edge is constructed from a *Return* node to either a *Call* node or an *Exit* node. A *Reaching* edge is constructed from a *Return* node to a *Call* node if the parameter that is associated with the *Return* node is also associated with another call site. For example, if a *root r* is connected to two *universal* operations, a *Reaching* edge is constructed from the *Return* node that is associated with *r* to the *Call* node that is also associated with *r*. A *Reaching* edge is con-

structed from a *Return* node that is associated with a formal parameter to the *Exit* node that is associated with the same formal parameter. These *Reaching* edges summarize the control flow structure of $IOCG_{(p)}$.

After constructing the aforementioned nodes and edges, we abstract DEF sets for every *Call* and *Exit* node, and UPCON sets for every *Entry* and *Return* node. The definition of the actual parameter that is connected to a call site constitutes the DEF set for a *Call* node. The DEF set for an *Exit* node that is associated with a formal parameter is equal to ø since a formal parameter cannot be redefined. The UPCON set for an *Entry* node that is associated with a formal parameter is the set of the du-associations related to all the datalinks connecting the formal parameter to other operations. The UPCON set of a *Return* node that is associated with an actual parameter, is the set of du-associations of all operations, other than the one responsible for the call site, that are connected to the actual parameter.

In the second Task, we construct *Binding* edges to connect the *IOCG* sub-graphs. *Binding* edges are divided into *Call-Binding* and *Return Binding* edges. The *Call-Binding* edges are constructed from *Call* nodes to *Entry* nodes, while *Return-Binding* edges are constructed from *Exit* nodes to *Return* nodes. *Call-Binding* and *Return-binding* edges are constructed based on the call structure of the program, and they correspond to the binding of formal and actual parameters.

In the third Task, we update the UPCON sets that were obtained in Task 2, by propagating information about other procedures over call and return paths in the $IOCG_{(P)}$ to obtain the interprocedural data-flow information. Finally, in Task four, we collect, for each actual parameter that is connected to a call site its interprocedural du-chain associations. Next we explain each Task in the context of the program that is depicted in Figure 4-10.

Task 1: Constructing the IOCG sub-graphs

For every formal parameter $\in p$, where p is a visual dataflow procedure, we construct an *Entry* node and an *Exit* node in the $IOCG_{(p)}$. For example, as depicted in Figure 4-12, nodes $E1_{(i)}$ and $X1_{(i)}$ are created in $IOCG_{(A)}$ for the formal parameter i in procedure A. Similarly, nodes $E2_{(j)}$ and $X2_{(j)}$ are created in $IOCG_{(B)}$ for the formal parameter j in procedure A.

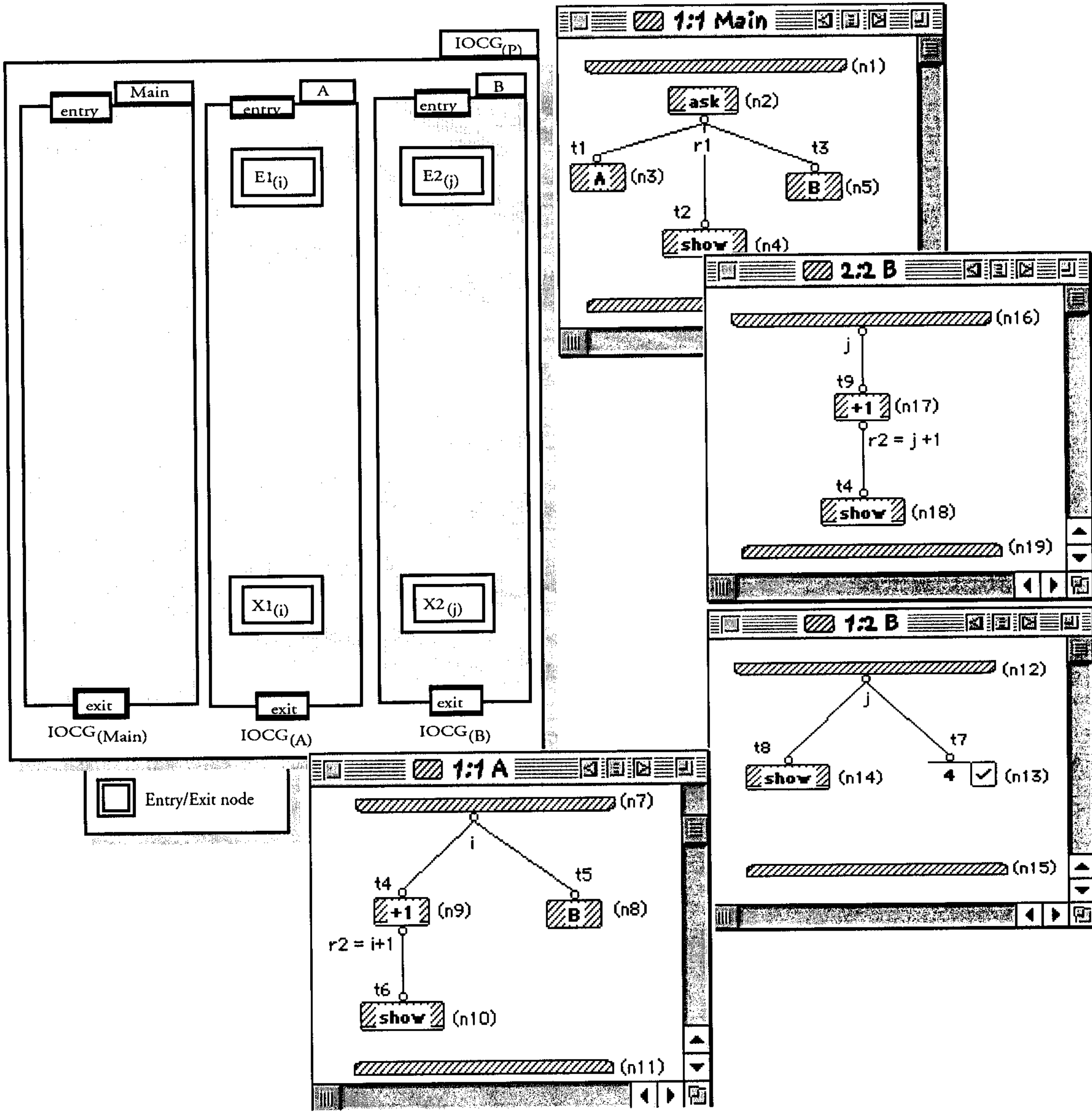

Figure 4-12. Creating the Entry and Exit Nodes for procedures A and B.

An *Entry* node represents the point prior to the entry into the procedure, whereas an *Exit* node represents the point after the end of the procedure. Note that since *Main* does not have any formal parameters, neither *Entry* nor *Exit* nodes will be constructed in $IOCG_{(Main)}$.

```
procedure ComputeInterDu-Chains (P) /* P is a collection of procedures p1, p2,....*/
{
                    /*DECLARATION*/
  NODES nodeSet[];        /* A set of nodes to process */
  EDGES edgeSet[];        /* A set of edges to process */
  DUC   duc[];            /* Array of definition-use chains */
                    /*Task 1: sub-graph construction for each procedure */
  for each p ∈ P do {
   for each root(f) ∈ p do {/*root(f) represents a formal parameter*/
     Construct an Entry node and an Exit node;
       for each root(a) ∈ p that is connected to a call site do {/*root(a) represents an actual parameter*/
         Construct Call node and Return node;
         Using the data flow information provided by the editing environment
         Construct Reaching edges for p;
         Construct Interreaching edges;
         Extract DEF[X] such that X ∈ nodeSet of {Call, Exit}; /* X is a node in IOCG(p) */
         Extract UPCON[Y] such that Y ∈ nodeSet of {Entry, Return}; /*Y is a node in IOCG(p) */
         }
     }
  }

                    /*Task 2: connecting the IOCG-sub-graphs */
   Construct Call-Binding edges among the IOCG-sub-graphs;
   Construct Return-Binding edges among the IOCG-sub-graphs;

                    /*Task 3: propagation of local information to obtain global information */
   for each node N ∈ IOCG(P) do { /* initialization N ∈ nodeSet of {Entry, Exit Call, Return} nodes*/
     INuse[N] = UPCON[N];
     OUTuse[N] = ø;
   }
                                        /* phase 1 */
   nodeSet = {Entry, Call, Return} /* nodes in IOCG(P) */
   edgeSet = {all edges in IOCG(P)}
   Propagate (nodeSet, edgeSet);
                                        /* phase 2 */
   nodeSet: = {all nodes in IOCG(P)}
   edgeSet: = {Return-Binding, Reaching, Interreaching} /* edges in IOCG(P) */
   Propagate (nodeSet, edgeSet)

                    /*Task 4: interprocedural definition-use chains computation */
     for each p ∈ P do {
       for each interprocedural definition of a root r in P do {
         DUC[r]: = ø;
          if (r is in DEF[Call]) DUC[r]:= DUC[r] U OUTuse[Call];
          if (r is in DEF[Exit]) DUC[r]:= DUC[r] U OUTuse[Exit];
         }
     }
} /* end of ComputeChains */
```

Figure 4-13. The algorithm to construct the $IOCG_{(P)}$ of a visual dataflow program P.

For every parameter ∈ p that is connected to a *terminal* t on a *universal* operation o, we construct a *Call* node and a *Return* node. For example, as depicted in Figure 4-14, nodes $C1_{(r1)}$ and $R1_{(r1)}$ are constructed in $IOCG_{(Main)}$ for the actual parameter $r1$ that is connected to the *universal* operation A *in Main*. Also, nodes $C2_{(r1)}$ and $R2_{(r1)}$ are constructed in $IOCG_{(Main)}$ for $r1$ since it is connected to the universal operation B in *Main*. Similarly, nodes $C3_{(i)}$ and $R3_{(i)}$ are constructed in $IOCG_{(A)}$ for the formal parameter i in $n7$ that is connected to $t5$ on the operation labeled $n8$ which represents a call to B from A. Note that since B does not contain call sites, neither *Call* nor *Return* nodes will be constructed in $IOCG_{(B)}$.

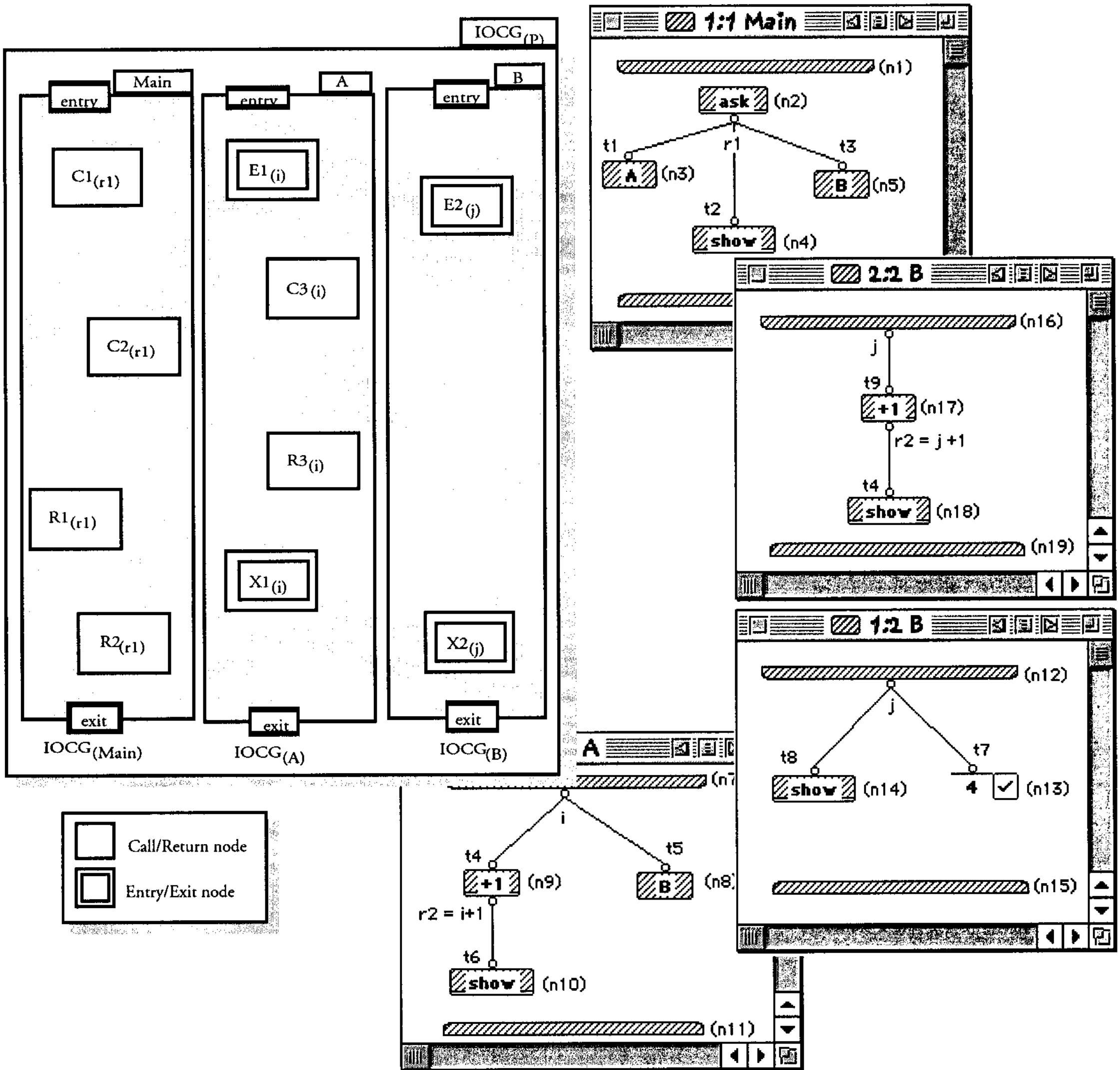

Figure 4-14. Creating the Call and Return Nodes for procedures A and B.

Next, we construct *Reaching* edges. For every formal parameter $f \in p$ that is connected to a call site, we construct a *Reaching* edge from the *Entry* node that is associated with f to the *Call* node that is also associated with f. Since $E1_{(i)}$ is the *Entry* node that is associated with formal parameter i and $C3_{(i)}$ is the *Call* node that is also associated with i, the *Reaching* edge ($E1_{(i)}$, $C3_{(i)}$), as depicted in Figure 4-15 is constructed from $E1_{(i)}$ to $C3_{(i)}$ in $IOCG_{(A)}$.

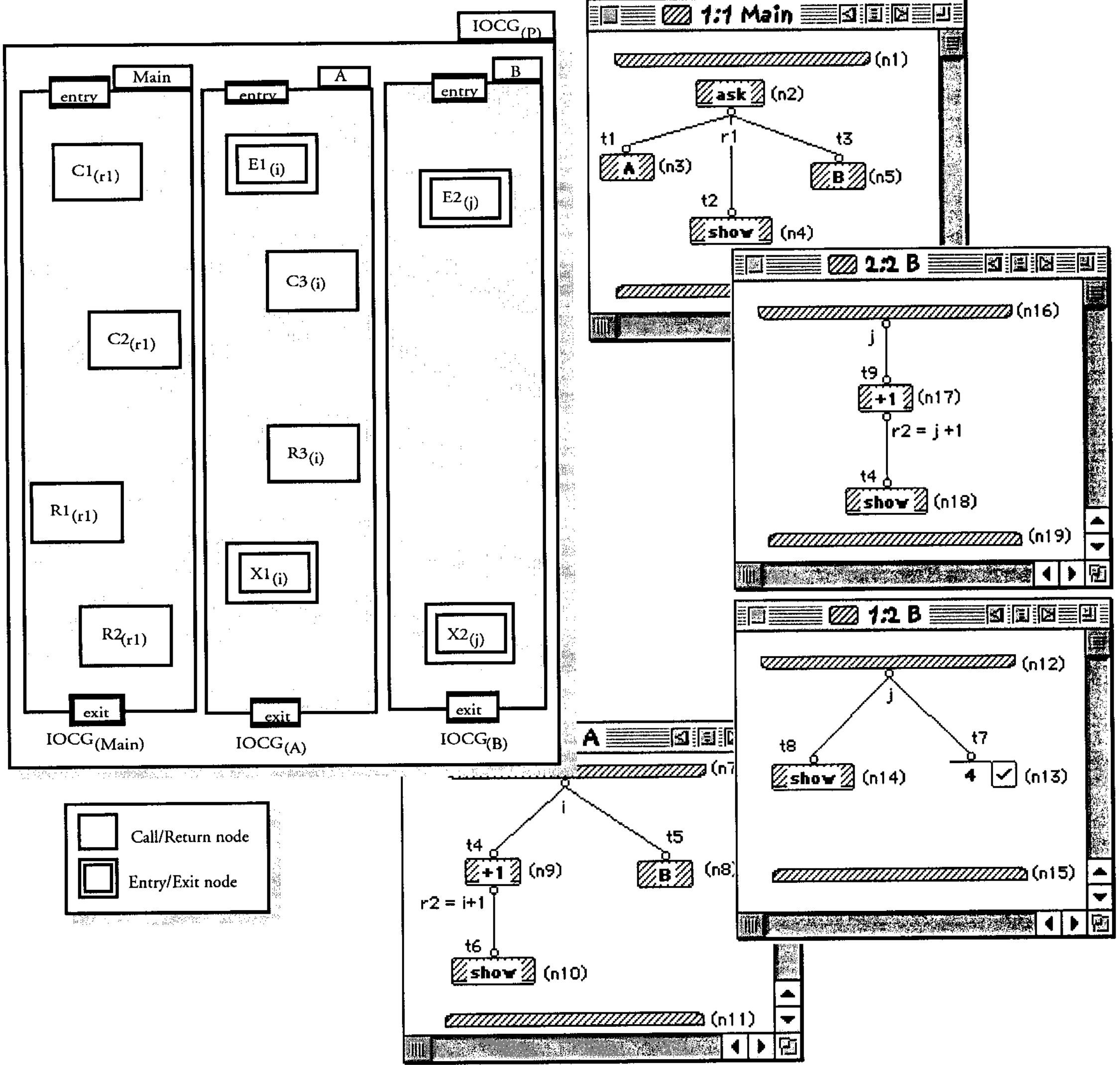

Figure 4-15. Reaching edges for procedures *Main, A*, and *B*.

For every formal parameter f that is not connected to any call site, we construct a *Reaching* edge from the *Entry* node that is associated with f to the *Exit* node that is also associated with f. Since formal parameter j is not connected to any call site, the *Reaching* edge ($E2_{(j)}$, $X2_{(j)}$), as depicted

in Figure 4-15 is constructed in $IOCG_{(B)}$ from $E2_{(j)}$, the *Entry* node associated with *j* to $X2_{(j)}$, the *Exit* node associated with *j*.

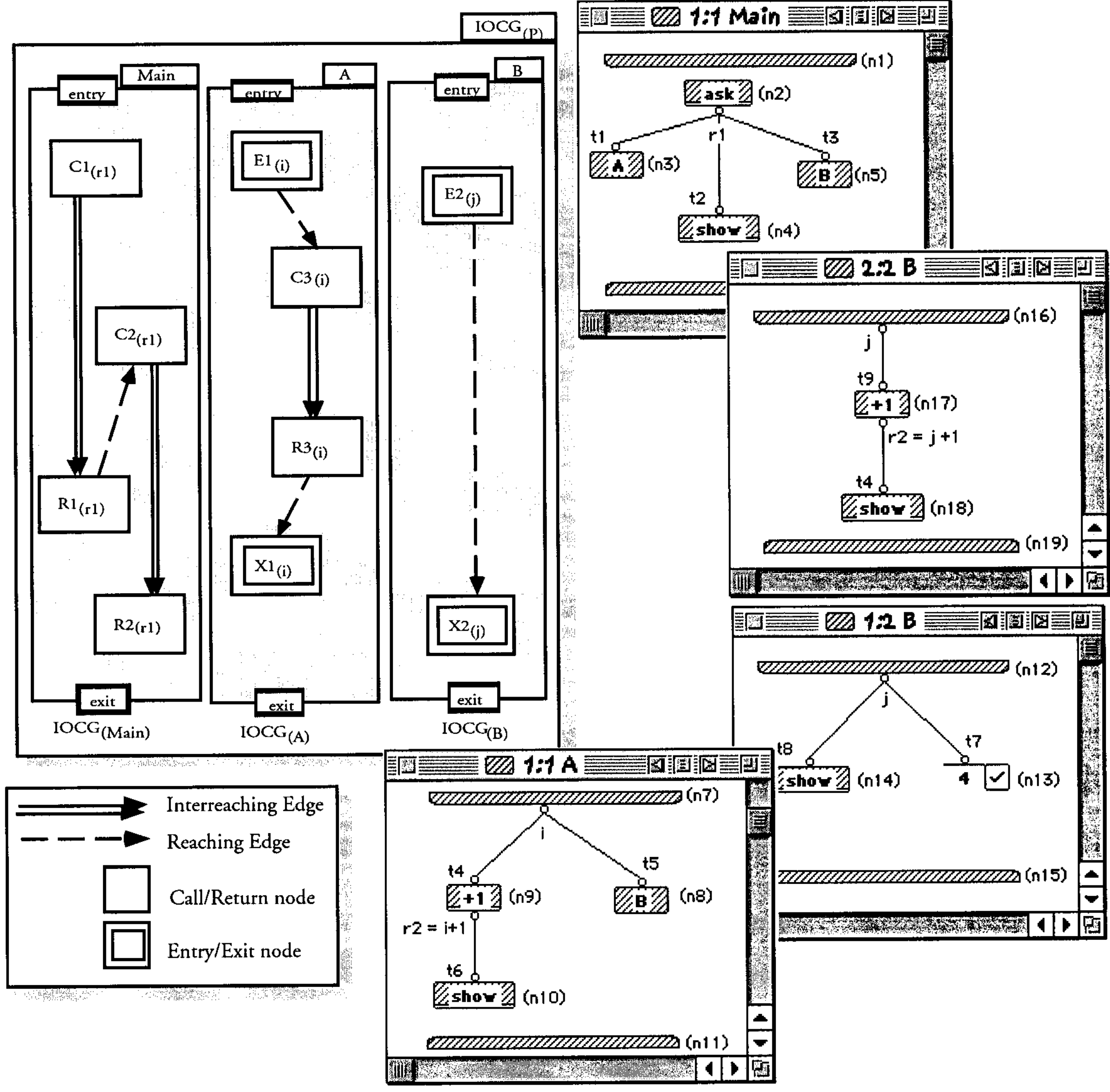

Figure 4-16. Interreaching edges for procedures *Main*, *A*, and *B* in Figure 4-10

A *Reaching* edge is constructed from the *Return* node to either a *Call* node or an *Exit* node. For every formal parameter *f* in *p* that is associated with a *Return* node, we construct a *Reaching* edge from the *Return* node to the *Exit* node that is associated with *f*. For example, as depicted in Figure 4-15, the *Reaching* edge ($R3_{(i)}$, $X1_{(i)}$) is constructed in $IOCG_{(A)}$ from $R3_{(i)}$, the *Return* node that is associated with *i* to $X1_{(i)}$, the *Exit* node that is associated with *i*. For every actual parameter *a* that is associated with a *Return* node, we construct a *Reaching* edge from the *Return*

node to the *Call* node that is associated with *a*. For example, as depicted in Figure 4-15, the *Reaching* edge ($R1_{(r1)}$, $C2_{(r1)}$) is constructed in $IOCG_{(Main)}$ from $R1_{(r1)}$, the *Return* node that is associated with *r1*, to $C2_{(r1)}$, the *Call* node that is associated with *r1*. This completes the construction process of all the *Reaching* edges for the *IOCG* sub-graphs of *Main*, *A*, and *B*.

With imperative languages, [19] the construction process of *Reaching* edges starts by solving the standard data flow problem of a reaching definition. That is, a definition of a variable *x* reaches a point *pnt* in a subroutine *S* if there is an execution path from the definition to *pnt* along which *x* is not redefined or killed. In imperative languages, a point *pnt* is said to exist before and after each statement *s* in a block *b*. In visual dataflow languages, we rely on the information collected by the editing environment to construct *Reaching* edges.

For *Call* and *Return* nodes that are associated with either a formal or actual parameter, we construct an *Interreaching* edge between the *Call* node and the *Return* node. Thus, as depicted in Figure 4-16, we construct *Interreaching* edges: ($C1_{(r1)}$, $R1_{(r1)}$) and ($C2_{(r1)}$, $R2_{(r1)}$) for *r1* in $IOCG_{(Main)}$, and ($C3_{(i)}$, $R3_{(i)}$) for *i* in $IOCG_{(A)}$. The main purpose of *Interreaching* edges is to facilitate the propagation of the UPCON sets to the *Call* and *Entry* nodes without traversing the *Call-Binding* edges. More on this in *Phase1* and *phase2* in Task3.

Table 4-2. The DEF and UPCON sets for the *IOCG* sub-graphs nodes of Figure 4-16.

IOCG nodes	DEF set	UPCON set
$C1_{(r1)}$	{r1}	∅
$R1_{(r1)}$	∅	{(n2, n4, (r1, t2)); (n2, n5, (r1, t3))}
$C2_{(r1)}$	{r1}	∅
$R2_{(r2)}$	∅	∅
$E1_{(i)}$	{i}	{(n7, n8, (i, t4)); (n7, n9, (i, t5))}
$C3_{(i)}$	{i}	∅
$R3_{(i)}$	∅	∅
$X1_{(i)}$	∅	∅
$E2_{(j)}$	{j}	{(n12, n13, (j, t1)); (n12, n14, (j, t2)); (n12, n14, (j, t2)); (n16, n17, (j, t3))}
$X2_{(j)}$	∅	

We next extract the DEF[X] and UPCON[Y] sets, where X = {*Call*, *Exit*} nodes of $IOCG_{(pi)}$, and Y = {*Entry*, *Return*} nodes of $IOCG_{(pi)}$. The DEF set for either a *Call* or *Exit* node is analogous to the DEF sets in imperative languages. The DEF set for a *Call* node that is associated with an actual parameter is equal to the *root* that is connected to the call site. For example, DEF[$C1_{(r1)}$] in Figure 4-16 is equal to {$r1$}. On the other hand, the DEF set for an *Exit* node that is associated with a formal parameter is equal ø since a formal parameter cannot be redefined. For example, DEF[$X1_{(i)}$] in Figure 4-17 is equal to ø.

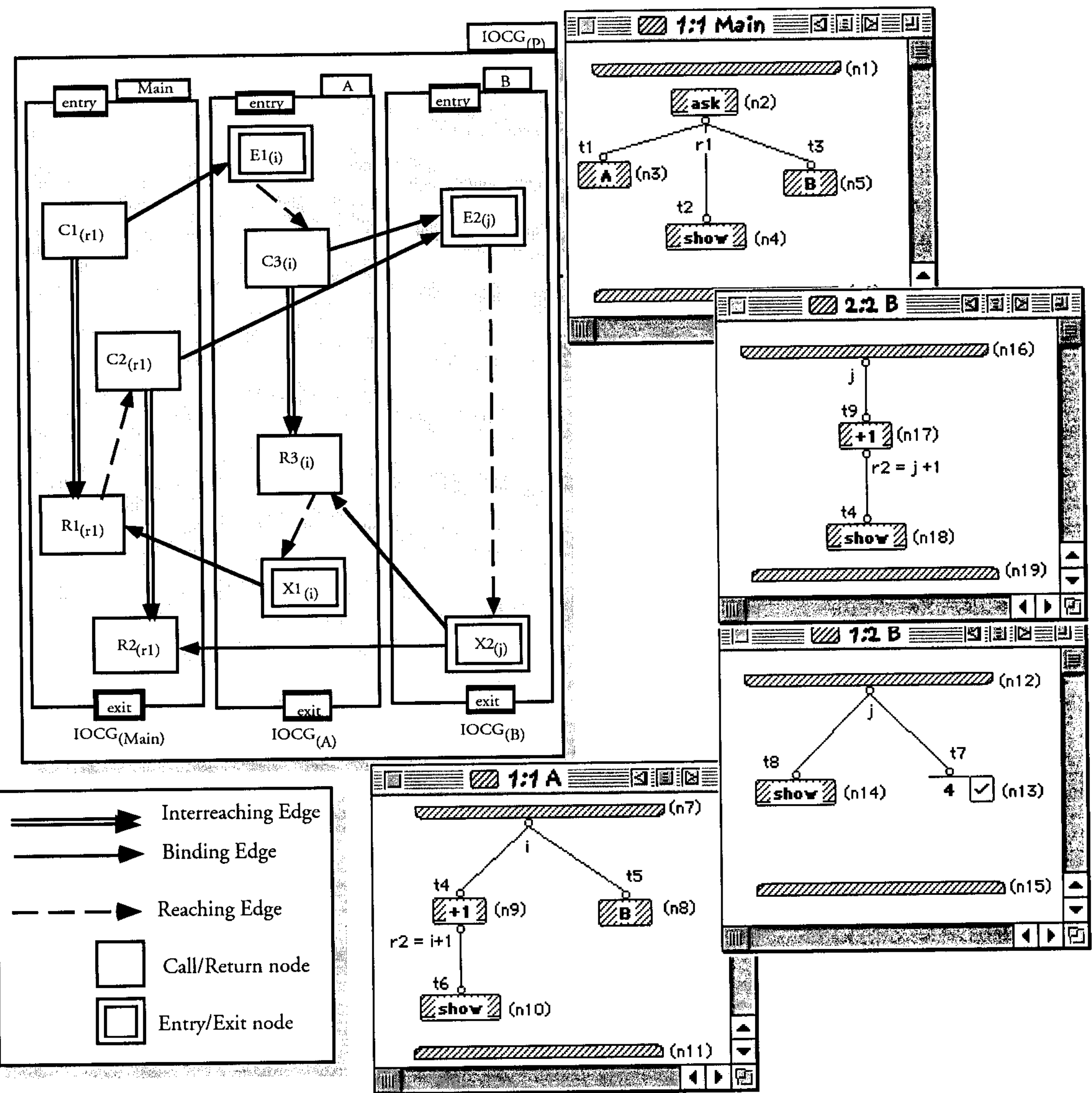

Figure 4-17. The connected *IOCG* sub-graphs for procedures *Main*, *A*, and *B* in.

The UPCON set for either an *Entry* or *a Return* node is analogous to the *upward exposed* set [1] in imperative languages. The UPCON set of an *Entry* node that corresponds to a formal parameter is the set of du-associations related to all the datalinks from the formal parameter to other operations. For example, the UPCON[$E1_{(i)}$] = {($n7$, $n8$, (i, $t4$)); ($n7$, $n9$, (i, $t5$))}. The UPCON set of an *Return* node is the set of du-associations of all operations, other than the one responsible for the call site, and to which the *root* or parameter is connected. For example, the UPCON[$R1_{(r1)}$] = {($n2$, $n4$, ($r1$, $t2$)); ($n2$, $n5$, ($r1$, $t3$))}. The DEF set for *Call* and *Exit* nodes, and the UPCON sets for *Entry* and *Return* nodes of Figure 4-17 are represented in Table 4-2.

Task 2: Connecting the IOCG-sub-graphs of P = {Main, A, B}

After all the sub-graphs have been processed in Task 1, we now connect all the IOCG-sub-graphs. To do that, we construct *Call-Binding* edges from *Call* nodes to *Entry* nodes, and *Return-Binding* edges from *Exit* node to *Return* nodes. A *Call-Binding* edge is an edge whose source is at a *Call* node and sink is at the *Entry* node. For example, as depicted in Figure 4-17, ($C1_{(r1)}$, $E1_{(i)}$), ($C2_{(r1)}$, $E2_{(j)}$), ($C3_{(i)}$, $E2_{(j)}$) are *Call-Binding* edges. A *Return-Binding* edge is an edge whose source is at the *Exit* node and sink is at the *Return* node. For example, as depicted in Figure 4-17, ($X1_{(r1)}$, $R1_{(r1)}$), ($X2_{(j)}$, $R3_{(i)}$), and ($X2_{(i)}$, $R2_{(r1)}$) are *Return-Binding* edges. *Binding* edges depend only on the call structure of the program, and not the internal structure of any *universal* method. This concludes the construction process of the $IOCG_{(P)}$ of Figure 4-17 whose program was originally depicted in Figure 4-10.

Task 3: propagating local information throughout the $IOCG_{(P)}$

After the first two tasks are completed, local (to each *IOCG* sub-graph) information such as DEF[N] and UPCON[N] is now available for each node $N \in IOCG_{(P)}$. In this task, we propagate the UPCON[N] sets backward throughout the $IOCG_{(P)}$ as far as they can be reached to obtain the interprocedural reachable uses. With imperative languages, the interprocedural reachable use problem is formulated as a simple distributive data-flow problem [55] which is solved by using procedure *Propagate* of Figure 4-18.

```
procedure Propagate (N, E)
 {
   input N set of node types to be processed
   E: set of edge types to be processed
     while dataflow changes do {
        for each node n of type N do
                for each nodes that is a successor over E of n do
                        OUTuse[n]= INuse[n] ∪ INuse[s]
                        INuse[n] = INuse[n] ∪ UPEXP[n] // UPCON[n] is substituted here
        } /* end for */
      } /* end for */
    } /* end while */

} /* end Propagate */
```

Figure 4-18. The procedure used to propagate the interprocedural uses information.

To adapt the *Propagate* algorithm described in Figure 4-18, we define for every node $N \in IOCG_{(P)}$, two sets: $IN_{use}[N]$; and $OUT_{use}[N]$. These sets denote the interprocedural uses that are reachable before and after the control points of procedure entry, procedure exit, procedure call, and procedure return. Before propagating, we initialize $IN_{use}[N]$ and $OUT_{use}[N]$ by assigning $In_{use}[N]$ to UPCON[N], and $OUT_{use}[N]$ to ø. At this point in Task 3, every node $N \in IOCG_{(P)}$ now has the following sets defined for it: DEF[N], In[N], OUT[N], and UPCON[N]. After initializing each node $N \in IOCG_{(P)}$, we adapt the *Propagate* algorithm of Figure 4-18 by substituting the UPEXP set with the UPCON set. Using the adapted *Propagate* algorithm of Figure 4-18, we can now propagate the UPCON[N] backward throughout the I$OCG_{(P)}$. The propagation in Task 3 is divided into two phases. In each phase, the call to *Propagate* iterates over a selected set of the nodes and edges in the $IOCG_{(P)}$ until the data-flow sets stabilize.

To explain the reason behind using a two-phase propagation technique, consider the backward propagation of the UPCON set for $R1_{(r1)}$ in Figure 4-17. The UPCON[$R1_{(r1)}$] consists of the use of *root r1* in operations labeled *n3, n4*, and *n5.* If UPCON[$R1_{(r1)}$] is propagated backward in the $IOCG_{(P)}$ of Figure 4-17, it would reach, among other, nodes $X1_{(i)}$, $R3_{(i)}$, $X2_{(j)}$, $E2_{(j)}$, and $C2_{(r)}$. However, this path does not match the return context. The problem occurs when this use is propagated backward over the *Call-Binding* edge ($E2_{(j)}$, $C2_{(r1)}$), since this edge does not

match the return context. The use however must be propagated, after it reaches $E2_{(j)}$, to nodes $C3_{(r1)}$, $E1_{(i)}$, and $C1_{(r1)}$, since this path does match the return context.

To solve this problem, the propagation is performed in two phases. In *phase-1*, information propagating to the *Exit* node is not computed, but all other information is allowed to flow across the *Call-Binding* edges. In *phase-2*, information reaching the *Exit* node is further propagated using the *Interreaching* edge, but no information is propagated over the *Call-Binding* edges. This two-phase propagation preserves the calling context of called procedures and ensures that only possible control paths in the program are considered. Therefore, we first process only the *Entry*, *Call*, and *Return* nodes, and propagate the uses that can be reached in called procedures over the *Call-Binding* edges, *Reaching* edges, and *Interreaching* edges. Thus, in *phase-1* of the propagation, successors of a node *N* consist of the immediate successors of *N* that are *Entry*, *Call*, or *Return* nodes. Next, we propagate the uses that can be reached in calling procedures over the *Return-Binding* edges, *Reaching* edges, and *Interreaching* edges. The propagation must be restricted to these edges to prevent traversal of paths through the $IOCG_{(P)}$ that do not represent control paths through *P*. Thus, in *phase-2*, successors of a node *N* consist of the immediate successors of *N* over all edges except the *Call-Binding* edges. Thus, processing includes all nodes, but no information is propagated over the *Call-Binding* edges. To incorporate the results of *phase-1* computation during *phase-2*, the $OUT_{use}[N]$ computed in *phase-1* is used in computing the new $OUT_{use}[N]$.

Cycles can occur in an $IOCG_{(P)}$ within a sub-graph or among sub-graphs. In particular, cycles occur (1) in the interconnections of sub-graphs because the return from one procedure reaches the call to another, and (2) in programs with recursive procedures. Cycles of type (1) result in a *Reaching* edge from the *Return* node to the *Call* node. Although there is a cycle in the *IOCG*, the subset of *IOCG* nodes and edges processed by procedure *Propagate* in either *phase1* or *phase2* is cycle free. To illustrate this type of cycle, consider the program and its *IOCG* given in Figure 4-17. The path in the $IOCG_{(P)}$ through nodes $C2_{(r1)}$, $E2_{(j)}$, $X2_{(j)}$, $R3_{(r1)}$, $X1_{(i)}$, $R1_{(r1)}$, and $C2_{(r1)}$ is an example of a cycle of this type. However, *phase-1* does not include nodes $X2_{(j)}$ or $X1_{(i)}$ or edges ($X2_{(j)}$, $R3_{(r1)}$), ($R3_{(r1)}$, $X1_{(i)}$) or ($X1_{(i)}$, $R1_{(r1)}$), and consequently, no cycle is processed by procedure *Propagate*. Since edge ($C2_{(r1)}$, $E2_{(j)}$) is not included in the processing

of *phase-2*, no cycle exists among the subset of nodes and edges, and thus, no iteration is required for *Propagate.* The presence of recursive procedures may cause cycles of type (2) during the propagation of *phase1*. In this case, iteration is required in the procedure *Propagate* to compute the data sets. To avoid infinite number of iteration, we associate the *Call-Binding* edge of every recursive call with a flag to ensure that it is traversed only once.

Task 4: Computing the du-chains throughout the $IOCG_{(P)}$

The reachable use information that was computed in Task 3 for the $IOCG_{(P)}$, will now be used with local DEF sets to compute the interprocedural definition-use chains. Interprocedural definition-use chains are computed by considering DEF sets for *Call* and *Exit* nodes associated with each procedure. If a *root* $r \in$ DEF[N], where N is a *Call* node, then the interprocedural du-chain of r consists of the elements in $OUT_{use}[N]$. If $r \in$ DEF[N] for more than one N associated with the procedure, then the interprocedural du-chain is the union of the OUT_{use} sets for all N where DEF[N] contains r. Consider the example of Figure 4-10 whose completed $IOCG_{(P)}$ is shown in Figure 4-17. In that example, the interprocedural du-chains of the definition of r in *n2* is the set of uses that can be reached from node $C1_{(r1)}$ or {*n8, n9, n13, n14, n17*}.

4.4.3 Dealing With Aliases

The technique we have described in Figure 4-13 for collecting interprocedural du-chains in visual dataflow languages has not yet dealt with the presence of aliases in the visual code. In this Section, we describe a technique to provide, without altering the algorithm of Figure 4-13, a safe computation of interprocedural du-chains in the presence of aliases.

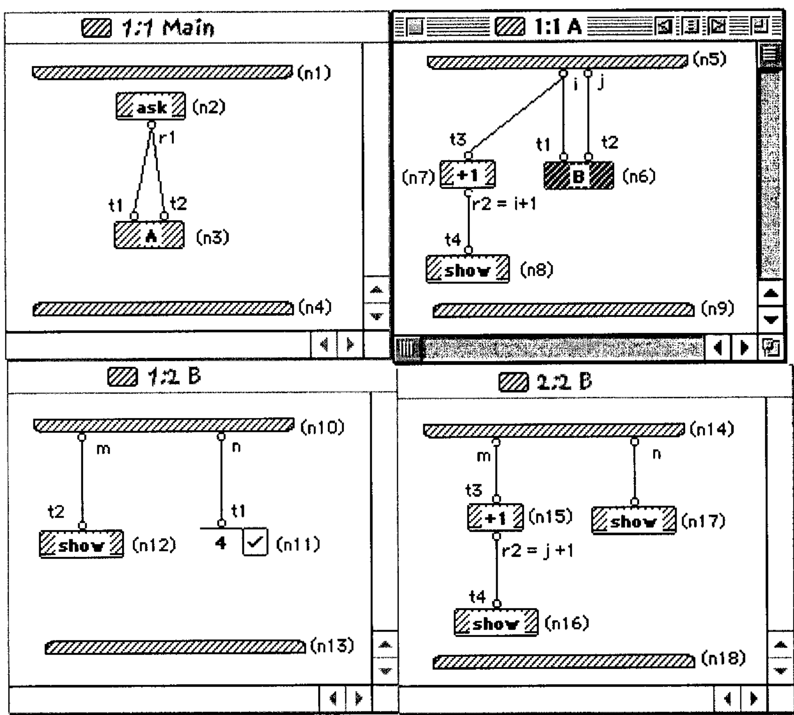

Figure 4-19. An example of an alias at a call site.

Assuming that alias pairs for reference parameters for a visual dataflow program $P = \{p_1, p_2, ..., p_n\}$ has been computed using an algorithm such as the one introduced by Cooper and Kermedy [22], we adapt a technique by [7] to unalias p in P using a program's activation tree which creates a new copy of the procedure for each different alias configuration. Thus, for each procedure p in P containing an alias, another version of procedure p' is created that contains only one formal parameter named as a linear combination of the names of all the aliased formal parameters. Then, all occurrences of formal parameters involved in the alias will be replaced with the new name. For example, in Figure 4-19, another copy of procedure A will be created, and i and j will be replaced with one formal parameter named ij. Thus, as depicted in Figure 4-20, the *IOCG* sub-graph of A' will contain only one *Entry* node, $E_{(ij)}$. Unaliasing the program in this way, although results in precise interprocedural du-chains, can be exponential in the number of parameters passed to a procedure. However, one can argue that the number of parameters passed to a procedure that result in aliasing is generally small.

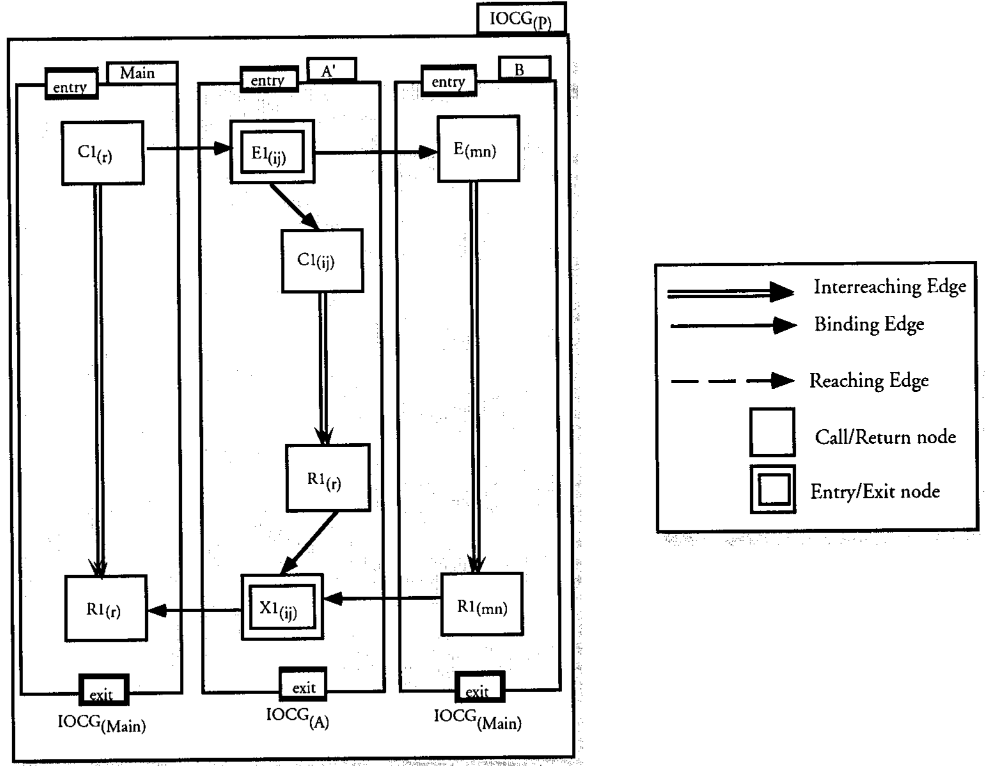

Figure 4-20. The unaliased IOCG for the program depicted in Figure 4-19.

4.5 An Example

The example in Figure 4-21 differs from that of Figure 4-6 in that the *factorial* method is not a *local*; rather, it is a *universal* method. This means that the input to *factorial*, *roots i7* and *i8*, are now reference parameters. The example in Figure 4-21 contains an error. The datalinks connection on the input to method *factorial* should have been interchanged. That is, *r1* should have been connected to *t4*, and *r3* should have been connected to *t5*. As depicted in Figure 4-21, the call to "factorial" in the method *iterative* is represented with a dashed box that contains an entry node or $n7_e$ and an exit node or $n7_x$. Since the method call is loop-annotated, there are two implicit definitions, *r4* and *r5* at $n7_e$. Next, we (1) show how the algorithm of Figure 4-13 is applied to collect the interprocedural du-chains of the Factorial program of Figure 4-21, and (2) show how interprocedural All-du-paths testing can catch the error.

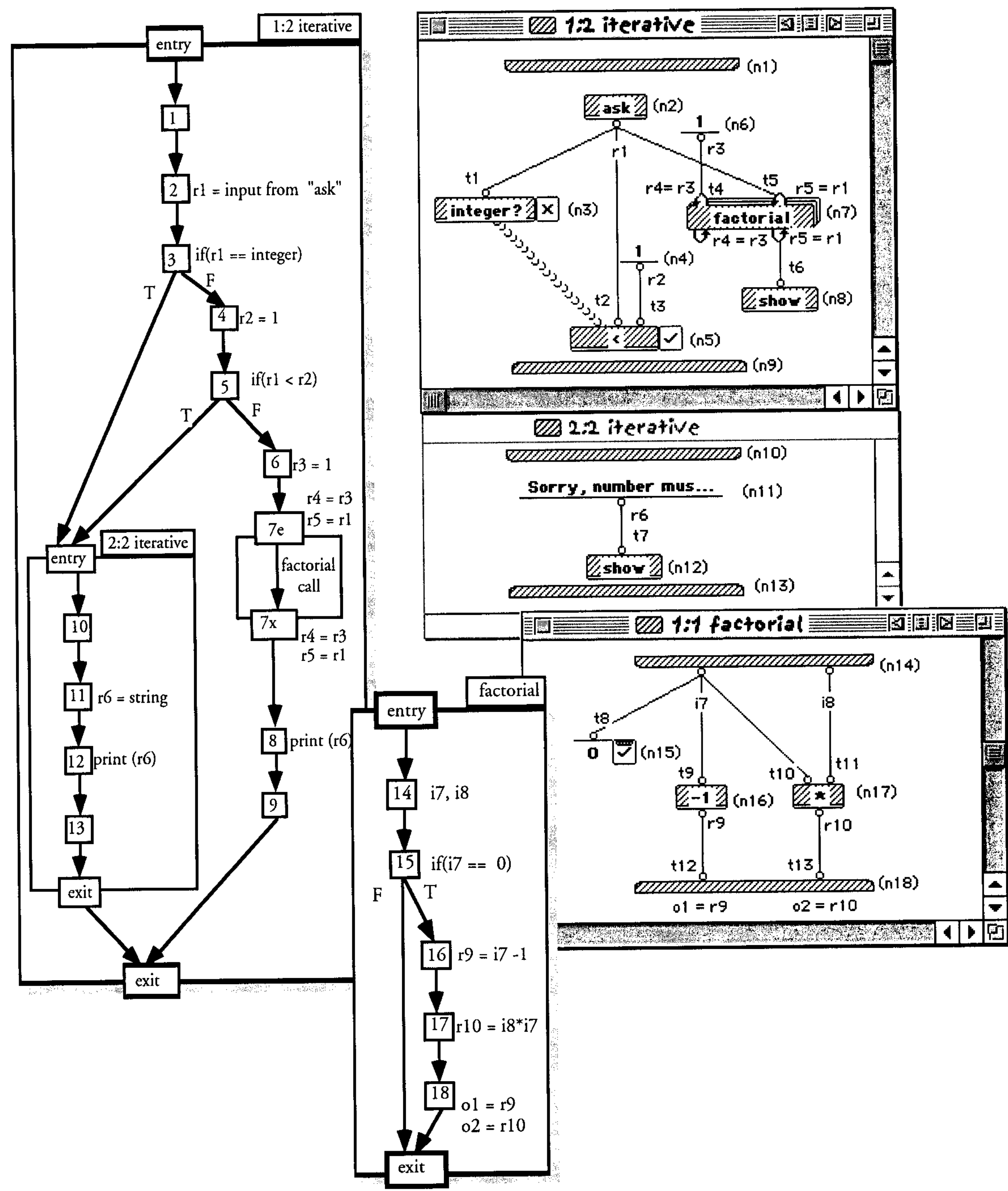

Figure 4-21. The OCGs (left) for methods iterative and factorials (right).

Task 1: Constructing IOCG sub-graphs: {factorial, iterative}

As depicted in Figure 4-22, we build $IOCG_{(factorial)}$ by first constructing an *Entry* node, $E1_{(i7)}$, and an *Exit* node, $X1_{(i7)}$, for formal parameter *i7*. Similarly, we construct an *Entry* node, $E2_{(i8)}$

and an *Exit* node, $X2_{(i8)}$ for formal parameter *i8*. Since neither *i7* nor *i8* are connected to a call site, we construct *Interreaching* edges ($E1_{(i7)}$, $X1_{(i7)}$) and ($E2_{(i8)}$, $X2_{(i8)}$). Next, we extract the DEF[*X*], where *X* = {*Call, Exit*} nodes in $IOCG_{(factorial)}$, and UPCON[*Y*], where *Y*= {*Entry, Return*} nodes in $IOCG_{(factorial)}$. As previously mentioned, a formal parameter cannot be redefined, and thus reaches, live, the *Exit* node. Since the DEF set of an *Exit* node is always ø, DEF[$X1_{(i7)}$] and DEF[$X2_{(i8)}$] = ø. To extract the UPCON[*N*], we attach the du-associations related to each formal parameter (obtained during the intraprocedural analysis) to its *Entry* node. Since *i7* is connected to *n15, n16,* and *n17,* the UPCON set of $E1_{(i7)}$ or UPCON[$E1_{(i7)}$] = {((*n14*, (*n15*, n_x), (*i7*, *t8*)); ((*n14*, (*n15*, *n16*), (*i7*, *t8*)); (*n14*, *n15*, (*i7*, *t9*)); (*n14*, *n17*, (*i8*, *t11*))}. Similarly, since *i8* is connected to *n17*, The UPCON set of $E2_{(i8)}$ or UPCON[$E2_{(i8)}$] = {((*n14*, *n17*, (*i8*, *t11*)); The DEF and UPCON sets for the *Entry* and *Exit* nodes for the *factorial* procedure are also illustrated in Table 4-3.

As depicted in Figure 4-22, we also construct the $IOCG_{(iterative)}$. Since the implicit definitions of *r4* and *r5* at $n7_e$ reach the call site to *factorial*, we construct a *Call* node, $C1_{(r4)}$, and an *Return* node, $R1_{(r4)}$, for the implicit definition of *r4* at $n7_e$. Similarly, we construct a *Call* node, $C2_{(r5)}$, and an *Return* node, $C2_{(r5)}$, for the implicit definition of *r5* at $n7_e$. Since the definitions of *r4* and *r5* at the *loop-root* in $n7_x$ are connected to the call site (*factorial*) via the wrap-around link, we construct a *Call* node, $C3_{(r4)}$, and an *Return* node, $R3_{(r4)}$, for the definition of *r4* at $n7_x$. Similarly, we construct a *Call* node, $C4_{(r5)}$, and an *Return* node, $R4_{(r5)}$, for the definition of *r5* at $n7_e$.

We next construct the following *Interreaching* edges: ($C1_{(r4)}$, $R1_{(r4)}$), ($C2_{(r5)}$, $R2_{(r5)}$), ($C3_{(r4)}$, $R3_{(r4)}$) and ($C4_{(r5)}$, $R4_{(r5)}$). Next, we extract the DEF[*X*], where *X* = {*Call, Exit*} nodes in $IOCG_{(iterative)}$, and UPCON[*Y*], where *Y* = {*Entry, Return*} nodes of $IOCG_{(iterative)}$. The DEF set for the implicit definition of *r4* at $n7_e$ or DEF[$C1_{(r4)}$] = {*r4*}. Similarly, the DEF set for the implicit definition of *r5* at $n7_e$ or DEF[$C2_{(r5)}$] = {*r5*}. The DEF set of the definition of the *loop-root r4* at $n7_x$ or DEF[$C3_{(r4)}$] = {*r4*}. Similarly, The DEF set of the definition of the *loop-root r4* at $n7_x$ or DEF[$C4_{(r5)}$] = {*r5*}. The UPCON set for the *Entry n*ode that is associated with formal parameter *i7* or UPCON [$E1_{(i7)}$] = {((*n14*, (*n15*, *nx*), (*i7*, *t8*)); ((*n14*, (*n15*, *n16*), *i7* *t8*)); (*n14*, *n15*, (*i7*, *t9*)); (*n14*, *n17*, (*i8*, *t11*))}. Similarly The UPCON set for the *Entry n*ode

that is associated with formal parameter *i8* or UPCON $[E2_{(i8)}] = \{((n14, n17, (i8, t11))\}$. The UPCON set that is associated with *loop-root r5* or $\text{UPCON}[R4_{(r5)}] = \{((n7_e, n8), (r5, t6))\}$ since *r5* is also connected to *t6* in *n8*. The UPCON sets of $R1_{(r4)}$, $R2_{(r4)}$, and $R3_{(r4)}$ is ø, since *r4* is not connected to any other operation beside the one that is responsible for the procedure call. Table 4-3 summarizes the DEF[*N*] and UPCON[*N*] sets for all the nodes of $IOCG_{(factorial)}$ and $IOCG_{(iterative)}$.

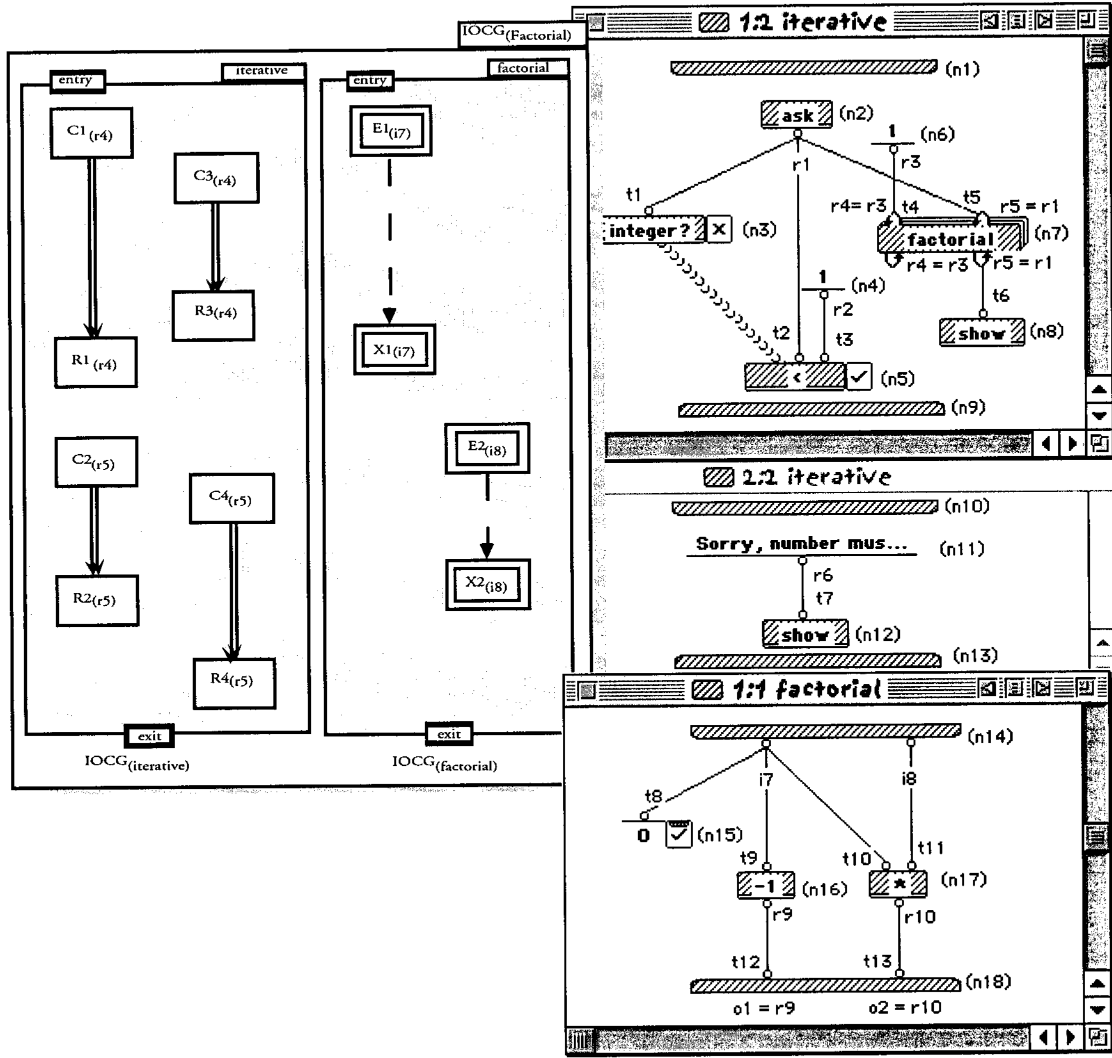

Figure 4-22. Interprocedural du-chains with looped universals.

Task 2: Constructing $IOCG_{(P)}$, P = {iterative, factorial}

As depicted in Figure 4-23, we connect the $IOCG_{(iterative)}$ and $IOCG_{(factorial)}$ sub-graphs by constructing a *Call-Binding* edge between every *Call* node associated with an actual parameter

and every *Entry* node associated with a formal parameter that is bound to the actual parameter at a call site. Thus, we construct the following *Call-Binding* edges: ($C1_{(r4)}$, $E1_{(i7)}$), ($C2_{(r5)}$, $E2_{(i8)}$), ($C3_{(r4)}$, $E1_{(i7)}$), and ($C4_{(r5)}$, $E2_{(i8)}$). We also construct a *Return-Binding* edge between every *Exit* node that is associated with a formal parameter to the every *Return* node that is associated with an actual parameter that is bound to the formal parameter at a call site. Thus, we construct the following *Return-Binding* edges: ($X1_{(i7)}$, $R1_{(r4)}$), ($X1_{(i7)}$, $R3_{(r4)}$), ($X2_{(i8)}$, $R2_{(r5)}$), and($X2_{(i8)}$, $R4_{(r5)}$). Figure 4-23 shows the $IOCG_{(factorial)}$ and $IOCG_{(iterative)}$ that correspond to the *factorial* and *iterative* methods, respectively.

Table 4-3. The DEF and UPCON sets for the program in Figure 4-21.

IOCG nodes	DEF	UPCON
E1(i7)	ø.	{((n14, (n15, nx), i7 t8)); ((n14, (n15, n16), i7 t8)); (n14, n15, (i7, t9)); (n14, n17, (i8, t11))}.
$X1_{(i7)}$	ø	ø
$E2_{(i8)}$	ø	{((n14, n17, (i8, t11))}.
$X2_{(i8)}$	ø	ø
$C1_{(r4)}$	{r4}	ø
$R1_{(r4)}$	ø.	ø.
$C2_{(r5)}$	{r5}.	ø.
$R2_{(r4)}$	ø.	ø.
$C3_{(r4)}$	{r4}.	ø.
$R3_{(r4)}$	ø.	ø.
$C4_{(r5)}$	{r5}.	ø.
$R4_{(r5)}$	ø.	(n7x, n8, (r5, t6))

In Task 3, we use the *Propagate* algorithm of Figure 4-13 to propagate the UPCON sets as far as they can be reached. In Task 4, we collect for each implicit or explicit definition that reaches the call site of *factorial* all of its interprocedural du-chains.

As previously mentioned, the example in Figure 4-21 contains an error. The datalinks connection on the input to method *factorial* should have been interchanged. That is, *r1* should have been connected to *t4*, and *r3* should have been connected to *t5*. Consider the intraprocedural All-du-paths test suite in Table 4-4. In that test suite intraprocedural All-du testing is applied

to each procedure in the example of Figure 4-21 separately. The results from third test case in the test suite of Table 4-4 shows incorrect output; however, the traversed intraprocedural du-associations for each method is adequate. This indicates that, although the error is recognized by the tester, it is caught by intraprocedural All-du testing. As illustrated in Table 4-5, testing the program as whole using interprocedural All-du testing does catch the error.

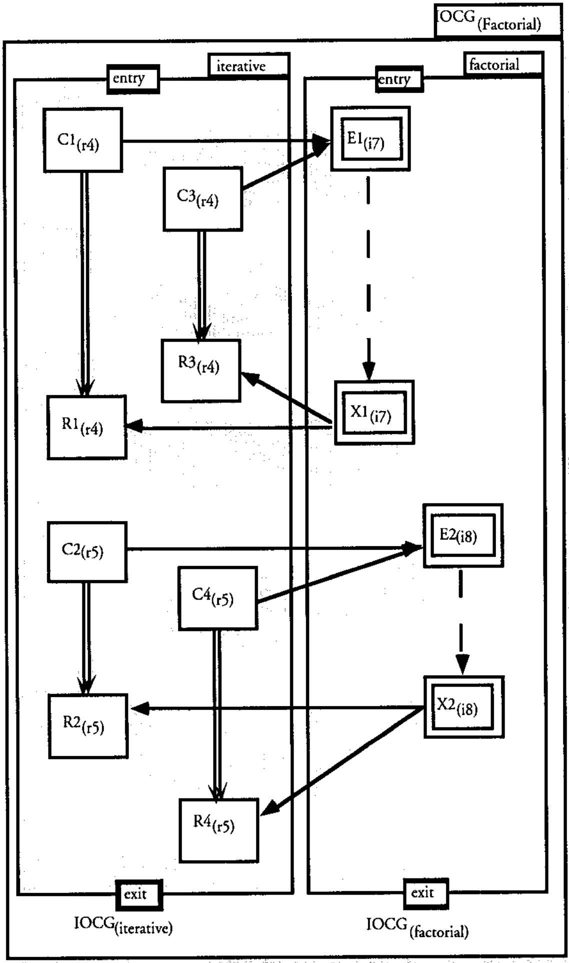

Figure 4-23. The connected IOCG for the *Factorial* program.

Consider the test suite of Table 4-5 that illustrates the interprocedural All-du testing for the Factorial program. In that test suite, the three test cases are not interprocedurally All-du paths adequate because: (a) the implicit definition of *r4* in $n7_e$ reached only one of its uses at *terminal t8* of the predicate operation *n15*; (b) the definition of *r4* in $n7_x$ reached only one of its uses on the predicate operation *n15,* and no uses at *terminals t9* and *t10;* and (c) the definition of *r5* in $n7_x$ did not reach its c-use at *terminals t11*. In (a), *r4* in $n7_e$ reached only one of its uses

on the predicate operation *n9* since *r4* in $n7_e$ could never receive the value "0", and thus *r4* could never be tested against "0". This is an example of an interprocedural infeasible du-association. In (b) and (c), *r4* and *r5* in $n7_x$ did not reach their uses since there exists no input that would cause the path which includes the loop edge to start at $n7_x$ and subsequently reaches *n16* and *n17.*

Table 4-4. Intraprocedural testedness of factorial and iterative methods.

rl	output	intraprocedural du-associations iterative	%	intraprocedural du-associations factorial	%
1.1	invalid domain	(n2(n3, n10)(r1, t1)), ((n11, n12)(r6, t17)).	13.3	ø	0.0
-1	invalid domain	(n2(n3, n4)(r1, t1)), (n4(n5, n6)(r2, t3)), (n2(n5, n10)(r1, t2)), ((n11, n12)(r6, t7)).	26.6	ø	0.0
2	2	(n2(n3, n10)(r1, t1)), (n4(n5, n6)(r2, t3)), (n2(n5, n6)(r1, t2)), ((n6, n7)(r3, t5)), ((n2, n7)(r1, t4)), ((n7, n18)(r5, t6)).	100.0	(n14(n15, n18)(i1, t8)), ((n14, n16)(i2, t9)), (n14, n17)(i1, t10)), ((n14, n17)(i2, t11)), (n16, n18)(r7, t12)), ((n17, n18)(r8, t13), (n14(n15, n_x)(i1, t8)).	100.0
3	3 "wrong result"	(n2(n3, n10)(r1, t1)), (n4(n5, n6)(r2, t3)), (n2(n5, n6)(r1, t2)), ((n6, n7)(r3, t5)), ((n2, n7)(r1, t4)), ((n7, n18)(r5, t6)).	100.0	(n14(n15, n18)(i1, t8)), ((n14, n16)(i2, t9)), (n14, n17)(i1, t10)), ((n14, n17)(i2, t11)), (n16, n18)(r7, t12)), ((n17, n18)(r8, t13), (n14(n15, n_x)(i1, t8)).	100.0

Table 4-5. Interprocedural testedness of the Factorial program in Figure 4-21.

rl	output	interprocedural du-chains for (iterative & factorial)	%
1.1	invalid domain	(n2(n3, n10)(r1, t1)), ((n11, n12)(r6, t17)).	16.6
-1	invalid domain	((n2(n3, n4)(r1, t1)), (n4(n5, n6)(r2, t3)), (n2(n5, n10)(r1, t2)), ((n11, n12)(r6, t7)).	47.0
2	2	(n2(n3, n10)(r1, t1)), (n4(n5, n6)(r2, t3)), (n2(n5, n6)(r1, t2)), ((n6, n7)(r3, t5)), ((n2, n7)(r1, t4)), ((n7, n18)(r5, t6))(n14(n15, n18)(i1, t8)), ((n14, n16)(i2, t9)), (n14, n17)(i1, t10)), ((n14, n17)(i2, t11)), (n16, n18)(r7, t12)), ((n17, n18)(r8, t13), (n14(n15, n_x)(i1, t8)).	78.0
3	3 "wrong result"	(n2(n3, n10)(r1, t1)), (n4(n5, n6)(r2, t3)), (n2(n5, n6)(r1, t2)), ((n6, n7)(r3, t5)), ((n2, n7)(r1, t4)), ((n7, n18)(r5, t6))(n14(n15, n18)(i1, t8)), ((n14, n16)(i2, t9)), (n14, n17)(i1, t10)), ((n14, n17)(i2, t11)), (n16, n18)(r7, t12)), ((n17, n18)(r8, t13), (n14(n15, n_x)(i1, t8)).	78.0

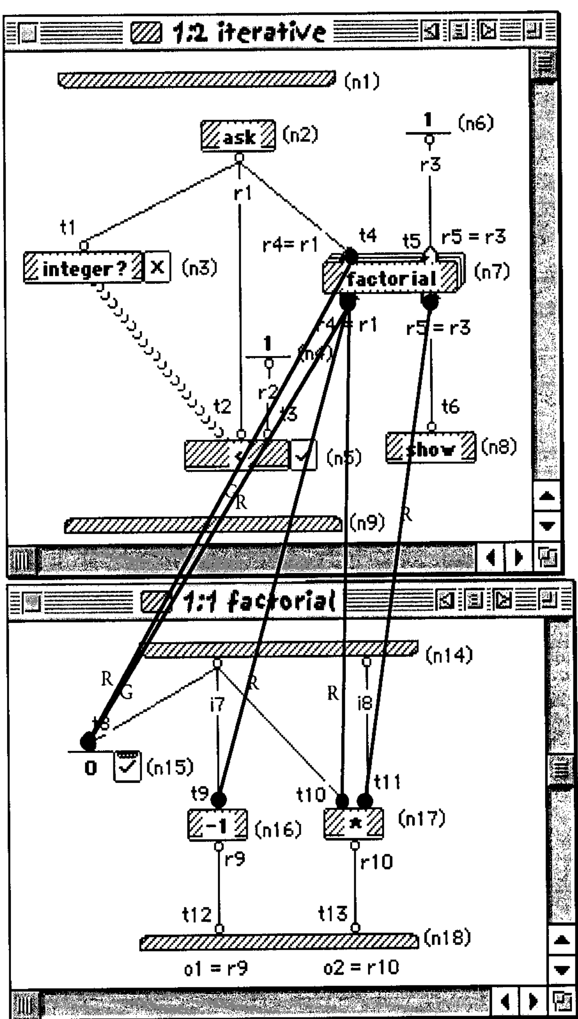

Figure 4-24. A visual communication of the testedness of the *factorial* method.

Since any input value > 2 will cause the value received at *n15* during the beginning of the second iteration of the looped *universal factorial* to be "0", no further testing will result in traversing the interprocedural du-associations of (a), (b), or (c), and thus they are flagged as potential errors. To reflect the untested interprocedural du-association of (a) we construct, as depicted in Figure 4-24, a link between *t4* (the *loop-terminal*) and *t9* at *n15,* and color it half green and half red. To reflect the untested du-associations of (b) and (c), we construct, as depicted in Figure 4-24, three red links and one half red/half green link. The red links are constructed as follows: a red link between the *loop-root r4* and *t9* in *n16*; a red link between the

loop-root r5 and *t10* in *n17*; and a red link between *r5* and *t11* in *n17*. The half red/half green link is constructed between the *loop-root r4* and *t8* in *n15*.

Now suppose we fixed the error in the *factorial* method, running the same test suite of Table 4-4 on the modified "factorial" program will result in a correct output and only 95.45% interprocedural All-du-paths testedness. Since the definition of *r4* in $n7_e$ could never be assigned a value that is less than 1, *r4* could never be tested against "0" on the predicate operation *n15*. Thus, the du-association ($n7_e$, (*n15*, $n7_x$), (*r4*, *t8*)) cannot be executed. Only when the value of the operation labeled *n4* is changed from 1 to 0, and the input "0" is added to the test suite in Table 4-4, that a 100% interprocedural All-du-paths testing is obtained.

4.6 Findings Summary and New Directions

We have described issues in, and adapted techniques for, data-flow testing of visual dataflow programs. We have shown ways to apply both intraprocedural and interprocedural All-du-paths testing for visual dataflow languages. We have also demonstrated by example, that the All-du-paths criterion can provide important error detection ability. Furthermore, as was illustrated in the results of Table 4-1, we have shown that, analogous to imperative languages [32], the All-du-paths criterion subsumes the All-nodes and All-edges criteria that were introduced in Chapter Three.

Since Prograph is an object oriented language with facilities for defining new datatypes as classes, testing these classes is essential to increase confidence when these classes are reused. Indeed, testing the visual object-flow side of Prograph is the topic of discussion of Chapter Five.

5 Testing Visual Object-flow Languages

5.1 Introduction

One of the most important benefits of Object Oriented Programming (OOP) is the ability to reuse classes in the development of other applications. Creating well designed and tested classes is essential to increase confidence when these classes are reused. The basic unit of testing in an OOP language is a class [40][41]. A class in the OOP paradigm is an information-hiding module that defines data or instance variables and operations or methods. Without loss of generality, access to instance variables and methods in a class can be either public or private. Public instance variables and methods can be accessed by users of the class, private instance variables and methods can be accessible only within the class. A programmer in an OOP language instantiates objects that interact with each others via message passing. While OOP languages may reduce some kinds of errors [12], the object oriented paradigm does not in any way rule out the basic motivation of software testing. Methods often consist of just few lines of code; however, coding errors are probably as likely as ever [12]. To help catch these errors structural-based testing is required. As discussed in Chapters Two and Four, data-flow testing is a structural-based testing technique used to test the data-flow interactions between variables in a program. With OOP languages, data interaction is generally related to instance variable access. There are three basic actions that can be performed on instance variables: (1) define action; (2) use action; and (3) kill action. A define action occurs in a message or statement that changes the concrete state of an instance variable. For example, $C.i = 3$ changes the memory location associated with the instance variable i of object C, and subsequently changes the concrete state of C. A use action occurs in a message or statement that gets the value of an instance variable without changing it. As with imperative languages [76], two types of uses have been identified, c-use and p-use. A c-use applies when the memory location associated with an instance variable

is fetched. For example, $a = C.i$ is a c-use of the instance variable i of C. On the other hand a p-use occurs when an instance variable is involved in a predicate statement. For example *if*($C.i$ == *3*) is a p-use with respect to the instance variable i. A kill action include any message, statement, or side effects that causes an instance variable to be decollated, released, or undefined. For example, the following code fragments: *foo.set*(x); and *~foo*(); first define the attribute x of *foo* then kill it.

With object oriented languages, there are three levels of data-flow testing in a class: *Intramethod; Intermethod;* and *Intraclass.* Intramethod testing tests data-flow interactions within individual methods in a class. This level of testing is similar to intraprocedural data-flow testing. Intermethod data-flow testing tests data-flow interactions between methods in a class that interact via messages. This level of testing is similar to interprocedural data-flow testing. Intraclass data-flow testing test data-flow interactions that result from sequences of public methods that can be invoked in an arbitrary way. Since the set of possible public method call sequences is infinite in large classes, intraclass testing tries to test only a subset of all possible sequences.

The integration of two or more classes in an object oriented program introduces another level of data-flow testing. For example, when a class C_1 sends a message to a class C_2, it is often desired to test the dataflow interactions between C_1 and C_2. Testing this type of data-flow interactions between classes is known as *Interclass* data-flow testing. Other data-flow interactions that are also of interest are those relevant to essential features of object oriented languages such as dynamic bindings. With dynamic binding, it is often desired to test data-flow interactions resulting from binding one message to all possible receiving methods. Another essential object oriented feature is inheritance. With inheritance, derived classes often modify, add, or delete methods or data inherited from the base class. Although experiments suggest that tests and testing information originally used to test a base class can be reused to test a derived class [41], it is often desired however to employ incremental data-flow analysis algorithms to reduce the cost of data-flow analysis in derived classes.

In this chapter, we investigate, from a data-flow testing perspective, differences between code-based object oriented languages and visual object-flow languages in the context of Prograph. Our findings reveal that, analogous to code-based object oriented languages, there are three levels of testing the dataflow interactions in a visual object-flow class. In each level, we show how code-based data-flow testing techniques can be adapted to collect that level's appropriate du-chains. For example, we show how code-based intraprocedural testing techniques can be adapted to collect the intramethod du-chains in visual object-flow languages such as Prograph. The rest of this chapter is organized as follows: in Section 5.2, we formally discuss the three aformentioned levels of data-flow testing for a class in textual object oriented languages, and briefly discuss some issues related to data-flow testing techniques for special object oriented features such as polymorphic binding and inheritance. In Section 5.3, we discuss issues relating to data-flow interactions between instance variables in Prograph. To that effect, we define two types of variables interactions: simple variable interactions; and instance variable interactions. Simple variables and are not part of the class's data, and are analogous to local variables defined inside a C++ method. The data-flow interaction associated with simple variables is analogous to data-flow interactions of atomic variables in visual dataflow languages. Instance variables in visual object-flow languages such as those found in Prograph's classes are analogous to instance variables in a C++ class. In Section 5.5, we show that intramethod and intermethod du-chains for instance variables in Prograph can be collected by applying code-based intraprocedural and interprocedural data-flow analysis techniques such as those found in [1] and [44], respectively. Finally we propose some techniques to visually validate, in Prograph, the data-flow interaction that stems from special object oriented features such as polymorphism.

5.2 Data-flow Testing for Text-based Object Oriented Languages

As previously mentioned, there are three levels of data-flow testing for a class C in an object oriented program P: intramethod testing; intermethod testing; and intraclass testing [40]. In this section we formally introduce each level.

- **Intramethod data-flow testing:** This technique tests methods individually, and is equivalent to unit testing of individual procedures in imperative languages. Intramethod data-flow testing on a class is performed by testing each method in a class separately. Computing intramethod du-chains for each method separately is analogous to computing procedural du-chains. The computation of procedural du-chains is obtained by using traditional iterative dataflow analysis methods [1]. An intramethod du-chain is a chain that starts with the definition of an instance variable and ends with its use. For example, in the class *A* of Figure 5-1, the definition of the instance variable *Y* at line 15 has uses at lines 16 and 17, respectively. Thus, the paths {15, 16} and {15, 17} are intramethod du-chains with respect to *Y*.

Formally, Let *m* be a method in a class *C* and S_d is a statement representing a definition of an instance variable of *C* and S_u a statement representing a use of that instance variable. We say that (S_d, S_u) is an intramethod du-chain in *M* if there exists a program *P* that calls *M* such that in *P*, (S_d, S_u) is exercised during a single invocation of *M*.

- **Intermethod data-flow testing:** This technique tests a public method together with other methods in its class that it calls directly or indirectly. This level of testing is equivalent to interprocedural testing in procedural languages. Computing intermethod du-chains is analogous to computing interprocedural du-chains [19][43][44]. Intermethod du-chains occur when methods within the calling context of a single public method interact, such that a definition in one method reaches across method boundaries to a use in some method called directly or indirectly, by the public method. For example, in the class *A* of Figure 5-1, when the public method *foo* is invoked, it calls both *foobar* and *barfoo*. Thus, intermethod data-flow testing of *foo* involves testing the du-chains of variables inside *foo* that reach the call sites of both *foobar*, and *barfoo*.

In the example of Figure 5-1, when *foo* is called, the definition of *Y* at line 15 is passed to the call sites of both *foobar* and *barfoo* at lines 16 and 17, respectively. Since *k*, the formal parameter in *foobar* that is bound to *Y* at the call site in line 16, is redefined in line 30, *Y*, does not reach, live, the call site to *barfoo* at line 17. Thus, the intermethod du-chains with respect to *Y* over the method calls to *foobar* and *barfoo* are: {15, 30}; and {15, 35}, respectively.

Formally, let m_i be a public method in C such that m_i is not in M and $M = <m_1, m_2, \ldots m_n>$ is the set of methods in C called directly or indirectly, when m_i is invoked. We say that (S_d, S_u) is an intermethod du-chain if (1) S_d is a statement representing a definition of an instance variable in m_i, (2) S_u is a statement representing a use of an instance variable in $m_j \in M$; and (3) there exists a program P that calls m_i which exercises (S_d, S_u) with a single invocation by P of m_i.

```
1 class A {
2 public:
3         ~A() { delete A;}
4         A (int x, int y){ X= x; Y = y;}
5         void foo (int i);
6         void bar (int j);
7         void barfoo (int k);
8         void foobar (int l);
9 private:
10        int X, Y;
11 };
12/********************/
13 void A::foo(int i) {
14     if(X > i) {
15       Y = X /2;
16        foobar(Y);
17        barfoo(-Y)
18        X ++;
19     }
20  }
21 /*******************/
22 void A::bar(int j ){
23      if( Y < j)
24      foobar(Y);
25       X--;
26      foobar(X);
27 }
28 /********************/
29 void A::foobar(int k) {
30       k = k*5;
31        prntf(k);
32 }
33 /********************/
34 void A::barfoo(int l) {
35       l = l/5*6;
36       printf(l);
37 }
38 /*********************/
```

Figure 5-1. The class and its call graph representation.

- **Intraclass data-flow testing**: This method tests the interactions of public methods when they are called in various sequences. This level is often problematic because a class is often destined to be reused in different applications, and the sequence in which its public methods are invoked and executed can be infinite. Intraclass du-chains occur when sequences of public

methods are invoked. Consider for example, in class *A*, the sequence <*foo*, *foo*>. In the first call to *foo*, if *X* is > *i*, line 18 sets *X*. In the second call to *foo*, line 14 fetches the value of *X*. Thus, {14, 18} is an intraclass du-chain. As a second example, consider the sequence <*foo*, *bar*>. The method *foo* sets the value of *X* at line 18. When bar is called, the value of *X* is fetched at line 25. Thus {18, 25} is a intraclass du-chain with regards to *X*.

Formally, Let m_i be a method in *C*, and *M* the set of methods that are called directly or indirectly when m_i is invoked. Let n_j be a public method in *C* (possibly $i = j$), and *N* the set of methods that are called directly or indirectly when m_j is invoked. We say that (S_d, S_u) is an intraclass du-chain if (1) S_d is a statement containing a definition in the set $\Delta = \{m_i, M\}$; (2) S_u is a statement that contains a use in the set $\Delta' = \{m_j, N\}$; (3) there exists a program *P* that calls m_i and m_j, such that (S_d, S_u) is a du-chain in *P*; and (4) S_d is executed, and before S_u is reached, the call to m_i terminates.

Research on code-based interclass testing has always focused on extracting method sequences. One approach by Parrish and Boir [72] constructs a graph *G* from a class *C* such that each public method *m* in *C* is represented with a node *n* in *G*. An edge is constructed from node n_i representing a method m_i to a node n_j representing a method m_j, if the user can invoke m_i followed by m_j. If m_i sets a condition *c* such that when that condition is satisfied, m_j may be invoked, then the control edge $(n_i\ n_j)$ is constructed with a *true* label, else it is a control edge with a *false* label. Given that graph *G*, test suites can be desiged to cover all nodes and/or edges. The technique provided by [72] specifies sequences of methods; however, it does not require any particular interclass code-based coverage to be applied to those sequences. For example, after a specific sequence of methods have been revealed, dataflow interactions resulting from those sequnences are not required to be covered.

Another code-based sequence selection technique focused on revealing and testing dataflow interactions that resulted at the intraclass level [40]. This technique treats a class *C* as a single entry single exit program, and generates a Class Call Garpgh (*CCG*) for *C*. The *CCG* is a directed graph whose nodes represent methods, and solid edges represent procedure calls between methods. To illustrate how the *CCG* for a class is constructed, consider the class *A* in

Figure 5-2. In the latter, since *foo* calls *foobar* and *barfoo,* edges are constructed from the node representing *foo* to the nodes representing *foobar* and *barfoo,* respectively. Figure 5-2 also depicts edges, shown as dashed lines, ending at the constructor, destructor, and each public method. The dashed edges represent messages sent to these public methods from outside the class.

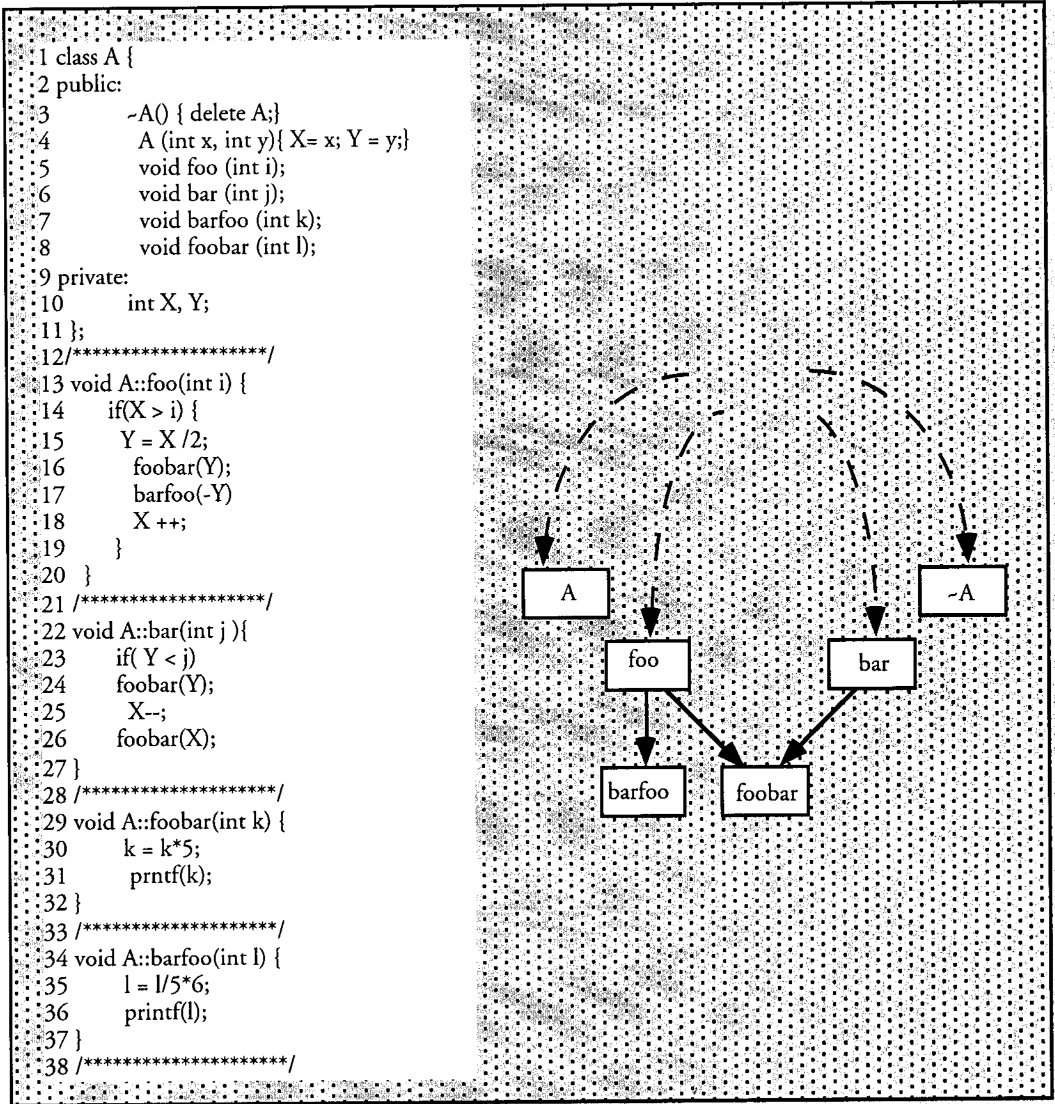

Figure 5-2. A class A (left) and its Class Call Graph (right).

To identify all the possible method sequences in a *CCG,* the *CCG* is enclosed in a *frame* to generate the method sequences. A frame is a driver for the public methods or an abstraction of a main program *P* in which calls to public methods are selected randomly by a switch statement

S. For each iteration, a new sequence of calls is generated. As depicted in Figure 5-3, a frame contains five nodes: frame entry and frame exit, which represent entry and exit from the frame; frame loop, which facilitates sequencing of methods; and frame call and frame return nodes, which represent the call to end and return from any public method, respectively. A frame also contains four edges: (frame entry, frame exit), (frame loop, frame call), (frame loop, frame exit), and (frame return, frame loop). Once the frame nodes and edges are constructed, each node *n* is replaced with its control-flow graph or *CFG*. The result is a class control flow graph (CCFG). For each new sequence generated, relevant du-chains can be collected by applying interprocedural data-flow analysis such as the ones found in [43][44].

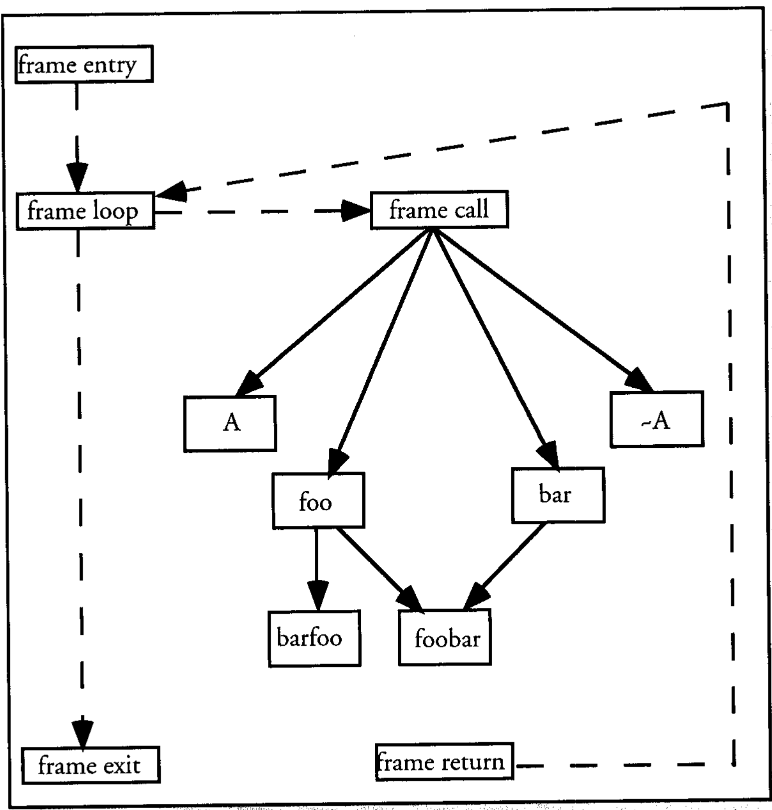

Figure 5-3. The representation of a Class Call Graph enclosed in a frame.

5.2.1 Other Issues in Data-flow Testing for Text-based OOP Languages.

Dynamic binding and complex inheritance structures create many opportunities for faults due to unanticipated bindings or misinterpretation of correct usage. Static analysis of source code

to identify paths - a common bedrock technique in procedural testing - is of little use when determining, for example, du-chains resulting from dynamic bindings .

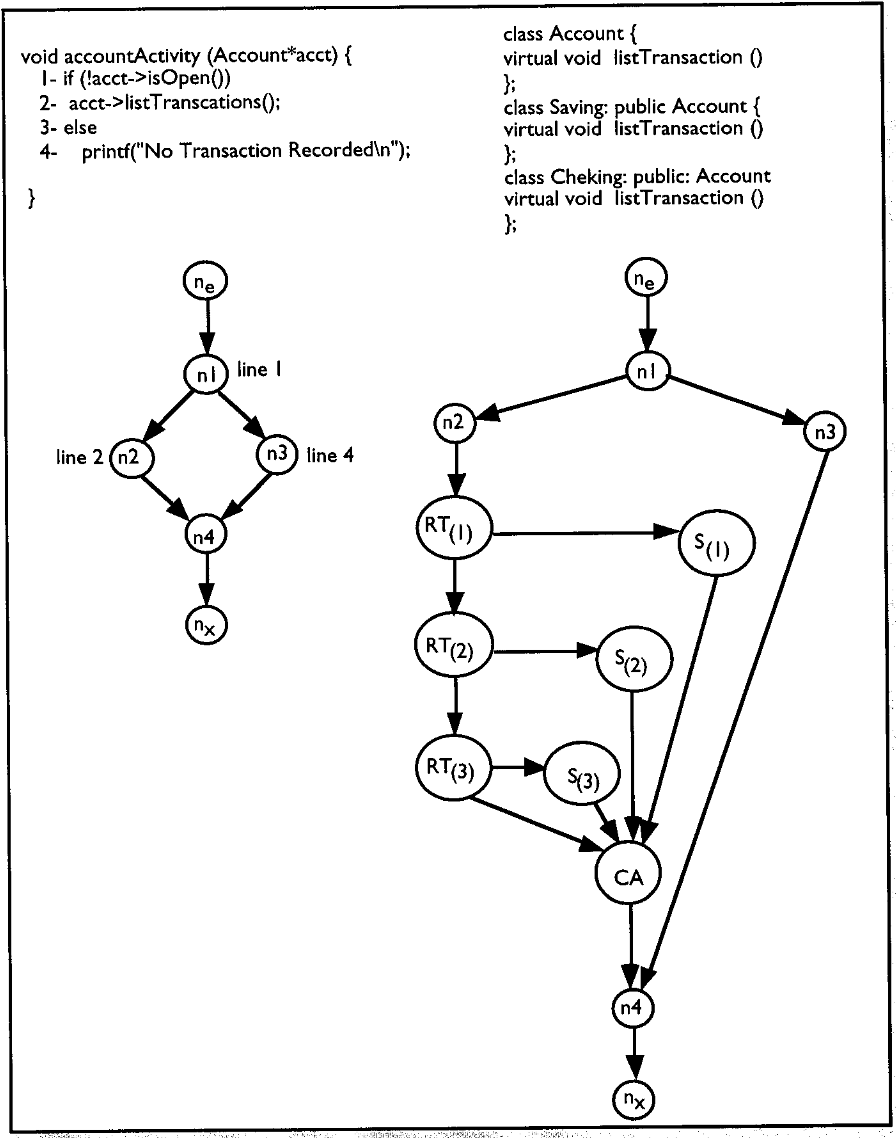

Figure 5-4. An example of a message and a polymorphic server.

- **Polymorphic Bindings**: Although a polymorphic message is a single statement, it is an interface to many different methods. Polymorphism and dynamic binding dramatically increase the number of execution paths. It is argued that exercising data-flow interactions that result from a single binding of a polymorphic server is "insufficient" [12]. The coverage can be considered complete when all the du-chains that result from all possible bindings have been

exercised. With larger polymorphic servers; however, test cases could be difficult to identify [12]. To test every possible binding, each polymorphic message must be exersised at least once. Thus, we expand the *CFG* of a method, and : (1) determine the number Δ of candidate bindings for each message; (2) expand the node containing the polymorphic message into Δ-way branch graph; and (3) add a node which represents the run-time binding or *RT*, a sequential node or *S* which represents the message send and return, and a Catch-All node or *CA* which represents run-time binding error; and (4) construct en edge from each $RT_{(i)}$ to each $RT_{(i+1)}$, and an edge from each $RT_{(i)}$ to each $S_{(i)}$, and finally an edge from each $S_{(i)}$ to *AC*, for $1 \leq i \leq \Delta$.

To illustrate, consider the example of Figure 5-4. In that example, the class hierarchy that contains the class *Account* and its children *Saving* and *Checking*, is a polymorphic serve since each level in the hierarchy contains a virtual method named *listTransaction.* The method named *acountActivity* is a polymorphic client because it contains a call to *listTransaction* which can bind at run time to the virtual *listTransaction* method that exists in each class of the hierarchy. Thus the *CFG* for the example of Figure 5-4 is expanded as follows: in (1) we determine that Δ = 3; in (2) we expand *n2* into 3-way branch graph; in (3) we add three RT nodes and three S nodes and an AC node; and finally in (4) we construct the appropriate edges. Using the expanded graph of Figure 5-4, dataflow testing techniques can be applied to adequatly test the data-flow interactions that result from dynamic bindings.

- **Inheritance and Incremental Class Testing.** The scope of retesting under inheritance established by Perry [75] is widely, if sometimes reluctantly, accepted [12]. Perry's result requires complete testing of derived classes. One technique to reduce the number of method-specific tests required for a derived classes was introduced by [41]. This technique suggests that a minimal set of inherited C++ derived class features be automatically determined and tested. A test suite is applied to each base class, then each derived class is flattened. Required test cases for derived classes are determined by incrementally updating the class testing history of the parent. Only new attributes or those inherited, affected attributes and their interactions are tested. The benefit of this technique is that it provides a saving both in time to analyze the class to determine what must be tested and in the time to execute test cases.

Another technique to reduce the cost of testing derived classes is to use incremental data-flow analysis algorithms to update du-chain information in derived classes.

5.3 The Object-flow in Prograph

As mentioned in Chapter 2, Prograph is an object oriented language with facilities for defining new datatypes as classes. Classes may contain methods of the same name, and a method may invoke different methods at different times during execution. Prograph has a single inheritance model. Methods and their attributes are by default public. There is no explicit way of enforcing data hiding. The visual code is organized around Sections as depicted in Figure 5-5. Each Section contains three parts: Classes; Universals; and Persistents.

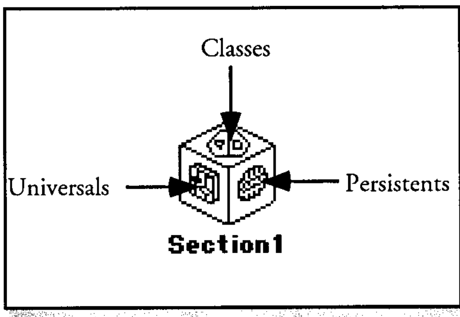

Figure 5-5. A pictorial representation of the Section icon in Prograph.

As depicted in Figure 5-5, the Classes part is where classes (attributes or states and methods or behavior) are declared. This information is visually encapsulated into the icon which is used for a class declaration. The Universals part is the where *universal* or procedural methods are defined, and the Persistents part is where *persistent* or global variables are defined. In this wrok we do not conider Persistents or global variables.

5.4 Testing Variable Interactions in Visual Object-flow Languages

We have thus far ignored the side effect associated with the execution order of data flow languages. With the presence of visual objects however, we have to address such a side effect and

its consequences. In visual object-flow languages such as Prograph, *get* and *set* methods are used to read and write to instance variables memory locations. When the order of execution between two operations that write to memory locations is not determined by neither the data nor the control dependency, two orders of executions are possible. Since each order could produce different results, we say that there is a potential error (not an actual error) present in the visual code. To deal with the side effect of such potential error, our testing environment should draw the user/tester's attention to enforce a *synchro* between any two *set* operations whose order of execution is not determined by neither the data nor the control dependency. To illustrate, consider the example in Figure 5-6. In that example, the operation labeled *n10* could execute before that of *n5* or vice versa. Each execution possibility can result in a different output.

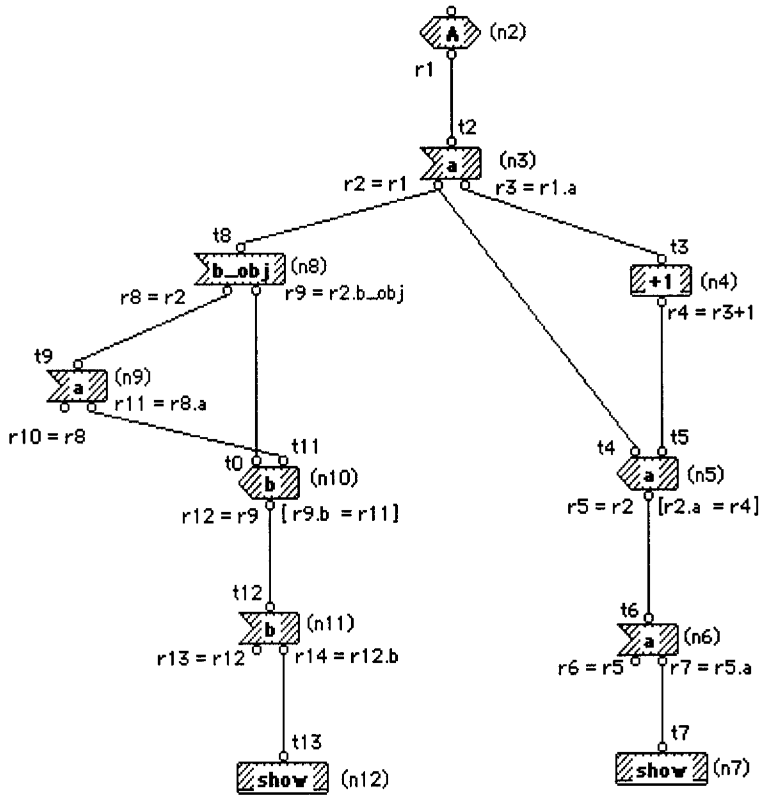

Figure 5-6. An example of get and set methods.

To solve the problem associated with the above mentioned side effect, we place a *Guard* that informs the user to enforce a *synchro* between any sequence of two operations *S* in Δ = {<*get*,

set>; <*set*, *set*>; <*set*, *get*>} such that, the order of execution between the elements of *S* is set by neither the data nor the control dependency, and *S's* elements are associated with the same attribute of an object *O*. Relying on the information provided by the editing environment, the *Guard* can determine the presence of a Δ in the visual code. Once found, the *Guard* will first highlight the operations corresponding to the elements of Δ, and then displays a dialog box indicating that testing cannot begin before a *synchro* is enforced between the operations. As illustrated in Figure 5-7, a *synchro* is enforced between *n5* and *n7* (*n10* in Figure 5-7). This *synchro* indicates that the order of execution has been adequately determined, and therefore will eliminate the potential error described earlier.

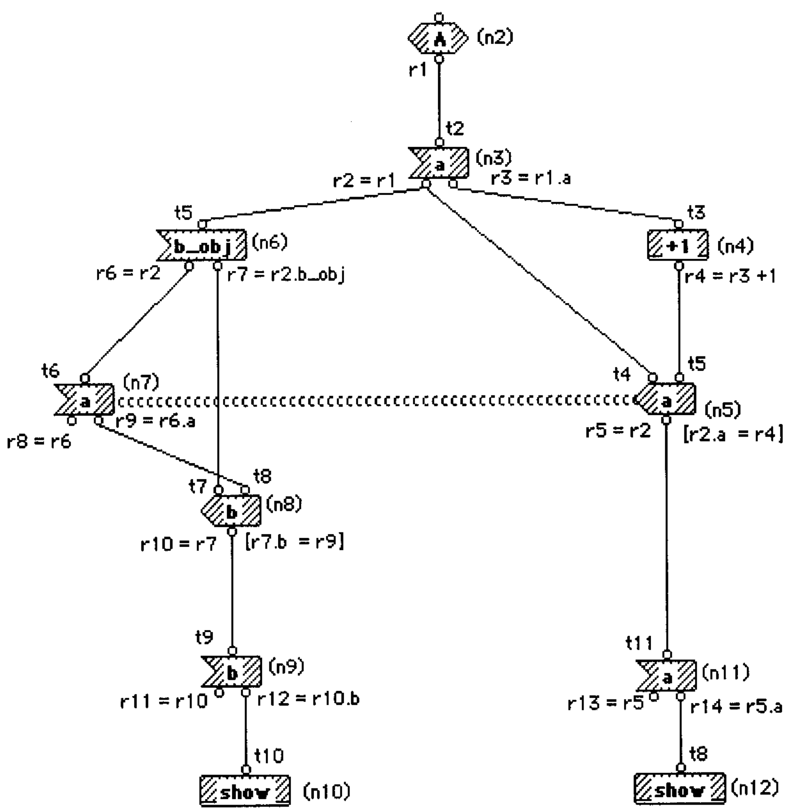

Figure 5-7. The example of Figure 5-6 with a synchro enforced between n5 and n8.

A *get* method has two *roots* and one *terminal*, while a *set* method has two *terminals* and one *root*. The right-hand side *root* of a *get* method g_1 references the appropriate attribute of the object that is being passed to g_1*'s terminal*. The left-hand side *root* of g_1 references the object that is being passed to g_1*'s terminal*. The *root* of a *set* method s_1 references the object that is being

passed to s_1's left-hand side *terminal*. Also at s_1, there is an implicit assignment that assigns the attribute indicated by s_1 to the value that is being passed to s_1's right-hand side *terminal*.

Formally, given a class C in a visual object-flow program P, such that m is any method $\in C$, *and* G and S are the sets of *get* and *set* operations, respectively in m. A *get* operation $g_1 \in m$ declares on its left-hand side *root* a reference variable and assigns it to the *root* connected to g_1's *terminal*. For example, in Figure 5-7, the statement "r2 = r1" at the *get* operation (labeled *n3*) declares *r2* as a reference parameter to *r1*. The right-hand side *root* of g_1 declares a reference variable and assigns it to the *root* connected to g_1's *terminal*. For example, in Figure 5-7, the statement "r14 = r5.a" at the *get* operation (labeled *n11*) declares *r14* as a reference parameter to *r5.a*. Similarly, the statement "r7 = r2.b_obj" at the operation labeled *n6*, declares *r7* as a reference parameter to *r2*.b_*obj*.

A *set* method $s_1 \in m$ declares on its *root* a reference variable and assigns it to the *root* connected to s_1's left-hand side *terminal*. For example, in Figure 5-7, the statement "r5 = r2" at the operation labeled *n5*, declares *r5* as a reference parameter to *r2*. At each *set* operation $s_1 \in m$, there is also an implicit assignment. This implicit assignment assigns the appropriate attribute of the *root* connected to the left-hand side *terminal* of s_1 to the *root* that is connected to the left-hand side *terminal* of s_1. For example, in Figure 5-7, the statement "r2.a = r4" at the operation labeled *n5* is a an example of such an implicit statement.

References created by *set* and *get* methods that belong to the same object *o* provide the user with more flexibility when choosing a *root* (or reference to *o*) from which to initiate a datalink. Furthermore, the flexibility can be effective in reducing the "spaghetti-like" datalinks in the visual code. To illustrate, consider the example in Figure 5-7. In that example, *r2*, *r5*, *r6*, *r8*, and *r13* all reference *r1*. Thus, any operation where either of these referecens is being passed, can be replaced by a datalink that is initiated from *r1* at *n2*. This is depicted in the example of Figure 5-8 which provides an alternative way of visually coding the example of Figure 5-7. Note that a synchro has been added between *n5* and *n11* since (1) these operations write and read the memory location associated with *r1.a*, respectively, and (2) the order of execution

between *n5* and *n11* is set by neither the data nor the control dependency. Similarly, a *synchro* is added between *n8* and *n9*.

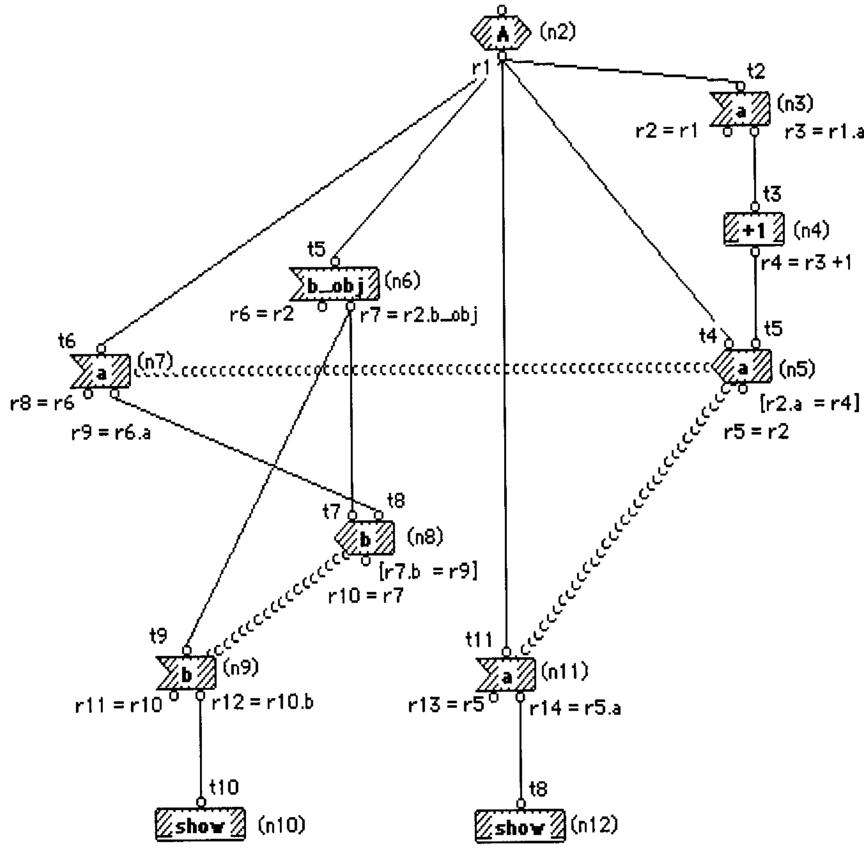

Figure 5-8. The alternative way of visually coding the example of Figure 5-7.

To identify all the references, created by the *get* and *set* operations, and belonging to the same object, we iteratively scan the data structure that hosts the visual operations, in search of *set* and *get* operations, if we encounter a *get* method $g_1 \in m$, we (a) remove the assignment that exists on g_1*'s* left-hand side *root*, and store only the name of the *root* that is connected to g_1*'s terminal*, and (b), we remove the assignment that exists on g_1*'s* right-hand side *root*, and store only the name of the *root* that is connected to g_1*'s terminal* and its appropriate attribute. For example, in Figure 5-9 (visual code originally depicted in Figure 5-7), the assignment statements at *n6* now reads: "r1" at the left-hand side *root*, and "r1.b_obj" at the right-hand side *root*. Similarly, the assignment statements at *n3* now reads: "r1" at the left-hand side *root*, and "r1.a" at the right-hand side *root*.

If we encounter a *set* operation $s_1 \in m$, we (a) remove the assignment that exists on s_1's *root*, and store only the name of the *root* that is connected to g_1's *terminal*, and (b) change the implicit assignment statement that exists at s_1 by replacing the name of the *root* on the right-hand side of implicit assignment with the name of the *root* that was replaced in (a). For example, in Figure 5-9 (visual code originally depicted in Figure 5-7), the variable shown at the *n5*'s *root* now reads: "r1" and the implicit statement reads: "r1.a = r4". Similarly, the *root* and implicit assignments at *n8* now read: "r1.b_obj" and "r1.b_obj.b = r1.a", respectively.

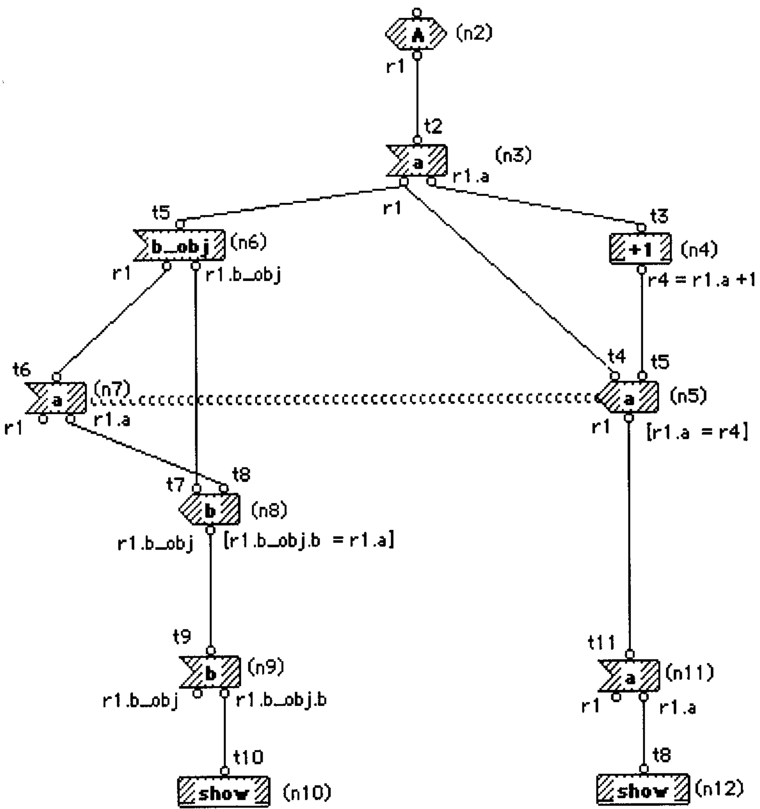

Figure 5-9. Modified labels after the scan and replace technique.

5.4.1 Definition-use Association for Object-flow Languages

For instance variable attributes in visual object-flow languages such as Prograph, there are two basic memory accesses: (1) define action and (2) use action. A define action occurs in operations that implicitly changes the concrete state of an instance variable. For example, in Figure 5-9, the implicit statement "r1.a = r4" at *n5* changes the value of memory location asso-

ciated with the attribute *r1.a*, and subsequently changes the concrete state of *r1*. A use action occurs in an operation that fetches the value from the memory location associated with an attribute without changing the concrete state of its instance variable. For example, in Figure 5-9, the statement "r4 = r1.a+1" at *n4* reads the value of *r1.a* without changing the concrete state of *r1*.

As with dataflow languages, there are two types of uses in visual object-flow languages: *c-use* or computational use and *p-use* or predicate use. A c-use occurs when the value of an attribute is used at a non control annotated operation. For example, in Figure 5-9, the attribute "r1.a" that is defined in *n5* is c-used in *n8*. Note that there is no direct datalink between *n5* and *n8*. This type of assignment in visual object-flow languages is possible via indirect datalinks. In Figure 5-9, the datalinks between *n5*, *n3*, *n6*, and *n8* are an example of an indirect datalink. A p-use occurs when the value of an attribute is used at a control annotated operation. For example, in Figure 5-10, the *root* labeled *i.b_obj* is p-used in the *primitive* operation *"instance?"* in *n4*.

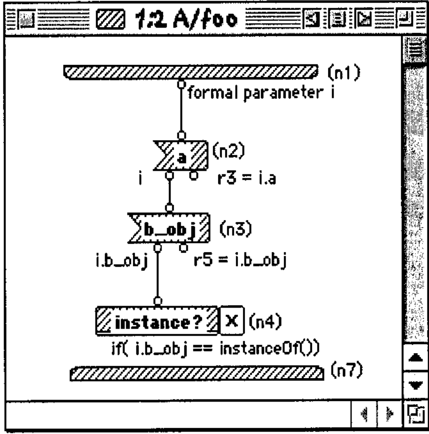

Figure 5-10. A p-use of attributes in visual object-flow languages such as Prograph.

Formally, given a class *C* in a visual object-flow program *P* with any instance variable *v* of *C*, let $O = \{o_1, o_2, \ldots, o_n\}$ be the set of operations in a method $m \in C$, and *N* the set of blocks in an *OCG* representing *O* of *m*. A def-c-use association in visual object-flow languages with regards to an attribute *a* of *v* is a triple $((n_i, n_j, (v.a))$, such that, n_i and n_j are nodes or blocks in *N* representing operations $o_i \in O$ and $o_j \in O$ respectively, there is either a direct or indirect datalink between o_j, and o_i, o_j is a non-control annotated operation, and there exists an assignment of

values to p's input, in which n_i reaches n_j. An example of a direct datalink between the operation where a definition occurs and the operation where it is c-used is illustrated in the example of Figure 5-8. In that example there is a direct datalink between the definition of *r1.a* in *n2* and its c-use in *n5*. An example of an indirect datalink between the operation where a definition occurs and the operation where it is c-used is illustrated in the example of Figure 5-9. In that example there is an indirect datalink (through *n1*) between *r1.a* in *n5* and its c-use in in *n8*.

A def-p-use association in visual object-flow languages with regards to an attribute a of v is a triple $((n_i, (n_j, n_k), (v.a))$, such that, n_i, n_j, and n_k are nodes or blocks in N representing operations o_i, o_j, and o_k in O respectively, there is either a direct or indirect datalink between o_j and o_i, o_j is a control annotated operation, and there exists an assignment of values to P's input, in which n_i reaches n_j, and causes the predicate associated with n_j to be evaluated such that n_k is the next node to be reached.

5.5 Data-flow Testing of Classes in Dataflow Languages

Given a visual object-flow program P, there are three levels of data-flow testing for a class C in P: intramethod testing; intermethod testing; and intraclass testing. In this section we formally introduce each level.

5.5.1 Intramethod Data-flow Testing

This technique test methods individually, and is equivalent to unit testing of an individual procedure in visual dataflow languages. Let m be an object-flow method in a class C such that O_d is an operation representing an implicit definition of an instance variable v of C and O_u an operation representing a use of that instance variable. We say that (O_d, O_u) is an intramethod du-pair in M if there exists a program P that calls M such that in P, (O_d, O_u) is exercised during a single invocation of M. We apply intramethod data-flow testing to each method m in a class C, by computing m's intramethod du-chains, and testing those du-chains. Next we discuss the process of collect the static du-associations in visual object-flow languages.

5.5.1.1 Collecting the Intramethod Static du-associations

For imperative languages, a wide range of analysis techniques for computing du-chains for individual procedures are well known Aho [1] and have been used in various tools, including data-flow testers introduced by: Frankl and Weyuker [33][34]; Harrold and Soffa [44]; and Korel and Laski [57]. These techniques propagate definitions along control flow paths to conservatively identify du-associations before they encounter a redefinition. To collect the static du -associations of instance variables attributes in visual object-flow languages, we adapt one of those aformentioned techniques. That is, intramethod du-chains associated with instance variables attributes will be collected in a way that is analogous to that of collecting variables du-chains in imperative languages. Formally, let *C* be a class with an instance variable *v*, let $O = \{o_1, o_2,..., o_n\}$ be the set of operations in a method *m* in *C*. Also, let *N* be the set of blocks or nodes in an *OCG* representing *O* in *m*. For every node $n \in N$, we create the following sets: (1) DEF[*n*], the set representing the locally generated *v's* attributes of *n*; (2) USE[*n*], the set of locally used *v*'s attributes in *n;* In[*n*], the set of *v*'s attribute definitions that reach the beginning of *n*; and (4) OUT[*n*], the set of *v's* attribute that are alive at the end of *n*. Once these sets are computed for each node *n*, the well known [1] dataflow equations for calculating du-chains, can be used on these sets to obtain the intraprocedural du-chains *m* in *C*.

5.5.1.2 Visually Representing Executed du-associations

To communicate the testing result in a way that complements both the environment of visual object flow languages, and the nature of its du-associations, we represent the validated results in two different ways. After a test suite is executed, results in the form of the percentage of exercised du-chains is inserted under the method name. The second method is similar to what we have introduced for du-association coverage for visual dataflow languages. To reflect the testedness of attribute's du-association, we first determine whether there is a direct datalink between the operation where the definition occurs and the operation where it is used. When a direct datalink that is associated with a definition c-use exists, it is colored green if exercised or red otherwise. If the direct datalink is associated with a p-use it is colored green when both outcome of the predicate are exercised, half green and half red when one of outcome is exercised,

and red otherwise. For those du-associations that are not represented with direct datalinks, we construct links from the operation where the attribute is defined to the operation where it is used. The validation process of constructed links is similar to that of datalinks.

5.5.2 Intermethod Testing

This level of testing is equivalent to interprocedural testing in procedural languages. It tests a public method together with other methods in its class that it calls directly or indirectly. An intermethod du-chain occurs when methods within the calling context of a single public method interact, such that a definition in one method reaches across method boundaries to a use in some method called, directly or indirectly, by the public method. To illustrate, consider the example in Figure 5-11. In that example, method *m* in class *A* calls method *n* in *A*. The instance variable *r1.i* that is defined in the operation labeled *n2* has a use in method *m* at the operation labeled *n4*, and also a use in method *n* at the operation labeled *n9*. The static data-flow analysis of intermethod du-chains associated with simple variables can be computed using the algorithm we have developed in Chapter Four for computing the interprocedural data-flow analysis for visual dataflow languages. On the other hand, intermethod static data-flow analysis associated with instance variables can be computed by adapting techniques developed for computing code-based intermethod static data-flow analysis.

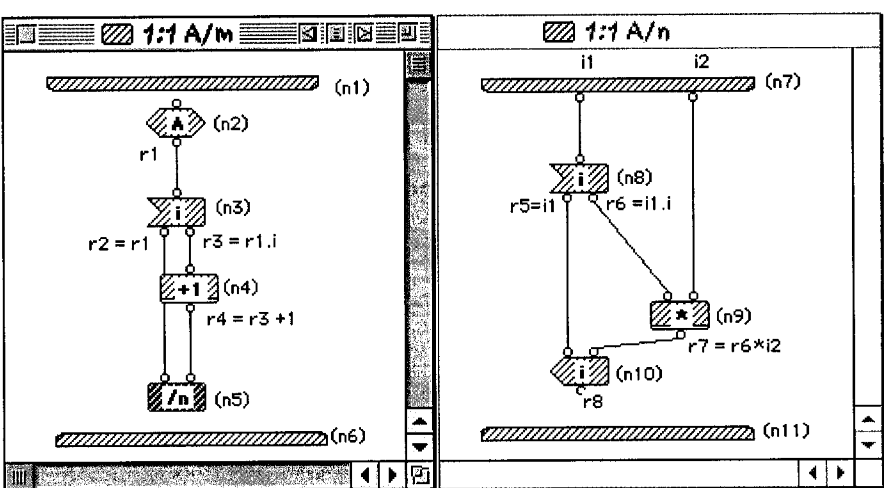

Figure 5-11. Intermethod du-associations in Prograph.

Formally, let m_i be a visual object-flow method in a class C such that m_i is not in M, and $M = <m_1, m_2, \ldots, m_n>$ is the set of public methods in C called directly or indirectly, when m_i is invoked. We say that (O_d, O_u) is an intermethod du-chain if (1) O_d is an operation that defines an instance variable v of C in m_i, (2) O_u is an operation that uses the memory location of v in $m_j \in M$, (3) there exists a program P that calls m_i which exercises (O_d, O_u) with a single invocation by P of m_i.

5.5.3 Intraclass Testing

This method tests the interactions of public methods when they are called in various sequences. This level is often problematic because a class is often destined to be reused in different applications, and the sequence in which its public method call sequences are invoked and executed can be infinite. In general, Let m_i be a method in C such that m_i is not in M and $M = <m_1, m_2, \ldots, m_k>$, $k > 1$ is the set of public methods in C called directly or indirectly, when m_i is invoked. Let n_j be a public method in C (possibly the same method as m_i), and let $<n_1, n_2, \ldots, n_l>$, $l > 1$ be the set of methods in C called, directly or indirectly, when m_j is invoked. Suppose d is in some method in $<m_1, m_2, \ldots, m_k>$, and u is in some method in $<n_1, n_2, \ldots, n_l>$, such that in P, (O_d, O_u) is a def-use pair, and such that after O_d is executed and before O_u is executed, the call to m_i terminates, then (O_d, O_u) is an inter-class def-use pair.

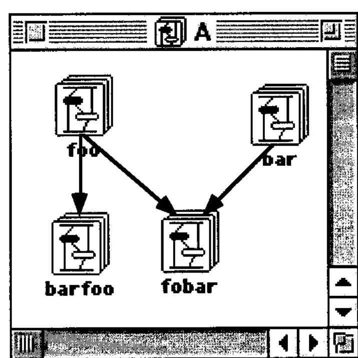

Figure 5-12. The visual representation of a Class Calling Graph.

To test this level of data-flow interaction in dataflow class A, we could build its Class Calling Graph or CCG, and, analogous to the approach described at the end of Section 5.2 and illus-

trated in Figure 5-3, enclose it in a Frame to test as many sequences as possible. To visually represent the *CCG*, we could provide a view in the window containing all the methods in a class with arrows between the icons representing the methods. Each arrow represents the edges between two methods in the *CCG*. To illustrate, consider the example in Figure 5-12. In that example, the *CCG* of the Prograph representation of the example of class *A* in Figure 5-2. The benefit of such view could be to aid the tester to visually keep track of tested sequences.

5.5.4 Polymorphic Testing

Data-flow testing of a polymorphic visual object-flow client method can be accomplished in a way analogous to that of code-based polymorphic client methods. For example in Prograph, every *OCG* representing a method that contains a call to a polymorphic server and can be bound to a number of methods in that server, can be expanded in a similar way to that of the *listTranscation* polymorphic client method of Figure 5-4. To Visually reflect the testedness of each possible binding, we add to the operation that represents the call to the polymorphic server arrows coming out of it. Each arrow represent a binding edge. As depicted in Figure 5-12, the *listTransaction* method in Prograph contains three arrows that represent each polymorphic binding. When these edges are traversed under a certain structural-based testing criterion, they are colored green, and red otherwise. This visual feedback helps the user keep track of which binding did or did not take place.

5.6 Summary and New Directions

In this chapter, we investigated, from a data-flow testing perspective, differences between code-based object oriented languages and visual object-flow languages in the context of Prograph. Our findings revealed that, analogous to code-based object oriented languages, there are three levels of testing the instance variable data-flow interactions in a Prograph class: intramethod, intermethod, and intraclass data-flow testing. In each level we showed how code-based data-flow testing techniques can be adapted to collect that level's appropriate du-chains.

In Chapters Three, Four and Five, we have described issues in, and strategies for, testing visual dataflow and visual object-flow programs in the context of Prograph. We presented test adequacy criteria that are based on both the control-flow and data-flow of Prograph programs. We have shown, by example, that the testing techniques we have presented can help the visual language community in developing a more trusted visual code. In Chapter Six, we introduce our future work which will shed some light on how to visualize the testedness of both the control and data-flow in imperative programs.

6 Concluding Remarks

6.1 Chapters Summary

Many code-based structural testing techniques have been used, published, and analyzed since the late 1960s. These techniques call for testing parts of an implementation that must be exercised to satisfy a particular adequacy criteria. Control-flow and data-flow test adequacy criteria are both structural-based, and have been applied to imperative, declarative, and logical languages. For visual dataflow languages however, we did not find any structural-based testing techniques. In visual dataflow languages, a program is represented as a directed graph where nodes represent operations, and datalinks represent edges through which data flow. Programming in the visual dataflow paradigm involves assembling operation elements that send and receive data. A visual dataflow language simply specifies data dependencies, and an operation is executed when all of its input data become available. This model of execution is known as the dataflow computational model. To allow the user to explicitly add control constructs such as those found in imperative languages, dataflow languages extended the pure dataflow model to include the necessary control constructs. Thus, a dataflow program can be characterized by both its data and control dependencies. Prograph is a representative of both commercial and research visual dataflow languages, and has been used for the development of a number of commercial applications. The lack of testing tools or structural-based testing techniques for visual dataflow languages, and our desire to provide dataflow programmers with a methodology to test their visual code, were the driving force behind the initial research direction of this work. In pursuit of that, we investigated, from structural testing perspectives, differences between imperative and dataflow languages. Our investigations revealed opportunities to adapt code-based structural testing to test dataflow languages. Based on our adaptation of code-based structural testing, we have developed an integrated Dataflow Testing Tool *DFTT* and used it to visually test dataflow programs. We provided a truly visual testing and validating environ-

ment in the *DFTT* by using visual annotations to reflect, based on a certain testing criteria, the testedness of an operation or a datalink in a dataflow program. The use of visual annotations was first introduced by the WYSIWYT methodology developed to reflect the testedness of cells in form-based programs [16]. Although form-based languages have many similarities to the visual dataflow paradigm, there are some differences, at least as the paradigm exists in Prograph. We have identified these differences. One difference that we have pointed out is that Prograph is not responsive. That is, results stemming from changes made to the visual code are not automatically evaluated and displayed. Some other differences pointed out were Prograph's handling device on *success/failure*, the case-by-case execution device, and the support for loops. Prograph also has side effects; however, this is not in the subset supported by the methodology described in the first part of this work. In the considered subset, we represented a dataflow program *P* with an Operation Case Graph or *OCG* that preserved both *P*'s data and control dependencies. Each block or node *n* in an *OCG* represents an operation *o* in *P*. Edges in an *OCG* represent, based on the control constructs, the control flow in *P*. To test the control-flow in *P*, we applied code-based control-flow testing criteria, such as All-nodes and All-branches, to the *OCG*. We then built upon the WYSIWYT methodology an approach to reflect, according to a certain control-flow testing criterion, the testedness of visual constructs in dataflow languages. For example, when applying the All-branches criterion, a control annotated operation *o* is colored green if both edges emanating from *n*, the node in the *OCG* representing *o*, are executed, half green and half red if one edge is tested, and red otherwise.

As with imperative languages, we recognized two data-flow testing levels in visual dataflow languages: intraprocedural data-flow testing; and interprocedural data-flow testing. To compute and test intraprocedural du-associations, we treated every *root* r in an operation o represented with a block n as a variable definition and every *terminal* t as a variable use. We defined a datalink between r and t as a definition-use association or du-association, and recognized two uses in visual dataflow languages: computational uses or c-uses; and predicate uses or p-uses. A c-use consisted of an operation o_1 connected to an operation o_2 such that the latter is not control annotated. On the other hand, a p-use consisted of an operation o_1 connected to control annotated operation o_2. A du-association in a visual dataflow program links a *root* with *terminals* that the *root* or definition can reach. We considered two types: a definition-c-use associa-

tion; and definition-p-use association. We recognized that, apart from loop-related *roots* which are always redefined at the Output Bar of a looped *case* and whose du-association can be statically resolved, a *root* or variable in visual dataflow languages cannot be redefined. This means that the memory location associated with that variable does not change. Thus, to collect static du-associations of a visual dataflow program *P*, we developed a technique that relied on the information collected and maintained by the editing environment. Other techniques such as the traditional data flow analysis techniques could also have been adapted; however, these techniques would have performed redundant and unnecessary computations. To reflect the testedness of a datalink that corresponds to a def-p-use association, our testing environment tracks the datalink in the visual code and colors it green if both outcomes of the evaluated predicate are traversed, half green and half red if only one outcome of the evaluated predicate is traversed, and red otherwise.

To compute and test interprocedural du-associations among interacting procedures for a visual dataflow program *P*, we processed each procedure *p* in any order, and extracted du-associations related to formal and actual parameters. The collected du-associations information corresponding to each procedure was then used to construct an Interprocedural Operation Case Graph for *P* or $IOCG_{(P)}$. The $IOCG_{(P)}$ is a collection of connected sub-graphs each of which represents a procedure or $IOCG_{(p)}$ in *P*. For each control point (procedure entry, exit, call, and return) in an $IOCG_{(p)}$, we computed the appropriate DEF and UPCON sets. We then adapted a code-based two-phase propagation technique to propagate these sets via the $IOCG_{(P)}$ to obtain, for each definition, its interprocedural uses. Our technique supports separate compilation, while taking into account the calling context of called procedures. It also provide safe analysis in the presence of aliases. Extending our technique to compute interprocedural definition-use chains for recursive procedures is part of our future work. To reflect an untested interprocedural du-chain, we construct a link from the *root* on operation in the calling procedure to the *terminal* on the operation in the called procedure, and color it red.

For visual object-flow languages, we investigate from a data-flow testing perspective, differences between code-based object oriented languages and visual object-flow languages in the context of Prograph. Our findings revealed that, analogous to code-based object oriented lan-

guages, there are three levels of testing the instance variable data-flow interactions in a Prograph's class: intramethod, intermethod, and intraclass data-flow testing. In each level we showed how code-based data-flow testing techniques can be adapted to collect that level's appropriate du-chains. For example, we showed how code-based intraprocedural testing techniques can be used to collect intramethod du-chains for instance variables in visual object-flow languages such as Prograph. We also showed how code-based interprocedural data-flow analysis techniques can also be adapted to collected the intermethod du-chains of instance variables in Prograph.

We have demonstrated, by example, the effectiveness of our intraprocedural and interprocedural data-flow testing methodologies and their role in providing the tester with a mechanism to locate errors in the visual code. With visual dataflow languages, our intraprocedural data-flow testing methodology was found to be particularly useful in uncovering faults related to untraversed du-chains of looped *local* indexes. Similarly, our interprocedural data-flow testing methodology was found to be particularly effective in catching interface errors. In visual object-flow languages such as Prograph, we showed how all three levels of data-flow testing can be accomplished for a Prograph class. As to the empirical evaluation of each level, we are in the process of extending our *DFTT* to enable us to empirically evaluate its effectiveness in catching coding errors in Prograph's object oriented programs.

6.2 Visual-based Testing of Imperative Languages

Existing Integrated Development Environments (IDEs) such as those from Metrowerks® and Borland®, provide integrated tools with visual features for the development and debugging of the source code; however, they do not provide any visual or non-visual structural-based testing capabilities. Structural-based testing performed on code developed on existing *IDEs* is normally accomplished using third party software testing tools. These structural-based testing tools are either source-level testers [33][34] or compiler-based testers [38]. Source-level testers translate a program under test to some intermediate form. On the intermediate form, control-flow and data-flow analysis are then performed to collect the required testing information. Such tools duplicate a large part of the computation that is usually performed by a compiler.

Aside from performing redundant tasks, the usefulness of a source-level testing tool is limited to testing programs written in the language for which it is developed. Compiler-based testing tools use existing compilers to translate the source code to an intermediate form. Since the information required for data-flow testing is quite similar to that required by compiler optimization, compiler-based testers make use of the data-flow information gathered by the compiler. Using a complier-based testing technique can thus (1) avoid costly translation and analysis required to compute control-flow and data-flow information, and (2) be extended to test other languages following only moderate changes [46]. Regardless of a tester type in use – source-level or compiler-based – testing results are normally communicated back to the tester in the form of a log file. Typically, a log file is a separate entity that makes finding the location of the construct(s) or variable(s) in the source code responsible for a reported error a time consuming task. Some attempt was taken to bridge the gap between the log file generated by the testing tools and the source code. An example of such an attempt is Combat [46]. Combat is a compiler - based testing tool that visually represented the control-flow in a *CFG*. Each node in a *CFG* was represented with a box containing the block number and its corresponding source code statements. Edges in the *CFG* were represented with arrows between blocks. Combat did not visually represent a program's control-flow information when testing. Furthermore, Combat did not represent the data-flow information of a program.

In this section we describe our new research direction which is geared towards providing *IDE* users with a compiler-based visual testing tools that can (1) represent both the data-flow and control-flow in a *CFG*; (2) reflect, based on a certain testing criterion, the testedness of parts of the implementation; and (3) help testers quickly identify and locate errors in the source code based on visual reflections.

6.2.1 Integrated Testing and Visual Validating Environment

Since it is fair to assume that most commercial applications are developed using *IDEs*, it would be of great benefit to allow the developer/tester to empirically and visually validate programs from within those *IDEs*. As shown in Figure 6-1, our proposed Integrated Testing and Visual Validating Environment (ITVVE) is a compiler-based system that is added to an existing *IDE*.

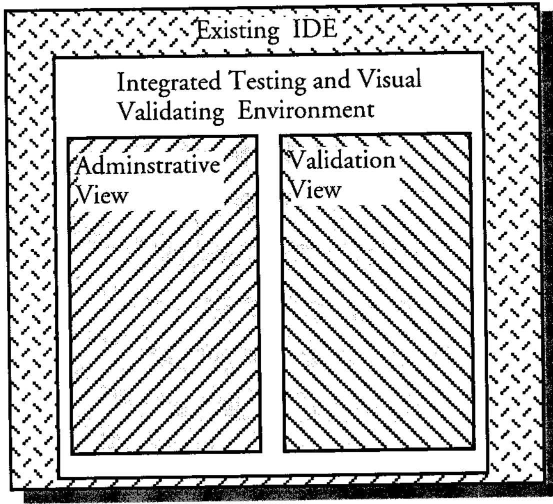

Figure 6-1. The integrated testing and validating environment in the proposed IDE.

The *ITVVE* consists of two main views, the Administrative View (AV) and the Validation View (VV). The *AV* is an environment that allows the user/tester to administer test cases to program units or components developed in the existing environment. Test cases can be extracted based on either the specification or the logic of the source code, or manually created by the tester. The *VV* offers a visual representation of both the control-flow and data-flow information of a *CFG*. Once a program unit has been successfully compiled, visual representations of the program parts that are of interest for structural testing will be automatically generated in the *VV*. The user at this point can switch to the *AV* and begin a testing session by choosing a structural-based testing session such as All-branches. Once a chosen test case is run, the user can switch to the *VV* and visually monitor the validated branches that were tested and those that were not.

6.2.2 Proposed System and Method

Consider the example in Figure 6-2 that computes the square root of a number between 0 and 1 to an accuracy *e*, such that $0 \; e \leq 1$. This example contains an error; the statements at lines 20 and 21 should be interchanged.

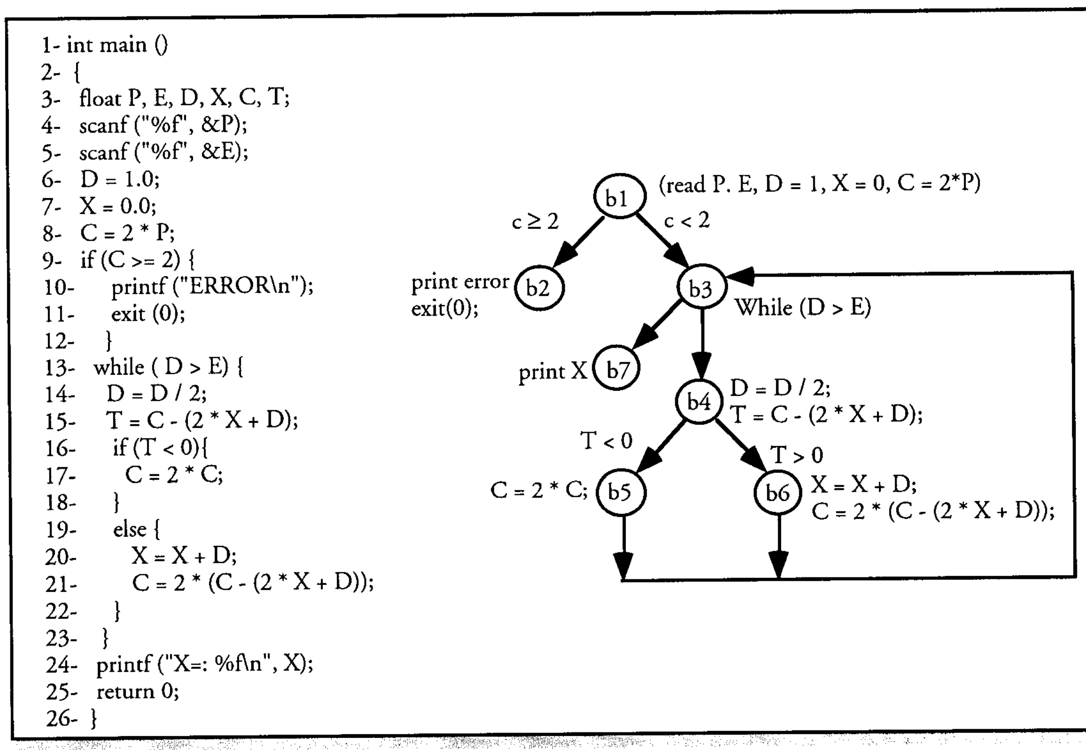

Figure 6-2. A C example containing an error (left) and its CFG (right).

Consider the test suite in Table 6-1 that applies the All-nodes and All-edges to the example of Figure 6-2. The output of the second and third test cases in the test suite report a wrong result; however, after administering all three test cases, the All-nodes and All-edges report 100% testedness. This means that neither one of those criteria has caught the error. To help catch this error, a more rigorous testing criterion such as the All-du-paths is required. Consider the test suite in Table 6-2 that applies the All-du-paths testing to the example of Figure 6-2. The output of the second and third test cases in the test suite report a wrong result; however with the All-du-paths testing criterion a 98% testedness is reported. The All-du-paths criterion is not satisfied since the path (*b6, b3, b4, b6*) is not traversed with regards to the definition of *C* in *b6*. This means that the definition of *C* in *b6* does not reach it use over the loop edge.

Using existing structural-based testing tools to test this program would reveal that, when All-du-paths testing is not satisfied, these tools generate a log file with a textual message to convey

that a du-association with regards to variable *C* has not been traversed. The problem with such a message is that it does not specifically identify the untested path with regards to *C* in the code.

Table 6-1. Node and edges test suite for the C program shown in Figure 6-2.

P	E	output	nodes	%	edges	%
1	--	invalid domain	b1, b2.	28.5	(b1, b2).	12.5
0.9	1	0.0 "wrong result"	b1, b3, b7.	28.5	(b3, b7).	12.5
0.9	0.004	0.625 "wrong result"	b1, b3, b4, b6, b3, b4, b5, b3, b4, b6, b3, b4, b5, b3, b4, b5, b3, b4, b5, b3, b4, b5, b3, b4, b5, b3, b4, b5, b3, b7.	100.0	(b1, b3); (b3, b4); (b4, b5); (b5, b3); (b3, b4); (b4, b6); (b6, b3); (b3, b4); (b4, b5); (b5, b3); (b3, b4); (b4, b5); (b5, b3); (b3, b4); (b4, b5); (b5, b3); (b3, b4); (b4, b5); (b5, b3); (b3, b4); (b4, b5); (b5, b3); (b3, b4); (b4, b5); (b5, b3); (b3, b7).	100.0

Assuming that the program in Figure 6-2 has been developed using an *IDE* that incorporates our proposed *ITVVE*, we will next illustrate how to visually represent both the control-flow and data-flow of a *CFG* in the *VV*. We will also show how these visual representation can aid the user in visually catching the untraversed du-association and, quickly and precisely locating the variable responsible for the untraversed du-association in the textual source code.

Table 6-2. All-du-paths test suite for the C program shown in Figure 6-2.

P	E	output	All-du-paths	%
1	1	invalid domain	ø.	0.0
0.9	1	0.0 "wrong result"	(b1, (b3, b7), D); (b1, (b3, b7), E).	12.0
0.9	0.004	0.625 "wrong result"	(b1, (b3, b4), D); (b1, (b3, b4), E); (b1, b4, D); (b1, b4, C); (b1, b4, X); (b4, (b4, b5),T); (b1, b5, C); (b4, (b3, b4), D); (b1, (b3, b4), E); (b4, (b4, b6),T) (b4, b4, D); (b5, b3, C); (b1, b4, X); (b1, b6, X); (b4, b6, D); ((b5, b6, C); (b4, (b3, b4), D); (b1, (b3, b4), E); (b4, (b4, b5),T); (b6, b5, C); (b4, (b3, b4), D); (b1, (b3, b4), E); (b4, (b4, b5),T); (b6, b5, C); (b4, (b3, b4), D); (b1, (b3, b4), E); (b4, (b4, b5),T); (b6, b5, C); (b4, (b3, b4), D); (b1, (b3, b4), E); (b4, (b4, b5),T); (b6, b5, C); (b4, (b3, b4), D); (b1, (b3, b4), E); (b4, (b4, b5),T); (b6, b5, C); (b4, (b3, b7), D); (b1, (b3, b7), E);	98.0

Once a program has been successfully compiled, its compiler-based static analysis will be used to generate visual control-flow and data-flow information in the *VV*. The visual representations of control-flow information of a *CFG* is constructed as follows: each block in the *CFG* of a program will be represented with a *local*-like icon and a number. For example the *local* labeled *n2* in Figure 6-3 corresponds to the node or block labeled *b2* in the *CFG* of Figure 6-2. Every predicate block in the *CFG* is represented with a *local* annotated with a [✓] on its right-hand side, and with a [×] on its left-hand side, to represent the predicate block's true and false edges, respectively. For example, the local labeled *n1* in Figure 6-3 corresponds to the node labeled *b1* in the *CFG* of Figure 6-2.

Every loop in a *CFG* is represented with a looped *local*. For example, the looped *local* labeled *n3* of Figure 6-3 corresponds to the loop that starts at the node labeled *b3* in the *CFG* of Figure 6-2. This looped *local*, once doubled clicked opens up. Inside, the looped *local* contains the set of locals {*n4*, *n5*, *n6*} that represent, the set {*b4*, *b5*, *b6*} of blocks in the *CFG* of Figure 6-2, respectively.

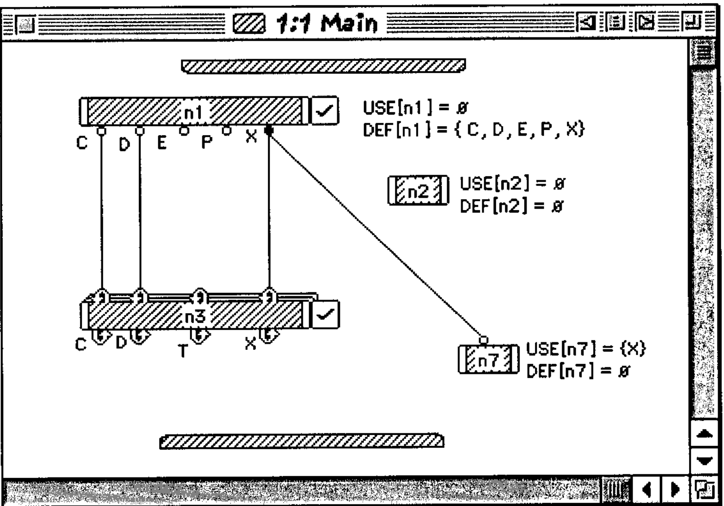

Figure 6-3. The control and data-flow information of the example in Figure 6-2.

The visual representations of data-flow information of a *CFG* is constructed as follows: Every formal parameter is represented with a *root* on the Input Bar of the window representing the method. Every formal parameter that reaches, live, the end of the method is represented with a *terminal* on the Output Bar of the window representing the method. Every variable definition

in a node of a *CFG* that reaches, live, the end of the block is represented with a *root* on the *local*. Every variable definition that reaches, live a use(s) in a block other than the one it was defined in, is represented with a *terminal* on the local corresponding to the block where it is used. For example, in Figure 6-2 variable *X* is defined in *b1* and reaches live via the path (*b1, b3, b7*) a use in block *b7*. Thus, a *terminal* is created on the local labeled *n7* of Figure 6-3. For example, the definition of *X* in node *b1* of Figure 6-2, reaches live the end of *b1*, and thus is represented with a *root* on the local labeled *n1* in Figure 6-3.

To represent the du-associations with respect to each variable in a *CFG*, we recognize three types of links between *roots* and *terminals*: direct links; extended links; and wrap-around links. A direct link is a straight link that connects a *root* to a *terminal*. For example, the link between the *root* labeled *X* on the *local* labeled *n1* in Figure 6-3 and the *terminal* on the *local* labeled *n7* is a direct link. An extended link is constructed in two cases: (1) a definition inside the body of loop that reaches, live, a use outside the loop, or (2) a definition outside the loop body that reaches, live, a use inside the body of the loop. For example, in the first case, the definition of *X* in node *b6* of Figure 6-2 reaches, live, a use in node *b7* via the path {*b6, b3 b7*}.

Thus, an extended link is constructed from the *root* labeled *C* on the local labeled *n6* in Figure 6-4 to the *terminal* on the Output Bar of the window labeled *n3*, and from the third right-hand side *root* on the looped predicate *local* labeled *n3* in Figure 6-3 to the *terminal* on the *local* labeled *n7*. The *terminals* on the Output Bar of a looped *local* represent ports through which definitions of variables inside the loop reach uses either outside the loop or back in the loop via the loop edge. An example of the second case is the extended link that starts with the *root* labeled *X* at the predicate local labeled *n1* in Figure 6-3 to the *terminal* on the looped predicate *local* labeled *n3* and from the third *root* on the Input Bar of the window labeled *n3* in Figure 6-4 to the third *terminal* on the predicate *local* labeled *n4*, and the third right-hand *terminal* on the local labeled *n6*. The *roots* on the Input Bar of a looped *local* represent ports through which definitions either inside or outside the body of the loop reach uses inside the loop.

A wrap-around link is constructed for every variable defined inside the body of a loop, and reaches, live, uses back in any block inside the loop via the loop edge. We construct a wrap-around link by: (a) constructing a link from the *terminal* where the variable is defined to the appropriate *terminal* on the Output Bar of the looped *local*; (b) constructing a loop-link on the link constructed in (a); (c) wrapping the link around the looped *local*, and into the appropriate loop *terminal*; (d) constructing links from the appropriate *root* on the Input Bar of the looped *local* to terminals that are associated with the uses; and (e) constructing loop-links on the links constructed in (d). If the links that are to be constructed in (d) already exists, we simply skip the construction process of (d), and apply (e). The loop-link is constructed to differentiate between the definition of a variable (outside the body of the loop) that reaches, live, uses directly inside the loop, and a redefinition of the same variable (inside the loop body) that reach, live, uses via the loop edge.

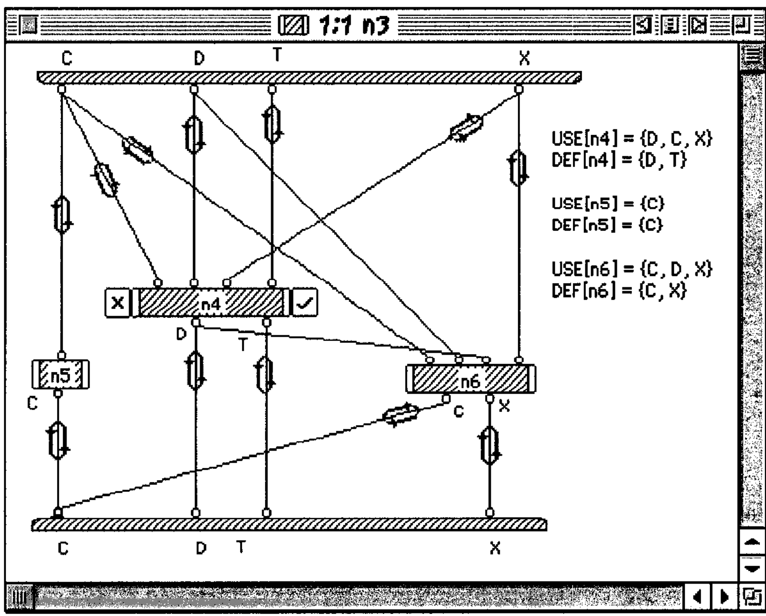

Figure 6-4. The collection of blocks inside the loop labeled n3 of Figure 6-3.

To illustrate, consider the example of Figure 6-2. In that example, the definition of *X* in *b1* reaches, live, uses in *b4*, and *b6*, and thus, as depicted in Figure 6-3 and Figure 6-4, extended links are created from *n1* to *n4* and *n6*. In that example also, the definition of *X* in *b6* reaches, live, via the loop edge, uses in *b4*, and *b6*. Thus, as depicted in Figure 6-4, we (a) construct a wrap-around link from the *terminal* that is associated with *X* in *b6*, (b) construct a loop-link

on the link that was constructed in (a), (c) wrap the link around the looped *local n3,* and into the third right-hand side loop *terminal,* (d) skip the construction process since the links that are associated with the uses in *n4* and *n6* already exist, and (e) construct the appropriate loop-links on the already constructed links.

6.2.3 Reflecting The Testedness

Under a control-flow testing criterion such as All-branches, these visual annotations will be colored both green to reflect that the two edges which are associated with the two possible outcomes of a predicate statement are traversed, one green and one red to indicate that only one edge has been traversed, and red otherwise. For example, after the first test case in the test suite of Table 6-1 is executed, the All-branches testing criterion will result in coloring all the left and right visual annotation of blocks *b1*, *b3*, and *b4* to green indicating that the program has resulted in a 100% testedness under the All-branches criterion. Under the All-du criterion; however, no amount of further testing or test cases would result in testing the definition of *C* in block *b6* that reaches uses in block *b4.* To reflect this untested du-association, we color red the loop-links of the wrap-around link that starts at the *root* that is associated with the definition of *C* at *n6,* and ends at the first right-hand *terminal* on the predicate *local n4.*

6.2.4 Locating the Error in the Source Code

To help the user locate the variable involved in the untraversed du-chain, the user can simply double click on the red link, and the environment will take the user into the source code where the suspected variable will be highlighted in some way. Or, analogous to the work in [46], we could include the source code statements that are associated with each block. By so doing the user can correct the above mentioned error directly from the *VV* by opening the block that the red link is connected to and observe the highlighted variables.

6.2.5 Benefits

The benefit of integrating our proposed visual testing tool in existing IDEs is two fold: one academic and the other is industrial. In an academic environment this tool can help in teaching various aspects of structural testing. For example, a student may be required to uncover a coding error such as the one introduced in the example of Figure 6-2. Such an exercise can help student develop a better understanding of structural testing criteria such as All-branches or All-du-paths. From an industrial point of view, the integration of this visual testing tool complements the cycle of visual development and debugging in existing IDEs. For example, after a program has been debugged and compiled, programmers can begin testing in the same IDE. Furthermore, the visual testing feedback provided by this tool allows programmers to intuitively identify and correct coding errors.

6.3 Conclusion and Future Work

We have shown that, with visual dataflow languages such as Prograph, visual constructs of the visual dataflow language can be used to reflect the program's testedness according to a particular testing criterion. For example, (1) with All-nodes, a Prograph operation is colored green when traversed, and red otherwise; (2) with All-branches, the control annotation on an operation is fully colored green when both outcome of the predicate are traversed, half green half red when one of the predicate outcome is traversed and fully red otherwise; and (3) with All-du-paths, a datalink or du associations is fully colored green when both outcome of the predicate are traversed, half green half red when one of the predicate outcome is traversed, and fully red otherwise. We believe that the testing methodologies we have provided for visual dataflow languages can help increase confidence in visual dataflow code, and perhaps provide yet another incentive for the integration of visual dataflow languages into mainstream programming. As for our proposed system, we believe that the *ITVVE* provides a visual testing environment that facilitates the task of approximating the location of errors in imperative languages.

As for our future work it will involve: (1) establishing a hierarchy of structural testing criteria for dataflow languages, and compare that with the hierarchy of structural testing criteria for

imperative languages, (2) improving the *ITVVE* to handle structured variables, and pointers, and (3) develop the Administrative View interface, and further improve that of the Visual Validation. We could also extend the visual testing environment to handle a variety of market-driven languages such as C++ and Java. Another future direction would be to generalize the tool so that it could be, with moderate changes, applied to any existing *IDEs*.

Bibliography

[1] Aho, A. V; Sethi, R; Ullman, J. D., Compilers, Principles, Techniques, and Tools. Addison-wesley, Boston, MA, 1986.

[2] Allen, F., E; Cocke J., A Program Data Flow Analysis Procedure. *Commun. Ass. Comput. Mach,* vol. 19, pp. 137-147, Marsh, 1976.

[3] Allen, F. E.; Burre, M.; Charles, P.; Cytron, R.; Ferrante, J., An Overview of the PTFL4N Analysis System for Multiprocessing. *In Proceedings of 1st International Conference on Supercomputing.* Springer-Verlag, New York, pp.194-211, June1987.

[4] Appleby, D; Vandekopple, J. J., Programming Languages: Paradigm and Practice, McGraw-Hill Book Computer Science Series, 1997.

[5] Azem, A.; Belli, F.; Jack, O., Jedrezejowicz, P., Testing and Reliability of Logic Programs. *In the Forth International Symposium on Software Reliability Engineering,* pages 318-327, 1993.

[6] Barth, J. M., A Practical Interprocedural Data Flow Analysis Algorithm. *Communication of ACM* 21, 9. Sept.1978. pp. 724-736

[7] Banning, J. P., An Efficient Way to Find the Side Effects of Procedure Calls and Aliases of Variables. *In Sixth Annual ACM Symposium on Principles of Programming Languages* ACM, New York, Jan. 1979.

[8] Belli, F.; Jack, O., A Test Coverage Notion for Logic Programming. *In the 6th International Symposium on Software Reliability Engineering,* pages 133-142, 1995.

[9] Bernini, M, Mosconi, M., Vipers: a Data Flow Visual Programming Environment Based on the Tcl Language, *in Proceeding of AVI'94, ACM Press,* 1994.

[10] Berry E. Paton., Sensors, Transducers & LabView, Prentice Hall, 1998.

[11] Bieman, J. M.; Schultz, J. L., An Empirical Evaluation (and Specification) of the all-du-paths Testing Criterion. *Software Engineering Journal*, Jan. 1992, (43-51).

[12] Binder, Robert, V., Testing Object-Oriented Systems. Models, Patterns, and Tools. Addison Wesley 1999.

[13] Blostein, D; Schurr, A., Visual Modeling and Programming with Graph Transformations. *14th IEEE Symposium on Visual Languages.* Halifax, NS, Canada, Sept. 1-4, 1998.

[14] Bratko I., PRLOG, Programming For Artificial Intelligence. Addison Wesley, Second edition 1990.

[15] Budd, T. A.; Angluin, D., Two Notions of Correctness and Their Relation to Testing. *Acta Inf.* 18, 31-45.

[16] Burnett, M; Dupuis, C; Li, L; Rothermel, G., What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs. *IEEE International Conference on Software Engineering*, pp.198-207 1998.

[17] Burnett, B; Li, L; Rothermel, Gregg., Testing Strategies for Form-Based Visual Programs. *IEEE on Software Reliability Engineering*, pp. 96-107, November 1997.

[18] Burnett, M; Djang R. W., Introduction to Visual Programming languages: Scaling-Up Issues, Tutorial. *IEEE Symposium on Visual Languages*, 1998.

[19] Callahan, D., The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Analysis. *In Proceeding of SIGPLAN'88 Conference on Programming Language Design and Implementation.* ACM SIGPLAN Not. 23, 7 July,1988.

[20] Clarke, L; Podgurski, A; Richardson, J; Zeil, S., A Formal Evaluation of Data Flow Path Selection Criteria, *IEEE Transaction on Software Engineering*, Vol. 15, No. 11, November 1989.

[21] Cox, P; Pietrzykowski, T., Prograph: A Step Towards Liberating the Programmer From Textual Conditioning. *Proceedings of the IEEE Workshop on Visual Languages*, pp. 150-156, 1989.

[22] Cooper,K.; Kennedy, K., Interprocedural Side-effect Analysis in Linear Time. *In Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation.* ACM SIGPLAN No. 23, 7 July 1988.

[23] Del Frate, F.; Grag, P.; Mathur, A.; Pasquin, A., On the Correlation Between Code Coverage and Software Reliability. *In the 6th International Symposium on Software Reliability Engineering,* pages 124-132, October 1995.

[24] Duesterwald, E. and Gupta, R. and Soffa, M.L., Rigorous Data Flow Testing Through Output Influences. *Proceedings: 2nd Irvine Software Symposium,* March,1992.

[25] D. Ungar, H. Lienberman & C. fry, Debugging and the Experience of Immediacy, Communication of the ACM April 1997, v40 no 4.

[26] Fosdick, L. D; Osterweil, L. J., Data Flow Analysis in Software Reliability. *Comput. Surveys,* vol. 8, pp. 305-330, Sept. 1976.

[27] Frankl, P; Weyuker, E., A Formal Analysis of the Fault-Detecting Ability of Testing Methods. *IEEE Transactions on Software Engineering,* Vol. 19, No.3, pp. 202-213, March 1993.

[28] Frankl, P; Weyuker, E., An Analytical Comparison of the Fault-detecting Ability of Data Flow Testing Techniques. *IEEE Transactions on Software Engineering,* October 1993.

[29] Frankl, P; Weyuker, E., An Applicable Family of Data Flow Criteria, *IEEE Transactions on Software Engineering,* Vol. 14, No.10, pp. 1483-1498, October 1988.

[30] Frankl, P; Weyuker, E., Provable Improvements on Branch Testing, *IEEE Transaction on Software Engineering.* SE-17, 6, pp. 553-564, June, 1991.

[31] Frankl, P; Weiss, S., An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE Transaction on Software Engineering,* Vol. 19, No. 8, August 1993.

[32] Frankl, P; Weiss, S., An Experimental Comparison of the Effectiveness of the all-uses and all-edges Adequacy Criteria. *Proceedings of the Symposium on Testing, Analysis, and Verification,* pp. 154-64, October 1991.

[33] Frankl, P; Weiss, S; Weyuker, E. J., ASSET: A System to Select and Evaluate Tests. *In Proceedings of the IEE Conference on software Tools*, pp. 72-79, April, 1985.

[34] Frankl, P; Weyuker, E. J., A Data Flow Testing Tool. *In ACM Softfair Proceedings* Dec. 1985. ACM, New York, 46-53.

[35] Ghezzi, C; Jazayeri, M; Mandrioli, D., Fundamentals of Software Engineering, Prentice Hall, Englewood Cliffs, NJ, 1991.

[36] Glinert, P. Ephraim, Visual Programming Environments Paradigms and Systems. *IEEE Computer Society Press, Los Alamitos*, 1990

[37] Goodenough, J. B.; Gerhart, S. L., Towards a theory of Test Data Selection. *IEEE Transaction on Software Engineering*, vol. SE-1, pp. 156-173, June 1975.

[38] Hamlet, R. G., Testing Programs with The Aid of a Compiler, IEEE Transactions on Software Engineering, vol. 3, no 4, pp. 279-290, July 1977.

[39] Hamlet, D.; Gifford, B., and Nikolik, B., Exploring Dataflow Testing of Arrays. *In proceeding of 15th ICSE*, 118-129, May 1993.

[40] Harrold, M. J; Rothmermel G., Performing Data Flow Testing on Classes, *SIGSOFT, ACM* pp. 154-163, 1994.

[41] Harrold, M. J; McGregor, J. D; Fitzpatrick, K. J., Incremental Testing of Object Oriented Class Structures. *In proceedings of 14th ICSE*, 68-80, May 1992.

[42] Harrold, M. J.; Soffa, M., An Incremental Approach to Unit Testing During Maintenance. *IEEE Proceedings of the Conference on Software Maintenance*, pp. 362-367, October 1988.

[43] Harrold, M. J.; Soffa, L. M., Interprocedural Data Flow Testing. *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pp. 158-167, December 1989.

[44] Harrold, M. J.; Soffa, L. M., Efficient Computation of Interprocedural definition-use Chains, *ACM Transaction on Programming Languages and Systems*, 16(2):175-204, March 1994.

[45] Harrold, M. J.; Soffa, L. M.,A, M. L., An incremental Data Flow Testing Tool. *In Proceedings of the 6th International Conference on Testing Computer Software.* Washington D. C., May 1989.

[46] Harrold, M. J.; Priyadarshan Kolte., Combat: A Compiler Based Data Flow Testing System,

[47] Herman, P., A Data Flow Analysis Approach to Program Testing. *Aust. Compu. J.* 8, 3, (Nov.), 92-96, 1976.

[48] Hetzel, W., The Complete Guide to Software Testing. Collins, 1984.

[49] Hils, D.D., Visual Languages and Computing Survey: Data Flow Programming Languages, *Journal of Visual Languages and Computing, vol. 3*, 1992, pp. 69-101.

[50] Horgan, J; London S., Data flow Coverage and the C Language. *In Proceedings of the Forth Symposium on Testing, Analysis, and Verification*, pp.87-97, October 1991.

[51] Horgan, D. M; Strooper, P., Automated Module Testing in Prolog. *IEEE Transaction on Software Engineering*, 17-9 (Sept.), 934-943, 1991.

[52] Howden, W. E., Reliability of the Path Analysis Testing Strategy. *IEEE Transaction on Software Engineering*, vol. SE-2, no. 3, pp. 208-215, 1976.

[53] Huang, J. C., Detection of Data Flow Anomaly Through Program Instrumentation, *IEEE Transaction on Software Engineering*, vol. SE-5, pp. 226-236, May 1979.

[54] Hutchins, M; Foster, H; Gordia, Experiments on the Effectiveness of Dataflow-and Control flow-Based Test Adequacy Criteria. *International conference on Software engineering*, pp. 191-200, May1994.

[55] Kildall, G. A Unified Approach to Global Program Optimization. *In ACM Symposium on Prmclples of Programming Languages.* ACM, New York, pp. 194–206, 1973.

[56] Laski, J. W.; Korel, B., A Data Flow Oriented Program Testing Strategy. I*EEE Transactions on Software Engineering*, Vol. SE-9, No. 3, pp. 347-54, May 1983.

[57] Laski, J. W.; Korel, B., A Tool for Data Flow Oriented Program Testing. *In ACM Softfair Proceedings.* ACM, New York, pp. 35-37, Dec.1985.

[58] Lomet, D. B., Data Flow Analysis in the Presence of Procedure Calls, *IBM J. Res. Dev.* 21, 6, pp. 559-571, Nov. 1977.

[59] Lou, G.; Bouchmann, G., Sarikaya, B.; Boyer, M., Control-flow based Testing of Prolog Programs. *In the 3rd International Symposium on Software Reliability Engineering,* pages 104-113, 1992.

[60] McCabe, T. J., Structured Testing. *IEEE Computer Society Pres.,* Los Alamos, CA, 1983.

[61] McCabe, T. J., A Complexity Measure. *IEEE Transaction on Software Engineering,* SE-2, 4, 308-320, 1976.

[62] McCabe, T. J, Schulmeyer, G. G., System testing Aided by Structured Analysis: A practical Experience. *IEEE Transaction on Software Engineering.* SE-11(9): 917-921, Sept., 1985.

[63] M.M Burnett, R.W Djang Introduction to Visual Programming languages: Scaling-Up Issues, Tutorial 1998 IEEE Symposium on Visual Languages.

[64] Myers, G. J., The art of Software Testing. John Wiley and Sons, New York, 1979.

[65] Myers, E. W., A Precise Interprocedural Data Flow Algorithm. *In Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, pp. 219-230, Jan. 1981.

[66] Ntafos, S. C., On Required Element Testing. *IEEE Transactions on Software Engineering,* 10(6), Nov., 1984.

[67] Ntafos, S. C., A Comparison of Some Structural Testing Strategies. *IEEE Transaction on Software Engineering,* SE-14, pp. 868-874, June 1988.

[68] Ostrand, T, J; Weyuker, E., Data flow-based Test Adequacy Analysis for Languages with Pointers. I*n Proceedings of SIGSOFT Symposium on Software Testing, and Verification* 4, pp. 74-86, Oct. 1991.

[69] Osterwel, L. J., The detection of Unexecutable Program Paths Through Static Data Flow analysis. *in Proc. IEEE COMPSAC 77,* Chicago, IL, pp. 406-413, Dec. 1977.

[70] Paige M. R., Program Graphs, an Algebra, and Their Implication for Programming, *IEEE Transaction on Software Engineering,* SE-1 No. 3, pp. 286-291., Sept. 1975.

[71] Pandi, H. D; Ryder, B. G; Landi, W., Interprocedural Def-use Associations in C Programs. *IEEE Transactions on Software Engineering*, 20(5):385-403, May 1994.

[72] Parrish, A. S; Borie, R. B; Cordes, D. W., Automated Flow Graph-based Testing of Object-Oriented Software Modules. *Journal of Systems Software*, 23: pp. 95-109, November 1993.

[73] Parrish, A; Zweben, S. H., Analysis and Refinement of Software Test Data Adequacy Properties. *IEEE Transactions on Software Engineering*. SE-17(6):565-581, June 1991.

[74] Parrish, A; Zweben, S. H., Clarifying Some Fundamental Concepts in Software Testing. *IEEE Transactions on Software Engineering*, 19(7):742-746, July, 1993.

[75] Perry, E.; Gail, K. G., Adequate Testing and Object-Oriented Programming. Journal of Object-Oriented Programming 2(5):13-19, Jan./Feb., 1990.

[76] Rapps, S.; Weyuker, E., Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 367-375, April 1985.

[77] Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill Book Company. 1992

[78] Ryder, B. G.; Paull, M. C., Incremental Data Flow Analysis Algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1-50, JAN. 1988.

[79] Schach R. Stephen: Software Engineering with Java, Irwin. 1997.

[80] Shafer Dan, The power of Prograph CPX, The reader Network, 1994.

[81] Sharp, John, A; Data Flow Computing, Ellis Horwood, 1985.

[82] Smedley, J. T.; Cox, T.Cox., Visual Languages for the Design and Development of Structured Objects*. *Journal of Visual Languages and Computing*, 8, 57-84, 1997.

[83] Soffa, M. L.; Pallock, L., An Incremental Version of Iterative Data Flow Analysis. *IEEE transaction on Software Engineering*, 15(12):1737-1549, Dec. 1989.

[84] Tanimoto, S., VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing*, 2(2), 127-139, June 1990.

[85] Ungar, U; Lienberman, H; Fry, C., Debugging and the Experience of Immediacy. *Communication of the ACM,* Vol.30 No 4, April 1997.

[86] Weyuker E., Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering,* 12(12):1128-1138, Dec.1986.

[87] Weyuker, E., More Experience with Data Flow Testing. *IEEE Transaction on Software Engineering,* Vol. 19, No. 9, September 1993.

[88] Weyuker, E. J., The Cost of Dataflow Testing: An Empirical Study. *IEEE Transactions on Software Engineering,* SE-16(2) pp121-128, February 1990.

[89] Weyuker E. J.; Ostrand T. J., Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transaction on Software Engineering,* vol. SE-6. pp. 236-246, May 1980.

[90] White, L. J; Wiszniewski, B., path testing of computer programs with loops using a tool for simple loop patterns. *Software Pract. Exper.* 21,(10), Oct. 1991.

[91] Wong, W.; Horgan, J.; London, S.; Mathur, A., Effect of Test Set Size and Block Coverage on the Fault Detection Effectiveness. *In the Fifth International Symposium on Software Reliability Engineering,* pages 230-8, Nov. 1994.

[92] Woodward, M R; Hedley, D; Hennel, M. A., Experience With Path Analysis and Testing of programs. *IEEE transaction on Software Engineering,* SE-6(5):278-286, May, 1980.

[93] Zhu, H., A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria. *IEEE Transaction Software Engineering,* Vol. 22, No. 4, April 1996.

[94] Zhu, H; Hall, P. A. V., Test Data Adequacy Measurements. *Software engineering Journal,* 8(1):21-30, January, 1993.

[95] Zhu, H; Hall, A. P; May, H. R. J., Software Unit Test Coverage and Adequacy. *ACM Computing Surveys,* Vol. 29, 29, No. 4, December 1997.