National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

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

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

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

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

AVIS

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

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

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

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

Canada

AUTOMATED TEST CASE SELECTION

by
Zhiyan Zhan

A M. Sc. Thesis

Submitted to the School of Graduate Studies and Research
of the University of Ottawa
in partial fulfillment of the requirements for the degree of

Master of Computer Science*

Department of Computer Science
University of Ottawa
Ottawa, Ontario

* The Master of Computer Science program is a joint program with Carleton University, administrated by the Ottawa-Carleton Institute for Computer Science.

ISBN 0-612-07814-0

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Test case selection in software testing is the process of identifying a set of test cases which satisfies a test criterion. Most of the existing test case selection systems are based on a pathwise approach, which consists of the following steps: control flowgraph construction, path selection, and test data construction. These systems have difficulties in dealing with path selection, loop iteration and array reference problems. Not only do they need user interactions, but also they waste significant computational effort in analyzing infeasible paths. Most of them are only capable of selecting test cases for control flow oriented testing.

In this thesis, we present an automatic test case selection system, which aims at selecting test cases for both control and data flow oriented testing. In our system, path expressions are employed, path expression generation and complete path selection algorithms are used to achieve the automation of path selection, and problems with loop iterations are addressed. Partial symbolic execution is applied to reduce the effort wasted in analyzing infeasible paths. Linear programing, path predicate simplification, and path predicate evaluation techniques are employed to solve the inequalities resulted from partial symbolic execution as well as symbolic execution. Our system is also capable of selecting different sets of test cases by applying different test selection criteria to achieve different test coverage which serve different test purposes.

In our system, first a flowgraph modeling both control and data flow information contained in the program under test is constructed. Second, by applying a control or data flow oriented test selection criterion to this flowgraph, a set of test units is built. Finally, through path expression generation, partial symbolic execution, symbolic execution, predicate simplification/evaluation, and linear programming, a set of test cases is selected

to cover feasible test units. Implementation details of a prototype ETSG are discussed and examples produced by ETSG are provided.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Hasan Ural, for all the guidance and valuable advice he has given me throughout my graduate studies at University of Ottawa. I especially wish to thank him for the great amount of time he devoted, and the patience and carefulness he showed during the review of this thesis.

I would like to thank the Telecommunication Software Engineering Research Group, in particular, to Dr. Robert Probert for stimulating discussions in an excellent research environment for the entire period of my study leading to this thesis.

I would like to thank BNR (Bell Northern Research) test tools development group for the research environment they provided in the development of Phase3 in ETSG. I acknowledge my colleagues of BNR, especially Keqin Zhu, for many valuable discussions and suggestions, and Alan William for his helpful insight in ETSG applications on SDL specifications.

I especially wish to thank my husband Wentao for his continuous encouragement and support which leads to the completeness of the thesis under my very tight schedule of a full-time job.

I also like to acknowledge NSERC (Natural Sciences and Engineering Research Council) and TRIO (Telecommunications Research Institute of Ottawa) for having financially supported this research and the School of Graduate Studies and Research of the University of Ottawa for providing me the entrance scholarship.

Introduction 1
1.1 Background 1
1.2 Contributions of the Thesis 4
1.3 Organization of the Thesis 6
Flowgraph Construction and Test Unit Derivation 7
2.1 Program Flowgraph Construction 7
2.2 Test Unit Derivation 15
2.2.1 Control Flow Oriented Testing 16
2.2.2 Data Flow Oriented Testing 22
Data Flow Related Concepts 22
Classification of Variable Occurrence 23
Data Flow Oriented Coverage Criteria 29
2.3 Other Aspects of Data Flow Analysis of Pascal Programs 32
2.3.1 Procedure/Function Calls 32
2.3.2 Arrays, Records, Pointers and File Variables 34
2.3.3 A More Complex Example 35
Path Expression and Complete Path Generation 40
3.1 Path Selection for a Test Unit 40
3.2 Representation of Path Expressions by Regular Expressions 42
3.3 Path Expression Derivation from Test Units 44
3.4 Application of Path Expressions in Data Flow Testing 47
3.4.1 Using Path Expressions to Identify Infeasible Test Units 48
3.4.2 Application of Path Expressions to Complete Path Selection 49
3.5 Complete Path Generation 52
Symbolic Evaluation 56
4.1 Symbolic Execution (SE) 57
4.2 Global Symbolic Execution (GSE) 62
4.3 Partial Symbolic Execution (PSE) 63
4.4 Comparison of Symbolic Evaluation Techniques 67
4.5 Path Predicate Simplification & Evaluation 68
4.5.1 Simplification of Arithmetic Expressions 69
4.5.2 Simplification of Relational Expressions 70
4.5.3 Linear Programming 71
Introduction to Linear Programming 71
Application of Linear Programming to Path Predicate Evaluation 75
4.5.4 Special Considerations in Path Predicate Evaluation 76
Arrays 76
Alphanumeric Constants 77
ETSG: A System Prototype 79
5.1 Language Translation and Test Unit Derivation 80
5.1.1 Phase1: Language Translation 80
5.1.2 Phase2: Test Unit Selection 82
5.2 Phase3: Test Case Selection 82
5.3 A Pascal Example 89
5.4 An SDL Example 91
Conclusions 94
6.1 Comparison with Related Work 94
6.2 Concluding Remarks 96

6.3 Future Enhancements 97
References 98
An SDL example 103
A.1 SDL Source File 103
A.2 Cgrr Flowgraph 107
A.3 Cgrr Flowgraph Text File 108
A.4 Cgrr Test Unit File For All-Edges Coverage 113
A.5 Cgrr Test Path File For All-Edges Coverage 113
A.6 Cgrr Test Unit File For All-Uses Coverage 114
A.7 Cgrr Test Path File For All-Uses Coverage 116
A.8 Infeasible Def-use Associations 117
Assumptions on Pascal and SDL Input 119
B.1 Assumptions on Pasflow Input 119
B.2 Assumptions on Sdlflow Input 120
Format of the Abstract Definition File 124
List of Functions 126
Files and Data Structures 148

FIGURE 1.1 Three Phases of Test Suite Construction in White Box Testing 2
FIGURE 2.1 A Pascal Example: Program A 13
FIGURE 2.2 Flowgraph of Program A 14
FIGURE 2.3 Relationship Test Selection Criteria 31
FIGURE 2.4 Program B and its Flowgraph 37
FIGURE 3.1 An Example of Path Expression Generation 47
FIGURE 3.2 Program C: Looping until done 49
FIGURE 3.3 System Flow of Identifying Infeasible Test Units 51
FIGURE 3.4 A Path Expression Expressed as a Binary Tree 53
FIGURE 4.1 Flowgraph of Bubblesort Program 59
FIGURE 5.1 ETSG Configuration 79
FIGURE 5.2 Phase3 System Flow 83
FIGURE 5.3 Module Structure of Phase3 88

TABLE 2.1 Mapping from Simple Statements to Subgraphs 9
TABLE 2.2 Mapping from Conditional Statements to Subgraphs 10
TABLE 2.3 Mapping from Repetitive Statements to Subgraphs 11
TABLE 2.4 Test Units for Some Testing Methods 16
TABLE 2.5 Definition and Uses of Variables within Pascal Statements 23
TABLE 2.6 Defs and Cuses for Program A 26
TABLE 2.7 Puses for Program A 27
TABLE 2.8 Def-cuse Associations for Each Def of a Variable 28
TABLE 2.9 Def-puse Associations for Each Def of a Variable 28
TABLE 2.10 Some Data Flow Oriented Coverage Criteria 30
TABLE 2.11 Test Unit File Required by All-uses Criterion for Program A 32
TABLE 2.12 Defs and Cuses for Program B 38
TABLE 2.13 Puses for Program B 38

Chapter 1

Introduction

1.1 Background

Software testing is one of the most widely used method for software quality assurance. The general goal of software testing is to reveal the existence of errors in a program through the application of a *test suite* (i.e. a finite set of *test cases*, each consisting of *test input* and corresponding *expected output*) in a controlled environment. Generally, software testing methods can be categorized into three classes: *black box testing* (also called *functional testing*), *grey box testing*, and *white box testing* (also called *structural testing*), depending on the resource used for the construction of test cases. Test cases are derived from specification, design, source code of the program under test in black, grey, or white box testing, respectively [24]. The study reported in this thesis adopts the white box testing, in particular, the test suite construction by using the explicit knowledge of the structure of the program under test.

Consider a program P and its specification S. Let I and O denote the input and output domains of P, respectively. In this context, a test suite T is a finite set of test cases, i.e. $T = \{(i, o) \mid (i, o) \in I \times O\}$. In white box testing [2-11], explicit knowledge of the structure of P is used for test suite construction which can be divided into three distinct phases [21] as shown in Fig 1.1:

- program flowgraph construction;
- test unit derivation;
- test case selection.

The first phase (program flowgraph construction) is to construct a directed graph $F = (V, E)$, called *flowgraph*, which represents P. Without loss of generality, it is assumed that P has a single entry and single exit. Every node in V corresponds to one simple statement in P [11]. There are two special nodes in V, namely s and t, denoting the entry and exit points of P. Every edge in E corresponds to the transfer of control between statements in P. The flowgraph representation of a program allows representing not only the control flow, but also the data flow information and thus can be used for precise description of both control and data flow oriented testing methods [30].

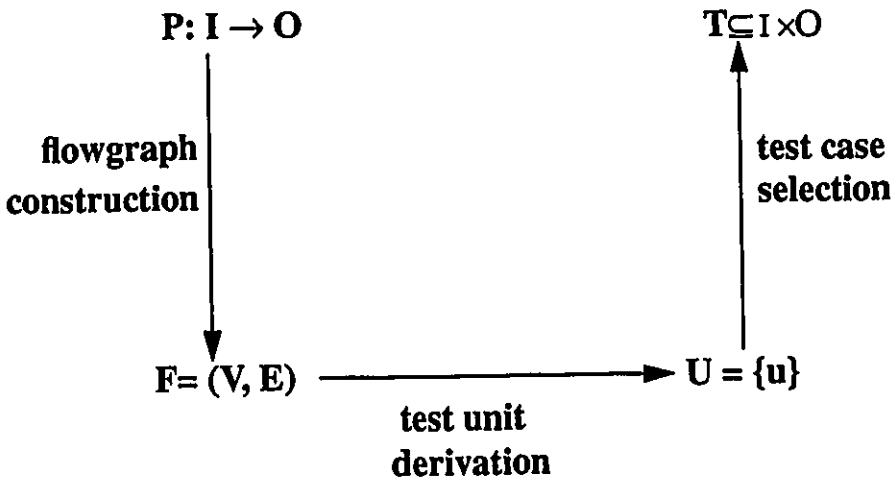


FIGURE 1.1 Three Phases of Test Suite Construction in White Box Testing

Given that F is obtained from P, one then enters the second phase: test unit derivation. In this phase, a finite set of test units U is derived from P to satisfy one or more testing methods that are defined in terms of control or data flow information exhibited by P. *Control flow information* is about the sequencing of the statements in P. *Data flow information* is about the relationships among the variables in the statements of P [22]. Accordingly, the white box testing methods can be divided into *control flow oriented*

testing methods and *data flow oriented testing methods*. Depending on the testing method selected, a *test unit* is a node (or statement), a branch (or decision outcome), a def-use association, etc. A test unit in P is called *feasible* if there exists an input data that can cause the execution of the test unit, otherwise, it is called *infeasible*.

When the set of test units required by the selected testing method is obtained, one then enters the third phase: test case selection in which a finite set of test cases $T = \{(i, o)\}$ (i is a test input, o is the corresponding expected output [20]) is selected. Test input i is selected from P's input domain I in order to impose the execution of the derived test units, and the expected output o is obtained from output domain O of S.

A collection of test inputs can be selected either manually or automatically. The manual approach requires testers to provide some initial inputs to P and identifies the test units executed by the inputs as well as the test units remained to be executed. Testers then analyze P and provide other input data until all the test units are executed or the test units not executed are identified as infeasible. ASSET [22] is an example tool which follows this approach. This approach is quite straightforward, but its disadvantage is that it relies on the tester's knowledge of the program under test and involves considerable human interaction.

The automatic approach, which is the focus of this thesis, includes three steps:

1) The feasibility check of a test unit: Path expression [7] and partial symbolic execution technique [6] are used to decide the feasibility of each test unit. If a test unit is decided infeasible, it will be deleted from the test unit set required to be executed by the test suite.

2) Complete path selection: A *complete path* (also called a *test path*) is a path whose first node is the entry node s of P and last node is the exit node t of P. If there exists some input data i which can cause the execution of a complete path, we say that the complete path is *executable* or *feasible*, and the test input is the input data i. Otherwise, the complete path is called *unexecutable* or *infeasible*. A feasible complete path is required for each test unit.

3) Test input selection: By employing symbolic execution [3][12][13][14] and path predicate evaluation techniques [26], a test input is selected from the feasible complete path obtained in step2.

Upon obtaining the test input i, the corresponding expected output o can be derived from the specification S, and hence, a test case (i,o) is obtained.

1.2 Contributions of the Thesis

Software testing is very labor-intensive and expensive. It accounts for approximately 50% of the cost of a software system development [2]. Software systems have been getting more complex, more difficult to understand, and consequently more difficult to test. With the current advances in software development, a crucial task is to achieve the automation of test suite construction.

This thesis aims to automate the process of test suite construction by following the automatic approach described in section 1.1. According to the system architecture of white box testing, the proposed system in this thesis can also be divided into three phases: flowgraph construction, test unit derivation, and test case selection. Compared with other related work, our system has the following advantages:

- Automation of both control and data flow oriented test suite construction is achieved.
- Formally defined testing methods are applied to minimize the defect risks.
- Different test unit selection criteria can be chosen to generate different sets of test cases, which can serve different test purposes.
- All the test cases generated are feasible, and the feasible complete paths which will be followed by the execution of the test cases are provided.
- Path expression [7] and partial symbolic execution [6] are employed to detect the infeasible test units even before complete paths are selected. This reduces the significant computation time wasted on analyzing infeasible test units.

The system proposed in this thesis is a comprehensive application of different test strategies and techniques, such as: control flow and data flow oriented testing methods [22], path expression generation [7], symbolic evaluation techniques, including partial symbolic execution [6] and symbolic execution [3][12-14], as well as predicate evaluation techniques such as linear programming. Some common problems encountered in test suite construction system, such as: flowgraph representation of procedure/function calls, arrays and pointers; path selection and loop iteration; and predicate evaluation/simplification techniques are discussed and solutions are provided. Algorithms for path expression generation, complete path selection, symbolic execution, and partial symbolic execution are given. A prototype of the proposed system is implemented and examples are analyzed.

Based on the automatic test suite generation model illustrated in Figure 1.1, the prototype of the system consists of three corresponding phases: Language Translation, Test Unit Generation, and Test Case Selection. The prototype is implemented in C Language under Unix. Phase2 and Phase3 are implemented by the author.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows: Based on the system architecture described in section 1.1, chapter 2 covers Phase1 (flowgraph construction) and Phase2 (test unit derivation) of the system. Some control flow and data flow related concepts are introduced and commonly used testing methods are also reviewed in this chapter. Phase3 of the system (test case selection) is discussed in Chapter 3 and 4. Chapter 3 describes path expression and complete path generation methods. Chapter 4 describes symbolic evaluation techniques, including symbolic execution, partial symbolic execution and global symbolic execution. Path predicate simplification and evaluation techniques are also studied. In chapter 5, a prototype of the proposed system is presented and some examples are analyzed. Comparison with other related systems, concluding remarks and future research directions are given in chapter 6.

Chapter 2

Flowgraph Construction and Test Unit Derivation

As mentioned in Chapter 1, in white box testing, test cases are constructed from the source code of a given program, and there are three seemingly distinct phases [21]:

- program flowgraph construction;
- test unit derivation;
- test case selection.

This chapter covers the first two phases of white box testing: program flowgraph construction and test unit derivation. Some terms used throughout this thesis are defined in this chapter. Control flow oriented and data flow oriented testing methods are reviewed. Several Pascal program examples are given to illustrate all the definitions introduced in this chapter.

Although the system can be applied to different programming or specification languages, from Chapter 2 to Chapter 4, Pascal is chosen as an example language to illustrate the methods applied in the system.

2.1 Program Flowgraph Construction

Without loss of generality, it is assumed that a program, which may be a main program or a subprogram, has a single entry and single exit. For ease of applying a control or data flow oriented testing method, a program is represented by a digraph $F = (V, E)$, called *flowgraph* [8], where V is a set of nodes corresponding to simple statements in the program, and E is a set of directed edges indicating the possible flow of control between statements in the program. In V, there is

a) a *statement node* (e.g. nodes 1, 2, 3, 4, 5, 6, 8, 9 and 11 in Figure 2.2) corresponding to each simple statement (i.e. input, output, assignment, or procedure statement) in the program.

b) a *branching node* (e.g. nodes 7 and 10 in Figure 2.2) corresponding to the branching point in the transfer of control implied by the decision part of each conditional statement (i.e. if-then, if-then-else, or case statement) or each repetitive statement (e.g. while, for, or repeat statement) in the program.

In addition, there are two special nodes in V, namely s and t, that correspond to the entry and exit points of the program, respectively. Each statement node in V contains a simple statement. All other nodes (i.e. s, t, and branching nodes) are assumed to be empty.

Every directed edge in E represents the possible flow of control between the nodes in V as implied by the transfer of control between statements in the program. Consequently,

a) For each statement node in V, there is a single outgoing edge.

b) For each branching node in V, there are at least two outgoing edges. Each outgoing edge of a branching node is associated with a predicate (i.e., a boolean expression) whose truth value is equivalent to a specific outcome of the decision that implies the branching point corresponding to the node.

c) For any pair of node i and j in V, there is at most one edge from node i to node j in E.

d) s has no incoming edge and t has no outgoing edge.

An edge (n_i, n_j) is called a *branch* if n_i is a branching node. The predicate associated with a branch in the flowgraph is referred to as a *branch predicate*, which describes the conditions under which the branch will be traversed.

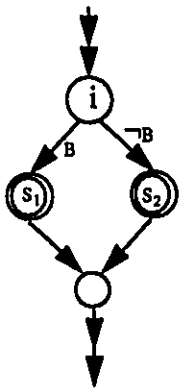
For Pascal programs, Tables 2.1-2.3 show the flowgraph segments to be constructed for each type of statement in Pascal. The flowgraph segments corresponding to statements in a Pascal program can be combined to construct the flowgraph representation of the program. Each node is labelled with the statement it contains. The edges emanating from a node where a branch occurs are labelled with the outcomes of the decision required to take these branches. In Tables 2.1-2.3, S_i denotes the subgraph corresponding to statement S_i [29]. If S_i is a simple statement, then its subgraph is a single node which is labeled by i.

TABLE 2.1 Mapping from Simple Statements to Subgraphs

Statement	Subgraph	Properties
S_i: Assignment statement $v := expr$;		None.
S_i: Input statement read (v_1,..., v_n); readln (v_1,..., v_n); read (f, v_1,..., v_n); readln (f, v_1,..., v_n);		None.
S_i: Output statement write (e_1,..., e_n); writeln (e_1,..., e_n); write (f, e_1,..., e_n); writeln (f, e_1,..., e_n);		None.

Statement	Subgraph	Properties
S_i: Procedure call statement Procedure_name (e_1,..., e_n);		None.

TABLE 2.2 Mapping from Conditional Statements to Subgraphs

Statement	Subgraph	Properties
S_i: If-then-else statement if B then S_1; if B then S_1 else S_2;		Let k and j be the entry nodes of S_1 and S_2, respectively. Then, edges (i, j) and (i, k) are associated with B and $\neg B$, respectively ($\neg$ stands for 'not' hereafter). If there is no "else" part then subgraph S_2 is eliminated by directing its incoming edge to the node reached by its outgoing edge.

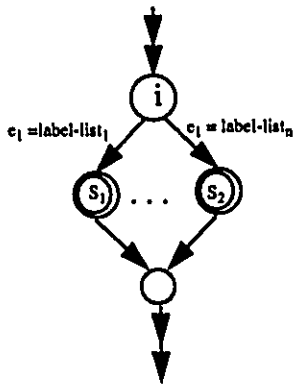
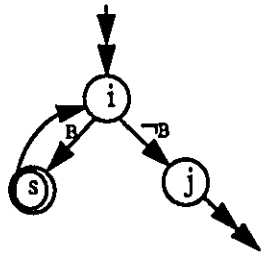
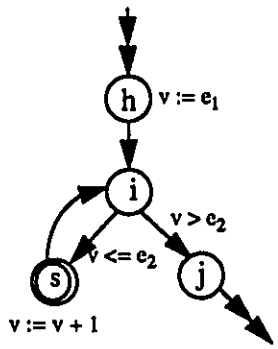
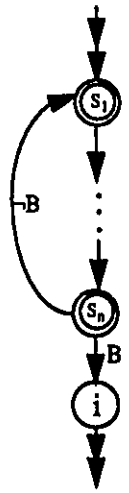
Statement	Subgraph	Properties
S_i: Case statement case e_1 of *label-list*$_1$: S_1; . . . *label-list*$_n$: S_n end;		Let $j_1,\ldots, j_n$ be the entry nodes of $S_1,\ldots,S_n$, respectively. Then, edges $(i, j_1),\ldots, (i, j_n)$ are associated with boolean expressions B_k ("e_1 = *label-list*$_k$"), $1 <= k <= n$, respectively. If "else" is used in the case statement, then a subgraph S_{n+1} is added such that it corresponds to the statement associated with $\neg(B_1$ or ... or $B_n)$ where j_{n+1} is the entry node of S_{n+1}.

TABLE 2.3 Mapping from Repetitive Statements to Subgraphs

Statement	Subgraph	Properties
S_i: While statement while B do S;		Let h be the entry node to subgraph S. Then, edges (i, h) and (i, j) are associated with B and $\neg B$, respectively.

Statement	Subgraph	Properties
S_i: For statement: for $v := e_1$ to e_2 do S; for $v := e_1$ downto e_2 do S; (Subgraph shows only statement "for $v := e_1$ to e_2 do S;")		Let f and g be the entry and exit nodes of S, respectively. Then, edges (i, f) and (i, j) are associated with boolean expressions "v (<= | >=) e_2" and "v (> | <) e_2", respectively. Node h is associated with the statement "$v := e_1$". Node S is associated with the statement "$v := v$ (+ | -) 1".
S_i: Repeat statement: repeat S_1, ..., S_n until B;		Let f and g be the entry node of S_1 and exit node of S_n, respectively. Then, edges (g, f) and (g, i) are associated with $\neg B$ and B, respectively.

To illustrate the format of a flowgraph, the following Pascal program will be used as an example. This program is used to do a statistic on the GRE scores in a class. It finds out the number of students whose total score of the three parts is greater than or equal to 1800 and the mark of part II (mathematics) is greater than or equal to 700, and display their score records.

```
program GRE_score (input, output);
var i, j, t, s, n, x, y, z: integer;
```

```
begin
    i := 0; j :=0;
    {Read the number of students in a class}
    readln(t);
    repeat
        {Read the record of a student using the format: student_number,
        scores for part I, II and III }
        readln(n,x,y,z);
        s := x + y + z;
        i := i + 1;
        if (s >= 1800) and (y >= 700)
        begin
            j := j + 1;
            writeln('Student number:',n, 'Part I',x, 'Part II',y, 'Part III',z, 'Sum',s);
        end
    until (i = t);
    writeln('number of students whose sum are equal to or over 1800, and
        Part II are equal to or over 700:', j);
end. {GRE_score}
```

FIGURE 2.1 A Pascal Example: Program A

Using the mappings given in the above tables, the example Pascal program in Figure 2.1 can be represented by the following flowgraph:

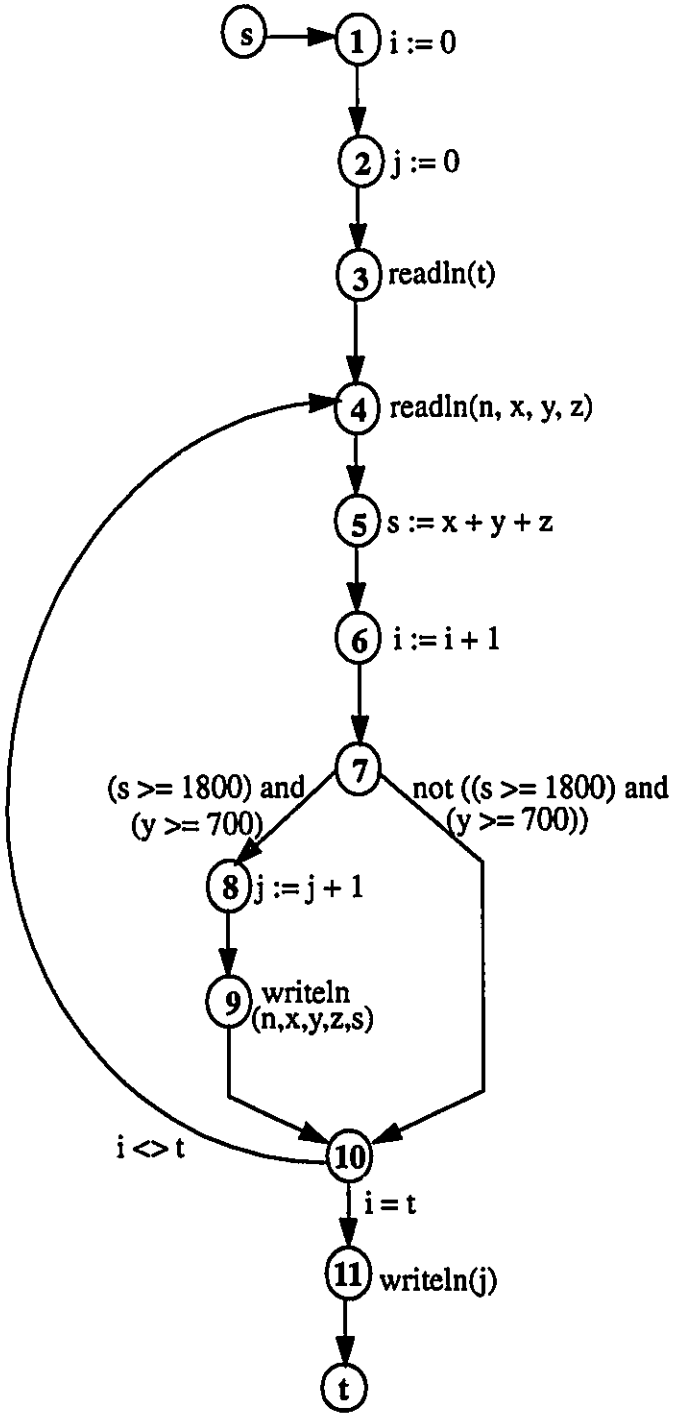


FIGURE 2.2 Flowgraph of Program A

A *path* $(n_1, n_2, \ldots, n_m)$ in a flowgraph $F=(V, E)$ is a sequence of nodes in F, such that for all i, $1 \le i \le m\text{-}1$, $m \ge 2$, $(n_i, n_{i+1}) \in E$. A *subpath* of a path $(n_1, \ldots, n_m)$ is a path $(i_1, \ldots, i_k)$ if there exists a ζ, $0 \le \zeta \le m - k$, such that for all j, $1 \le j \le k$, $i_j = n_{j+\zeta}$. A *loop-free path* is a path in which all nodes are distinct. A *simple path* is a path in which all nodes except possibly the first and the last are distinct. A *complete path* in F is a path whose first node is s and whose last node is t. For example, path (s, 1, 2, 3, 4, 5, 6, 7, 10, 11, t) is a complete path for the flowgraph in Figure 2.2. A complete path is *executable* or *feasible* if input data which causes its execution exists, and *unexecutable (infeasible)* otherwise. A path is executable if it is a sub-path of an executable complete path. Clearly, the question of whether a path is executable depends on the semantics of program itself, not just on the underlying graph structure.

Let p be a complete paths for the flowgraph of a given program, we say that a *node i is covered by p* if p is a complete path $(n_1, \ldots, n_m)$ such that $i = n_j$ for some j, $1 \le j \le m$. Similarly, an *edge (i_1, i_2) is covered by p* if p is a complete path $(n_1, \ldots, n_m)$ such that $i_1 = n_j$ and $i_2 = n_{j+1}$ for some j, $1 \le j \le m - 1$. A *path $(i_1, \ldots, i_k)$ is covered by p* if p is a complete path $(n_1, \ldots, n_m)$ and path $(i_1, \ldots, i_k)$ is subpath of path $(n_1, \ldots, n_m)$.

2.2 Test Unit Derivation

After the program flowgraph $F = (V, E)$ has been constructed, one then enters the second phase: test unit derivation. *A test unit* is a node, an edge, a du_path or a def_use association, etc. A file containing all the test units based on a given testing method is called a *test unit file*. The types of test units produced will depend on the chosen testing method. These are summarized in Table 2.4.

TABLE 2.4 Test Units for Some Testing Methods

Testing Method	Test Unit
All_nodes	node n_i
All_edges	edge (n_i, n_j)
All_uses	def-use association
All_du_paths	du_path

Basically, white box testing can be categorized as two classes: control and data flow oriented testing. *Control flow oriented testing* (also called *path testing*) is based on the analysis of the program's control flow. On the other hand, *data flow oriented testing* is based on the analysis of the program's data flow information. In order to reduce the potentially infinite exhaustive testing process, these testing methods aim to choose representative elements from the structural information provided by the program under test. Although the proposed system aims to produce test cases for data flow oriented testing, it also includes most control flow oriented testing methods. The commonly used control and data flow oriented testing methods are discussed in the following sections.

2.2.1 Control Flow Oriented Testing

Each control flow oriented testing method aims at generating a set of test cases that covers (i.e. causes execution of) all occurrences of a control flow oriented test unit (e.g. node, edge, branch, outcome of a decision, etc.). There are various control flow oriented testing methods: all_nodes (statement coverage) [4, 10], all_edges [4, 10], all_paths [4,10], decision coverage (branch coverage), condition coverage, decision/condition coverage and multiple condition coverage.

- All_nodes (Statement Coverage)

A set of test cases satisfies the *all_nodes* criterion (or *statement coverage* criterion) in a flowgraph if by executing this set of test cases, each node (statement) in the flowgraph is executed at least once. This is likely to be the easiest and weakest of all the testing methods. The test units required to be covered by all_nodes criterion for the example program in Figure 2.2 is listed below:

All_Nodes
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

- All_edges

A set of test cases satisfies the *all_edges* criterion in a flowgraph if by executing this set of test cases, each edge in the flowgraph is executed at least once. The test unit file required to be covered by all_edges criterion for the example program in Figure 2.2 is listed below:

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

- All_paths

A set of test cases satisfies the *all_paths* criterion in a flowgraph if by executing this set of test cases, each possible complete path in the flowgraph is executed at least once. If we achieve this prescription, we are said to have achieved **100% path coverage.** This is the strongest criterion among the testing methods, which is generally impossible to achieve.

- Decision (Branch) Coverage

A *decision* is a boolean expression [18]:

<decision> := <component> AND <component>
| <component> OR <component>
| NOT <component>

A component of a decision can be a decision or a *condition*, which is defined as follows:

<condition> := <var1> = <var2>
| <var1> > <var2>
| <var1> < <var2>
| <var1> >= <var2>
| <var1> <= <var2>
| <var1> != <var2>
| etc.

where var1, var2 may be optionally replaced by function names or constants.

A set of test cases satisfies the *decision coverage* criterion [18] (or *branch coverage* criterion) in a flowgraph if by executing this set of test cases, each outcome of each decision (i.e. branch) in the flowgraph is executed at least once. The test unit file required to be covered by the decision coverage criterion for the example program in Figure 2.2 is listed below:

Decision Coverage
(7, 8), (7, 10), (10, 4), (10, 11)

However, if there are more than one condition in a decision, decision coverage may not expose the errors that could be present in a particular condition. For example, for testing the decision (*A* or *B*), setting *A* to true and *B* to false in one test case, and *A* to false and *B* to false in another would satisfy the decision coverage criterion, but it could mask a possible error in condition *B*. A shortcoming of the decision coverage criterion is that it does not guarantee coverage of each condition within each decision.

- Condition Coverage

A set of test cases satisfies the *condition coverage* criterion [18] in a flowgraph if by executing this set of test cases, every outcome of each condition in each decision in the flowgraph is executed at least once. The test units required to be covered by the condition coverage criterion for the example program in Figure 2.2 is listed below:

```
Condition Coverage
From node 7 to node 8
    : CONDITION (s >= 1800) TRUE
    : CONDITION (y >= 700) TRUE
From node 7 to node 10
    : CONDITION (s >= 1800) FALSE
    : CONDITION (y >= 700) FALSE
From node 10 to node 4
    : CONDITION ( i = t) FALSE
From node 10 to node 11
    : CONDITION (i = t) TRUE
```

Although the condition coverage criterion states that every outcome of each condition in each decision will be executed, it does not however guarantee that all outcomes of every decision will be taken and hence it does not guarantee the coverage of all statements or all branches in a flowgraph. For example, for testing the decision (*A* or *B*), setting *A* to true and *B* to false in one test case, and *A* to false and *B* to true in another would satisfy the condition coverage criterion, but it would not satisfy the decision coverage criterion, since it would fail to test the false outcome of the decision.

- Decision/Condition Coverage

A set of test cases satisfies the *decision/condition coverage* criterion [18] in a flowgraph, if by executing this set of test cases, each outcome of each condition and each

outcome of each decision in the flowgraph is executed at least once. The test units required to be covered by the decision/condition coverage criterion for the example program in Figure 2.2 is listed below:

```
Decision/Condition Coverage
From node 10 to node 4:
    DECISION (i = t) FALSE
    : CONDITION (i = t) FALSE
From node 10 to node 11:
    DECISION (i = t) TRUE
    : CONDITION (i = t) TRUE
From node 7 to node 8:
    DECISION ((s >= 1800) AND ((y >= 700)) TRUE
    : CONDITION (s >= 1800) TRUE
    : CONDITION (y >= 700) TRUE
From node 7 to node 10:
    DECISION ((s >= 1800) AND ((y >= 700)) FALSE
    : CONDITION (s >= 1800) FALSE
    : CONDITION (y >= 700) FALSE
```

To compensate for the deficiencies of condition coverage and decision coverage, decision/ condition coverage ensures that each outcome of each condition and each outcome of each decision will be executed. However, it is still possible not to uncover the error of having "and" instead of "or" in a decision. For example, for testing the decision (*A* and *B*), setting *A* and *B* both to true in one test case, and *A* and *B* both to false in another would satisfy the decision/condition coverage criterion, but it would be possible to replace the *"and"* for an *"or"* without affecting the result of the tests.

- Multiple Condition Coverage

A set of test cases satisfies the *multiple condition coverage* criterion [18] in a flowgraph, if by executing this set of test cases, each possible combination of outcomes of

conditions in each decision in the flowgraph is executed at least once. The test units required to be covered by the multiple condition coverage criterion for the example program in Figure 2.2 is listed below:

Multiple Condition Coverage
From node 10 to node 4
: CONDITION (i = t) FALSE
From node 10 to node 11
: CONDITION (i = t) TRUE
From node 7 to node 10:
: CONDITION (s >= 1800) FALSE
: CONDITION (y >= 700) FALSE
From node 7 to node 10:
: CONDITION (s >= 1800) FALSE
: CONDITION (y > 700) TRUE
From node 7 to node 10:
: CONDITION (s >= 1800) TRUE
: CONDITION (y >= 700) FALSE
From node 7 to node 8:
: CONDITION (s >= 1800) TRUE
: CONDITION (y >= 700) TRUE

Multiple condition coverage eliminates the deficiencies of the previous criteria, since test cases must be generated to exercise all possible combinations of all condition outcomes in each decision.

It is widely accepted that statement coverage is rather weak, and branch coverage criterion is a minimal standard of achievement in white box testing [31]. Other criteria (multiple condition, decision/condition, condition) are difficult to achieve in a program of more than moderate complexity [31].

2.2.2 Data Flow Oriented Testing

Most test case selection systems are based on control flow testing, which examines the branch and loop structures of a program. We believe that data flow testing, which is widely used in code optimization, should be considered as well.

Data flow information is about the relationships among the variables in each statement of a program, and is made explicit by data flow analysis. Rather than selecting program paths based solely on the control structure of a program, data flow analysis tracks input variables through a program, following them as they are modified, until they are ultimately used to produce output values. Data flow oriented testing is based on the analysis of the data flow characteristics of the program being tested. Data flow oriented testing criteria are constructed so that critical associations between the definition of a variable and its uses are examined during program testing. Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed [10].

2.2.2.1 Data Flow Related Concepts

Data flow analysis was originally used for compiler optimization [22]. It aims to trace the program variables as they are initialized and modified during the program execution. Data flow analysis classifies each variable occurrence in a statement associated with a node or in a boolean expression associated with an edge in a flowgraph as a definition or a use [10, 22]. A *definition* (or *def*) of a variable x at node n (denoted by d^x_n) is an occurrence of x by which x receives a value. A *use* of a variable x is an occurrence of x by which the value of x is referenced. Uses of variables are further classified as *computational uses* (cuses) and *predicate uses* (puses) [10, 22]. A *cuse* of a variable x at

node n (denoted by c^x_n) is a use of x which directly affects the computation being performed (e.g., an occurrence of x on the RHS of an assignment statement, for instance, the use of x at node 5 in Figure 2.2) or allows one to see the result of some earlier definition (e.g., an occurrence of x in the list of variables of an output statement, for instance, the use of x at node 9 in Figure 2.2). A *puse* of a variable x on edge (n, m) (denoted by $p^x_{(n, m)}$) is a use of x which directly affects the control flow of the program (i.e., an occurrence of x in the boolean expression associated with an outgoing edge of a branching node n in a flowgraph, for instance, the use of s on edge (7, 8) in Figure 2.2).

2.2.2.2 Classification of Variable Occurrence

A sequence of definitions and/or cuses is associated with each node and a set of puses is associated with each edge in a flowgraph. Table 2.5 shows the classification of variable occurrences in different Pascal statements. In addition, the entry node is considered to have a definition of each parameter, each non-local variable which occurs in the subprogram, and the input buffer 'input^'. In Table 2.5, S_i denotes the subgraph corresponding to statement S_i.

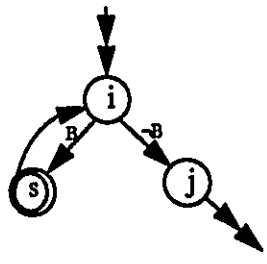
TABLE 2.5 Definition and Uses of Variables within Pascal Statements

SIMPLE STATEMENTS

Assignment statement *v*:= *expr*; Node *i* has cuses of each variable in *expr* followed by a definition of *v*.	

Input statement read (v_1,..., v_n); readln (v_1,..., v_n); read (f, v_1,..., v_n); readln (f, v_1,..., v_n); Node i has definitions of v_1,..., v_n. File variable f is ignored.	
Output statement write (e_1,..., e_n); writeln (e_1,..., e_n); write (f, e_1,..., e_n); writeln (f, e_1,..., e_n); Node i has cuses of each variable occurring in e_1,..., e_n. File variable f is ignored.	
Procedure call: P (e_1,..., e_n); Node i has cuses of each variable occurring in the expressions e_1,..., e_n. These are followed by definitions of each actual parameter which corresponds to a pass-by-reference parameter.	

REPETITIVE STATEMENTS

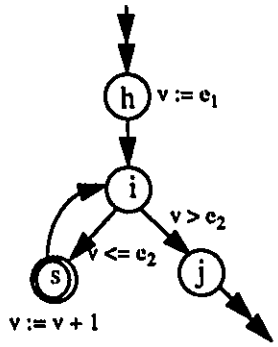
while statement: while B do S; Let h be the entry node to subgraph S. Edges (i, h) and (i, j) have puse of each variable in the boolean expression B.	

for statement:

for $v := e_1$ to e_2 do S;

for $v := e_1$ downto e_2 do S;

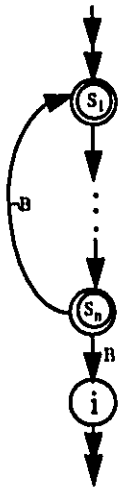
Let f and g be the entry and exit nodes of S, respectively. Node h has cuses of each variable in e_1 followed by a def of v. Edge (i, f) and (i, j) have puses of v and each variable in e_2. Node g has a cuse of v followed by a def of v.



repeat statement:

repeat S_1,..., S_n until B;

Let f be the entry node of S_1, and let g be the exit node of S_n. Edges (g, f) and (g, i) have puses of each variable in the boolean expression B.

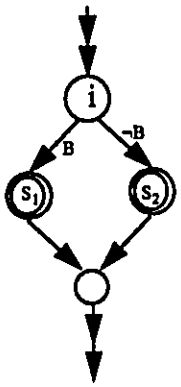


CONDITIONAL STATEMENTS

<table>
<tr>
<td>if-then-else statement

if B then S_1;

if B then S_1 else S_2;

Let k and j be the entry nodes of S_1 and S_2, respectively. Edges (i, j) and (i, k) have puses of each variable in the boolean expression B. If there is no "else" part then subgraph S_2 is eliminated by directing its incoming edge to the node reached by its outgoing edge.</td>
<td> 
</td>
</tr>
<tr>
<td>Case statements

case e_1 of

$label\text{-}list_1$: S_1;

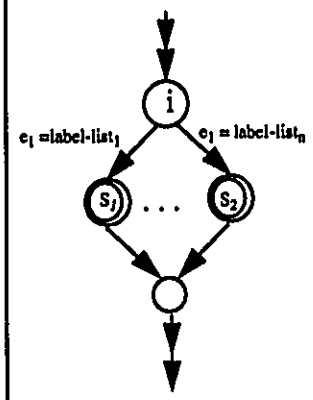
.

.

.

$label\text{-}list_n$: S_n

end;

Let j_1,..., j_n be the entry nodes of S_1,..., S_n, respectively. Edges (i, j_1),..., (i, j_n) have puses of each variable in the expression e_1.</td>
<td> 
</td>
</tr>
</table>

In a flowgraph, a definition, or a cuse takes place within a node. A puse takes place on an edge. The following tables shows the definitions and cuses in the example Program A in Figure 2.2.

TABLE 2.6 Defs and Cuses for Program A

Node	Variables defined	Variables cused
1	i	--
2	j	--
3	t	(input)

TABLE 2.6 Defs and Cuses for Program A

Node	Variables defined	Variables cused
4	n, x, y, z	(input)
5	s	x, y, z
6	i	i
8	j	j
9	(output)	n, x, y, z, s
11	(output)	j

The following table shows the puses in Program A.

TABLE 2.7 Puses for Program A

Edge	Variables pused
(7, 8)	s, y
(7, 10)	s, y
(10, 4)	i, t
(10, 11)	i, t

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

Data flow information in a program P is obtained from the flowgraph of P by associating with each node i the sets $cuse(i)$ = {variables which have cuses in node i} and $def(i)$ = {variables which have definitions in node i}, and associating with each edge (i, j) the set $puse(i, j)$ = {variables which have puses on edge (i, j)}. We also define sets of nodes $dcu(x, i)$ = {nodes j such that $x \in cuse(j)$ and there is a def-clear path with respect to x from i to j} and $dpu(x, i)$ = {edges (j, k) such that $x \in puse(j, k)$ and there is a def-clear path with respect to x from i to (j, k)}.

A *def-cuse association* is a triple (i, j, x) where i is a node containing a definition of x and $j \in dcu(x, i)$. A *def-puse association* is a triple $(i, (j, k), x)$ where i is a node containing a definition of x and $(j, k) \in dpu(x, i)$. *A def-use association* is a def-cuse association or a def-puse association. For the example program in Figure 2.2, the def-cuse associations and def-puse associations for each def of each variable is given in the following tables.

TABLE 2.8 Def-cuse Associations for Each Def of a Variable

Variable	Defined at node	Cused at node
i	1	6
j	2	8
j	2	11
x	4	5
y	4	5
z	4	5
n	4	9
x	4	9
y	4	9
z	4	9
s	5	9
i	6	6
j	8	8
j	7	11

TABLE 2.9 Def-puse Associations for Each Def of a Variable

Variable	Defined at node	Pused at edge
s	5	(7, 8)
s	5	(7, 10)
y	4	(7, 8)
y	4	(7, 10)
i	6	(10, 4)

Variable	Defined at node	Pused at edge
i	6	(10, 11)
t	3	(10, 4)
t	3	(10, 11)

We say that a def-use association is *feasible* (*executable*) if a def-clear path related to the association is a subpath of some executable complete path; otherwise, it is *infeasible* (*unexecutable*).

A path $(n_i, \ldots, n_j, n_k)$ is a *du-path* with respect to a variable x if n_i has a definition of x and either

a) n_k has a cuse of x and $(n_i, \ldots, n_j, n_k)$ is a def-clear simple path with respect to x, or

b) (n_j, n_k) has a puse of x and $(n_i, \ldots, n_j, n_k)$ is a def-clear loop-free path with respect to x.

2.2.2.3 Data Flow Oriented Coverage Criteria

Data flow oriented testing has been motivated to span the gap between all_paths and all_edges coverage [10]. A family of data flow oriented coverage criteria is defined in [22]. The criteria attempt to cover various combinations between defs and uses of variables. Roughly speaking, the family of data flow oriented coverage criteria is based on the requirement that the test cases execute def-clear paths from each node containing a definition of a variable to specified nodes containing cuses and edges containing puses of that variable. For each variable definition, it is required that all/some def-clear paths with respect to that variable from the node containing the definition to all/some of the uses/ cuses/puses reachable by some such paths be executed. These criteria are defined precisely in Table 2.10.

TABLE 2.10 Some Data Flow Oriented Coverage Criteria

Criterion	Association Required
All-defs	Some (i, j, x) such that $j \in dcu(x, i)$ or some $(i, (j, k), x)$ such that $(j, k) \in dpu(x, i)$.
All-cuses	All (i, j, x) such that $j \in dcu(x, i)$.
All-puses	All $(i, (j, k), x)$ such that $(j, k) \in dpu(x, i)$.
All-puses/some-cuses	All $(i, (j, k), x)$ such that $(j, k) \in dpu(x, i)$. In addition, if $dpu(x, i) = \emptyset$ then some (i, j, x) such that $(j, k) \in$ dcu (x, i). Note that since i has a definition of x, $dpu(x, i) = \emptyset \Rightarrow dcu(x, i) \neq \emptyset$.
All-cuses/some-puses	All (i, j, x) such that $j \in dcu(x, i)$. In addition, if $dcu(x, i) = \emptyset$ then some $(i, (j, k), x)$ such that $j \in dpu(x, i)$. Note that since i has a definition of x, $dcu(x, i) = \emptyset \Rightarrow dpu(x, i) \neq \emptyset$.
All-uses	All (i, j, x) such that $j \in dcu(x, i)$ and all $(i, (j, k), x)$ such that $(j, k) \in dpu(x, i)$.
All-du-paths	All du-paths from i to j with respect to x for each $j \in dcu(x, i)$ and all du_paths from i to (j, k) with respect to x for each $(j, k) \in dpu(x, i)$.

If variable x has a definition in node i, the all-defs criterion requires the test cases to exercise some def-clear path wrt x from node i to some node or edge at which the value assigned to x in node i is used. The all-uses criterion requires the test cases to exercise at least one such path to each such node and to each such edge. The all-du-paths criterion requires that all of the du-paths from node i to each such node and each such edge be exercised. The criteria all-puses, all-cuses, all-puses/some-cuses, and all-cuses/some-puses place emphasis on either cuses or puses. Note that any program has only finitely many def-use associations, so none of the data flow testing methods requires an infinitely many of test units. Figure 2.3 illustrates the relationships between control and data flow oriented test selection criteria. As we can see from Figure 2.3, every data flow oriented test selection criterion is stronger than all-edges criterion. To find out more about the

advantages of data flow oriented testing over control flow oriented testing, please refer to reference [10] for detail.

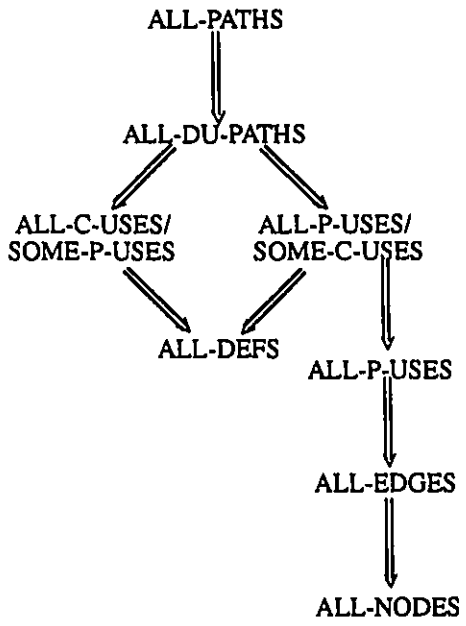


FIGURE 2.3 Relationship Among Test Selection Criteria

As a technical convenience, we assume that the subprogram or program being tested has no **goto** statements, no **with** statements, no variant records, no functions having **var** parameters, and no procedural or functional parameters. It would not be difficult to relax these assumptions [22]. We also assume that in every conditional statement, the boolean expression which determines the flow of control has at least one occurrence of a variable or a call to the function *eof* or to the function *eoln*.

For illustration purposes, we present the application of the all-uses criterion to the example Program A in Figure 2.2. The all-uses criterion requires that a set of test cases be selected to exercise each def-use association in a flowgraph at least once during testing. The test unit file required to be covered by the all-uses criterion for the example program in Figure 2.2 is given below:

TABLE 2.11 Test Unit File Required by All-uses Criterion for Program A

All Uses Coverage

Def-cuse Associations:

(i, 1, 6)	(j, 2, 8)	(j, 2, 11)	(x, 4, 5)
(y,4, 5)	(z, 4, 5)	(n, 4, 9)	(x, 4, 9)
(y, 4, 9)	(z, 4, 9)	(s, 5, 9)	(i, 6, 6)
(j, 8, 8)	(j, 8, 11)		

Def-puse Associations:

(s, 5, (7, 8))	(s, 5, (7, 10))
(y, 4, (7, 8))	(y, 4, (7, 10))
(i, 6, (10, 4))	(i, 6, (10, 11))
(t, 3, (10, 4))	(t, 3, (10, 11))

Number of Test Units: 22

2.3 Other Aspects of Data Flow Analysis of Pascal Programs

In section 2.2.2, the classification of variable occurrences in a flowgraph of a Pascal program as defs, cuses, and puses was defined. Here, we will discuss how data flow analysis handles function/procedure calls, structural variables like arrays and records as well as pointer variables and file variables.

2.3.1 Procedure/Function Calls

If a procedure/function definition contains only assignment statements, and has no local variable, the procedure/function is called *a simple procedure/function*, Otherwise, it is called *a complex procedure/function*. In this thesis, we consider only those programs that contain only simple procedure/function definitions. For the interpretation of a call to a complex procedure/function, the technique of extended flowgraph [30] can be used to merge the flowgraph of each called procedure/function into the flowgraph of main

program to generate test units for both intra-procedural and inter-procedural data and control flow. Interested reader may refer to [30] for more details.

In general, there are two types of interpretation of any call to a simple procedure. The first type of interpretation is based on the classification given by Rapps and Weyuker for the actual parameters of a procedure call in Pascal programs. That is, the parameter list of a procedure statement contains cuses of all actual pass-by-value and pass-by-reference parameters followed by defs of actual pass-by-reference parameters. In fact, this is the interpretation given in Table 2.5. Obviously, this classification is sufficient for the family of data-flow oriented coverage criteria reviewed in Section 2.2.2.3, since these criteria are based on individual def-use associations. Other data flow oriented coverage criteria [5, 11] require a more detailed classification since they are based on identification of chains of def-use associations. Thus, the second type of interpretation for the actual parameters of a procedure statement is based on the abstract definition of each procedure *pn* referred to in the Pascal program so that the manner each actual pass-by-reference parameter and nonlocal variable takes a value within *pn* can be determined. After substituting actual parameters for formal parameters, the use of pseudo assignment statements in the abstract definition of the called procedure realizes the second type of interpretation.

An abstract definition for a procedure determines how each pass-by-reference parameter takes a value in the called procedure. That is, for each formal pass-by-reference parameter, there exists a pseudo assignment statement of the form "*v*:= *expression*", where expression is defined over all or some of the formal parameters and possibly nonlocal variables of the called procedure. The same applies to each nonlocal variable updated in the called procedure. Therefore, each nonlocal variable updated in a called procedure must be defined by a pseudo assignment statement in the abstract definition of that procedure.

As in the case of call to a simple procedure, there is an abstract definition of each called user-defined function. It is assumed that in the definition of each called user-defined function, no nonlocal variable is updated and the formal parameters are all pass-by-value. An abstract definition of a called function defines the manner the function takes a value by an assignment statement of the form "*function_name* := *expression*", where expression is defined over the formal parameters of the function. Interested reader may refer to *Appendix B.1: Assumptions on Pasflow Input* for more detail.

2.3.2 Arrays, Records, Pointers and File Variables

Since it is not possible, in general, to determine the particular array element which is being defined or used in an occurrence of an array variable a, any definition of the variable $a[e]$ will consist of a cuse of each variable occurring in the expression e, followed by a definition of a. Any use of $a[e]$ will consist of a use of each variable occurring in e, followed by a use of a. For example, assignment statement '$y := x[a, b] + c$' has cuses of x, a, b, and c. In the assignment statement '$x\,[y, z] := 1$', y and z are cused, and x is defined. In a branch predicate '$x\,[y, z] >= 1$', y, z, and x are pused.

Another method to treat arrays is to consider each array element as a distinct data-flow object. Interested reader may refer to [9] for more details.

Each field of a record is treated as an individual variable. Any unqualified occurrence of a record is treated as an occurrence of each field of the record.

We will treat pointers purely syntactically, making no attempt to perform data flow analysis on dereferenced pointers. If p is a pointer variable, the dereference of p is expressed by $p^\wedge$. A definition of $p^\wedge$ consists of a cuse of p followed by a definition of $p^\wedge$; and a use of $p^\wedge$ consists of a use of p followed by a use of $p^\wedge$. Since it is not possible to

determine statically the memory location to which a pointer points, we will ignore the definitions and uses of $p^{\wedge}$.

Occurrences of file variables in I/O statements are ignored (Refer to Table 2.5).

2.3.3 A More Complex Example

In order to explain the flowgraph representation of programs with procedure statements and arrays, Figure 2.4 gives a more complex Pascal program example and its flowgraph.

```
program BubbleSort (input, output);
var switch: Boolean;
    n:  integer; {size of the array}
    i:  integer;
    temp:integer;

    procedure Exchange (var x, y: integer)
    {Switches x and y}
    begin
        temp := x; x := y; y := temp;
    end; {Exchange}

    begin {BubbleSort}
        switch := TRUE; n := 5;
        for i:= 1 to 5 do read(a[i]);
        while n >= 2 and switch
        begin
            switch := FALSE;
            for i:= 1 to n - 1 do
                if a[i] > a[i+1] then
```

```
            begin
                switch := TRUE;
                Exchange (a[i], a[i+1]);
            end
        n := n - 1;
    end
    writeln ('Array is sorted');
end. {BubbleSort}
```

Abstract Definition File

```
procedure Exchange (var x, y: integer)
temp := x; x := y; y := temp.
```

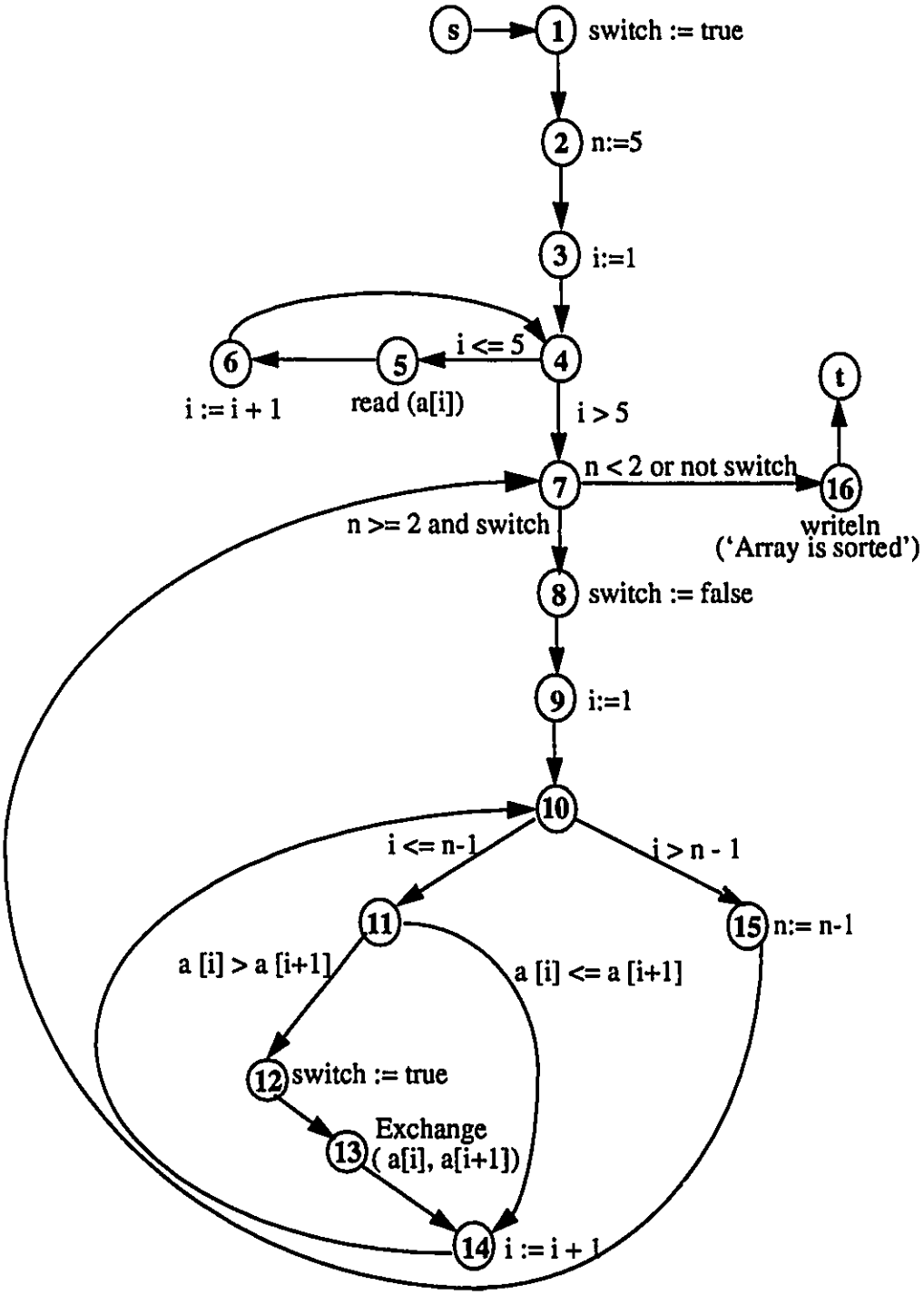


FIGURE 2.4 Program B and its Flowgraph

From Figure 2.4, we know that in procedure *Exchange (x, y)*, both *x* and *y* are pass_by_reference, *temp* is a global variable. From the definition of *Exchange (x, y)*, it is

obvious that *temp* is defined by a cuse of *x*; *x* is defined by a cuse of *y*; *y* is defined by a cuse of *temp*. The second interpretation of procedure/function calls uses these information which is contained in the abstract definition file. Based on this interpretation, in node 13, cuses of *a* and *i* are followed by definition of *temp*; cuses of *a* and *i* are followed by definition of *a*; and cuses of *i* and *temp* are followed by definition of *a*.

The first interpretation of procedure/function calls ignores the information contained in the abstract definition file. Based on this interpretation, on node 13, variables *a* and *i* are cused, and variable *a* is defined. Based on this interpretation, the following table shows the definitions, cuses and puses for each variable in the example Program B.

TABLE 2.12 Defs and Cuses for Program B

Node	Variables defined	Variables cused
1	switch	- -
2	n	- -
3	i	- -
5	a	i
6	i	i
8	switch	- -
9	i	- -
12	switch	- -
13	a	a, i
14	i	i
15	n	n

TABLE 2.13 Puses for Program B

Edge	Variable pused
(4, 5)	i
(4, 7)	i
(7, 8)	n, switch
(7, 16)	n, switch

Edge	Variable pused
(10, 11)	i, n
(10, 15)	i, n
(11, 12)	a, i
(11, 14)	a, i

Note that our model of data flow may not reflect the actual data flow in the subprogram being tested completely accurately. For example, we have made no attempt to perform any interprocedural data flow analysis, have ignored dereferenced pointers, and have ignored potential side effects. In an optimizing compiler, it is imperative that conservative assumptions be made about the flow of data, lest a code transformation which changes the semantics of the program be performed. In the context of data flow testing, however, such caution is not strictly necessary. On the other hand, it seems reasonable to expect that more accurate data flow analysis will force the selection of better test data.

Chapter 3

Path Expression and Complete Path Generation

3.1 Path Selection for a Test Unit

As stated in section 1.2, given a test unit, the first task for test case selection in the second approach of white box testing is to find a *test path* or *complete path* to cover each test unit identified in the given flowgraph. We say a test unit is *covered* by a complete path p when

- i) if the test unit is a node i and node i is in p;
- ii) if the test unit is an edge (i_1, i_2) and edge (i_1, i_2) is in p;
- iii) if the test unit is a def-cuse association (x, n_1, n_2) and there is a def-clear path from n_1 to n_2 wrt variable x, the def-clear path is a subpath of p.

We say *a complete path is feasible* if there exists a test input which causes the execution of the program to follow the complete path. If a test unit is not covered by any feasible complete path, then it is said infeasible.

There are usually many complete paths to cover a test unit. Sometimes, the number of complete paths covering a test unit may even be infinite and the length of some of these paths may be unbounded. In a flowgraph, every decision doubles the number of potential paths, and every loop multiplies the number of potential paths by the number of different iterations possible for the loop. If a routine has one loop, each pass through that loop (once, twice, three times, and so on) constitutes a different path through the routine, even though the same code is traversed each time.

The goal of path selection is to select feasible complete paths to cover every test unit identified in a flowgraph. Path selection can be manual or automatic as well as static or dynamic. Manual dynamic path selection requires a user to select the next statement whenever a branching node is encountered. This is very tedious, inefficient, and difficult. Manual static path selection requires a user to completely specify program paths before analysis is initiated. However, users tend to inadvertently select mainly unexecutable paths. In automatic static selection, paths are automatically selected prior to symbolic execution. This method is usually based on the flowgraph representation of the program, without additional semantic information. It has the same drawbacks as the previous two methods. Automatic path selection requires selecting all the feasible paths, and therefore, has the disadvantage of overwhelming the user with paths [26].

To reduce the significant computational effort wasted in analyzing infeasible paths, and to achieve the automation of test path selection, we are interested in characterizing the conditions under which any one of a set of paths, which are from a *source* node to a *target* node and cover a certain test unit, can be executed. In addition, we are interested in characterizing the result of executing such a path. We will represent a set of paths by a regular expression, which is called *path expression*. By symbolically analyzing a path expression, we can decide the feasibility of the set of paths presented by the path expression. Path expression representation also makes it possible to generate a set of complete paths to satisfy a certain control or data flow oriented coverage criterion.

Because some paths in a set of complete paths $\{p_i\}$ covering a test unit may be infeasible, our task includes two aspects: First, decide the feasibility of a test unit. Second, select a feasible complete path to cover the test unit. Our techniques involve representing the (possibly infinite) set of paths from program entry s to program exit t by a path expression, and applying a generalized form of symbolic execution (called *partial*

symbolic execution) to the path expression. If the path expression is determined infeasible, it means that the test unit which is covered by the path expression is infeasible. Otherwise, a complete path is generated from the path expression. By applying symbolic execution to the selected complete path, the feasibility of the complete path can be determined. If it is feasible, a test input can be generated from the path predicate obtained during symbolic execution. If infeasible, another complete path has to be generated from the path expression until a feasible complete path is found. If no feasible complete path can be found, the test unit will be declared infeasible.

In this chapter, the representation of path expressions and algorithms of generating path expressions and complete paths are discussed. Partial symbolic execution and symbolic execution techniques are discussed in chapter 4.

3.2 Representation of Path Expressions by Regular Expressions

We are interested in representing the (possible infinite) set of paths from program entry s to program exit t by a regular expression, which is called a *path expression.*

Let Σ be a finite alphabet disjoint from $\{\lambda, \varnothing, (,)\}$, where λ denotes the empty string, and $\varnothing$ denotes the empty set. A *regular expression* over Σ is any expression built under the following rules.

1. λ and $\varnothing$ are atomic regular expressions; for any $a \in \Sigma$, a is an atomic regular expression.
2. If R_1 and R_2 are regular expressions, then (R_1+R_2), $(R_1 \bullet R_2)$, and $(R_1)^*$ are *compound* regular expressions, where R_1 and R_2 are *sub-expressions.*

In a regular expression, + denotes choice, • denotes concatenation, and * denotes reflexive, transitive closure under concatenation. Thus each regular expression R over Σ represents a set $\sigma(r)$ of strings over Σ. $\sigma(r)$ is defined as follows:

3. $\sigma(\lambda) = \{\lambda\}$; $\sigma(\varnothing) = \{\varnothing\}$; $\sigma(a) = \{a\}$ for $a \in \Sigma$.

4. $\sigma(R_1+R_2) = \sigma(R_1) \cup \sigma(R_2) = \{w | w \in \sigma(R_1) \text{ or } w \in \sigma(R_2)\}$;

5. $\sigma(R_1 \bullet R_2) = \sigma(R_1) \bullet \sigma(R_2) = \{w_1w_2 | w_1 \in \sigma(R_1) \text{ and } w_2 \in \sigma(R_2)\}$;

6. $\sigma(R^*) = \cup^{\infty}_{k=0}\sigma(R)^k$, where $\sigma(R)^0 = \{\lambda\}$ and $\sigma(R)^i = \sigma(R)^{i-1} \bullet \sigma(R)$.

Two regular expressions R_1 and R_2 are *equivalent* if $\sigma(R_1) = \sigma(R_2)$. A regular expression R is *simple* if $R = \varnothing$ or R does not contain $\varnothing$ as an expression. We can transform any regular expression R into an equivalent simple regular expression by repeating the following transformations until none is applicable:

(i) replace any sub-expression of the form $\varnothing \bullet R_1$ or $R_1 \bullet \varnothing$ by $\varnothing$;

(ii) replace any sub-expression of the form $\varnothing + R_1$ or $R_1 + \varnothing$ by R_1;

(iii) replace any sub-expression of the form $\varnothing^*$ by λ.

We now define the syntax and semantics of path expressions in a given flowgraph $F = (V, E)$. A *path expression* from node v to node w in V is a simple regular expression r over V such that every string in $\sigma(r)$ is a path from node v to node w. Every sub-expression of a path expression is a path expression, and it can be determined as follows.

7. Let r be a path expression from node v to node w.

- If $r = r_1 + r_2$, then r_1 and r_2 are sub-path expressions from v to node w.
- If $r = r_1 \bullet r_2$, then there must be a unique node u such that r_1 is a sub-path expression from node v to node u and r_2 is a sub-path expressions from node u to node w.

- If $r = r_1^*$, then $v = w$ and r_1 is a sub-path expressions from node v to node v.

A path expression r actually represents a set of paths, which we refer as PATHS(r). A path expression r is *valid* if every element of PATHS(r) is actually a path in F.

3.3 Path Expression Derivation from Test Units

As stated before, a test unit can be either a node, an edge, or a def-use association. Based on Tarjan's path expression derivation algorithm [7], path expression derivation is described below where the aim is to derive a complete path expression for a given test unit. A path expression is called *complete path expression* if it starts from the entry node s and ends at the exit node t.

For flowgraph $F = (V, E)$,

A) if the test unit is a node n_i

i) derive the path expression r_1 from the entry node s to node n_i. If $n_i = s$, then $r_1 = s$;

ii) derive the path expression r_2 from node n_i to the exit node t.

iii) the complete path expression r to cover the test unit node n_i is $r = r_1 \bullet r_2$.

B) if the test unit is an edge (n_i, n_j)

i) derive the path expression r_1 from the entry node s to node n_i. If $n_i = s$, then $r_1 = s$.

ii) the path expression for the edge (n_i, n_j) is $r_2 = n_i \bullet n_j$

ii) derive the path expression r_3 from node n_j to the exit node t.

iv) the complete path expression r of the test unit is $r = r_1 \bullet r_2 \bullet r_3$.

C) In the case that the test unit is a def-use association (x, n_i, n_j) or $(x, n_i, (n_j, n_k))$, we introduce a new concept called *def-clear path expression.* A path expression r is

called a *def-clear path expression with respect to* (*wrt*) *a variable x* from node n_i to node n_k, if r represents all the def-clear paths with respect to variable x from node n_i to n_j; a path expression r is called a *def-clear path expression wrt variable x* from node n_i to edge (n_j, n_k), if r represents all the def-clear paths with respect to variable x from node n_i to edge (n_j, n_k). Based on Tarjan's path expression derivation algorithm [7], we developed an algorithm to generate a def-clear path expression from node n_i to node n_j with respect to variable x.

[Algorithm 1]

Def-clear path expression derivation from node n_l to node n_j wrt x in $F = (V, E)$

[1] Let S represents a set of nodes. Initialize $S = \varnothing$.

[2] For each node $n_k \in V$, if x is defined at node n_k, then $S = S \cup \{n_k\}$.

[3] Generate path expression r from node n_l to node n_j using Tarjan's path expression derivation algorithm.

[4] Repeat for each node n_k in S

if n_k is not a node in path expression r, do nothing.

Otherwise, apply the following rule to r to take out every occurrence of node n_k:

Assume that any one of r_1, r_2, r_3 and r_4 is $\varnothing$ or a sub-path expression of r.

- If n_k occurs in a sub-path expression $r' = r_1 \bullet n_k \bullet r_2$ of r, taking out n_k results in $r' = \varnothing$.
- If n_k occurs in a sub-path expression $r' = r_1 + n_k + r_2$ of r, taking out n_k results in $r' = r_1 + r_2$.
- If n_k occurs in a sub-path expression $r' = r_3 \bullet (r_1 \bullet n_k \bullet r_2)^* \bullet r_4$ of r, taking out n_k results in $r' = r_3 \bullet r_4$.
- If n_k occurs in a sub-path expression $r' = (r_1 + n_k + r_2)^*$ of r, taking out n_k results in $r' = (r_1 + r_2)^*$.

[5] The resulting path expression r is the def-clear path expression from node n_i to node n_j wrt x.

In case that the test unit is a def_cuse association (x, n_i, n_j), where n_i and $n_j \in V$, x is a variable defined at node n_i and cused at node n_j, and there is a def_clear path from node n_i to node n_j with respect to x, the process of deriving path expression r is as follows:

i) derive path expression r_1 from entry node s to node n_i.

ii) derive def-clear path expression r_2 from node n_i to node n_j with respect to variable x.

iii) derive path expression r_3 from node n_j to the exit node t.

iv) the complete path expression r of the test unit is $r = r_1 \bullet r_2 \bullet r_3$.

If the test unit is a def-puse association $(x, n_i, (n_j, n_k))$, where n_i, n_j and $n_k \in V$, $(n_j, n_k) \in E$, x is a variable defined at node n_i and pused at edge (n_j, n_k), and there is a def_clear path from node n_i to edge (n_j, n_k) with respect to x, the process of deriving path expression r is as follows:

i) derive path expression r_1 from entry node s to node n_i.

ii) derive def-clear path expression r_2 from node n_i to node n_j with respect to variable x.

iii) the path expression for the edge (n_j, n_k) can be represented as $r_3 = n_j \bullet n_k$.

iv) derive path expression r_4 from node n_k to the exit node t.

v) the complete path expression r of the test unit is $r = r_1 \bullet r_2 \bullet r_3 \bullet r_4$.

The following is an example of the path expression for a def-use association. The flowgraph is shown in Fig 3.2.

TEST UNIT:

def-puse association (done, 1, (2, 8))

path expression from s to 1:

1

def-clear path expression from 2 to 8 wrt done:

(2 3 4 5 7) * 2 8

path expression from 8 to t:

8

the path expression for the test unit:

1 (2 3 4 5 7) * 2 8

FIGURE 3.1 An Example of Path Expression Generation

3.4 Application of Path Expressions in Data Flow Testing

Most test case selection systems do not use path expressions. They generate a complete path directly from a test unit, which can be a node, an edge or sometimes a def_use association. However, as stated earlier, the automatic static derivation of complete paths yields a considerable number of complete paths which are unexecutable. As we shall see below, even very simple programs often have some infeasible test units such as def-use associations. All of the complete paths generated for an infeasible test unit will also be infeasible. Thus, it is important to decide whether a given test unit is feasible before a complete path is generated for the test unit. Path expressions can be used to control the complete path selection.

3.4.1 Using Path Expressions to Identify Infeasible Test Units

Given a test unit, for example, a def-use association (v, d, u), we would like to determine whether there is an executable complete path covering it. Thus, we are interested in examining the (possibly infinite) set of paths which begin at the entry node, have a def-clear subpath with respect to variable v form node d to node u, and end at the exit node. As described above, we can derive a path expression r which represents that set of paths. By applying partial symbolic execution, which is described in chapter 4, a set of constraints in the form of a path predicate can be derived from the path expression r. By examining the constraint set, the feasibility of the test unit can be determined. If it is equivalent to False, then we know that none of the complete paths represented by path expression r are executable, and the def-use association (v, d, u) is definitely infeasible. If it is not equivalent to false then the association may or may not be feasible. The following example illustrates how to use a path expression to identify an infeasible test unit.

Consider the program whose flowgraph is shown in Figure 3.2. We assume that **done** is not redefined in statements S1, S2 or S3. Consider a def-use association (done, 1, (2, 8)), which arises from the fact that **done** is defined by the assignment statement **done:= false** in node 1, used in the exit condition from the loop, and there is a def-clear path with respect to **done** from node 1 to edge (2, 8).

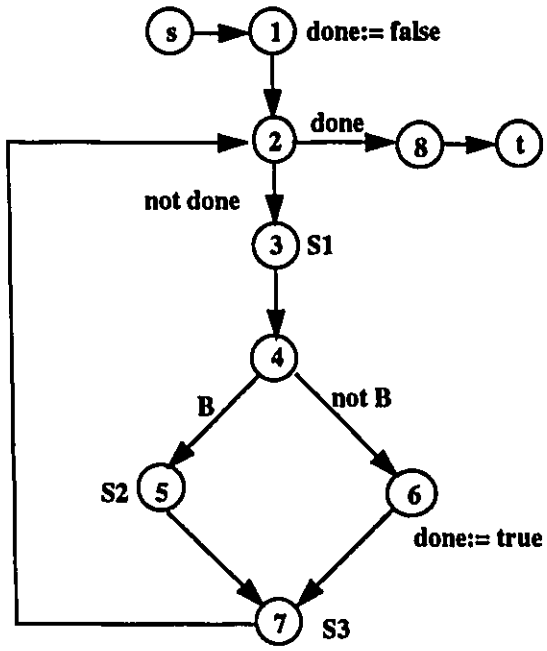


FIGURE 3.2 Program C: Looping until *done*

A path expression can be generated to represent the set of all paths covering the test unit. According to the algorithms defined above, the process of generating path expression p = 1 (2 3 4 5 7) * 2 8 for test unit: def-puse association (done, 1, (2, 8)) is shown in Figure 3.1.

Now, we obtain the path express p = 1 (2 3 4 5 7) * 2 8, which represents the set of complete paths covering the def-cuse association. Next, by applying partial symbolic execution, we obtain partial path predicate PPPp: (false = true), which cannot be satisfied. Thus, the test unit is declared infeasible.

3.4.2 Application of Path Expressions to Complete Path Selection

Path selection and loops have always been an area that poses problems for automatic test case selection. Methods of path selection employed in most existing

systems are based on simple strategies only, such as taking the true branch first or generating the shortest path. Most of the complete paths selected by applying these strategies are usually infeasible and the problem of identifying feasible paths that include the non-covered test units from the a set of paths remains unsolved. Path expression characterizes every branch and every loop contained in the program from the entry to the exit. In order to achieve the automation of complete path selection, path expression is employed to select a feasible complete path by analyzing each branch and each loop. Details are discussed in the next section.

We use path expressions in the proposed system to achieve the automation of test case construction by identifying infeasible test units and controlling the selection of complete paths. The system flow of applying path expression and symbolic evaluation techniques to determine the feasibility of test unit and to control complete path selection is shown in Figure 3.3.

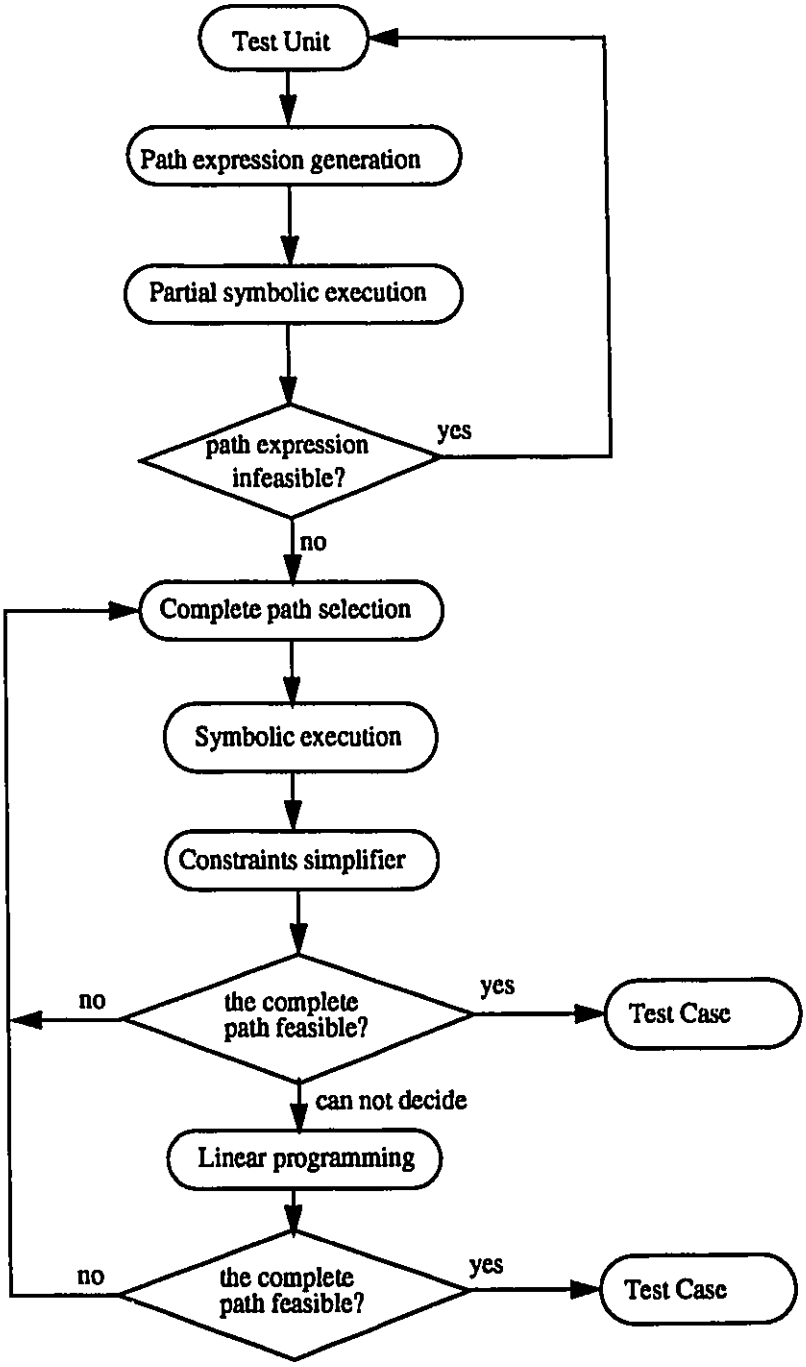


FIGURE 3.3 System Flow of Identifying Infeasible Test Units

3.5 Complete Path Generation

After a path expression is determined not infeasible, the next task is to select a feasible complete path from the path expression. Generally, the number of complete paths represented by a path expression is very large or infinite and the path length is sometimes unbounded. However, we need a feasible complete path among those represented by a feasible path expression. Thus, a complete path is selected from the path expression and by applying symbolic execution to the selected complete path, the feasibility of the complete path can be determined. If it is feasible, a test input can be generated from the path predicate obtained during symbolic execution. If infeasible, another complete path has to be generated from the path expression until a feasible complete path is found. If no feasible complete path can be found, the test unit will be declared infeasible.

First, we will discuss the numbering of "*" and "+" in the path expression. A path expression, which characterizes every branch and every loop contained in the program from the entry to the exit, can be represented by a binary tree, where every leaf node is a node in the path expression and every nonleaf node is one of "• ", "+", or "*". Consider the complete path p = 1 2 (3 (4 5 + 4 6) 7 2)* 8 in Program C (Figure 3.2), which covers the test unit edge (2, 8). Put back the concatenation "• " implied between any two nodes, the path expression becomes:

$$p = 1 \bullet 2 \bullet (3 \bullet (4 \bullet 5 + 4 \bullet 6) \bullet 7 \bullet 2)^* \bullet 8$$

It can be expressed by the following binary tree:

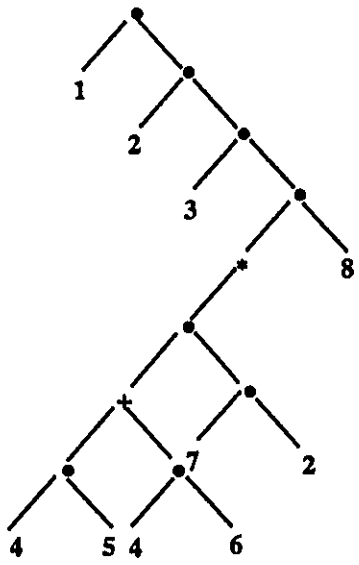


FIGURE 3.4 A Path Expression Expressed as a Binary Tree

In order to distinguish the positions of "*" and "+" in the path expression for use in the algorithm discussed afterwards, we number the "*" and "+" respectively in the order of pre-order-traversal of the tree. After numbering, the original path expression becomes the one below, where the subscript 1 of the "+" and the superscript 1 of the "*" indicate that they are the first "+" and "*" symbol traversed:

$$1\ 2\ (3\ (4\ 5\ +_1\ 4\ 6)\ 7\ 2)^{*\ 1}\ 8$$

The objective of complete path selection is to exhaust all the complete paths from the path expression until we find a feasible complete path. A path expression represents a very large set of paths when a "*" (a loop) is contained in the expression. It is impractical to exhaust all of the paths. We must find a way to reduce this complexity. First, we limit the number of iterations of each loop to 2 times, i.e., the sub-path-expression under a "*" can be 0, 1 or 2. This reduces the number of paths represented by a path expression with loops. But it is not enough by doing this only, if we select each side of every choice in the

path expression, for a path expression containing 10 "*" and 20 "+", the maximum number of complete paths is $3^{10} * 2^{30}$, and a lot of duplicated paths may be generated. When testing communication protocols, we found that there always exist many "+" symbols in a complex path expression, and they are often nested. Setting an upper bound for the repeat times when handling "+" helps to reduce the duplicates. In our algorithm, an upper bound is employed to limit the different complete paths generated by the combination of the choices, and random numbers are used to decide whether the left or right side of a choice is to be included in the complete path. By testing communication protocols, we found very few duplicate paths are generated and a feasible complete path can be selected quickly. The algorithm is described in detail as follows:

[Algorithm 2]

Complete_path_selection

begin

[1] Pre-order traverse the tree structure of the path expression and number the "+" and "*" respectively. Let *star_num* be the total number of "*" and *plus_num* be the total number of "+".

[2] for i := 1 to 3^{star_num}

```
begin { for i := 1 to 3 star_num}

{The following for loop is to set the iteration number, which can be 0,
1, or 2, for the kth star, and store it in array star[k]}
temp := i;
for k := 1 to star_num do
    begin
    star[k] := temp mod 3;      temp := temp div 3;
    end

{The following if statement is to set an upper bound plusb for the
iteration number when handling "+"}
if 2^plus_num <= 128
then plusb := 128
```

```
        else plusb := 2^plus_num;

        {The following for loop selects a complete path p}
        for j := 1 to plusb do
            begin
            pre-order-traverse the tree structure of the path expression
            according to the following rule:
            .. When a leaf-node or "•" is encountered, add it into the
               complete path p directly.
            .. When *^K is encountered, if star[k] = 0, do not traverse the
               sub-tree; if star[k] = 1, traverse the sub-tree once; if star[k] =
               2, traverse the sub-tree twice.
            .. When +_K is encountered, randomly choose the left sub-tree
               or the right sub-tree to traverse.
            end
        end { for i := 1 to 3 star_num }

    end. {Complete_path_selection}
```

In the above algorithm, the sub-path expression under every star will have at least 3^{star_num-1} chances to be traversed zero time, once and twice, respectively. Obviously, the time complexity (i.e. number of complete paths generated) of the above algorithm is $3^{star\_num} * \min(2^{plus\_num}, 128)$.

This algorithm is found efficient in selecting feasible complete paths. In the case that no feasible complete path is selected after executing the algorithm, a feasible subpath starting from the entry node to cover the test unit will be generated by backwards searching the flowgraph from the test unit. A complete path expression is then generated by extending the subpath to the exit node of the flowgraph. By applying the algorithm to this path expression, a feasible complete path can be easily generated. If no feasible subpath can be generated to cover the test unit, the test unit is considered infeasible.

Chapter 4

Symbolic Evaluation

Once a path or path expression is generated, symbolic evaluation techniques are used to generate path predicates, which are conjunctions of branch predicates expressed in terms of program input variables. Input data satisfying a path predicate will result in the execution of the corresponding path. Symbolic evaluation of a program does not involve the execution of the program. It is a method of symbolically defining data that forces program paths to be executed.

The goal of symbolic evaluation is to derive, from the program text, the algebraic descriptions for the conditions to be satisfied and the outputs to be generated by the execution of a given path or a path expression. Symbolic evaluation is based on the flowgraph of a program. It traverses the flowgraph from an entry point, along a particular path or a path expression to an exit point. During a path or path expression traversal, each input variable is given a symbolic value rather than an actual value. Thereafter, each assignment statement is evaluated so that it is expressed in terms of input variables and constants. The results are a set of symbolic output expressions (one expression per output) and a set of path predicates.

Basically, there are three kinds of symbolic evaluation techniques: symbolic execution (SE), global symbolic execution (GSE) and partial symbolic execution (PSE). This chapter introduces and compares the three symbolic evaluation techniques. Algorithms are provided for SE and PSE.

For the discussions in this chapter, we assume for now that the path being symbolically executed contains no procedure or function calls. As we discussed in chapter

2, since the node corresponding to the procedure call can be replaced by a particular path through the called procedure and a function call can be replaced by the corresponding path through a user-defined function, this assumption is reasonable.

4.1 Symbolic Execution (SE)

The goal of symbolic execution (SE) is to derive, from the program text or flowgraph, an algebraic description of the function corresponding to a given path [6]. Typically, this description consists of two parts: a predicate called the *path predicate*, which describes the conditions under which the path can be executed, and a symbolic expression called the *output computation*, which describes how execution of the path forms the output.

Given a path $p = (n_1, \ldots, n_k)$, a set of input variables $V_1, \ldots, V_m$, (where m is the number of visible input variables), symbolic execution computes two expressions. The path predicate, $PP_p(V_1, \ldots, V_m)$ describes the conditions under which p will be executed. That is, control will follow path p if and only if at the top of node n_1, input variables satisfy PP_p. The output computation $OUT_p(V_1, \ldots, V_m)$ describes the output symbolic expression caused by the execution of path p. That is, after the execution of path p, every output in the path p can be expressed by the input variables and their relationship satisfies OUT_p. The result of symbolic execution can be summarized in a single predicate $SE_p = PP_p(V_1, \ldots, V_m) \cap OUT_p(V_1, \ldots, V_m)$.

There are two approaches to symbolically executing a path: forward expansion and backward substitution. *Forward expansion* commences at the start of the path and works towards the end, while *backward substitution* is less obvious and commences at the end of the path and works towards the start. But backward substitution is easier to implement, and it is used in this section to illustrate the process of symbolic execution.

In backward substitution, first a path p is reversed, PP_p is set to TRUE and OUT_p is set to $\varnothing$. Then, each node and each edge encountered on the path p is symbolically executed. While symbolically executing a node, the following types of statements are encountered: input, assignment, and output. The following describes how each statement is symbolically executed:

1. If an output statement *output S* is encountered, expression S is concatenated into OUT_p, i.e. $OUT_p = OUT_p \cup S$;

2. If an assignment statement $V :=$ <*expr*> is encountered, two substitutions take place. Variable V in PP_p and OUT_p is substituted by its current expression <*expr*>. For example, assuming before the substitution, $OUT_p = A + X$ and $PP_p = (A - 1 < 3 * B) \cap (A > 0)$. If we encounter assignment statement $A = B + C$, symbol A in OUT_p and PP_p, is replaced by symbol '$B + C$'. The result, $OUT_p = B + C + X$ and $PP_p = (B + C - 1 < 3 * B) \cap (B + C > 0)$.

3. If an input statement *input S* is encountered, the variable S in OUT_p and PP_p is marked as an input variable.

In backward substitution, after symbolically executing a node n_i $(1 < i)$, one needs to symbolically execute the edge (n_{i-1}, n_i) by simply conjoining the predicate on the edge (or branch) (n_{i-1}, n_i) to the path predicate PP_p. For example, if a predicate @ is encountered on the edge, the predicate is conjoined to the PP_p, i.e. $PP_p = PP_p \cap @$.

The result of a symbolic execution is a complex expression $SE_p = PP_p(V_1, \ldots, V_m) \cap OUT_p(V_1, \ldots, V_m)$. Once the symbolic execution of the path p is complete, PP_p has to be assessed for contradictions among the component conjuncts. If a contradiction is found, then the path p is declared infeasible; otherwise it may be feasible. OUT_p gives the symbolic expressions for the outputs. For a program without any data flow abnormality,

the result of symbolically executing a complete path p, OUT_p and PP_p can always be represented as functions defined over the input variables of the program.

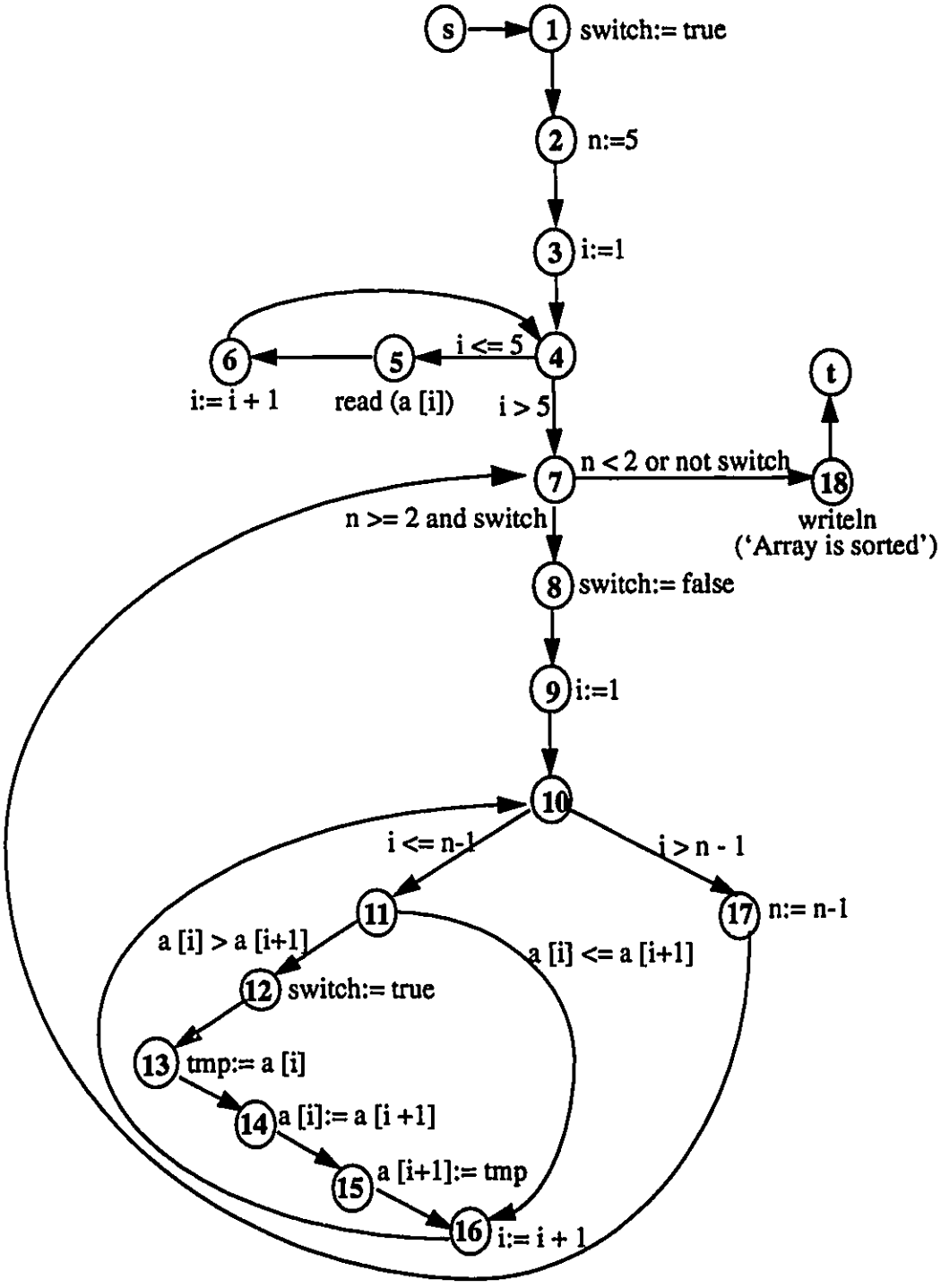


FIGURE 4.1 Flowgraph of Bubblesort Program

We now present an example of symbolic execution. Consider the bubblesort program shown in Figure 4.1. Let p be a complete path

$$p = s \bullet 1 \bullet 2 \bullet 3 \bullet 4 \bullet 7 \bullet 8 \bullet 9 \bullet 10 \bullet 17 \bullet 7 \bullet 18 \bullet t$$

Here we use the example to illustrate backward substitution symbolic execution. In order to use backward substitution, we first reverse the path p to p' and eliminate s and t, and get $18 \bullet 7 \bullet 17 \bullet 10 \bullet 9 \bullet 8 \bullet 7 \bullet 4 \bullet 3 \bullet 2 \bullet 1$. We then symbolically execute the nodes and edges starting from node 18, then 7, 17, ..., until 1.

1. Initialize PP_p = TRUE, and $OUT_p = \varnothing$.

2. In node 18, output statement 'Writeln ('Array is sorted')' is encountered. Concatenating the constant into OUT_p, we get OUT_p= 'Array is sorted'. Symbolically execute edge (7, 18). Conjoining the predicate '$n < 2$ or not *switch*' into PP_p, we get $PP_p = (n < 2$ or not *switch*).

3. No statement in node 7 and no predicate for edge (17, 7).

4. In node 17, assignment statement $n := n - 1$ is encountered. Substituting n as $n - 1$ in PP_p and OUT_p, we get $PP_p = (n - 1 < 2$ or not *switch*) and OUT_p = 'Array is sorted'. Symbolically execute edge (10, 17). A new branch predicate $i > n$-1 is conjoined into PP_p, and we get $PP_p = (n - 1 < 2$ or not *switch*) $\cap$ $(i > n$-1).

5. No statement in node 10 and no predicate for edge (9, 10).

6. In node 9, an assignment statement i :=1 is encountered. Replacing i by '1' in PP_p and OUT_p, we get $PP_p = (n - 1 < 2$ or not false) $\cap$ $(1 > n$-1) and OUT_p = 'Array is sorted'. No predicate on edge (8, 9).

7. In node 8, an assignment statement *switch*:= false is encountered. Replacing *switch* by 'false' in PP_p and OUT_p, we get PP_p = (n - 1 < 2 or not false) $\cap$ (1 > n-1) and OUT_p = 'Array is sorted'. Symbolically execute edge (7, 8). Branch predicate 'n >=2 and *switch*' is encountered. Conjoining it into PP_p, we get PP_p = (n - 1 < 2 or not (false)) $\cap$ (1 > n-1) $\cap$ (n >= 2 and *switch*).

8. No statement in node 7. Symbolically execute edge (4, 7). Conjoining the predicate (i > 5) in edge (4, 7) into PP_p, we get:

   PP_p = (n - 1 < 2 or not false) $\cap$ (1 > n-1) $\cap$ (n >= 2 and *switch*) $\cap$ (i > 5).

9. No statement in node 4 and no predicate on edge (3, 4).

10. Symbolically execute node 3. Substituting i by 1 in PP_p and OUT_p, we get:

    PP_p = (n - 1 < 2 or not false) $\cap$ (1 > n-1) $\cap$ (n >= 2 and *switch*) $\cap$ (1 > 5)

    OUT_p = 'Array is sorted'.

    No predicate on edge (2,3).

11. Symbolically execute node 2. Substituting n by '5' in PP_p and OUT_p, we get:

    PP_p = (5 - 1 < 2 or not false) $\cap$ (1 > 5 -1) $\cap$ (5 >= 2 and *switch*) $\cap$ (1 > 5)

    OUT_p = 'Array is sorted'.

    No predicate on edge (1, 2).

12. Symbolically execute node 1. Substituting '*switch*' by 'true' in PP_p and OUT_p, we get:

    PP_p = (5 - 1 < 2 or not (false)) $\cap$ (1 > 5 -1) $\cap$ (5 >= 2 and true) $\cap$ (1 > 5);

    OUT_p = 'Array is sorted'.

From the above PP_p, we can easily find PP_p is equivalent to False, it means that the path p = 1• 2•3•4•7•8•9•10•17•7•18 is infeasible.

4.2 Global Symbolic Execution (GSE)

Symbolic execution is limited by the fact that it can handle only a single path at a time. Global symbolic execution [10][11], on the other hand, attempts to derive an algebraic expression summarizing the effect of an entire program unit, i.e., the effect of executing any path through the whole program. The difficulty lies in the fact that programs usually have conditional statements and loops, thus may have a large or even infinite number of paths.

Global symbolic evaluation intends to deal with this problem by deriving conditional predicates describing conditional statements, and deriving and solving recurrence relations describing loops. For example, the statement 'for i := 1 to 5 do {null}' would give rise to the recurrence relation

$$i_0 = 1$$
$$i_{k+1} = i_k + 1$$

in which i_{k+1} represents the value of i after k iterations of the loop. Solving the recurrence relation yields $i_k = k$. Unfortunately, such computations become substantially more complicated when loops involve complicated series of assignments, nested conditions, and procedure or function calls.

Earlier work on global symbolic evaluation recognized that more research was needed before the technique could be practically used. Little if any progress has been made in overcoming these difficulties. In fact, it is unlikely that any substantial progress can be made for global symbolic evaluation, since, in essence, global symbolic evaluation amounts to trying to perform program verification without the benefit of inductive assertions.

4.3 Partial Symbolic Execution (PSE)

It is shown in the previous sections that symbolic execution is not a sufficiently general technique for many applications and that global symbolic execution is not a realistic alternative. Another symbolic evaluation technique, partial symbolic execution (PSE) of path expressions is described in this section. Like the global symbolic execution, it can treat infinite sets of paths, and like symbolic execution, it is reasonably efficient. In order to achieve this, partial symbolic execution sacrifices some information of the program being symbolically evaluated, thus derives a *partial* characterization of a set of paths.

Partial symbolic execution of a sub-path-expression which represents a single path is the same as symbolic execution of that path. Some information is sacrificed when evaluating a sub-path-expression which represents more than one path (such as a loop): namely, any variable which is potentially redefined on any of those paths is given a fresh symbolic variable as its value. This amounts to saying that we do not know anything at all about the value of such a variable when control reaches the target of that sub-path-expression.

More precisely, given a path expression r, a set $V_1, \ldots, V_m$, (where m is the number of input variables to the program), partial symbolic execution computes two predicates: *the partial path predicate*, $\mathrm{PPP}_r(V_1, \ldots, V_m)$, which describes the necessary conditions for the execution of any paths represented by r; *the partial output computation* $\mathrm{POUT}_r(V_1, \ldots, V_m)$, which describes the output symbolic expressions upon executing any of the paths represented by r. The result of partial symbolic execution can be summarized in a single predicate $\mathrm{PSE}_r(V_1, \ldots, V_m) = \mathrm{PPP}_r(V_1, \ldots, V_m) \cap \mathrm{POUT}_r(V_1, \ldots, V_m)$.

Let r be a valid path expression. Intuitively, PPP_r will partially describe the common features of the paths in PATHS(r), and $POUT_r$ will partially describe the common features of output symbolic expressions corresponding to paths in PATHS(r).

Let r_0 be a sub-path-expression of r consisting only of nodes, •, λ, and (). Then PATHS(r_0) consists of a single path π. On the other hand, if r_0 contains + or * then PATHS(r_0) will have more than one element. Unlike global symbolic execution, partial symbolic execution will not attempt to consider all of the details of each of the (possibly infinitely many) elements of PATHS(r_0). Rather, it will assign a new symbolic value to any variable which has a definition on at least one path in PATHS(r_0), then we will treat the value of that variable as being unknown after partially symbolically evaluating r_0.

Partial symbolic execution is first introduced by Phyllis G. Frankl in [6]. Derivation of the partial path predicate of a path expression r is similar to derivation of the path predicate of a single path. There are also two approaches to the partial symbolic execution of a path expression: forward expansion and backward substitution. Unlike what is done in [6], we implement partial symbolic execution by backward substitution, which we found that it simplifies the algorithm for PSE and makes the implementation easier. Partial symbolic execution also needs to symbolically execute the node and symbolically execute the edge which both follow the same rules as in symbolic execution. The only difference is when a loop head symbol is encountered, a new symbolic value will be assigned to every variable which is potentially redefined in that loop.

To illustrate partial symbolic execution, we will use the bubblesort program in Figure 4.2. We would like to concentrate on covering a test unit edge (7, 18) required by all-edges coverage. First, the path expression is generated:

$$r = 1 \bullet 2 \bullet 3 \bullet (4 \bullet 5 \bullet 6)^* \bullet 4 \bullet (7 \bullet 8 \bullet 9 \bullet (10 \bullet 11 \bullet (12 \bullet 13 \bullet 14 \bullet 15 \bullet 16 + 16))^* \bullet 10 \bullet 17)^* \bullet 7 \bullet 18$$

which represents the set of all paths from the entry node to the exit node and covers the required test unit. To make it clear we will take a sub-path-expression of r which is

$$r = 1\bullet 2\bullet 3\bullet 4\bullet(7\bullet 8\bullet 9\bullet(10\bullet 11\bullet(12\bullet 13\bullet 14\bullet 15\bullet 16+16))^{*}\bullet 10\bullet 17)^{*}\bullet 7\bullet 18$$

By using loop header, the path expression r call be expressed as follows:

$$r = 1\bullet 2\bullet 3\bullet 4\bullet H7\bullet 7\bullet 18$$

Here, $H7 = (7\bullet 8\bullet 9\bullet(10\bullet 11\bullet(12\bullet 13\bullet 14\bullet 15\bullet 16+16))^{*}\bullet 10\bullet 17)^{\wedge}$

H7 represents the loop headed by node 7, Obviously, r represents the infinite set of paths which start at node 1, go through nodes 2, 3 and 4, take an arbitrary path through the loop headed by node 7, then go to nodes 7 and 18.

In order to perform backward substitution, we have to reverse the path expression r to $r' = 18\bullet 7\bullet H7\bullet 4\bullet 3\bullet 2\bullet 1$ first. We also need to initialize PPP_r = TRUE, and $POUT_r = \varnothing$. Then start partial symbolic execution by symbolically executing the nodes and edges starting from node 18.

1. In node 18, output statement 'writeln ('Array is sorted')' is encountered. Concatenating the constant into $POUT_r$, we get $POUT_r$= 'Array is sorted'. Symbolically execute edge (7, 18). Conjoining the predicate '$n < 2$ or not *switch*' into PPP_r, we get $PPP_r = (n<2$ or not *switch*).

2. No statement in node 7 and no predicate on edge (17, 7).

3. Symbolically execute loop symbol H7. This is where partial symbolic execution differs from symbolic execution. Note that sub-expression H7 represents an infinite set of paths. Rather than making any attempt to deduce the details of the loop, we simply assign a new symbolic value to each variable which is potentially

redefined within the loop. Since H7= (7•8•9•(10•11•(12•13•14•15•16+16))* •10•17)*, from Figure 4.2, it is obvious that variable *a* is modified by assignment statements *a*[*i*] := *a*[*i*-1] and *a*[*i*+1] := *temp*, *n* is modified by *n* := *n*-1, and *switch* is modified by *switch* :=true. Each of these variables is given a fresh symbolic value, a^1 is given to *a*, n^1 is given to *n*, and $switch^1$ is given to *switch*. Replacing the variables in PPP_r and $POUT_r$, we get

$PPP_r = (n^1 < 2$ or not $switch^1)$.

$POUT_r$= 'Array is sorted'.

4. No statement in node 7. Symbolically execute edge (4, 7). Conjoining the predicate ($i > 5$) on edge (4, 7) into PPP_r, we get

   $PPP_r = (i > 5) \cap (n^1 < 2$ or not $switch^1)$.

5. No statement in node 4 and no predicate on edge (3, 4).

6. Symbolically execute node 3. Substituting *i* by 1 in PPP_r and $POUT_r$, we get:

   $PPP_r = (1 > 5) \cap (n^1 < 2$ or not $switch^1)$.

   $POUT_r$ = 'Array is sorted'.

   No predicate on edge (2,3).

7. Symbolically execute node 2. Since *n* is neither in PPP_r nor in $POUT_r$, PPP_r and $POUT_r$ are the same as before.

   No predicate on edge (1, 2).

8. Symbolically execute node 1. Since *switch* is neither in PPP_r nor in $POUT_r$, PPP_r and $POUT_r$ are the same as before.

Conjoining PPP_r and $POUT_r$ gives us the partial symbolic execution predicate:

PSE_r = (output('Array is sorted')) $\cap$ ((1 > 5) $\cap$ ($n^1 < 2$ or not $switch^1$))

From PPP_r we can easily deduce that PPP_r = false. It means that the path expression r = 1•2•3•4•(7•8•9•(10•11•(12•13•14•15•16+16))*•10•17)*•7•18 is infeasible.

Note that in the above example, when partially symbolically executing the loop head symbol H7, new symbolic values have been given to all redefined variables, e.g., a^1 is given to a, n^1 is given to n, and $switch^1$ is given to *switch*, because each of these variables is potentially redefined in the loop headed by node 7.

4.4 Comparison of Symbolic Evaluation Techniques

The following relationships between the symbolic execution, partial symbolic execution, and global symbolic execution are derived from their definitions.

For a path expression r from node v to node w in a flowgraph,

1. For each $p \in$ PATHS(r), $SE_p(v, w) \Rightarrow PSE_r(v, w)$.
2. $GSE_r(v, w) \Rightarrow PSE_r(v, w)$.
3. If PATHS(r) = $\{p\}$ then $SE_p(v, w) \equiv PSE_r(v, w) \equiv GSE_r(v, w)$.

If PATHS(r) consists of a single path, the three methods are equivalent. Global symbolic execution is more powerful than partial symbolic execution, but partial symbolic execution is much more practical, being essentially no more difficult than symbolic execution.

We now compare the cost of the three symbolic execution techniques. The cost of symbolically executing a path p (without simplifying the resulting expression) is linear in the sum of the lengths of the expressions occurring in the nodes and on the edges of p. Since the length of each expression is bounded by the length of the program P, $|P| \cdot |p|$

gives a crude upper bound on the cost of symbolic execution. Note that p may be arbitrarily long if it contains a loop.

Similarly, the cost of partial symbolic execution of a path expression r is bounded by $|P| \cdot |r|$, once the required data flow information has been derived. This information, a list of which variables are potentially redefined in each sub-expression, can be derived from the list of variables defined in each node in a single pass over r. As mentioned above, r can be computed in time linear in the number of edges. Again, in realistic situations one must also add the cost of simplification.

Global symbolic execution, even in those situations where it is well defined, is very expensive. For example, global symbolic execution of programs with sequences of conditional expressions can result in GSE predicates whose size is exponential in the length of the program [14]. Thus partial symbolic execution provides a much less expensive (but much less powerful) alternative to global symbolic execution .

4.5 Path Predicate Simplification & Evaluation

After symbolically executing a complete path, a set of constraints (i.e. path predicate) is generated. A test input can be generated by solving these constraints. But how to solve these constraints is a difficult task which poses some complex problems.

To obtain the test input from a path predicate generated during the symbolic execution, the first step is to simplify the constraints. Generally speaking, path predicate simplification can be divided into two parts. The first is to simply the arithmetic expressions. The second is to simplify the relational expressions. These two path predicate simplification techniques will be discussed in section 4.5.1 and 4.5.2.

For some path predicates, their feasibility can be determined after the simplification and test inputs can be generated immediately. For those path predicates whose feasibility can not be decided after path predicate simplification, linear programming has to be employed to determine their feasibility and to generate the test inputs. Linear programming is discussed in section 4.5.3. Section 4.5.4 will focus on some difficulties in the area of path predicate evaluation.

4.5.1 Simplification of Arithmetic Expressions

The simplification of arithmetic expressions includes the following rules:

1. Erase unnecessary brackets.

2. '*A* op 0' can be simplified to '*A*', where *A* can be any variable and op ∈ {+, -}. For example, expression '*z* + 0' can be reduced to '*z*'.

3. '0 + *A*' can be simplified to *A*; '0 - *A*' can be simplified to '-*A*'.

4. '0 * *A*' can be simplified to '0'.

5. '0 op *A*' can be simplified to '0', where *A* <> 0 and op ∈ {div, /, mod}.

6. '*cons1* op *cons2*' can be simplified to another constant which is the result of the operation, where *cons1* and *cons2* can be any real constant, and op ∈ {+, -, *, div, / }. For example, '4 * 5' can be simplified to '20'.

7. Simplify the arithmetic expression, i.e. calculate the coefficients of every variable appeared in the expression. For example: 1.5*x* +2*x* can be reduced to 3.5*x*.

The original constraints generated from symbolic execution may look like the following:

((((y/2) *3) + ((y/2) +z + 0)) >= 0) or (1 >= 4)

For the above expression, obviously there are at least 2 parts that can be simplified according to the rules listed above:

- Erase unnecessary brackets. After removing the brackets, the expression becomes

  y/2 * 3 + y/2 + z + 0 >= 0 or 1 >= 4.

- Simplify the arithmetic expressions, including the coefficients calculation of every variable appeared in the expressions. The example expression can be further simplified to

  2y + z >= 0 or 1 >= 4

Since the path predicate expression can be stored in a tree structure, it is not difficult to achieve the above goal.

4.5.2 Simplification of Relational Expressions

Simplification of relational expressions can be used to calculate relational expressions, and generate input value for some variables. It includes the following simplification rules:

1. '*expl* ∪ TRUE' can be determined to be TRUE,

2. '*expl* ∪ FALSE' can be reduced to '*expl*',

3. '*expl* ∩ TRUE' can be reduced to '*expl*',

4. '*expl* ∩ FALSE' can be determined to FALSE.

5. '*cons1* rel_op *cons2*', where *cons1* and *cons2* are two constants, rel_op ∈ {>, >=, <, <=}, can be determined to be either true or false. For example, '1 >= 4' can be determined false.

In the above rules, *exp1* can be a boolean expression or a symbolic expression. Applying rule 5, '(1 >= 4)' in the previous example can be decided to be FALSE, the expression becomes

$$(2y + z >= 0) \cup \text{FALSE}$$

Applying rule 2, the expression is reduced to

$$2y + z >= 0$$

If the feasibility of a path predicate can be determined right after the simplification of relation expressions, the test input can be generated if the path predicate is determined feasible. As an example, consider the path predicate '$(x = 1) \cap (2y = 1 + 3) \cap (z = 4) \cap (5 >= 4)$'. After simplification, it is obvious that the path predicate is feasible, and the test input is '$x = 1, y = 2, z = 4$'.

If the feasibility of a path predicate cannot be determined after the simplification of relation expressions, linear programming, which will be discussed in the next section, has to be used to decide the feasibility and to generate the test input.

4.5.3 Linear Programming

4.5.3.1 Introduction to Linear Programming

Linear programming can be used to determine the feasibility of the path where the path predicate contains only linear functions. For example, $x^3 + y^2 >= 5$ is nonlinear, but $x + 2y >= 5$ is linear. A linear programming problem is a mathematical program in which the objective function is linear in the unknowns and the constraints consist of linear qualities and linear inequalities. The exact form of these constraints may differ from one problem to

another, but as shown below, any linear program can be transformed into the following *standard form* [32]:

$$
\begin{aligned}
\text{minimize} \quad & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\
\text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
& a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\
& \vdots \qquad\qquad \vdots \\
& a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \\
\text{and} \quad & x_1 >= 0,\ x_2 >= 0,\ \ldots,\ x_n >= 0
\end{aligned}
$$

where the b_i's, c_i's and a_{ij}'s are fixed real constants, and the x_i's are real numbers to be determined. We always assume that each equation has been multiplied by minus unity, if necessary, so that each $b_i >= 0$.

In more compact vector notation, this standard problem becomes

$$
\begin{aligned}
\text{minimize} \quad & \mathbf{cx} \\
\text{subject to} \quad & \mathbf{Ax} = \mathbf{b} \text{ and } \quad \mathbf{x} >= \mathbf{0}.
\end{aligned}
$$

Here, $\mathbf{x}$ is an n-dimensional column vector, $\mathbf{c}$ is an n-dimensional row vector, $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{b}$ is an m-dimensional column vector. The vector inequality $\mathbf{x} >= \mathbf{0}$ means that each component of $\mathbf{x}$ is nonnegative. We say that $\mathbf{CX}$ is the *objective function* of the problem and the equations and inequalities are *constraints*.

Example 1 to 3 illustrate how various other forms of linear programs can be converted to the standard form.

Example 1 (Slack variables). Consider the problem

$$\begin{aligned}
&\text{minimize} && c_1x_1 + c_2x_2 + \ldots + c_nx_n, \\
&\text{subject to} && a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n <= b_1 \\
& && a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n <= b_2 \\
& && \vdots \\
& && a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n <= b_m \\
&\text{and} && x_1 >= 0, x_2 >= 0, \ldots, x_n >= 0.
\end{aligned}$$

In this case the constraint set is determined entirely by linear inequalities. The problem may be alternatively expressed as

$$\begin{aligned}
&\text{minimize} && c_1x_1 + c_2x_2 + \ldots + c_nx_n \\
&\text{subject to} && a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n + y_1 = b_1 \\
& && a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n + y_2 = b_2 \\
& && \vdots \\
& && a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n + y_m = b_m \\
&\text{and} && x_1 >= 0, x_2 >= 0, \ldots, x_n >= 0, \\
&\text{and} && y_1 >= 0, y_2 >= 0, \ldots, y_m >= 0.
\end{aligned}$$

The new positive variables y_i introduced to convert the inequalities to equalities are called *slack variables* (or more loosely, *slacks*). By considering the problem as one having $n + m$ unknowns $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$, the problem takes the standard form. The $m \times (n + m)$ matrix that now describes the linear equality constraints is of the special form [**A**, **I**], (that is, its columns can be partitioned into two sets; the first n columns make up the original **A** matrix and the last m columns make up an $m \times m$ identity matrix).

Example 2 (Surplus variables). If the linear inequalities of Example 1 are reversed so that a typical inequality is

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n >= b_i,$$

it is clear that this is equivalent to

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n - y_i = b_i$$

with $y_i >= 0$. Variables, such as y_i, adjoined in this fashion to convert a 'greater than or equal to' inequality to equality are called *surplus variables.*

It should be clear that by suitably multiplying by minus unity, and adjoining slack and surplus variables, any set of linear inequalities can be converted to standard form if the unknown variables are restricted to be nonnegative.

Example 3 (Free variables). If a linear program is given in standard form except that one or more of the unknown variables is not required to be nonnegative, the problem can be transformed to standard form by the following simple technique.

To describe the technique, suppose in the standard form, for example, that the restriction $x_i >= 0$ is not present and hence x_i is free to take on either positive or negative values. We then write

$$x_1 = u_1 - v_1$$

where we require $u_1 >= 0$ and $v_1 >= 0$. If we substitute $u_1 - v_1$ for x_1 everywhere in the standard form, the linearity of the constraints is preserved and all variables are now required to be nonnegative. The problem is then expressed in terms of the $n + 1$ variables $u_1, v_1, x_2, x_3, \ldots, x_n$.

There is obviously a certain degree of redundancy introduced by this technique, however, since a constant added to u_1 and v_1 does not change x_1(that is, the representation of a given value x_1 is not unique). Nevertheless, this does not hinder the method of solution.

Interested readers may refer to reference [32] for further details on linear programming.

4.5.3.2 Application of Linear Programming to Path Predicate Evaluation

Path predicate evaluation, i.e., finding a set of values for the variables in a path predicate such that all the constraints in the path predicate are satisfied, is a problem that can be transformed to the linear programming problem described above. The only difficulty is the absence of an objective function, that is, an expression to be minimized. However, the precise nature of the objective function is not important in practice, and a simple objective function, such as the sum of all the variables present in the path predicate, can be created. As an example, consider the following path predicate

$2x + y + z <= 6$

$x + 2y + 3z <= 15$

$2x + 2y + z <= 18$

$x >= 0, y >= 0, z >= 0.$

To transform the problem into the standard form so that the linear programing algorithm can be applied to, we first generate an object function - x - y - z, next introduce three nonnegative slack variables x_2, x_3, x_4. We then obtain the standard form as follows:

minimize $-x - y - z$

subject to $2x + y + z + x_2 = 6$

$$x + 2y + 3z \quad +x_3 \quad = 15$$

$$2x + 2y + z \quad + x_4 = 18$$

$$x >= 0, \quad y >= 0, \quad z >= 0,$$

$$x_2 >= 0, \quad x_3 >= 0, \quad x_4 >= 0.$$

The optimal solution to this linear programing problem is $x = 1/15$, $y = 0$, $z = 8/15$, $x_2 = 0$, $x_3 = 0$, $x_4 = 4$ with a corresponding value of the object of -3/5 [34]. Thus, we can say the path predicate is feasible, and the test input is $x = 1/15$, $y = 0$, $z = 8/15$.

To generate test case that satisfies any kinds of conditional statements on the path requires that the system be capable of solving an arbitrary set of inequalities, which has been identified as an undecidable problem [8]. Therefore, to determine the existence of a test case for a specific program path is an undecidable problem and hence heuristic approaches have been advocated. Our solution is based on the hypothesis that the inequalities will usually be relatively simple and often linear. Statistics indicates that nonlinear predicate interpretation are rarely encountered and are unlikely to pose a problem for the majority of software programs [19]. Should this assumption prove valid, then linear programming techniques can be employed to solve the systems of inequalities in most cases. If this premise fails then more powerful methods have to be found to solve the inequalities.

4.5.4 Special Considerations in Path Predicate Evaluation

4.5.4.1 Arrays

Arrays have been a problem for almost every system which applies symbolic evaluation techniques. There is no difficulty for an expression like $a(5) > b$, because $a(5)$ is unique in the same sense that b is unique. Both refer to a specific element and this

$$(\text{not } a(i+2) > 10) \cap (a(i+1) > a(i+2))$$

where i is an input variable. Because the identification of the element within the array cannot be defined, it is said to be an ambiguous array element. This ambiguity poses a problem for the evaluation of the path predicate. One approach to overcome this problem is to replace each element by a new variable, and treat each element separately. In the above expression, replace $a(i+1)$ and $a(i+2)$ respectively by variable x and y, it becomes

$$(\text{not } y > 10) \cap (x > y).$$

4.5.4.2 Alphanumeric Constants

For numerical programs where predicates tend to consist only of operators, numeric variables, and numeric constants, formulating the path predicates into a format suitable for input to a linear programming routine is a relatively straightforward matter. But some cases, programs may contain an additional class of predicate component, the alphanumeric. Constraints that contain only numeric variables, numeric constants, and alphanumeric variables but not alphanumeric constants can be processed by a linear programming routine by treating all variables as numeric variables. The solution in this case will not necessarily be useful as a test case where alphanumeric variables are present, but its feasibility will have been assessed.

The problem exists for the predicates that contain alphanumeric constants which cannot be easily passed to a linear programming routine that requires numeric data. One solution to this problem is to replace each alphanumeric constant in a predicate with a numeric token, and the numerical tokens must give the same sort sequence as the constants that they represent. Then the numeric tokens along with the variables and numeric constants can be passed to the linear programming package in the usual way. For example, for the following path predicate

$$(\text{not } x > 15) \cap (s1 = \text{'a'}) \cap (\text{not } x < 10) \cap (s2 = \text{'b'}) \cap (s1 = s2)$$

Since 'a' < 'b', we can use token 4 and 6 to replace 'a' and 'b'. Now, we get

$$(\text{not } x > 15) \cap (s1 = 4) \cap (\text{not } x < 10) \cap (s2 = 6) \cap (s1 = s2)$$

which is obviously infeasible.

Chapter 5

ETSG: A System Prototype

As a prototype of the system described in this thesis, ETSG (Extended Test Sequence Generation System) is developed cooperatively by University of Ottawa and Bell Northern Research. It is implemented in C language and runs on HP workstations under UNIX. The objective of ETSG is to provide a tool set to automatically generate an executable test suite to test the source codes or specifications written in one of the following two languages: SDL and Pascal. Based on the automatic test suite generation approach proposed in this thesis, ETSG consists of three major parts, which is shown in Figure 5.1:

Phase1: Language Translation

Phase2: Test Unit Derivation

Phase3: Test Case Selection

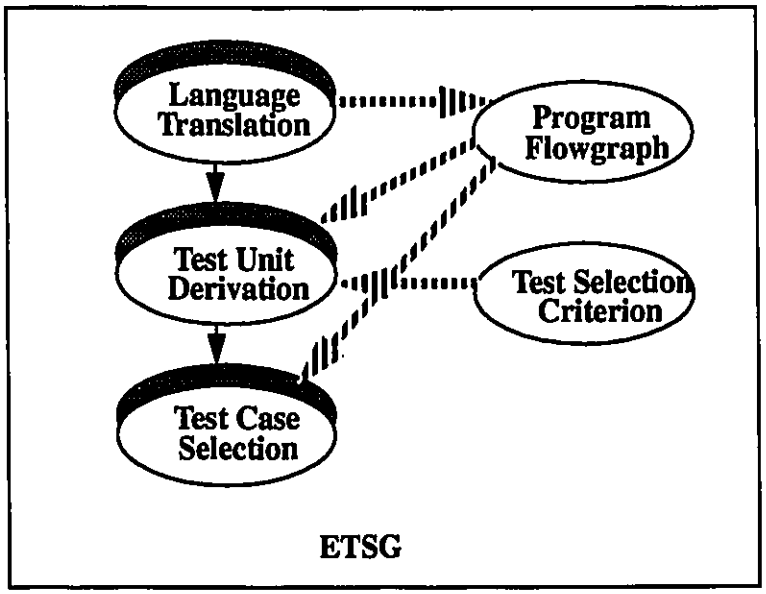


FIGURE 5.1 ETSG Configuration

The process performed by ETSG can be described as follows. First, a flow graph modeling the flow of both control and data implied in the given program is constructed through language translation. Second, by applying a formally defined control or data flow oriented test selection criterion to this flowgraph, a set of test units is built through test unit derivation. Third, for each test unit, a path expression is generated through path expression generation to represent a set of paths covering the test unit. Partial symbolic execution is then used to determine the feasibility of the path expression. If the path expression is determined infeasible, it means the test unit is infeasible. Otherwise, a complete path will be selected from the path expression. Symbolically executing the complete path will result in a set of constraints, and after that, by evaluating these constraints, the feasibility of the complete path can be determined and the input data for the test case is generated. If the complete path is declared infeasible, another complete path will be selected and symbolically executed until a feasible complete path is found. If no feasible complete path can be found from the corresponding path expression, the test unit is declared infeasible.

Even though the phase1 in ETSG is called language translation, it is actually equivalent to the first part, flowgraph construction, of the system proposed in this thesis. In sections 5.1, the implementation of phase1 and phase2 are briefly introduced. The implementation of phase3 is described in detail in section 5.2.

5.1 Language Translation and Test Unit Derivation

5.1.1 Phase1: Language Translation

Phase1 (Language Translation) converts an input, i.e., a system specification (written in SDL specification language) or the source code of a system module (written in Pascal) into an internal flowgraph representation, called Bflow data structure, which

accurately describes both control flow and data flow information in the input. The input to the language translators consists of two parts: the specification or the source code, accompanied by related abstract definition files. In ETSG, two translators are implemented for two different languages. They are Sdlflow for SDL and Pasflow for Pascal.

- **Pascal and Pasflow**

Pasflow derives both control and data flow information from a single entry and single exit program or procedure written in a subset of ISO standard Pascal. The translator consists of a Lex lexical analyzer and a YACC parser.

There are some limitations on the input Pascal program to Pasflow, they are: file and pointer variables are ignored; file identifiers may exist in file-oriented I/O but are ignored; there is no go to statement, no arrays within records, and no records within arrays. It is further assumed that in a function, formal parameters are all pass-by-values, and no global variable is updated in a called function. Interested reader may refer to *Appendix B.1: Assumptions on Pasflow Input* for more details.

- **SDL and Sdlflow**

SDL (Specification and Description Language) [16,17] is developed and recommended by CCITT (the International Telegraph and Telephone Consultative Committee) for the specification of telecommunication systems. Sdlflow accepts as input an SDL specification with a single entry and a single exit process together with related abstract definition files, and produces flowgraph containing both control and data flow information.

There are some limitations on the input SDL process to Sdlflow. Interested reader may refer to *Appendix B.2: Assumptions on Sdlflow Input* for more details.

5.1.2 Phase2: Test Unit Selection

Phase2 (Test Unit Selection) takes as input the flowgraph constructed in Phase1 and generates a set of test units based on a user-selected test coverage criterion. To allow a user to choose the test selection criterion has the flexibility of obtaining various test coverage for different test purposes. At present, the following two categories of test selection criteria are implemented in ETSG.

Control flow oriented test selection criteria:

all_nodes, all_edges, decision coverage, condition coverage,
decision/condition coverage, multiple condition coverage

Data flow oriented test selection criteria:

all_defs, all_cuses, all_puses, all_cuses/some_puses,
all_puses/some_cuses, all_uses, all_du_paths

5.2 Phase3: Test Case Selection

Test Case Selection (ETSG Phase3) first generates a path expression for each test unit and checks the feasibility of the path expression corresponding to the test unit by applying partial symbolic execution technique to the path expression. Then, a complete path is selected for each non-infeasible path expression and a test case for the test unit is generated by symbolically executing the complete path and evaluating the resulting path constraints.

Phase3 consists of the following modules: Path expression generation, Partial symbolic execution, Complete path selection, Symbolic execution, Path predicate simplification, Linear programming, Coverage check and Path file generation. The system flow of Phase 3 is illustrated in Fig. 5.2.

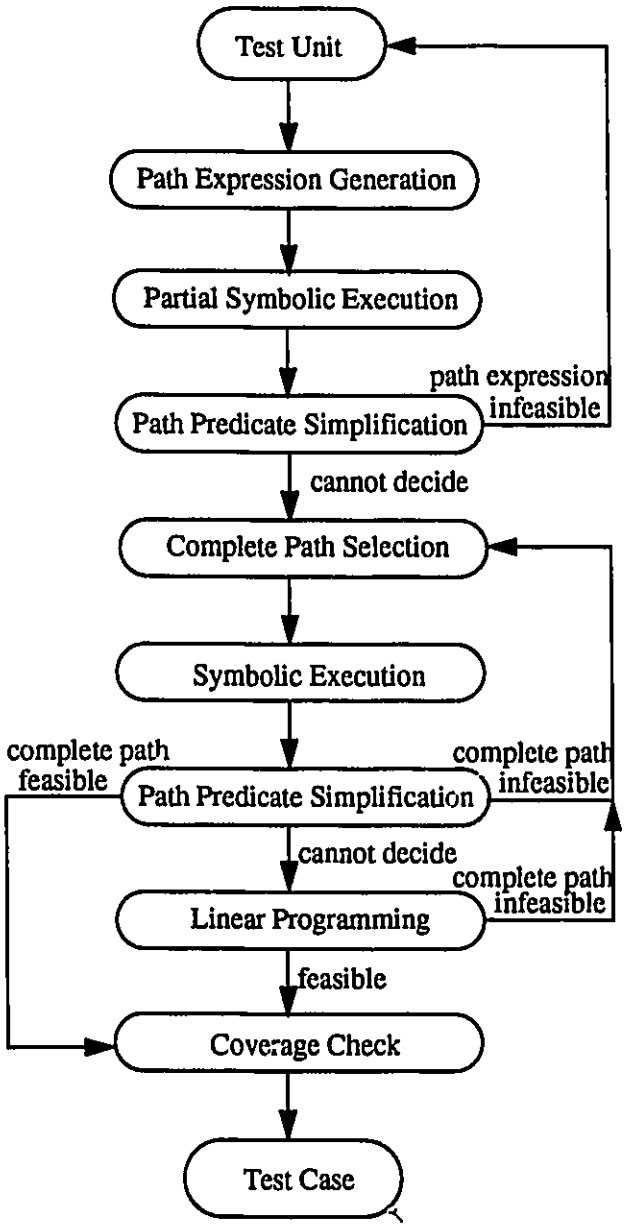


FIGURE 5.2 Phase3 System Flow

A short description for each module is given below. The functions referred to in these descriptions are given in Appendix D.

- **Path Expression Generation**

It is responsible for generating a path expression, which represents a set of complete paths from the entry to the exit of the program to cover a test unit. The input to this module is a test unit file. The following C functions implement this module: *ph3_read_unit.c*, *ph3_testunit.c* and *ph3_uses_unit.c* read a test unit from the input test unit file; *ph3_reduce.c* generates a path expression between two arbitrary nodes in a flowgraph; *ph3_def_clr.c* generates a path expression that represents all def-clear paths between two nodes with respect to a certain variable; *ph3_multi_pexp.c*, *ph3_dupath.c* and *ph3_uses_pexp.c* generate a complete path expression which covers the test unit. *ph3_search.c* finds a feasible subpath starting from the entry node to cover the given test unit, it is used to simplify the path expression and speed up the process of finding a feasible complete path. The output of this module is a complete path expression covering the chosen test unit from the test unit file. It is stored as a binary tree.

- **Partial Symbolic Execution**

It takes as input the complete path expression obtained above, and generates partial path conditions for a path expression and evaluate the feasibility of the path expression by checking the conflicts in the partial path condition. If the path expression is found to be infeasible, this test unit will be removed from the given test unit set and written into a file which stores all the infeasible test units. The function implementing this module is *ph3_PSE.c*.

- **Complete path selection**

If the path expression is not decided to be infeasible by *partial symbolic execution*, this module is invoked to select an arbitrary complete path from the path expression. Function *ph3_compl_pth.c* implements this module.

- **Symbolic Execution**

It is implemented by function *ph3_SYME.c* to symbolically execute the complete path selected by *complete path selection*, and to generate a path predicate for the path.

- **Predicate Simplification and Evaluation**

This module simplifies the path predicate generated by *symbolic execution*, and evaluates its feasibility. It includes two parts: arithmetic expression simplifier and relational expression simplifier. Arithmetic expression simplification simplifies arithmetic operations (+, -, *, /, div) before passing the result to the relational expression simplifier. Relational expression simplification reduces boolean-valued expressions to True, False, or neither. After path predicate simplification, if the path predicate is determined True(False), then it means that the corresponding complete path is feasible(infeasible). If the feasibility of the path predicate can not be determined, module *Linear Programming* has to be invoked. *Path predicate simplification and evaluation* is implemented by functions *ph3_simplifier.c* and *ph3_evaluate.c*.

- **Linear Programming**

It takes as input the simplified path predicate, determines the feasibility of the path predicate and then generates test input. If the path predicate is feasible, a set of input

values is generated for the input variables in the path predicate, which forms the test input. If the path predicate is infeasible, it means that the corresponding complete path is infeasible. Linear Programing is implemented by functions *ph3_minmain.c* and *ph3_minit.c*.

- **Coverage Check**

After a complete path is determined feasible as a result of *Predicate Simplification and Evaluation* or *Linear Programing*, *Coverage Check* is called to check if it covers other test units in the test unit set. If it does, those test units covered by this complete path will be removed from the test unit set. Since a complete path may cover many test units, this check is necessary to reduce the number of test cases for a certain test selection criterion. For example, by applying all-edges coverage to Program A in Figure 2.2, a feasible complete path *s* 1 2 3 4 5 6 7 8 9 10 11 *t* is generated. It covers the following test units in the test unit file: edge (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10) and (10, 11). After the coverage check, there are only two test units left in the test unit file to be covered: edges (7, 10) and (10, 4). For different test unit types, there are different kinds of coverage checkers, e.g., node coverage checker for all-statement coverage, def-use coverage checker for all-uses coverage, and etc. They are all implemented by function *ph3_cov_chk.c*.

- **Path File Generation**

Functions *ph3_output.c* and *ph3_pathfile.c* implement the *Path File Generation* module. *ph3_output.c* generates the expected output from the given specification based on the selected test input. *ph3_pathfile.c* generates a file called *path file* to store the feasible complete paths, the test input and expected output. *Appendix E : Files and Data Structures* gives the format of the path file.

There are also some utilities implemented in phase3, such as *ph3_inflow.c,* which displays all the data and control flow related information stored in the Bflow data structure.

Although the system is designed to be automatic, choice of human interference can also be made to speed up the process of test case selection process. For programs with many loop iterations, by specifying a sub-path to be covered by a test unit, a feasible complete path can be found more quickly. Human intelligence can also be used to add additional constraints into the path predicate resulted from symbolic execution, or to simplify or evaluate a path predicate if the path predicate is non-linear.

The program structure of phase3 is shown in Figure 5.3, which indicates the main control relations between the programs. These C programs implement the process of phase3 described above.

Note that:

a) the actual module name should be prefixed with "ph3_".

b) the dotted lines mean that the control may happen when necessary.

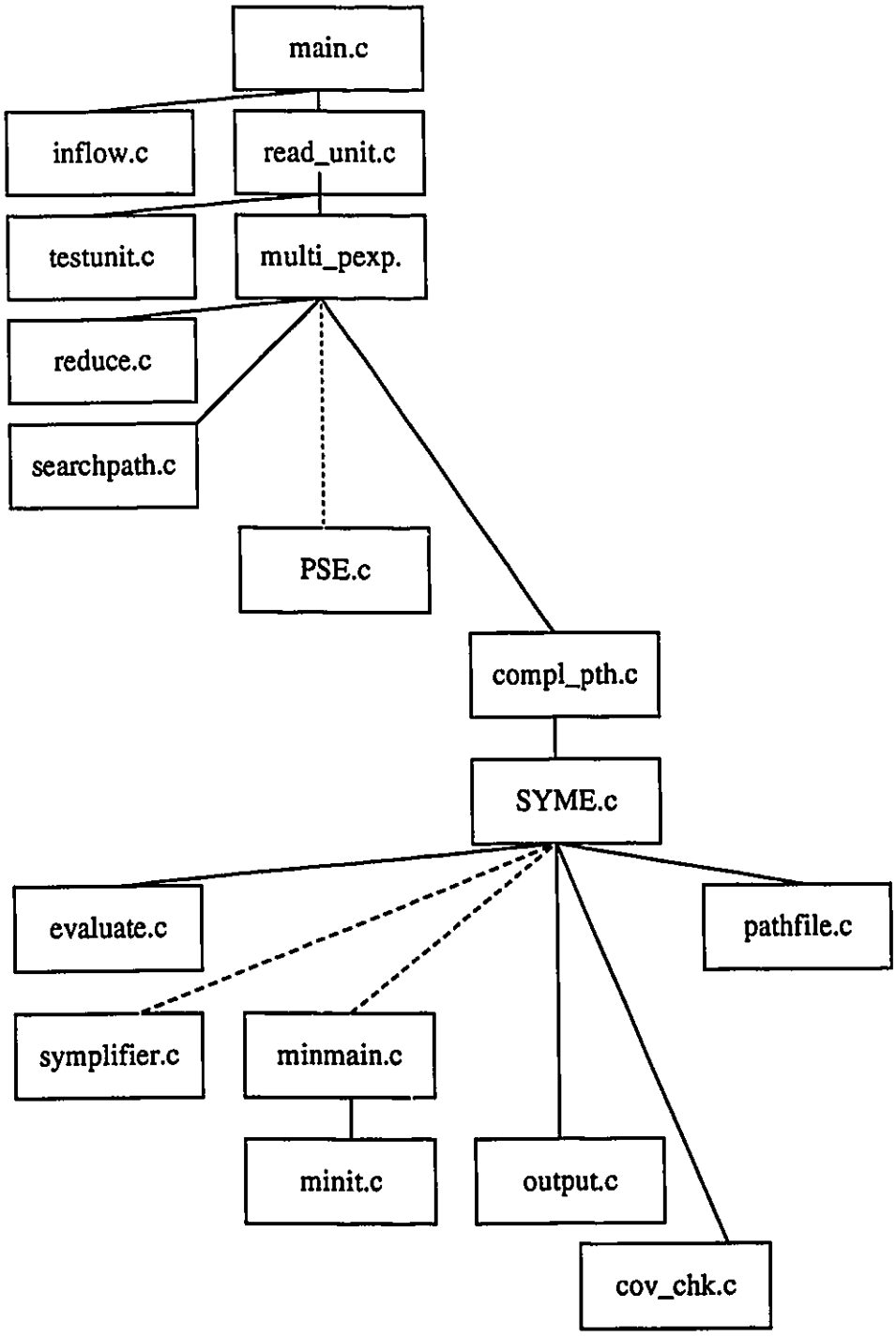


FIGURE 5.3 Module Structure of Phase3

5.3 A Pascal Example

Let us consider a Pascal example, whose source code is shown in Figure 2.1, and flowgraph is shown in Figure 2.2. The test unit file by applying all-uses criterion to the program is shown in Table 2.11, in which we number the test units and list them here as follows:

ALL USES COVERAGE

Def-cuse Associations:

#1 -- (i, 1, 6)	#2 -- (j, 2, 8)
#3 -- (j, 2, 11)	#4 -- (x, 4, 5)
#5 -- (y, 4, 5)	#6 -- (z, 4, 5)
#7 -- (n, 4, 9)	#8 -- (x, 4, 9)
#9 -- (y, 4, 9)	#10 -- (z, 4, 9)
#11-- (s, 5, 9)	#12 -- (i, 6, 6)
#13 -- (j, 8, 8)	#14 -- (j, 8, 11)

Def-puse Associations:

#15 -- (s, 5, (7, 8))	#16 -- (s, 5, (7, 10))
#17 -- (y, 4, (7, 8))	#18 -- (y, 4, (7, 10))
#19 -- (i, 6, (10, 11))	#20 -- (i, 6, (10, 4))
#21 -- (t, 3, (10, 11)	#22 -- (t, 3, (10, 4))

NUMBER OF TEST UNITS: 22

Based on the test unit file, ETSG analyses the flowgraph and collects all of the inputs and outputs along with each complete path (or test path). The inputs and outputs are identified by their corresponding node in the flowgraph. Each test path then consists of a sequence of nodes that represent the flow of data in and out of the flowgraph. All the test

paths start at the entry point and ends at the exit point. For the above example, the test path file generated is as following:

Test paths:

test path #1:

1 2 3(t = 1) 4(n=1, x=700, y=600, z=700) 5 6 7 10 11(O: j=0)

covers remaining test units:

1 -- (i, 1, 6)	3 -- (j, 2, 11)
4 -- (x, 4, 5)	5 -- (y, 4, 5)
6 -- (z, 4, 5)	
16 -- (s, 5, (7, 10))	18 -- (y, 4, (7, 10))
19 -- (i, 6, (10, 11))	21 -- (t, 3, (10, 11)

test path #2:

1 2 3(t = 1) 4(n=1, x=500, y=700, z=600) 5 6 7 8 9(O: n=1, x=500, y=700, z=600, s=1800) 10 11(O: j=1)

covers remaining test units:

2 -- (j, 2, 8)	7 -- (n, 4, 9)
8 -- (x, 4, 9)	9 -- (y, 4, 9)
10-- (z, 4, 9)	11-- (s, 5, 9)
14 -- (j, 8, 11)	
15 -- (s, 5, (7, 8))	17 -- (y, 4, (7, 8))

test path #3:

1 2 3(t = 2) 4(n=1, x=500, y=700, z=600) 5 6 7 8 9(O: n=1, x=500, y=700, z=600, s=1800) 10 4(n=2, x = 500, y=600, z=600) 5 6 7 10 11(O: j=1)

covers test units:

12 -- (i, 6, 6)	13 -- (j, 8, 8)
20 -- (i, 6, (10, 4))	22 -- (t, 3, (10, 4))

5.4 An SDL Example

The SDL version of ETSG is currently used at BNR to generate test suites from SDL specifications. It takes as input an SDL specification and generates test suite from it. The test suite is then used to test the program implementing the specification. During the system testing of ETSG, more than 40 randomly selected SDL processes were used. It turns out that our system can handle SDL processes with nodes fewer than 750 sufficiently well. Memory requirements have proved to be insufficient for the process of large SDL specifications. Complex data representation (such as array, record and pointer variable) and procedure/function calls are not implemented in the SDL version of prototype since they seldom appear in the specifications of communication protocols.

Appendix A: An SDL Example includes an SDL example and the test paths generated by applying all-edges and all-uses coverage criteria to that SDL specification. Appendices *A.1* and *A.2* give the SDL specification and the corresponding flowgraph. Appendix *A.3* is the flowgraph file of the SDL specification. Appendices *A.4* and *A.5* describe respectively the test unit file and the test path file generated by applying all-edges coverage criterion to the SDL specification. Appendices *A.6* and *A.7* describe the test unit file and the test path file generated by applying all-uses coverage criterion to the SDL specification. Appendix *A.8* lists the infeasible test units given in Appendix *A.6*.

Since all-uses coverage criterion strictly includes all-edges coverage criterion, as we can see from Appendices A.5 and A.7, all_uses coverage criterion generates 2 more test paths than all-edges coverage criterion. All-uses criterion also generates 60 pairs of infeasible def-puse associations (x, *node1*, *edge2*), which indicates that the definition of x in *node1* is not used in *edge2*. If the definition of a variable has never been used in the program, it indicates an error in the program, but it is not the case for this example.

The test paths listed in Appendices A.5 and A.7 are generated automatically by ETSG. It takes less than 10 minutes to achieve the test case selection for all-edges and all-uses coverage criteria respectively, and the system achieves 100% coverage for the feasible test units. ETSG also has the choice of human interactions. For some large complex programs, by introducing human interference, ETSG can also achieve test case selection very quickly.

To illustrate our theory that data flow oriented testing pose advantages over all-edges criterion, we assume that the program under test corresponding to the specification in Appendix A.1 contains two errors: the predicate in edge (4, 29) is ' $r = 0$' , and the predicate in edge (4, 43) is ' $l = 0$'.

The set of complete paths in Appendix A.5 satisfies all-edges criterion, but would not detect these two errors. The first error is not detected because the definition of l in node 25 does not reach edge (4, 29) unless the set of complete paths includes a def-clear path w.r.t. l from node 25 to edge (4, 29). Thus, we could not expect to detect this error, in general, unless the path (25, 4, 29) is included in the set of complete paths. The second error is not detected because the definition of r in node 21 does not reach edge (4, 43) unless the set of complete paths includes a def-clear path w.r.t. r from node 21 to edge (4, 43). Thus, we could not expect to detect this error, in general, unless the path (21, 4, 43) is included in the set of complete paths.

All-defs and all-cuses/some-puses are not stronger enough to detect these errors, either. But all-puses and all-puses/some-cuses do require that def-puse associations (l, 25, (4, 29)) and (r, 21, (4, 43)) are covered. Thus, the paths (25, 4, 29) and (21, 4, 43) must be included in the set of selected complete paths. Thus, both errors should be detected. The all-uses criterion includes both all-puses/some-cuses and all-cuses/some-puses criteria,

and should be able to detect all the errors described above. However, as we see in Figure 2.3, this criterion may not test all-possible combinations of predicate outcomes. This should be accomplished by all-du-paths.

Chapter 6

Conclusions

6.1 Comparison with Related Work

The concepts of data flow oriented testing are not new. Up to now, quite a few papers [4, 5, 8, 10, 22, 23] have proposed and analyzed different test selection criteria based on data flow oriented testing. The work studied in this thesis is based on these research.

Symbolic execution is also widely discussed and used [12-15, 20, 28] to identify the feasibility of a complete path or to generate input data for a test case. Similar to our system introduced in these thesis, the basic operation of symbolic execution based systems consist of language translation, path selection, execution of paths symbolically, constraints simplification, and inequality solving [26]. Some of the well-known symbolic execution systems include SELECT [33], EFFIGY, ATTEST, DISSECT[3], and SMOTL[27].

Among these systems, EFFIGY and DISSECT do not provide test case selection. EFFIGY is mainly used as an interactive debugging tool. It accepts programs written in a subset of PL/1 that allows only integer values and provides features such as tracing, breakpoints, and state savings. DISSECT analyzes ANSI Fortran programs to determine the computations along selected paths, the set of symbolic values which causes the path to be executed, and the symbolic values of the output variables. It does not have the function to solve inequalities. Both EFFIGY and DISSECT need user interactions to specify loop iteration and to select test paths.

SELECT, ATTEST, and SMOTL are test case selection systems. SELECT selects test cases and creates a symbolic representation of the output variables for programs written in a subset of Lisp. SELECT has limited capability to handle procedure calls. Also, the path constraints and output variable representations are currently represented as a Lisp list rather than a more human recognizable form [28]. ATTEST analyzes ANSI Fortran programs. Both SELECT and ATTEST need user interactions to define paths and loop iterations. SMOTL is the only one which achieves automatic test case selection. However, it only achieves branch coverage and cannot achieve complete coverage even for the feasible branches because of the way it deals with loop iteration: If there are two paths which differ only in the number of loop traversals, the longer path is not considered. SMOTL achieves up to 75% coverage for all the feasible branches.

The systems mentioned above are all research prototypes except SMOTL, which is oriented towards data-processing application. For these systems, besides the limitation of processing large programs, they also have difficulties in dealing with path selection, loop iteration and array references. And neither of them aims at data flow oriented testing.

Our work attempts to identify the possibilities and difficulties of the automatic test case selection, especially in the field of data flow oriented testing, by constructing a prototype system. In the prototype system, we used path expression generation and complete path selection algorithms to solve the path selection and loop iteration problems. To reduce the significant effort wasted in analyzing infeasible paths, which is a weakness of most pathwise test case selection systems, a new technique, partial symbolic execution is employed to identify the infeasible test units even before any complete path covering these test units is selected. The SDL version of the prototype system achieves 100% *all_nodes* and *all_edges* coverage for all such feasible test units. In our system, arrays do not pose any problem for flowgraph construction, test unit generation, path expression

generation, partial symbolic execution and symbolic execution. Some techniques are discussed to deal with the difficulties raised by arrays in path predicate evaluation.

6.2 Concluding Remarks

The system introduced in this thesis intends to solve the problem of automation of test suit construction for data flow testing. The given source code or specification is first transformed into a flowgraph which represents both control flow and data flow information contained in the source code or specification. Based on this information and a user-chosen test selection criterion, a set of test units is generated. After path expression generation and partial symbolic execution, the feasibility of the test unit can be decided. If the test unit is decided to be feasible, a complete path is selected based on the path expression corresponding to the test unit. By symbolically executing the complete path, a path predicate is generated for the complete path. Evaluation of this path predicate determines the feasibility of the complete path and generates the test input if the path is feasible. If the complete path is not feasible, then another complete path has to be selected from the path expression, and go through symbolic execution and path predicate execution until a feasible complete path is found or the test unit is arbitrarily declared infeasible after some reasonable number of iterations.

Linear programming routines can be used to solve the constraints in the path predicate for a large class of software. But unfortunately, there are some problems in transforming the data format used in software programs to a format applicable to linear programming, such as arrays, alphanumeric constants, and expression like "$a \neq b$". Those problems have been discussed and solutions have been provided.

6.3 Future Enhancements

One area for future work is to investigate the solutions for the problems raised by arrays in path predicate evaluation, especially when array subscripts depend on the input data, and implement them in the system prototype.

The second major area for future work is to include more Pascal types and operations in the Pascal version of the prototype, such as: records, pointer variables, file variables, and etc. The most important of these is the pointer type, the implementation of which will make the system more usable for a larger class of programs.

Another area of future enhancement is to make the system prototype capable of constructing test suites for programs containing complex procedure/function calls by implementing the technique of extended flowgraph [30] to merge the flowgraph of procedure/function calls into the flowgraph of the main program.

References

[1] F. Sato, K. Katsuyama, T. Mizuno, 'TENT: Test Sequence Generation Tool for Communication Systems', *Proc. of Formal Description Techniques*, II, pp. 1-5, 1990.

[2] B. Korel, 'Automated Software Test Data Generation', *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870-879, Aug. 1990.

[3] W. E. Howden, 'Symbolic Tesing and the DISSECT Symbolic Evaluation System', *IEEE Trans. Software Eng.*, vol. 6, no. 4, pp. 266-278, July 1977.

[4] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, 'A Formal Evaluation of Data Flow Path Selection Criteria', *IEEE Trans. Software Eng.*, vol. 15, no. 11, pp. 1318-1332, Nov. 1989.

[5] H. Ural and B. Yang, 'Test Path Selection via IP/O-chains Coverage Criteria', Technical Report TR-91-32, Dept. of Computer Science, University of Ottawa, July, 1991.

[6] P. G. Frankl, 'Partial Symbolic Evaluation of Path Expressions', Technical Report PUCS-110-90, Dept. of Computer Science, Polytechnic University, July 1990.

[7] R. E. Tarjan, 'Fast Algorithms for solving path problems', *Journal of the Association for Computing Machinery*, vol. 28 , no. 3, pp. 594-614, July 1981.

[8] E. J. Weyuker, 'The applicability of program schema results to programs', *Int. J. Comput. Inform. Sci.*, vol. 8, pp. 387-430, Nov. 1979.

[9] D. Hamlet, B. Gifford, and B. Nikolik, 'Exploring Dataflow Testing of Arrays', *Proc. of ICSE 1993*, pp. 118-129, May 1993.

[10] S. Rapps and E. J. Weyuker, 'Selecting Software Test Data Using Data Flow Information', *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 367-375, April, 1985.

[11] H. Ural and B. Yang , 'A Test Sequence Selection Method for Protocol Testing', *IEEE Trans. Comm.*, vol. 39, no.4, pp. 514-523, April 1991.

[12] T. E. Cheatham, G. H. Halloway, and J. A. Townley, 'Symbolic evaluation and the analysis of programs', *IEEE Trans. Software Eng.*, vol. SE-5, no. 4, pp. 402-417, Jan. 1979.

[13] P. D. Coward, 'Symbolic Execution and Testing', *Information and Software Technology*, vol. 33, no. 1, pp.53-64, Feb. 1991.

[14] L. A. Clarke and D. J. Richardon, 'Symbolic evaluation methods for program analysis', In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

[15] R. A. Kemmerer and S. T. Eckmann, 'UNISEX: a UNIX-based Symbolic EXecutor for Pascal', *Software Practice and Experience*, vol. 15, no. 5. pp. 439-458, May 1985.

[16] 'SDL, Recommendation Z.100', CCITT, 1992.

[17] H. Ural and A. Williams, 'Test Generation by Exposing Control and Data Dependencies within System specifications in SDL', *Proc. of FORTE'93*, pp. 339-354, 1993.

[18] ETSG group, 'ETSG -- A Test Tool', (internal material), Department of Computer Science, University of Ottawa, 1991.

[19] L. J. White, and E. I. Cohen, 'A domain strategy for computer program testing', *IEEE Trans. Software Eng.*, vol. 6, no. 3, pp. 247-257, March 1980.

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

[21] R. E. Prather, and J. P. Jr. Myers, 'The path prefix software testing strategy', *IEEE Trans. Software Eng.*, vol. 13, no. 7, pp.761-766, July 1987.

[22] P. G. Frankl, and E. J. Weyuker, 'An Applicable Family of Data Flow Testing Criteria', *IEEE Trans. Software Eng.*, vol. 14, no. 10, pp. 1483-1498, Oct. 1988.

[23] S. Rapps and E. J. Weyuker, 'Data flow analysis techniques for program test data selection', *Proc. of Sixth International Conference on Software Eng.*, Tokyo, Japan, 1982, pp. 272-278.

[24] R. L. Probert, 'Life-cycle/Grey-Box Testing', *Congressus Numerantium*, Vol. 34, pp. 87-117, 1982.

[25] ETSG group, 'ETSG---A Testing Tool', Computer Science Department, University of Ottawa, 1991.

[26] R. A. DeMillo, W. M. McCracken, R.J. Martin and J. F. Passafiume, *Software Testing and Evaluation*, Benjamin/Cummings Publishing Company, Inc., 1987.

[27] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins and E. F. Miller, 'SMOTL---A system to construct samples for data processing program debugging', *IEEE Trans. Software Eng.*, vol. 5, no. 1, pp. 60-66, Jan. 1979.

[28] L. Clarke, 'A system to generate test data and symbolically execute programs', *IEEE Trans. Software Eng.*, vol. 2, no. 3, pp. 215-222, Sept. 1976.

[29] ETSG group, 'Flowgraph Representation of a Specification, Design or Source Code', (internal material), Department of Computer Science, University of Ottawa, 1992.

[30] B. Yang and H. Ural, 'Program Modeling and Data Flow Testing', *Proc. of 15th ICSE*, pp. 277-286, May 1993.

[31] B. Beizer, *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold Inc., 1990.

[32] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley Publishing Company, 1973.

[33] R. S. Boyer, B. Elspas and K. N. Levitt, 'SELECT--- a formal system for testing and debugging programs by symbolic execution', *Proc. Int. Conf. Reliable Software*, pp. 234-244, April 1975.

Appendix A

An SDL example

A.1 SDL Source File

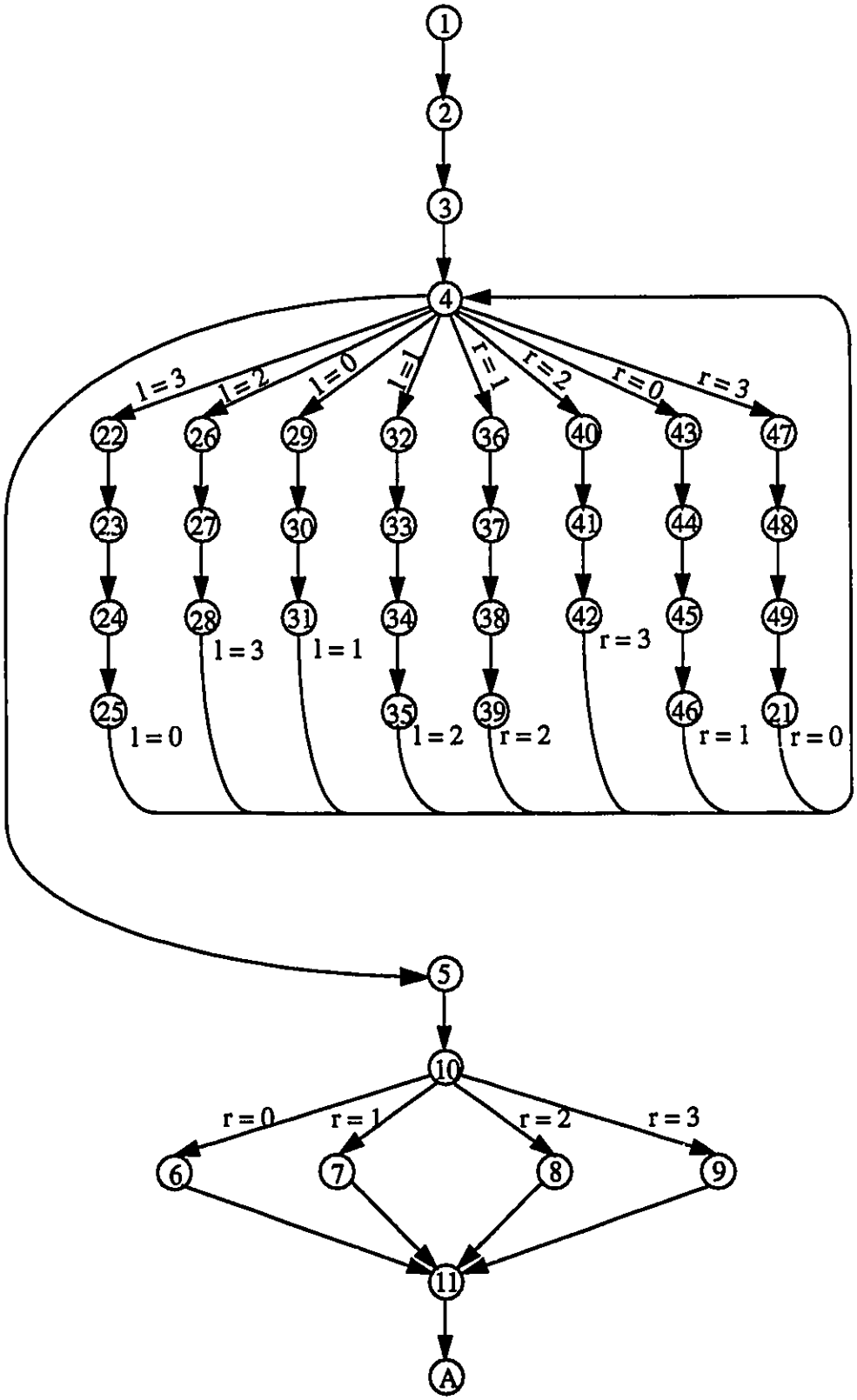
```
PROCESS cgrr;

SIGNAL
        BLA,
        BLO,
        Blocking,
        BlockingConf,
        BlockingInd,
        BlockingReq,
        BlockingResp,
        GRA,
        GRS,
        GrpResetComplete,
        MaintInd,
        UBA,
        UBL,
        Unblocking,
        UnblockingConf,
        UnblockingInd,
        UnblockingReq,
        UnblockingResp;

DCL     bstatus INTEGER,
        remoteblock INTEGER,
        localblock INTEGER;
    START ; /* #Partition 1 */
    TASK localblock
    := 0;
```

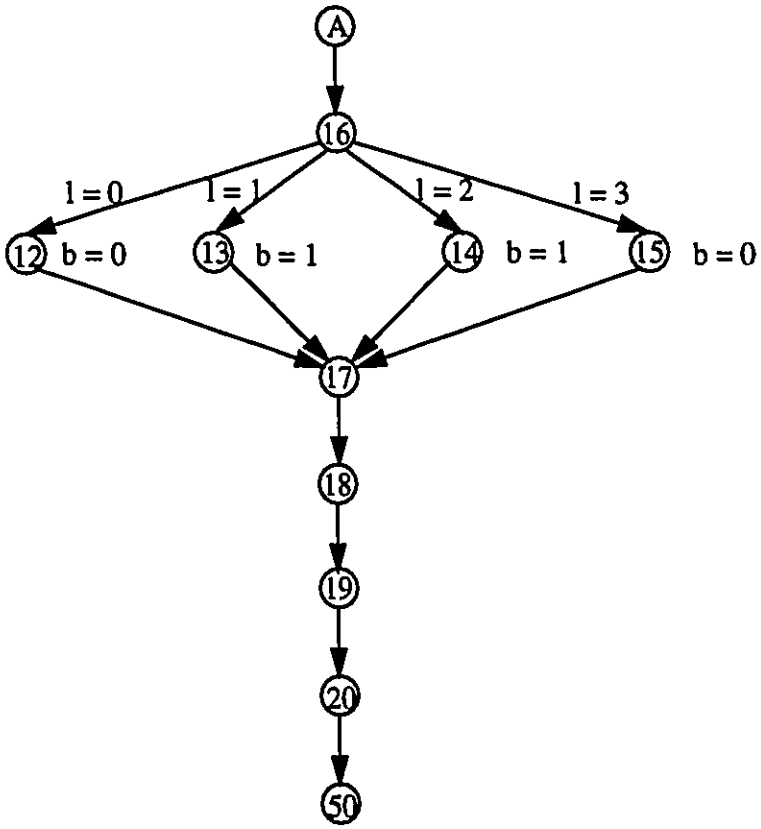
```
    TASK remoteblock := 0;
    NEXTSTATE Idle;

STATE Idle; /* #Partition 1 */
    INPUT GRS ;
    DECISION remoteblock;
        (= 0):
        (= 1):
        (= 2): OUTPUT Unblocking;
        (= 3): OUTPUT Unblocking;
    ENDDECISION ;

    DECISION localblock;
        (= 0): TASK bstatus := 0;
        (= 1): TASK bstatus := 1;
        (= 2): TASK bstatus := 1;
        (= 3): TASK bstatus := 0;
    ENDDECISION ;
    NEXTSTATE WaitForGroupResetComplete;
ENDSTATE ;

STATE WaitForGroupResetComplete; /* #Partition 1 */
    INPUT GrpResetComplete ;
    OUTPUT GRA ( bstatus );
    STOP ;
ENDSTATE ;

STATE Idle; /* #Partition 2 */
INPUT UBA ;
    PROVIDED localblock = 3 ;
        OUTPUT UnblockingConf;
        OUTPUT Unblocking;
        TASK localblock := 0;
        NEXTSTATE Idle;
```

```
INPUT UnblockingReq ;
    PROVIDED localblock = 2 ;
        OUTPUT UBL;
        TASK localblock := 3;
        NEXTSTATE Idle;

INPUT BlockingReq ;
    PROVIDED localblock = 0 ;
        OUTPUT BLO;
        TASK localblock := 1;
        NEXTSTATE Idle;

INPUT BLA ;
    PROVIDED localblock = 1 ;
        OUTPUT BlockingConf;
        OUTPUT Blocking;
        TASK localblock := 2;
        NEXTSTATE Idle;

INPUT BlockingResp ;
    PROVIDED remoteblock = 1 ;
        OUTPUT BLA;
        OUTPUT Blocking;
        TASK remoteblock := 2;
        NEXTSTATE Idle;

INPUT UBL ;
    PROVIDED remoteblock = 2 ;
        OUTPUT UnblockingInd;
        TASK remoteblock := 3;
        NEXTSTATE Idle;

INPUT BLO ;
    PROVIDED remoteblock = 0 ;
        OUTPUT MaintInd;
        OUTPUT BlockingInd;
```

```
        TASK remoteblock := 1;
        NEXTSTATE Idle;

    INPUT UnblockingResp ;
        PROVIDED remoteblock = 3 ;
        OUTPUT UBA;
        OUTPUT Unblocking;
        TASK remoteblock := 0;
        NEXTSTATE Idle;

    ENDSTATE ;

    ENDPROCESS cgrr;
```

A.2 Cgrr Flowgraph





A.3 Cgrr Flowgraph Text File

cgrr

50

1

50

EDGES AND P-USES

1 1 2 0 /0

2 1 3 0 /0

3 1 4 0 /0

4 9 5 0 /0 22 1 3 3#3#= 26 1 3 3#2#= 29 1 3 3#0#= 32 1 3 3#1#= 36 1 2 2#1#= 40 1 2 2#2#= 43 1 2 2#0#= 47 1 2 2#3#=

5 1 10 0 /0

6 1 11 0 /0

```
7 1 11 0 /0
8 1 11 0 /0
9 1 11 0 /0
10 4 6 1 2 $2$#0#= 7 1 2 $2$#1#= 8 1 2 $2$#2#= 9 1 2 $2$#3#=
11 1 16 0 /0
12 1 17 0 /0
13 1 17 0 /0
14 1 17 0 /0
15 1 17 0 /0
16 4 12 1 3 $3$#0#= 13 1 3 $3$#1#= 14 1 3 $3$#2#= 15 1 3 $3$#3#=
17 1 18 0 /0
18 1 19 0 /0
19 1 20 0 /0
20 1 50 0 /0
21 1 4 0 /0
22 1 23 0 /0
23 1 24 0 /0
24 1 25 0 /0
25 1 4 0 /0
26 1 27 0 /0
27 1 28 0 /0
28 1 4 0 /0
29 1 30 0 /0
30 1 31 0 /0
31 1 4 0 /0
32 1 33 0 /0
33 1 34 0 /0
34 1 35 0 /0
35 1 4 0 /0
36 1 37 0 /0
37 1 38 0 /0
38 1 39 0 /0
39 1 4 0 /0
```

40 1 41 0 /0

41 1 42 0 /0

42 1 4 0 /0

43 1 44 0 /0

44 1 45 0 /0

45 1 46 0 /0

46 1 4 0 /0

47 1 48 0 /0

48 1 49 0 /0

49 1 21 0 /0

50 0

NODES, DEFS, AND C-USES

1 0

2 1 2 3 0 #0#

3 1 2 2 0 #0#

4 0

5 1 1 0

6 0

7 0

8 1 4 0 0

9 1 4 0 0

10 0

11 0

12 1 2 1 0 #0#

13 1 2 1 0 #1#

14 1 2 1 0 #1#

15 1 2 1 0 #0#

16 0

17 0

18 0

19 1 1 0

20 1 4 1 1 1 1

21 1 2 2 0 #0#

22 1 1 0

23 1 4 0 0

24 1 4 0 0

25 1 2 3 0 #0#

26 1 1 0

27 1 4 0 0

28 1 2 3 0 #3#

29 1 1 0

30 1 4 0 0

31 1 2 3 0 #1#

32 1 1 0

33 1 4 0 0

34 1 4 0 0

35 1 2 3 0 #2#

36 1 1 0

37 1 4 0 0

38 1 4 0 0

39 1 2 2 0 #2#

40 1 1 0

41 1 4 0 0

42 1 2 2 0 #3#

43 1 1 0

44 1 4 0 0

45 1 4 0 0

46 1 2 2 0 #1#

47 1 1 0

48 1 4 0 0

49 1 4 0 0

50 0

SYMBOL TABLE

3

1 _var bstatus

2 _var remoteblock

3 _var localblock

NODE TABLE

50

1 S START

2 T Node_2

3 T Node_3

4 S Idle

5 I GRS

6 T Node_6

7 T Node_7

8 O Unblocking

9 O Unblocking

10 T Node_10

11 T Node_11

12 T Node_12

13 T Node_13

14 T Node_14

15 T Node_15

16 T Node_16

17 T Node_17

18 S WaitForGroupResetComplete

19 I GrpResetComplete

20 O GRA

21 T Node_50

22 I UBA

23 O UnblockingConf

24 O Unblocking

25 T Node_25

26 I UnblockingReq

27 O UBL

28 T Node_28

29 I BlockingReq

30 O BLO

31 T Node_31

32 I BLA

33 O BlockingConf

34 O Blocking

35 T Node_35

36 I BlockingResp

37 O BLA

38 O Blocking

39 T Node_39

40 I UBL

41 O UnblockingInd

42 T Node_42

43 I BLO

44 O MaintInd

45 O BlockingInd

46 T Node_46

47 I UnblockingResp

48 O UBA

49 O Unblocking

50 S STOP

A.4 Cgrr Test Unit File For All-Edges Coverage

{(1,2)(2,3)(3,4)(4,5)(4,22)(4,26)(4,29)(4,32)(4,36)(4,40)(4,43)(4,47)(5,10)(6,11)(7,11)(8,11)(9,11)(10,6)(10,7)(10,8)(10,9)(11,16)(12,17)(13,17)(14,17)(15,17)(16,12)(16,13)(16,14)(16,15)(17,18)(18,19)(19,20)(20,50)(21,4)(22,23)(23,24)(24,25)(25,4)(26,27)(27,28)(28,4)(29,30)(30,31)(31,4)(32,33)(33,34)(34,35)(35,4)(36,37)(37,38)(38,39)(39,4)(40,41)(41,42)(42,4)(43,44)(44,45)(45,46)(46,4)(47,48)(48,49)(49,21)}

A.5 Cgrr Test Path File For All-Edges Coverage

ALL-EDGE COVERAGE

1 2 3 4 5 10 6 11 16 12 17 18 19 20 (0) 50

1 2 3 4 29 30 31 4 32 33 34 35 4 26 27 28 4 22 23 24 25 4 5 10 6 11 16 12 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 36 37 38 39 4 5 10 8 11 16 12 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 36 37 38 39 4 40 41 42 4 5 10 9 11 16 12 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 36 37 38 39 4 40 41 42 4 47 48 49 21 4 5 10 6 11 16 12 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 5 10 7 11 16 12 17 18 19 20 (0) 50

1 2 3 4 29 30 31 4 5 10 6 11 16 13 17 18 19 20 (1) 50

1 2 3 4 29 30 31 4 32 33 34 35 4 5 10 6 11 16 14 17 18 19 20 (1) 50

1 2 3 4 29 30 31 4 32 33 34 35 4 26 27 28 4 5 10 6 11 16 15 17 18 19 20 (0) 50

A.6 Cgrr Test Unit File For All-Uses Coverage

DEF_CUSE ASSOCIATIONS: 4

(b, 12, 20)

(b, 13, 20)

(b, 14, 20)

(b, 15, 20)

DEF_PUSE ASSOCIATIONS: 80

(l, 2, (4, 22))	(r, 3, (4, 36))
(l, 2, (4, 26))	(r, 3, (4, 40))
(l, 2, (4, 29))	(r, 3, (4, 43))
(l, 2, (4, 32))	(r, 3, (4, 47))
(l, 25, (4, 22))	(r, 39, (4, 36))
(l, 25, (4, 26))	(r, 39, (4, 40))

(l, 25, (4, 29))	(r, 39, (4, 43))
(l, 25, (4, 32))	(r, 39, (4, 47))
(l, 28, (4, 22))	(r, 42, (4, 36))
(l, 28, (4, 26))	(r, 42, (4, 40))
(l, 28, (4, 29))	(r, 42, (4, 43))
(l, 28, (4, 32))	(r, 42, (4, 47))
(l, 31, (4, 22))	(r, 46, (4, 36))
(l, 31, (4, 26))	(r, 46, (4, 40))
(l, 31, (4, 29))	(r, 46, (4, 43))
(l, 31, (4, 32))	(r, 46, (4, 47))
(l, 35, (4, 22))	(r, 21, (4, 36))
(l, 35, (4, 26))	(r, 21, (4, 40))
(l, 35, (4, 29))	(r, 21, (4, 43))
(l, 35, (4, 32))	(r, 21, (4, 47))
(l, 2, (16, 12))	(r, 3, (10, 6))
(l, 2, (16, 13))	(r, 3, (10, 7))
(l, 2, (16, 14))	(r, 3, (10, 8))
(l, 2, (16, 15))	(r, 3, (10, 9))
(l, 25, (16, 12))	(r, 39, (10, 6))
(l, 25, (16, 13))	(r, 39, (10, 7))
(l, 25, (16, 14))	(r, 39, (10, 8))
(l, 25, (16, 15))	(r, 39, (10, 9))
(l, 28, (16, 12))	(r, 42, (10, 6))
(l, 28, (16, 13))	(r, 42, (10, 7))

(l, 28, (16, 14))	(r, 42, (10, 8))
(l, 28, (16, 15))	(r, 42, (10, 9))
(l, 31, (16, 12))	(r, 46, (10, 6))
(l, 31, (16, 13))	(r, 46, (10, 7))
(l, 31, (16, 14))	(r, 46, (10, 8))
(l, 31, (16, 15))	(r, 46, (10, 9))
(l, 35, (16, 12))	(r, 21, (10, 6))
(l, 35, (16, 13))	(r, 21, (10, 7))
(l, 35, (16, 14))	(r, 21, (10, 8))
(l, 35, (16, 15))	(r, 21, (10, 9))

A.7 Cgrr Test Path File For All-Uses Coverage

ALL-USES COVERAGE

1 2 3 4 5 10 6 11 16 12 17 18 19 20 (0) 50

1 2 3 4 29 30 31 4 5 10 6 11 16 13 17 18 19 20 (1) 50

1 2 3 4 29 30 31 4 32 33 34 35 4 5 10 6 11 16 14 17 18 19 20 (1) 50

1 2 3 4 29 30 31 4 32 33 34 35 4 26 27 28 4 5 10 6 11 16 15 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 5 10 7 11 16 12 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 36 37 38 39 4 40 41 42 4 47 48 49 21 4 5 10 6 11 16 12 17 18 19 20 (0) 50

1 2 3 4 29 30 31 4 32 33 34 35 4 26 27 28 4 22 23 24 25 4 29 30 31 4 5 10 6 11 16 13 17 18 19 20 (1) 50

1 2 3 4 43 44 45 46 4 36 37 38 39 4 40 41 42 4 47 48 49 21 4 43 44 45 46 5 10 7 11 16 12 17 18 19 20 (0) 50

1 2 3 4 29 30 31 4 32 33 34 35 4 26 27 28 4 22 23 24 25 4 5 10 6 11 16 12 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 36 37 38 39 4 5 10 8 11 16 12 17 18 19 20 (0) 50

1 2 3 4 43 44 45 46 4 36 37 38 39 4 40 41 42 4 5 10 9 11 16 12 17 18 19 20 (0) 50

A.8 Infeasible Def-use Associations

Infeasible Def_cuse Associations: None

Infeasible Def-puse Associations: 60

(1, 2, (4, 22))	(r, 3, (4, 36))
(1, 2, (4, 26))	(r, 3, (4, 40))
(1, 2, (4, 32))	(r, 3, (4, 47))
(1, 25, (4, 22))	(r, 39, (4, 36))
(1, 25, (4, 26))	(r, 39, (4, 43))
(1, 25, (4, 32))	(r, 39, (4, 47))
(1, 28, (4, 26))	(r, 42, (4, 36))
(1, 28, (4, 29))	(r, 42, (4, 40))
(1, 28, (4, 32))	(r, 42, (4, 43))
(1, 31, (4, 22))	(r, 46, (4, 40))
(1, 31, (4, 26))	(r, 46, (4, 43))
(1, 31, (4, 29))	(r, 46, (4, 47))
(1, 35, (4, 22))	(r, 21, (4, 36))
(1, 35, (4, 29))	(r, 21, (4, 40))
(1, 35, (4, 32))	(r, 21, (4, 47))
(1, 2, (16, 13))	(r, 3, (10, 7))
(1, 2, (16, 14))	(r, 3, (10, 8))
(1, 2, (16, 15))	(r, 3, (10, 9))

(l, 25, (16, 13))	(r, 39, (10, 6))
(l, 25, (16, 14))	(r, 39, (10, 7))
(l, 25, (16, 15))	(r, 39, (10, 9))
(l, 28, (16, 12))	(r, 42, (10, 6))
(l, 28, (16, 13))	(r, 42, (10, 7))
(l, 28, (16, 14))	(r, 42, (10, 8))
(l, 31, (16, 12))	(r, 46, (10, 6))
(l, 31, (16, 14))	(r, 46, (10, 8))
(l, 31, (16, 15))	(r, 46, (10, 9))
(l, 35, (16, 12))	(r, 21, (10, 7))
(l, 35, (16, 13))	(r, 21, (10, 8))
(l, 35, (16, 15))	(r, 21, (10, 9))

Appendix B

Assumptions on Pascal and SDL Input

B.1 Assumptions on Pasflow Input

Input to pasflow is a single program or procedure with single entry and single exit in which

- definitions of any other procedure, function or constants may exist but are ignored
- there is at least one conditional or repetitive statement
- any function call is for a function whose formal parameters are all pass-by-value
- get, put, reset, rewrite, new, and comments may exist but are ignored
- file and pointer variables may exist but are ignored
- file identifiers may exist in file-oriented i/o but are ignored
- there is no

a) with statements;

b) variant records;

c) procedure calls containing anything other than a variable for each actual parameter;

d) procedural parameters in a procedure or function call;

e) conformant arrays;

f) arrays within records;

g) records within arrays;

h) pointer variables;

i) go to statements;

j) tabs and spaces on string constraints;

k) array referencing in condition part of a conditional or a repetitive statement.

It is assumed that no global variable is updated in a called function.

It is assumed that an abstract definition of each called procedure is given as part of the input to Pasflow. In an abstract definition of a procedure, the manner each formal pass-by-reference parameter takes a value within the called procedure can be determined. That is, for each formal pass-by-reference parameter, there exists an assignment statement "v := expression", where expression is defined over the formal parameters and nonlocal variables of the called procedure. The same applies to each nonlocal (i.e., global) variable updated in the called procedure. Therefore, each global variable updated in a called

procedure must be defined by an assignment statement in the abstract definition of that procedure.

It is assumed that an abstract definition of each function whose definition is given in the Pascal source file is input to Pasflow. In the abstract definition of a called function, the manner the function takes a value is defined by an assignment statement "function_name := expression", where expression is defined over the formal parameters of the function.

There are three interpretations of each procedure call: Type-1 classification is based on the classification given by Rapps and Weyuker for the actual parameters of a procedure call in Pascal programs. That is, the parameter list of a procedure statement contains cuses of all actual pass-by-value and pass-by-reference parameters followed by defs of actual pass-by-reference parameters. Obviously, this classification is sufficient for the data flow oriented test selection criteria proposed by Rapps and Weyuker since these criteria are based on individual du-associations. Other data flow oriented test selection criteria require a more detailed classification since they are based on identification of chains of du-associations. Thus, Type-2 classification for the actual parameters of a procedure statement is based on the abstract definition of each procedure *pn* referred to in the Pascal source file so that the manner each actual pass-by-reference parameter and nonlocal variable takes a value within *pn* can be determined. After substituting actual parameters for formal parameters, the use of the assignment statements in the abstract definition of the called procedure realizes the Type-2 classification. Finally, Type-3 classification of actual parameters in the parameter list of procedure statement is based on the classification proposed for interprocedural or intermodular data flow analysis. This classification is a part of composition of flowgraphs of procedures, modules, or processes, and thus will be used when more than one Pascal procedure are represented by their flowgraphs. Type-3 classification will eliminate the need from interpreting procedure calls since each procedure call will be replaced with connecting edges to the flowgraph of the called procedure.

B.2 Assumptions on Sdlflow Input

The program expects input in the form of the text format defined by CCITT standard Z.100, section 2 through 5, with the following exceptions (section number in Z.100 is in parentheses)

Remote definitions (2.4.1)

System (2.4.2)

Block (2.4.3)

Procedure (2.4.5)

Channel (2.5.1)

Signal route (2.5.2)

Connection (2.5.3)

Signal list definition (2.5.5)

View definition (2.6.1.2)

Save (2.6.5)

Create (2.7.2)

Procedure call (2.7.3)

Partitioning (3.2)

Signal refinement (3.3)

Macro (4.2)

Generic systems (4.3)

Asterisk state (4.4)

Asterisk input (4.6)

Asterisk save (4.7)

Dash nextstate (4.9)

Service (4.10)

Imported and exported value (4.13)

Data kernel language (5.2)

Passive SDL data (5.3)

Extended data definition constructs (5.4.1)

Synonym (5.4.2.3)

Indexed primary (5.4.2.6)

Conditional ground expression (5.4.2.7)

Accessing variables (5.5.2)

Indexed variable (5.5.3.1)

Field variable (5.5.3.2)

Default assignment (5.5.3.3)

Imperative operators (5.4)

Operators not implemented: XOR, REM, //, and =>.

Input to Sdlflow is a single process or procedure with single entry and single exit in which

- definitions of any channel or signal route, procedure, or function may exist but are ignored
- there is at least one transition definition
- there is at least one transition whose state id is *initial*
- there is at least one transition whose next state id is *final*
- there is at least one parameter (constant or variable) for each output interaction
- each parameter of an input interaction is a variable
- any function call is for a function whose formal parameters are all pass-by-value
- comments may exist but are ignored
- there is no

a) variant records;

b) procedure calls containing anything other than a variable for each actual parameter;

c) procedural parameters in a procedure or function call;

d) arrays within records;

e) records within arrays;

f) redundant operators (i.e., x = +y);

g) tabs and spaces on string constraints.

It is assumed that no global variable is updated in a called function.

It is assumed that an abstract definition of each called procedure is given as part of the input to Sdlflow. In an abstract definition of a procedure, the manner each pass-by-reference parameter takes a value within the called procedure can be determined. That is, for each pass-by-reference parameter, there exists an assignment statement "v := expression", where expression is an n-ary function defined over the parameters of the

called procedure. The same applies to each nonlocal variable updated in the called procedure. Therefore, each global variable updated in a called procedure must be shown to be a pass-by-reference parameter in the abstract definition of that procedure and in each procedure statement corresponding to a call to that procedure.

Thus, there are two interpretations of each procedure call:

Type 1 Interpretation: Each parameter is a cuse, and each pass-by-reference parameter is a def

Type 2 Interpretation: After substituting actual parameters for formal parameters, interpret the assignment statements in the abstract definition of the called procedure.

Appendix C

Format of the Abstract Definition File

The abstract definition file must accompany every Pascal source file or SDL process/procedure which is converted into a flowgraph file. Note that quoted identifiers in the text are used to indicate terminal symbols.

An abstract definition file is a list of abstract procedure and/or function definitions.

An abstract definition file must contain

a) an abstract procedure definition for each procedure called within the Pascal source file or SDL process/procedure.

b) an abstract function definition for each of those functions whose definitions can be identified (i.e., those that are user defined).

Each abstract procedure definition is:

1) A syntactically valid Pascal formal procedure declaration.

2) A comma-separated and semicolon terminated list of assignment statements for each pass-by-reference formal parameter and each global variable whose value is updated in the called procedure. Each assignment statement defines one of the pass-by-reference formal parameters or one global variable in terms of the procedure declaration's formal parameters and/or global variables.

A syntactically valid Pascal formal procedure declarations is:

"procedure" procedure_name "("formal_parameter_declaration_part")"

where procedure_name and formal_parameter_declaration_part are defined in a Pascal syntax guide.

A comma-separated list of assignment statements is:

assignment_statement ","..."," assignment_statement ";"

An assignment_statement is:

formal_pbr_parameter_from_parameter_declaration_part ":=" expression

where pbr is an abbreviation for pass-by-reference; procedure and expression are defined in a Pascal syntax guide.

Each abstract function definition is:

1) A syntactically valid Pascal formal function declaration whose formal parameters are all pass-by-value.

2) An assignment statement defines the function in terms of the function's formal parameters.

A syntactically valid Pascal formal function declaration is:

"function" function_name "(" formal_parameter_declaration_part ")" ":"

type_denoter assignment_statement ";"

where function_name, formal_parameter_declaration_part and type_denoter are defined in a Pascal syntax guide.

An assignment_statement is: function_name ":=" expression, where expression is defined in a Pascal syntax guide.

Appendix D

List of Functions

D.1 Path Expression Creation

D.1.1 ph3_read_unit.c

read_a_test_unit

Control the reading of test units and invoke functions to generate feasible complete paths to cover them.

```
void read_a_test_unit(fflow)
char *fflow;                    /* flowgraph file name */
```

edge_unit

Read test units from a branch test unit file *<fn>.flow.bcc*(for branch coverage criterion), and store them in *(Tun_edge) eunit* and *einf.*

```
void edge_unit(fflow)
char *fflow;                    /* flowgraph file name */
```

edge_cov_info

Write the infeasible branch test units to file *<fn>.inf.*

```
void edge_cov_info(punit)
Tun_edge *punit;                /* branch test unit list */
```

node_unit

Read test units from a node test unit file *<fn>.flow.scc* (for statement coverage criterion), and store them in file *work_unit.*

```
void node_unit(fflow)
char *fflow;                    /* flowgraph file name */
```

uses_unit

Read test units from a def-use association test unit file *<fn>.flow.use* (*.def* , *.pus* , *.pcu* or *.cpu,* respectively for all_defs, all_puses, all_puses/some_cuses, all_cuses/some_puses coverage), and store them in file work_unit.

```
void uses_unit(fflow)
char *fflow;                    /* flowgraph file name */
```

D.1.2 ph3_uses_unit.c

dpuse_unit

Read def-puse associations from a def-use association test unit file, and store them in file *work_unit.*

```
void dpuse_unit(funit)
FILE *funit;                    /* test unit file pointer */
```

dcuse_unit

Read def-cuse associations from a def-use association test unit file, and store them in file *work_unit.*

```
void dcuse_unit(funit)
FILE *funit;                    /* test unit file pointer */
```

D.1.3 ph3_testunit.c

tun_edge_add

Add a test unit to the branch test unit list.

```
void tun_edge_add(punit, ind, hnode, tnode)
Tun_edge **punit;               /* branch test unit list */
int ind;                        /* branch test unit index number */
int hnode;                      /* head node of a branch */
int tnode;                      /* tail node of a branch */
```

tun_edge_prt

Print all test units in the branch test unit list.

```
void tun_edge_prt(punit)
Tun_edge *punit;                /* branch test unit list */
```

tun_edge_del

Delete a test unit from the branch test unit list.

```
void tun_edge_del(punit, ind)
Tun_edge **punit;               /* branch test unit list */
int ind;                        /* branch test unit index number */
```

tun_edge_des

Destroy the branch test unit list.

```
void tun_edge_des(punit)
Tun_edge **punit;                    /* branch test unit list */
```

tun_edge_get

Get a test unit from the branch test unit list. This function returns 1 if successful; otherwise, it returns 0.

```
int tun_edge_get(punit, ind, hnode, tnode)
Tun_edge **punit;                    /* branch test unit list */
int *ind;                            /* branch test unit index number */
int *hnode;                          /* head node of a branch */
int *tnode;                          /* tail node of a branch */
```

D.1.4 ph3_multi_pexp.c

edge_pexp

Generate a path expression to cover the branch test unit in the branch test unit list *(Tun_edge) eunit* and save it in *path *whole* and *char pthexp[]*. This function returns 1 if the path expression is successfully generated; otherwise, it returns 0.

```
int edge_pexp()
```

save_to_pthexp

Convert the path expression in data structure *path* to a string.

```
void save_to_pthexp(pthexp, R)
char *pthexp;                        /* path expression string */
path *R;                             /* path expression in data structure path */
```

node_pexp

Generate a path expression to cover the node test unit in file *work_unit* and save it in *path *whole* and *char pthexp[]*. This function returns 1 if the path expression is successfully generated; otherwise, it returns 0.

```
int node_pexp()
```

update

Remove the test unit with the given index number from the test unit file *work_unit*.

```
void update(order)
int order;                           /* test unit index number */
```

D.1.5 ph3_searchpath.c

edge_subp

Find a feasible subpath starting from the entry node to cover the given branch test unit. This function returns a string of a sub-path if found; otherwise, it returns 0.

```
char *edge_subp(bf, fnode, snode)
Bflow_type *bf;          /* flowgraph in Bflow */
long fnode;              /* head node of an edge */
long snode;              /* tail node of an edge */
```

searchpath

Search backward to find a subpath starting from *node* which satisfies path predicate *ccond*. This function returns 1 if found; otherwise, it returns 0.

```
int searchpath(bf, node, ccond, subpath)
Bflow_type *bf;          /* flowgraph in Bflow */
Tid_type node;           /* node id */
Eval_weld *ccond;        /* path predicate */
Pth_type subpath;        /* sub-path */
```

cycling

Check whether adding a node into a subpath results a cycle. if true, this function returns 1; otherwise, it returns 0.

```
int cycling(bf, subpath, node)
Bflow_type *bf;          /* flowgraph in Bflow */
Pth_type subpath;        /* sub-path */
Tid_type node;           /* node id */
```

self_circuit

Check whether a node is in a self circuit of a flowgraph. A self circuit is a path starting and ending with the same node, and except for this node, the in-degree and out-degree of each node in the path are 1. If true, this function returns 1; otherwise, it returns 0.

```
int self_circuit(bf, node)
Bflow_type *bf;          /* flowgraph in Bflow */
Tid_type node;           /* node id */
```

subp_eval_copy

Copy *source_cond* to *dest_cond*.

```
void subp_eval_copy(dest_cond, source_cond)
Eval_weld *dest_cond;    /* destination */
```

```
Eval_weld source_cond;          /* source */
```

subp_pth_copy

Copy *source_pth* to *dest_pth*.

```
void subp_pth_copy(dest_pth, source_pth)
Pth_type dest_pth;              /* destination */
Pth_type source_pth;            /* source */
```

uses_subp

Find a feasible subpath starting from the entry node to cover the given def-use association test unit. This function returns a string of the subpath if found; otherwise, it returns 0.

```
char *uses_subp(bf, cur_node)
Bflow_type *bf;                 /* flowgraph in Bflow */
long cur_node;                  /* node */
```

node_subp

Find a feasible subpath starting from the entry to cover the given node test unit. This function returns a string of the subpath if found; otherwise, it returns 0.

```
char *node_subp(bf, cur_node)
Bflow_type *bf;                 /* flowgraph in Bflow */
long cur_node;                  /* node */
```

D.1.6 ph3_reduce.c

ReadGraph

Read flowgraph from file *graph* and store it in *G* in matrix form; store the number of nodes of the flowgraph in *nn*. In File *graph*, the flowgraph is stored as adjacency lists.

```
void ReadGraph(G, nn)
int G[MAXNODES][MAXNODES]; /* flowgraph in matrix form */
int *nn;                        /* number of nodes in the flowgraph */
```

simple_reduce

Generate a path expression between two nodes. This function returns NULL when no path expression is generated; otherwise, it returns pointer to the path expression.

```
path *simple_reduce(ro, ta)
int ro, ta;                     /* two nodes */
```

no_repeat_reduce

Generate a path expression between two nodes without repeating these two nodes. This function returns NULL when no path expression is generated; otherwise, it returns a pointer to the path expression.

```
path *no_repeat_reduce(ro, ta)
int ro, ta;                    /* two nodes */
```

getnode

Return a pointer to a node.

```
path *getnode(n)
int n;                         /* node id */
```

savepathpt

Manage pointer to each node in path *(struct pathpt) *pthp.*

```
void savepathpt(ppt)
path *ppt;                     /* path */
```

freepathpt

Free pointer to each node in path *(struct pathpt) *pthp* .

```
void freepathpt()
```

dot

Return (R1• R2).

```
path *dot(R1,R2)
path *R1;                      /* path expression */
path *R2;                      /* path expression */
```

star

Return (R)*.

```
path *star(R)
path *R;                       /* path expression */
```

un

Return (R1 U R2).

```
path *un(R1, R2)
path *R1;                      /* path expression */
```

```
path *R2;                    /* path expression */
```

GetPathExp

Generate path expressions from *(int) root* (global variable) to all other reachable nodes and save them in *(path *) solution[]*. If successful, it returns 1; otherwise, it returns 0.

```
int GetPathExp()
```

initialize

Initialize global variables.

```
void initialize()
```

DFS

Perform depth-first search on subgraph of *G* rooted at *root*; store a reverse-post-ordering of the nodes in the array *RPost*.

```
void DFS(G, root, RPost, num)
int G[MAXNODES][MAXNODES]; /* flowgraph in matrix form */
int root;                  /* root node */
int RPost[MAXNODES];       /* reverse-Post-ordering of the nodes */
int *num;                  /* ordering number */
```

reorder

Reorder the flowgraph *G* based on the reverse-Post-ordering of the nodes.

```
void reorder()
```

FindDomAux

Initialize global variables and call *FindDominators*.

```
void FindDomAux(G)
int G[MAXNODES][MAXNODES]; /* flowgraph in matrix form */
```

FindDominators

Find the immediate dominator and the set of dominators of each node in the flowgraph. Store this information in the global variables *IDom* and *Dominators*.

```
void FindDominators(G, rPOST, NumNodes)
int G[MAXNODES][MAXNODES]; /* flowgraph in matrix form */
int rPOST[MAXNODES];       /* reverse-post-ordering of the nodes */
int NumNodes;              /* number of nodes in the flow-graph */
```

ReducePath

Reduce a subgraph of *G*. if it is non-reducible, returns 0; otherwise, it returns 1.

```
int ReducePath(r)
int r;                          /* root node */
```

reducible

Check if all the edges going into *node* come from the same region. If so, returns 1; otherwise, returns 0.

```
int reducible(node)
int node;                       /* node */
```

contract

Contract *node* into a region.

```
void contract(node)
int node;                       /* node */
```

eval

Return an unambiguous path expression from root to *node*.

```
path *eval(node)
int node;                       /* node */
```

finalize

Finalize the path expression for each node in the flowgraph.

```
void finalize()
```

PPrintpath

Output path expression *e* and remove the duplication of node *n* in *e*.

```
void PPrintpath(e, n)
path *e;                        /* path expression */
int n;                          /* node */
```

TKOUT

Remove node *n* from path expression *pe*.

```
path *TKOUT(pe, n)
path *pe;                       /* path expression */
int n;                          /* node */
```

takeout

Delete node *n* from path expression *e*.

```
void takeout(e, n)
path *e;                    /* path expression */
int n;                      /* node */
```

cpy

Copy path expression *p* to *e*.

```
void cpy(e, p)
path *e, *p;                /* path expression */
```

printpath

Print a path expression.

```
void printpath(R)
path *R;                    /* path expression */
```

D.1.7 ph3_uses_pexp.c

dpuse_pexp

Generate a path expression to cover the def-puse test unit in file *work_unit*and save it in *path *whole* and *char pthexp[]*. This function returns 1 ifthe path expression is successfully generated for it; otherwise, it returns 0.

```
int dpuse_pexp()
```

dcuse_pexp

Generate a path expression to cover the def-cuse test unit in file *work_unit* and save it in *path *whole* and *char pthexp[]*. This function returns 1 if the path expression is successfully generated for one of the test units; otherwise, it returns 0.

```
int dcuse_pexp()
```

input_subpath

Find a feasible sub-path that ends at node *fnode*. This function returns 1 if found; otherwise, it returns 0.

```
int input_subpath(fnode)
int fnode;                  /* node */
```

save_and_show

Convert the path expression in *(path *) whole* to a string *(char *) pthexp*, and display the path expression.

```
void save_and_show()
```

D.1.8 ph3_def_clr.c

def_clr_path

Generate a def-clear path expression from *nod1* to *nod2* wrt. *str.*

```
path *def_clr_path(nod1, nod2, str)
int nod1, nod2;                    /* nodes */
char *str;                         /* variable name */
```

D.1.9 ph3_dupath.c

dupath_unit

Read a test unit from a du-path test unit file *<fn>.flow.dup*, and save them in file *work_unit.*

```
void dupath_unit(fflow)
char *fflow;                       /* name of the flowgraph file */
```

dupath_pexp

Generate a path expression to cover the du-path test unit in file *work_unit* and save it in *path *whole* and *char pthexp[].* This function returns 1 if the path expression is successfully generated; otherwise, it returns 0.

```
int dupath_pexp()
```

D.2 Partial Symbolic Execution

D.2.1 ph3_PSE.c

PSE_main

Invoke partial symbolic evaluation function.

```
void PSE_main()
```

PSE

Evaluate the feasibility of the path expression *(char *) pthexp.*

```
void PSE()
```

geta

Get a node from a path expression.

```
void geta(pi)
int pi;                          /* position */
```

chk_cuses

Symbolically execute a node and store PPP in *(Eval_weld) ep.*

```
void chk_cuses(a)
int a;                           /* node */
```

chk_puses

Symbolically execute an edge and store PPP in *(Eval_weld) ep.*

```
void chk_puses(last, a)
int last, a;                     /* head and tail node of an edge */
```

rpl_cuses

Replace variable *var* with expression *value* in expressions *(Eval_weld) ep.*

```
void rpl_cuses(var, value)
Tid_type var;                    /* variable id */
Exp_tree_node *value;            /* expression */
```

PSE_chk_rpl

Replace variable *var* with expression *value* in expression *exp.*

```
Exp_tree_node *PSE_chk_rpl(var, value, exp)
Tid_type var;                    /* variable id */
Exp_tree_node *value;            /* expression */
Exp_tree_node *exp;              /* expression */
```

rpl_input

Mark variable *var* as 'input' in expressions *(Eval_weld) ep.*

```
void rpl_input(var)
Tid_type var;                    /* variable id */
```

var_input

Mark variable *var* as 'input' in the expression *exp.*

```
void var_input(var, exp)
```

```
Tid_type var;                    /* variable id */
Exp_tree_node *exp;              /* expression */
```

restore_input

Remove 'input' mark for variables in expression *poi*.

```
void restore_input(poi)
Eval_weld poi;                   /* expression */
```

del_input

Remove 'input' mark for variables in an expression.

```
void del_input(exp)
Exp_tree_node *exp;              /* expression */
```

chk_def

Check if there is any variable defined in node *a*, and invoke the function to mark the defined variables as 'ambiguous' in path predicates.

```
void chk_def(a)
int a;                           /* node */
```

prcs_new_var

Mark variable *var* as 'ambiguous' in expression *(Eval_weld) ep*.

```
void prcs_new_var(var)
char *var;                       /* variable name */
```

var_mark

Mark variable *var* as 'ambiguous' in the expression *exp*.

```
void var_mark(var, exp)
char *var;                       /* variable name */
Exp_tree_node *exp;              /* expression */
```

restore_ambig

Remove 'ambiguous' mark for variables in expression *(Eval_weld) ep*.

```
void restore_ambig()
```

del_mark

Remove 'ambiguous' mark for variables in expression *exp*.

```
void del_mark(exp)
Exp_tree_node *exp;                    /* expression */
```

PSE_exp_dup

Duplicate an expression.

```
Exp_tree_node *PSE_exp_dup(value)
Exp_tree_node *value;                  /* expression */
```

PSE_prt_eval

Print the partial symbolic path predicates *(Eval_weld) ep.*

```
void PSE_prt_eval()
```

eval_destroy

Destroy the list of expressions.

```
void eval_destroy(ep)
Eval_weld ep;                          /* list of expressions */
```

D.3 Complete Path Generation

D.3.1 ph3_compl_pth.c

Comm

Invoke *compl_pth.*

```
void Comm()
```

compl_pth

Initialize global variables and invoke complete path generator.

```
void compl_pth()
```

automatic

Generate complete paths from a path expression automatically, and invoke the function to check their feasibility.

```
void automatic()
```

choosepath

Select a complete path from path expression *(path *)whole* and save it in *(char *) pthstr.*

```
void choosepath()
```

chk_unit_n

Return the number of + operators in a path expression.

```
int chk_unit_n(R)
path *R;                        /* path expression */
```

showpath

Count the number of * and + operators in a path expression and save them in *(int) strnum* and *(int) unnum* respectively.

```
void showpath(R)
path *R;                        /* path expression */
```

D.4 Symoblic Execution

D.4.1 ph3_SYME.c

SYME_main

Evaluate the feasibility of the complete path stored in *(char *) pthstr.*

```
void SymE_main()
```

SYME_chk_cuses

Symbolically execute a node. Store PP and POUT in *(Eval_weld) ep* and *(Eval_weld) outep* respectively.

```
void SYME_chk_cuses(a)
int a;                          /* node */
```

SYME_chk_puses

Symbolically execute an edge. Store PP and POUT in *(Eval_weld) ep* and *(Eval_weld) outep* respectively.

```
void SYME_chk_puses(last, a)
int last;                       /* tail node of an edge */
int a;                          /* head node of an edge */
```

SYME_rpl_cuses

Replace variable *var* with expression *value* in the list of expressions *poi.*

```
void SYME_rpl_cuses(poi, var, value)
```

```
Eval_weld poi;              /* list of expressions */
Tid_type var;               /* variable id */
Exp_tree_node *value;       /* expression */
```

SYME_chk_rpl

Replace variable *var* with expression *value* in expression *exp*.

```
Exp_tree_node *SYME_chk_rpl(var, value, exp)
Tid_type var;               /* variable id */
Exp_tree_node *value;       /* expression */
Exp_tree_node *exp;         /* expression */
```

SYME_rpl_input

Mark variable *var* as input in the list of expressions *poi*.

```
void SYME_rpl_input(poi, var)
Eval_weld poi;              /* list of expressions */
Tid_type var;               /* variable id */
```

SYME_var_input

Mark variable *var* as input in expression *exp*.

```
void SYME_var_input(var, exp)
Tid_type var;               /* variable id */
Exp_tree_node *exp;         /* expression */
```

SYME_exp_dup

Duplicate an expression.

```
Exp_tree_node *SYME_exp_dup(value)
Exp_tree_node *value;       /* expression */
```

SYM_prt_eval

Print out the PP stored in *(Eval_weld) ep* and POUT stored in *(Eval_weld) outep*.

```
void SYM_prt_eval()
```

D.5 Path Predicate Simplification and Evaluation

D.5.1 ph3_evaluate.c

constant_eval

Evaluate the feasibility of a path predicate *(Eval_weld) ep* based on the constant information. This function returns 1 if the path predicate is infeasible; returns 0 if it is feasible; returns 2 if it is inconclusive.

```
int constant_eval()
```

exp_eval

Evaluate the feasibility of a constraint. This function returns 1 if the constraint is infeasible; returns 0 if it is feasible; returns 2 if it is inconclusive.

```
int exp_eval(exp)
Exp_tree_node *exp;              /* expression */
```

kill_cur

Remove one constraint *(Eval_weld) cur* from path predicate *(Eval_weld) ep* .

```
void kill_cur()
```

constant_rpl

Replace variable *var* with expression *value* in path predicates *(Eval_weld) ep*.

```
void constant_rpl(var, value)
Tid_type var;                    /* variable id */
Exp_tree_node *value;            /* expression */
```

var_chk_rpl

Replace variable *var* with expression *value* in expression *exp*.

```
Exp_tree_node *var_chk_rpl(var, value, exp)
Tid_type var;                    /* variable id */
Exp_tree_node *value;            /* expression */
Exp_tree_node *exp;              /* expression */
```

PSE_constant_eval

Evaluate the feasibility of a path predicate with 'ambiguous' variable *(Eval_weld) ep* based on the constant information. This function returns 1 if path predicate is infeasible; returns 0 if it is feasible; returns 2 if it is inconclusive.

```
int PSE_constant_eval()
```

PSE_exp_eval

Evaluate the feasibility of one constraint with 'ambiguous' variable. This function returns 1 if the constraint is infeasible; returns 0 if it is feasible; returns 2 if it is inconclusive.

```
int PSE_exp_eval(exp)
Exp_tree_node *exp;                    /* expression */
```

D.5.2 ph3_simplifier.c

simplify

Simplify the path predicate in *(Eval_weld) ep*, and form a linear programming problem. The result is stored in file *minit_input* which will be used as the input for the LP package.

```
void simplify()
```

input_num

Return the number of input variables.

```
int input_num()
```

display_exp

Display a simplified expression.

```
void display_exp(C)
float *C;                              /* coefficients of an expression */
```

count

Simplify an expression, and return the coefficients of the simplified expression.

```
float *count(cond)
Exp_tree_node *cond;                   /* expression */
```

D.6 Path File Generation

D.6.1 ph3_output.c

output_evaluate

Evaluate all the symbolic output expressions stored in *(Eval_weld) outep*.

```
void output_evaluate()
```

get_outp_value

Return the value of an output expression.

```
float get_outp_value(exp)
Exp_tree_node *exp;                    /* expression */
```

test_data_display

Display the value of all input variables.

```
void test_data_display(w)
float *w;                              /* data for input variables */
```

D.6.2 ph3_pathfile.c

get_pathfile_name

Get the path file name, which is stored in *(char *) pathflow.*

```
void get_pathfile_name()
```

empty_pathfile

Initialize the path file.

```
void empty_pathfile()
```

path_file

Write the feasible complete path and the input/output data into the path file.

```
void path_file()
```

chk_input

Find input data in a node and write them into the path file.

```
void chk_input(pathfl, nod)
FILE *pathfl;                          /* path file pointer */
int nod;                               /* node */
```

chk_output

Find output data in a node and write them into the path file.

```
void chk_output(pathfl, nod)
FILE *pathfl;                          /* path file pointer */
int nod;                               /* node */
```

simple_path_file

Write the feasible complete path into the path file, when there is no input/output in the complete path.

```
void simple_path_file()
```

D.7 Linear Programing

D.7.1 ph3_minmain.c and ph3_minit.c

Solve the linear programming (LP) problem:

```
max C X subject to
AX <= B
X >= 0
```

This LP package takes file *minit_input* as input, and store the result in *(float) x*.

D.8 Coverage Check

D.8.1 ph3_cov_chk.c

cov_chk

Invoke different coverage check functions based on the type of test units.

```
void cov_chk()
```

edge_cov_chk

From a set of edge test units, find the test units covered by the given complete path in *(char *) pthstr*, and remove it from the test unit set.

```
void edge_cov_chk()
```

chk_edge

From a set of edge test units, find all the test units covered by complete path *(char *) pthstr*. This function returns 1 if found any; otherwise, it returns 0.

```
int chk_edge(aj, ak)
int aj, ak;                    /* head and tail node of an edge */
```

node_cov_chk

From a set of node test units, find all the test units covered by complete path *(char *) pthstr* , and remove them from the test unit set.

```
void node_cov_chk()
```

chk_node

From a set of node test units, find all the test unit covered by complete path *(char *) pthstr.* The function returns 1 if found; otherwise, it returns 0.

```
int chk_node(node)
int node;                    /* node */
```

update_inf

Update the set of test units for coverage calculation.

```
void update_inf(order)
int order;                   /* index number of a test unit */
```

cuse_cov_chk

From a set of def-use association test units, find all the test units covered by complete path *(char *) pthstr*, and remove them from the test unit set.

```
void cuse_cov_chk()
```

chk_cuse

From a set of def-use association test units, find all the test units covered by complete path *(char *) pthstr.* This function returns 1 if found any; otherwise, it returns 0.

```
int chk_cuse(str, ai, aj)
char *str;                   /* variable name */
int ai;                      /* node */
int aj;                      /* node */
```

nod_def_chk

Check if a variable *str* is defined in node *nod.* This function returns 1 if it is; otherwise, it returns 0.

```
int nod_def_chk(str, nod)
char *str;                   /* variable name */
int nod;                     /* node */
```

puse_cov_chk

From a set of def-use association test units, find all the test units covered by complete path *(char *) pthstr*, and remove them from the test unit set.

```
void puse_cov_chk()
```

chk_puse

From a set of def-puse association test units, find if there is any test unit covered by complete path *(char *) pthstr*. This function returns 1 if found any; otherwise, it returns 0.

```
int chk_puse(str, ai, aj, ak)
char *str;                    /* variable name */
int ai;                       /* node */
int aj, ak;                   /* head and tail node of an edge */
```

dupath_cov_chk

From a set of du-path test units, find all the test units covered by complete path *(char *) pthstr*, and remove them from the test unit set.

```
void dupath_cov_chk()
```

chk_dupath

Check if a du-path test unit is covered by the complete path *(char *) pthstr* (global variable). This function returns 1 if it is; otherwise, it returns 0.

```
int chk_dupath(pth)
int *pth;                     /* du-path */
```

D.9 Others

D.9.1 ph3_main.c

main

Manage options and parameters in the command line, and invoke functions to generate feasible complete paths.

```
void main(argc, argv)
int argc;
char **argv;
```

clearfile

Remove internal temporary files.

```
void clearfile()
```

D.9.2 ph3_inflow.c

print_flow_file

Print information stored in a Bflow data structure, and create an internal file *graph* in which the flowgraph is simply stored as adjacency lists.

```
void print_flow_file(bf)
Bflow_type *bf;                    /* Bflow */
```

print_c_uses

Print information about nodes, defs, and c-uses in a Bflow data structure.

```
void print_c_uses(bf)
Bflow_type *bf;                    /* Bflow */
```

D.9.3 ph3_utility.c

add_elem

Append a node to a path.

```
void add_elem(pth_id, node_id)
Pth_type pth_id;                   /* path */
Tid_type node_id;                  /* node id */
```

Note that the list order of functions within a module is according to their positions in the module.

Appendix E

Files and Data Structures

E.1 Input Files

There are two input files for phase3: flowgraph file and test unit file.

E.1.1 flowgraph File

The flowgraph file is generated by program *sdlflow*. The file name is *<system name>.flow*. The format of the flowgraph file is described in [29].

E.1.2 Test Unit File

The test unit file is generated by program *nuflowc*. For branch coverage criteria, the file name is *<system name>.flow.bcc*. The syntax of the test unit file is specified by the following grammar.

test_unit_file ::= '{' { branch } + '}' '\n' { bcc_tp } +

/* where inclusion in { and } means 0 or more occurrences of the enclosed unit. When the closing } is followed by +, it signifies 1 or more occurrences of the enclosed unit. */

branch ::= '(' ' ' node_number ' ' ',' ' ' node_number ' ' ')' ' '

bcc_tp ::= path_number '.' ' ' '{' node_list '}' '\n'

node_list ::= { ' ' node_number ' ' } +

node_number ::= UNSIGNEDINT

/* where UNSIGNEDINT is a natural number. */

path_number ::= UNSIGNEDINT

E.2 Output File

There is one output file for phase3: test path file.

E.2.1 Test Pathfile

The file name is *<system name>.path*. The syntax of the test path file is specified by the following grammar.

path_file ::= { paths_per_coverage }

/* where inclusion in { and } means 0 or more occurrences of the enclosed unit. When the closing } is followed by +, it signifies 1 or more occurrences of the enclosed unit. */

paths_per_coverage::='\t' coverage_header '\n' { path_info } +

coverage_header::= TEXTLINE

/* where the text in TEXTLINE is the description of the test path generation criterion. */

path_info ::= node_info { node_info } + '\n'

node_info ::= (node_number | node_name) { parameter_info } ' '

/* The symbol '|' above represents an alternative and '(' and ')' are used for grouping. */

node_number ::= UNSIGNEDINT

/* where UNSIGNEDINT is a natural number representing the node. */

node_name ::= IDENTIFIER

/* where IDENTIFIER is a string of letters and digits starting with a letter. This results in the specification of a path using node names instead of node numbers. */

parameter_info::= '(' parameter_value { ',' parameter_value } ')'

parameter_value::= (+ | -) unsigned_constant | _

/* where '_' means that the value is not decided yet. */

unsigned_constant::=unsigned_integer | character_string | boolean_value

unsigned_integer::= (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) +

character_string::= "" (any character except ' | ") * ""

boolean_value ::= TRUE | FALSE

E.3 Internal File

For branch coverage criteria, there are two internal files which are used to transfer data between functions: *graph* and *minit_input*.

E.3.1 Graph

File graph is generated by function *print_flow_file()* in module ph3_inflow.c and used by function *ReadGraph()* in ph3_reduce.c. It represents an adjacency list representation of a flowgraph in the following format:

- each line consists of a node, followed by a list of successors to that node, followed by a zero;
- the end of the file is signalled by a zero.

E.3.2 Minit_input

File *minit_input* is used by the linear programming package. Corresponding to the linear programming (LP) problem:

max C X subject to

AX <= B

X >= 0

the file is in the following format:

<# of variables> <# of inequality constraints> <# of equality constraints>

<matrix A>

<vector B>

<vector C>

Example:

Corresponding to the LP problem

Maxmize $6x_1 + x_2 - x_3 - x_4$ subj. to

$x_1 + 2x_2 + x_3 + x_4 <= 5$

$3x_1 + x_2 - x_3 <= 8$

$x_2 + x_3 + x_4 = 1$

$x_1, x_2 x_3, x_4 >= 0$

the file is:

```
4 2 1

1 2 1 1
3 1 -1 0
0 1 1 1

5 8 1
6 1 -1 -1
```

E.4 Data Structure

E.4.1 Bflow

Bflow data structure is used to store information of a flowgraph. The description of Bflow can be found in [25].