PROGRAM-BASED, STRUCTURAL TESTING OF SHARED MEMORY PARALLEL PROGRAMS

by

Cheer-Sun David Yang

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer and Information Sciences

Summer, 1999

PROGRAM-BASED, STRUCTURAL TESTING OF SHARED MEMORY PARALLEL PROGRAMS

by

Cheer-Sun David Yang

Approved: ______________________________

Errol L. Lloyd, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: ______________________________

John C. Cavanaugh, Ph.D.
Vice Provost for Academic Programs and Planning

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____________________________________
Lori L. Pollock, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____________________________________
Errol L. Lloyd, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____________________________________
B. David Saunders, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____________________________________
David Binkley, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

First and foremost, hats off to Dr. L. L. Pollock, my major advisor, who has virtually been the technical referee, coach, and most important of all, the cheerleader during the past. Without any of the roles she played, I would not be able to finish my dissertation alone. Her technical excellence and charisma have influenced my way of thinking technically.

Thanks go to my two sisters Sonia and Maria, both younger, but finished their Ph.D. degrees earlier than me. They have become my role models especially when I began to realize how challenging it is to get a Ph.D. Without their encouragement, I would not have the courage to quit my job and return to school.

My parents have been my long-term supporters. They have long been my heros all these years for raising, educating, and above all, tolerating me. They deserve my whole-hearted love and highest respect.

I am indebted to James B. Fenwick Jr., who provided extremely valuable assistance about SUIF when I needed it the most. Again, Jay, thanks!

I would like to acknowledge my fellow graduate students and friends in the High Performance Computing Lab: John Graham, who provided me program examples that use UNIX *threads*; Amie L. Souter, who implemented the *Path Visualizer* of the *della pasta* testing tool.

Thanks go to all faculty members! Because of you all, the CIS department at UD has become a wonderful place to receive an excellent education and an enjoyable place to get “stuck” for all these years.

To all committee members, thanks for reading my somewhat lengthy proposal and my dissertation. Your comments have been extremely valuable.

Finally, this research work was partially funded through the collaborative participation in the Advanced Telecommunication/Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under Cooperative Agreement DA AL01-96-2-0002. [1] With this grant, it was possible for me to explore the new technical area of parallel program testing.

[1] The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

TABLE OF CONTENTS

LIST OF FIGURES . . . . **xii**
LIST OF TABLES . . . . **xvi**
ABSTRACT . . . . **xvii**

Chapter

1 INTRODUCTION **1**
2 BACKGROUND **6**

2.1 Software Testing 6

2.1.1 Testing Methodologies 7

2.1.1.1 Overview of Static Analysis Methodologies 8
2.1.1.2 Overview of Dynamic Testing Methodologies 9

2.1.2 Testing Criteria for Path Coverage Methodologies 11
2.1.3 Subsumption Hierarchy for Path Coverage Criteria 19
2.1.4 Research Directions in Path Testing 20

2.2 Parallel Computing 22

2.2.1 Parallel Programming Paradigms 23

2.2.1.1 Explicit vs. Implicit Parallelism 24
2.2.1.2 Data Parallelism vs. Control Parallelism 26
2.2.1.3 Shared Memory vs. Message Passing Systems 27

2.2.2 Memory Consistency Models for Shared Memory Systems . . . 31

2.3 Testing and Programming Models Targeted by this Research 34

3 **PARALLEL PROGRAM TESTING AND ANALYSIS: CHALLENGES AND STATE OF THE ART** **36**

3.1 Potential Errors in Parallel Programs 36

3.1.1 Race Conditions 36
3.1.2 Deadlock 39

3.2 Detecting Errors via Debugging Tools 42

3.2.1 Detecting Race Conditions 43
3.2.2 Deadlock Detection 45
3.2.3 Execution Replay and Nondeterminism 46

3.3 Structural Testing 49

3.3.1 Concurrent Program Testing 49
3.3.2 Implicitly Parallel Programs 55
3.3.3 More General Explicitly Parallel Codes 55

3.3.3.1 Static Analysis of Explicitly Parallel Programs . . . 55
3.3.3.2 Theoretical Framework for Parallel Program Testing 60
3.3.3.3 Applicability of Path Coverage Testing Methodologies 62

3.3.4 Timing-Related Testing: Nondeterministic versus Deterministic Execution 64
3.3.5 The Size of a Temporal Test Suite 67

3.4 Summary 67

4 **A STRUCTURAL TESTING FRAMEWORK FOR PARALLEL PROGRAMS** **70**

4.1 Parallel Program Model and Notation 71

4.1.1 Parallel Programming Language Model 71

4.1.2 Program Representation for Static Analysis 74

4.1.2.1 Shared Memory Programs 75

4.1.3 Path Test Coverage for a Parallel Program 80

4.2 Software Testing Framework 82

4.2.1 Parallel Program Testing Paradigm 82
4.2.2 Test Data Adequacy Criteria 84
4.2.3 Partial Ordering of Coverage Criteria 86
4.2.4 Nondeterminism and the Testing Process 87
4.2.5 Summary 89

5 FINDING A DU-PATH FOR SHARED MEMORY PARALLEL PROGRAMS 90

5.1 Inherent Problem Exhibited 91
5.2 A Classification of du-path Coverage for Parallel Programs 93

5.2.1 Acceptability of a du-path Coverage 94
5.2.2 W-runnability of a du-path Coverage 95

5.3 Computational Complexity Issues of du-path Testing 96
5.4 State of the Art in Path Testing Methodologies 112

5.4.1 Path Testing of Sequential Programs 112
5.4.2 Path Analysis of Concurrent Programs 123

5.5 Applying Path Finding Algorithms to Parallel Programs 125
5.6 A Hybrid Approach 128

5.6.1 Data Structures 129
5.6.2 The du-path Finding Algorithm 132
5.6.3 Example of Generating a $PATH_a$ 134
5.6.4 Correctness and Complexity 135

5.7 Summary and Contributions 138

6 TEMPORAL TESTING 140

6.1 Delay Execution Approach 141

6.1.1 Execution Timing Chart 141
6.1.2 Locating Delay Points 142
6.1.3 Inherent Problem of Delay Insertion 144

6.2 Redundant Delays 146

6.2.1 Redundant Delay Theorem 149
6.2.2 Detecting Redundant Delays 154
6.2.3 Computing Reach(n) 155
6.2.4 Computing Intrusion and Redundancy 156
6.2.5 An Example 157
6.2.6 Proof of Correctness 158
6.2.7 Complexity of the Algorithm for Identifying Redundant Delays 159
6.2.8 Eliminating Test Cases By Identifying Redundant Delays . . . 159

6.3 Generating and Tracking Temporal Test Cases 161

6.3.1 Representing A Temporal Test Case Suite 161
6.3.2 Using the Test Case Suite Representation 163

6.4 Summary and Contributions 165

7 THE DESIGN AND IMPLEMENTATION OF DELLA PASTA . 167

7.1 The Static Analyzer 167

7.1.1 PPFG Construction 168

7.1.1.1 The SUIF Representation Hierarchy 168
7.1.1.2 Symbolic information 171
7.1.1.3 Annotations 172
7.1.1.4 PPFG based on SUIF 172

7.1.2 Du-pair Computation 174

7.1.3 Du-path Coverage and Classification 177

7.2 Path Visualizer Illustrated 177
7.3 Path Handler 181
7.4 Summary and Current Status 183

8 EMPIRICAL STUDIES OF TESTING METHODOLOGIES FOR SHARED MEMORY PROGRAMS 185

8.1 Others' Experimental Results on Testing Methodologies 186

8.1.1 Sequential Programs 186
8.1.2 Temporal Testing of Concurrent Programs 187

8.2 Experimental Studies Targeting Four Research Questions 188

8.2.1 Producer/Consumer 189

8.2.1.1 Problem Description 191
8.2.1.2 Finding a du-path 191
8.2.1.3 Redundant Delay Points 193

8.2.2 Pipelining 195

8.2.2.1 Problem Description 195
8.2.2.2 Finding a du-path 195
8.2.2.3 Redundant Delay Points 197

8.2.3 TCP and UDP Name Caching Service 198

8.2.3.1 Problem Description 198
8.2.3.2 Finding a du-path 203
8.2.3.3 Redundant Delay Points 203

8.3 Summary 203

9 CONCLUSIONS AND FUTURE DIRECTIONS 206

9.1 Summary of the Work 206
9.2 Contributions 207

9.3 Future Directions 208

9.3.1 Extension to Other Parallel Programming Paradigms 210

9.3.1.1 Rendezvous Communication 210
9.3.1.2 Message Passing Programs 212

9.3.2 Future Directions in Incremental Testing of Parallel Programs 214

BIBLIOGRAPHY 217

LIST OF FIGURES

2.1 Software Testing Problem Illustrated 7

2.2 All-statement and All-branch Coverage Example 12

2.3 All-du-path and LCSAJ Example 14

2.4 All-path Coverage Problem Illustrated 16

2.5 A Program with Infeasible Paths 18

2.6 Partial Ordering of Coverage Criteria 19

2.7 Explicit Parallelism with Parallel Sections 24

2.8 Implicit Parallelism 24

2.9 Code Segment Using Compiler Directives 25

2.10 A Program Segment in an SPMD Program 27

2.11 Statement Orderings Illustrated 30

3.1 Data Race Example 37

3.2 An Internally Determinate Program 38

3.3 A Non-internally Determinate Program 39

3.4 A Program Example with Synchronization 40

3.5 A Program Example with a Synchronization Error 41

3.6 Example of Order/Conflict Graph 44

3.7 Examples of SYN-sequences 47

3.8 Example Representation of State Transitions for Concurrent Program 50

3.9 An Example Petri Net 53

3.10 An Example ER Net 53

3.11 Illustration of Finding Matching Synchronization Calls 57

3.12 Example of Weiss's Serialization Method 61

3.13 Partial Paths Found by Sequential Program Path Coverage Algorithms 62

3.14 Yang and Chung's Infeasible Paths 64

4.1 Simple Example of PPFG 74

4.2 PPFG of the Producer/Consumer Problem 77

4.3 A More Complex PPFG Example 78

4.4 Testing Criteria Subsumption Hierarchy for Parallel Programs . . . 86

5.1 A du-path can potentially cause an infinite wait. 92

5.2 Du-pair is incorrectly covered. 93

5.3 Path Classification Algorithm 98

5.4 Three Threads Reduced from Three Clauses 103

5.5 Vertical Constraint 104

5.6 Horizontal Constraint 105

5.7 A Parallel Program Reduced from the Boolean Expression B 106

5.8 Final PPFG (some threads and synchronizations shown partially) . 108

5.9 Reduction of an Arbitrary Graph to an Acyclic Graph 114

5.10 Gabow et al's Algorithm for Existence of a Multiple Node Constrained Path 115

5.11 An Example of Dominator Trees and Implied Trees 117

5.12 The CFG G of a Sequential Program P 119

5.13 A ddgraph, dominator, postdominator trees of the program P . . . 120

5.14 A Sub-ddgraph (G, e_6, e_{13}) 121

5.15 Illustrating C-paths and C-routes in concurrent programs 124

5.16 Example of the path finding algorithm 127

5.17 Phase 1: Annotate the PPFG. 130

5.18 Phase 2: Generate the du-path coverage 131

6.1 Temporal Testing Criteria 142

6.2 Temporal Testing Hierarchy 144

6.3 Observation of Delay Correlations 146

6.4 Illustrating Delay Redundancy 148

6.5 Possible Memory Accesses 152

6.6 Illustrating Redundant Delays 153

6.7 Identifying Redundant Delays 154

6.8 Redundant Delays 157

6.9 A Binomial Heap for n = 4 161

6.10 Generating the Test Suite Representation 164

6.11 Eliminating Redundant Tests 165

7.1 SUIF Intermediate Representation 169

7.2 SUIF Layered Structure 173

7.3 Generating a Static Analyzer Using Nsharlit 174

7.4 nsharlit Framework **n_solve()** 176

7.5 **della pasta** Path Visualizer User Interface 178

7.6 Highlighted Du-Path Illustrated 179

7.7 Resizing PPFG with Highlighting Illustrated 180

8.1 A Producer/Consumer Program 190

8.2 Pattern of a Matching *post* And *wait* Causing $non-PATH_w$ 191

8.3 A Timing Chart of the Producer/Consumer Program 194

8.4 A Pipelining Program 196

8.5 A Timing Chart for the Pipelining Program 197

8.6 A TCP and UDP Name Caching Program 198

8.7 The PPFG of the TCP and UDP Name Caching Service Program . 199

8.8 (Cont.)The PPFG of the TCP and UDP Name Caching Service Program 200

8.9 A Timing Chart for the Name Caching Program 202

9.1 An Ada Program 211

9.2 An MPI Program 213

LIST OF TABLES

3.1 Results of a 1981 Study of Program Faults 37

3.2 Concurrent States in the Dining Philosopher Code 51

6.1 Reducing Test Cases Using $d_3 \rightarrow d_4$ 160

8.1 Number of *post/wait* Nodes 201

8.2 Results of the Path Finding Algorithm 203

8.3 Redundant Delay Point Results 204

ABSTRACT

As computational scientists continue to demand higher performance, the use of parallelism is becoming more pervasive. Workstations with multiple CPU's are no longer unusual. With higher performance networks, clusters of workstations or PCs networked together have become a cost-effective parallel machine for high performance computing. A variety of parallel programming languages and sophisticated compiler technology has been developed for gaining efficient use of the available parallelism. Parallelism has become the way of life for many scientific programmers.

Unfortunately, writing correct parallel programs remains a challenging task for several reasons. Programmers often have to be concerned with ensuring that communication and synchronization between processes are correctly achieved in addition to programming computational components. For efficiency, most parallel programming paradigms allow for nondeterministic execution, in which processes do not synchronize after every statement. Instead processes execute at different speeds such that different sequences of statements are executed during different runs of the program with the same input data. Parallel programmers find software tools for performance analysis and debugging very valuable, but otherwise there is a lack of software tools for parallel programmers. In particular, little to no support for testing parallel programs exists now, and very little research has focused on this important stage in the software life cycle of parallel programs.

Many structural testing methodologies and algorithms for generating test data automatically have been developed and successfully used for testing sequential programs. Some progress has been made in the analysis, testing, and debugging of

Ada-like *concurrent* programs. Also, there has been considerable work in analyzing parallel programs for the detection of deadlock and race conditions, debugging, and optimizing analysis and transformation. The current testing methodologies target concurrent programs with rendezvous communication, hence, do not support the testing of programs using more general synchronization and communication features. Without further consideration, structural testing methodologies used for testing sequential programs cannot be applied to parallel programs. For example, if we apply sequential path finding techniques to find a path for covering a node in a parallel program, these algorithms can return incorrect paths.

The main objective of this research has been to demonstrate that program-based methodologies can be used to generate test cases automatically for shared memory parallel programs. That is, even when a formal specification is lacking, software testing of shared memory parallel programs can still be performed in a systematic fashion with the aid of software testing tools.

The contributions of this research include the following. The framework for testing sequential programs is extended to support parallel program testing, taking into account the nondeterministic nature of parallel programs. A novel algorithm is presented to generate a du-path coverage with respect to a define and use pair, i.e., du-pair, for shared memory parallel programs. Modifications to this algorithm in order to test parallel programs with message passing and rendezvous communication are described. With the goal of exposing synchronization errors, a temporal testing method is presented with the capability of automatically generating temporal test suites. Because the total number of possible temporal test cases is exponential, several approaches to reducing the size of a temporal test suite are presented. The methodologies developed in this dissertation have been implemented and incorporated into a software testing tool, called *della pasta*, which provides a practical and user-friendly interface for visualizing a du-pair and a path coverage in a shared

memory parallel program.

Chapter 1

INTRODUCTION

As computational scientists continue to demand higher performance, the use of parallelism is becoming more pervasive. Workstations with multiple CPU's are no longer unusual. With higher performance networks, clusters of workstations or PCs networked together have become a cost-effective parallel machine for high performance computing. A variety of parallel programming languages and sophisticated compiler technology has been developed for gaining efficient use of the available parallelism. Parallelism has become the way of life for many scientific programmers.

Unfortunately, writing correct parallel programs remains a challenging task. Programmers quickly find that writing correct parallel programs and verifying the results, disregarding efficiency goals, are much more difficult than writing sequential programs for the following three main reasons. First, within a single parallel program, multiple tasks may be executed in parallel. It is more difficult for a programmer to keep track of the exact execution paths of a program with multiple tasks. Hence, verifying the results becomes more difficult. Second, data communication and synchronization among threads are frequently needed. A programmer must consider not only the logic within one thread, but also the data communication and synchronization among threads. Finally, when synchronization errors or race conditions exist, a situation referred to as nondeterministic execution can cause the results of one execution run to be different from another run even using the same input data. Due to the property of nondeterministic execution, a "bug" in a parallel program may not be detected even when the program is executed multiple times.

Software testing plays an important role in detecting various program faults, and is considered an essential stage within the life cycle of computer software, as it directly influences the quality of a program and hence, the cost of software development. Moreover, inadequate testing may cause fatal crashes and result in serious outcomes. For example, software used in military defense or the space program cannot tolerate the outcome of system failure due to inadequate testing. A major obstacle to users in ensuring the correctness and reliability of parallel software is the current lack of software testing tools for this programming paradigm.

Researchers have proposed methodologies and algorithms for generating test cases automatically for testing sequential programs [18, 111, 43, 101, 149, 64, 7, 143, 102, 66, 105, 115, 158, 134, 138, 47, 133, 55, 125, 135, 109, 45, 71, 114, 34, 9, 72, 139, 106, 113] as well as distributed systems [120, 121, 17, 136]. Also some researchers have studied the problem in the context of concurrent program testing [15, 137, 89, 157, 124, 89, 28, 128, 131, 21, 30, 76]. Other work has focused on the detection of race conditions or debugging [31, 28, 6, 24, 94, 95, 126]. The ultimate goal of most of these efforts has been to be able to reproduce a test run. Many tools which incorporate these methodologies have been developed. Automatic and semi-automatic testing tools have aided sequential and concurrent software developers during and after the stage of software development. Various software testing methodologies and tools have provided programmers with systematic ways to perform system testing. However, the intrinsic nature of parallel programs significantly complicates the task of providing *program-based testing tools* to developers of *parallel programs.*

Nondeterminism makes it difficult to reproduce a test, or replay an execution for debugging. It also implies that a given test data set may not actually force the intended path to be covered during a particular testing run. The concept of generating test data to cover all statements or all branches is not well-defined in the context of parallel programs. When executing a parallel program more than once using a test suite, it could happen that all statements are executed in one execution run, but not all statements are not executed in another run.

The premise of this dissertation is that, *with some extension, test data adequacy criteria for sequential programs are still applicable to testing parallel programs under various models of communication.* The structure of a parallel program is analyzed with the goal of automatically or semi-automatically generating test data. The framework for testing sequential programs is extended to support parallel programs taking the issue of nondeterministic execution into consideration. A novel algorithm is presented to generate a du-path coverage with respect to a define-use pair, i.e., du-pair, for shared memory parallel programs. With the goal of exposing synchronization errors, a temporal testing method is presented with the capability of automatically generating temporal test data. Because the total number of possible temporal test cases is exponential, several approaches to reduce the size of a temporal test data suite are presented. The methodologies developed in this dissertation have been implemented and incorporated into a software testing tool, called *della pasta*, which provides a practical and user-friendly interface for visualizing a du-pair and a path coverage in a shared memory parallel program.

Testing parallel programs can be performed by a specification-based functional approach for which the parallel software is treated as a black box or a program-based structural approach for which the control and data flow of the program are analyzed in order to generate test cases. Program-based testing offers the promise that a particular level of testing coverage has been achieved by the generated test cases. Program-based analysis techniques also enable one to identify and retest only those parts of the program that are affected by a code modification, enabling selective regression testing. Hence, this dissertation focuses on program-based structural software testing techniques.

The remainder of this dissertation is organized as follows:

- Chapter 2 contains background information. The general nomenclature used in current software testing literature is introduced. Parallel programming is discussed with a focus on programming languages, programming paradigms, and execution models. The parallel programming model targeted throughout this research is described.

- Chapter 3 outlines the specific challenges for testing shared memory parallel programs. Related research in the area of concurrent program testing is described in detail.

- Chapter 4 defines the program representation used in this dissertation, and presents the testing framework that lies at the base of the testing algorithms presented in this dissertation.

- Chapter 5 examines the challenges of finding a du-path coverage for parallel programs. Two types of path coverages are characterized, and computational complexity issues are addressed. An algorithm for finding a du-path coverage is presented and analyzed.

- Chapter 6 focuses on temporal testing for detecting synchronization errors. A method is presented to automatically change the execution time of synchronization or communication events along paths identified for program-based testing. The majority of this chapter presents a method to automatically identify redundant test cases in order to reduce the size of a temporal test suite.

- Chapter 7 describes the design and implementation of the **PA**rallel **S**oftware **T**esting **A**id(*della pasta*) which allows a user to view a parallel program in a graphic or text mode, and incorporates the algorithms presented in this dissertation.

- Chapter 8 describes the experimental studies conducted for exploring the effectiveness of the algorithms presented in this dissertation.
- Chapter 9 summarizes major contributions and future work including the necessary modifications for applying the du-path coverage and temporal testing techniques to message passing systems as well as concurrent programs with rendezvous communications.

Chapter 2

BACKGROUND

This chapter provides the necessary background on software testing and parallel program models. Section 2.1 describes the fundamental problems of software testing as well as the general taxonomy of software testing methodologies. In particular, terminology used in the context of software testing is introduced. Section 2.2 provides the background on parallel systems. The distinctions between methodologies for testing distributed and parallel systems are discussed. Several forms of parallelism are illustrated. Various memory consistency models for shared memory programming are presented. The software testing problems and parallel programming paradigms targeted by this research are described in Section 2.3.

2.1 Software Testing

Software testing has always been considered to be an uncertain and grueling task throughout the life cycle of computer software. Usually, the success of software testing is not easily determined, yet failures are easily noticed. Marc Roper mentioned in his book *Software Testing* [116] that in addition to some practical reasons, (e.g., the lack of a formal specification or poor documentation), software testing is challenging, mainly due to the following two fundamental problems:

1. Exhaustive testing is intractable due to a potentially infinite number of test cases. This is easily illustrated using an example. In Figure 2.1, the variables a and c are two input variables. Without the knowledge of the number of

```
begin
  input a;
  input c;
  for i = 1 to a
     c = c - 5;
     b = 100 / c;
  endfor;
  c = b * 5;
end;
```

Figure 2.1: Software Testing Problem Illustrated

iterations of the loop, the number of possible input data sets for testing this program exhaustively is infinite!

2. Weyuker [140] has shown that there is no algorithm to determine whether or not a given statement in a program may be exercised or whether or not every statement may be exercised. Without knowing which statements may be exercised before a program is executed, test case generation becomes a challenging task.

These problems confronted in exhaustive software testing have led researchers to develop methodologies for achieving less ambitious goals, and others to develop formal methods to support software development as well as software testing.

2.1.1 Testing Methodologies

There are many taxonomic ways to classify software testing methodologies. If a testing method considers the software being tested as a "black box," such that only the input and output are of interest in generating test cases, the software testing approach is called *black-box testing.* In contrast, if a testing method requires analysis of the control and/or data constructs of the program, the testing method is called *white-box testing.* For example, a functional testing approach is a black-box testing, whereas a structural testing approach is a white-box testing. If a program testing approach only takes a program as input, it is considered *program-based*, whereas if a *formal specification* is available for generating test cases, the software testing is called *specification-based*. Early testing literature also classifies testing methodologies as either *static* or *dynamic* approaches. If an approach does not require a program to be executed, it is called a *static approach*. In contrast, *dynamic approaches* actually execute a program with test cases. Many testing methodologies today combine static analysis with dynamic testing. That is, the program is analyzed statically and information obtained from the static analysis is used to aid in the selection of specific program statements(later referred to as *paths*) and input data. Sometimes static analysis only selects test cases, while other times static techniques are used to automatically generate test cases.

2.1.1.1 Overview of Static Analysis Methodologies

The goal of static analysis is to expose as many program errors as possible without executing a program. Software inspection, symbolic execution, and program verification all fall into this category. *Software inspection* [32, 3] actually involves "walking through" a program manually by a software inspection team. Specific software engineering disciplinary rules are followed during the inspection. Discovered errors and the severity of the errors are recorded. Static error analysis for errors such as data flow anomalies, (e.g., referencing undefined variables or dead variable definitions,) can also be automatically performed by a compiler or software tool built from technology similar to a compiler [100, 132].

Symbolic execution [73] takes both a program and the input variables of the program as input, and generates symbolic representations of output variables in terms of the program's input variables. The symbolic representation is specified in a human-readable manner that facilitates error detection as well as assertion generation and automatic program documentation.

Program verification [61, 85], also called program proving or program correctness, applies formal mathematical proofs to demonstrate that a program terminates and satisfies the program's specifications. Assertions about the program's variables are made at various points in the code, and then theorem proving techniques are employed to verify the correctness of these assertions.

2.1.1.2 Overview of Dynamic Testing Methodologies

Dynamic testing requires the target program to be executed. Some typical examples of dynamic testing techniques are specification-based, fault-based, and program-based testing [77, 98, 102, 97, 91].

For *specification-based testing*, test data is generated to test the program with respect to the requirements. One of the most well-known examples of a testing technique in this category is the methodology employed for testing the conformance of communication protocols [1] for computer or telephone networks to the specifications. *Conformance testing* [11] requires that a finite state machine(FSM) be derived from the specification of a protocol. Using the FSM, one can generate some unique input-output(UIO) sequences for identifying each state and transition in the FSM. These UIO sequences are used to test the program that implements the protocol.

Other examples of specification-based testing are domain testing and cause-effect analysis. *Domain testing* analyzes the domain of the input data based on the specification and generates various test data sets with data selected from each subdomain, or partition, of the original domain. Partition analysis, equivalence partitioning, and boundary value analysis all belong to this category [72, 158]. Domain testing is motivated by the observation that many program errors occur in conditional statements. The predicate used in an *if statement* is often incorrectly coded such that the incorrect branch is taken. Such errors can change the domain of the

[1] A communication protocol is a set of rules which governs the exchange of messages between a sender and a receiver [54].

input data. Various specification-based techniques have been developed to partition the input domain into valid and invalid regions, and test data selected from both regions are used for testing the program.

Based on a study of the cause and effect of specific program errors, *cause-effect analysis* provides researchers with clues for designing techniques specifically for detecting these errors [91, 97]. The specification is analyzed and the relationships between causes(input data, or any user actions) and effects(trace and execution results, errors, changes in the system behavior) are identified. The relationships that link the causes and the effects are then expressed by a boolean graph. Test cases are produced with respect to each possible combination of cause and effect. The strength of this technique is that it provides an opportunity to investigate possible combinations of input cases. However, the complexity of the cause and effect for various program errors makes it a difficult task to create the boolean graph. The quality of the cause-effect analysis depends on an accurate specification which is often not available.

The second type of dynamic testing, *fault-based testing*, is based on the analysis of program faults to develop various techniques for detecting these program errors. *Mutation testing* is a well-known example of the fault-based approach [98, 134, 138, 149]. Originally, the goal of mutation testing was to manually identify the test data that can detect a specific program fault. However, constraint-based techniques have been developed to automate the test data generation process [26]. In mutation testing, a specific program fault is deliberately planted into a program, and the program is executed with input generated by other techniques. If the expected program error is exposed during the execution, the input data is considered to be adequate with respect to this specific program fault. The program with the embedded fault is called a *mutant*. If the test data is adequate, the mutant is said to be *killed* by the test data. If one of the mutants is not killed by the test data, the test suite is not adequate with respect to the specific program fault.

Most dynamic testing techniques fall into the third category called *program-based dynamic testing.* The various program-based, structural testing methodologies provide many different ways of generating test data. In general, these techniques are called *path coverage* methodologies. Since a major part of this dissertation focuses on path coverage for testing shared memory parallel programs, path coverage methodologies are discussed in detail in the next several sections.

2.1.2 Testing Criteria for Path Coverage Methodologies

Path coverage testing methodologies are among the most widely used program-based testing methodologies. A control flow graph is usually generated to represent a program prior to test data generation [9, 72, 139, 106, 113]. A control flow graph representation of a program is a directed graph consisting of nodes and edges where each statement(or basic block[2]) is represented by a node and each possible transfer of control from one statement to another is represented by an edge.

In the first step of test data generation, specific sequences of statements, generally referred to as *paths*, are usually produced based on a specific criteria. A *path* is an alternating sequence of nodes and edges starting from the *begin* (or *entry*) node in a control flow graph to the *end* (or *exit*) node. Symbolic execution [20, 77] is then performed to generate a set of test data associated with these paths. Finally, the program is executed with the test data that was generated by the static analysis. As the set of test data is used to execute the program, the computed paths will be “passed through”, or *covered by this set of test data.* The statements in these paths are said to be *covered by the paths.* Examples of path coverage techniques include

[2] A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

```
input a;
c = 0;
b = 3;
if a < b then
   c = 5;
c = c + 1;
```

COVERAGE	TEST SUITE
all-statement	a = {1}
all-branch	a = {1,4}

Figure 2.2: All-statement and All-branch Coverage Example

executing every *statement* or every *branch* in a program at least once. These methods are called *all-statement coverage* and *all-branch coverage*, respectively.

In general, dynamic testing methodologies generate test cases based on some set of rules, known as *criteria* [56]. For example, if we generate a set of test data to execute all statements within a program at least once, this set of test data is generated based on the *all-statement test criterion.* Moreover, these criteria are also considered criteria for evaluating the *adequacy of test data.* For example, the set of test data generated by using the all-statement test criterion is said to be *adequate* with respect to the all-statement test criterion. A *test data adequacy criterion* is formally defined as follows:

A set of test data T is considered C adequate for a given program P, if there exist some test cases in T, such that the test data adequacy criterion C is satisfied for each component of P. If T is C adequate for P, it is said that the value of C(P,T) is true[141].

For example, test set T is *all-statement adequate* for program P, provided that for every statement s in P, there exists at least one test case TC in T such that the statement s is executed when this program is executed with TC. Since the test set T is a finite set, if all test cases in this set are executed at least once, the specific criterion is satisfied by the test set T. Therefore, a *test data adequacy criterion* may also be used to determine how much testing is needed and when the testing process can be stopped.

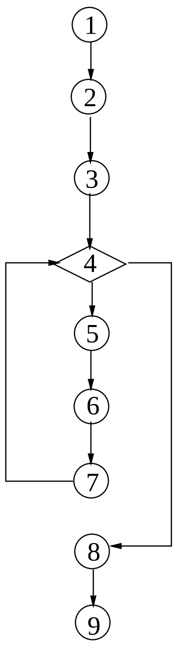
In Fig. 2.2, two test data suites, which are all-statement and all-branch adequate respectively, are illustrated. When the input value of a equals 1, the conditional expression used in the *if* statement, i.e., $a < b$, evaluates to TRUE, and statement 5 will be executed. As a result, all statements will be executed once. Therefore, the test suite $\{1\}$ is all-statement adequate. Similarly, when the input value of a equals 4, the statement 5 will be skipped due to the fact that $a \geq b$. Hence, the test suite $\{1, 4\}$ is all-branch adequate.

In addition to all-statement and all-branch coverages, other major test data adequacy criteria are:

- *Data flow testing* - A number of data flow testing methodologies have been developed in which the data flow relationships in a program are used to select the set of test paths. One of the major data flow testing criteria is *all-define-use-path*, or *all-du-path*, coverage. A variable is *defined* in a statement when it receives a value, most commonly by an assignment or input statement. A variable is *used* in a statement when the variable is referenced in an expression of a computation, conditional, or output statement. A *define-use pair*, also known as a *define-use association(dua)*, is formally represented by a 3-tuple *(V, D, U)*, where *D* and *U* indicate the statements where the variable *V* is defined and used, respectively, such that there is at least one path in the control flow graph from *D* to *U* with no redefinition of variable *V*. This method generates all paths with respect to a define-use pair in the program. If there are multiple du-paths from D to U, these paths must all be generated. The

```
1.  begin
2.     input a;
3.     c = 10;
4.     for i = 1 to a
5.        c = c - 5;
6.        b = 100 / c;
7.     endfor;
8.     c = b * 5;
9.  end;
```

(a)



(b)

all-du-path

def	use	var
2	4	a
3	5	c
5	5	c
5	6	c
6	8	b

all-du-path test suite

a = {3}

(c)

LCSAJ

No.	begin	end	target
(1)	1	4	8
(2)	1	7	4
(3)	4	7	4
(4)	4	4	8
(5)	8	9	exit

LCSAJ test suite

a = {-1. 2}

(d)

Figure 2.3: All-du-path and LCSAJ Example

key motivation behind the all-du-path coverage is to generate paths that follow the pattern of data flow within a program.

Figure 2.3(b) shows the control flow graph of the program shown in Figure 2.3(a). Figure 2.3(c) shows define-use pairs and a test suite that is all-du-path adequate. All of the paths for the define-use pairs should be covered when the value 3 is used as the input if the program were free of errors. However, the example program has two program faults. First, the statement 6 may cause an error when the value of c equals zero. Second, the variable b in statement 8 is not defined when the body of the *for-loop* is not executed. Although some compilers can detect program errors such as the potentially undefined variable b in the example, some compilers assume that each loop is executed at least once when the static analysis is performed. Hence, this error may not be detected by the compiler if this code segment is embedded in a complicated program. The first error can be detected by testing via all-du-path coverage, whereas the second error can be detected by other structural testing techniques such as LCSAJ(see below). A family of data flow testing data adequacy criteria exists with different resulting sizes and types of test suites [112].

- *Linear Code Sequence and Jump(LCSAJ)* - An LCSAJ is a 3-tuple (s_1, s_2, s_3), where s_1 is the first statement of a program or the target statement of a control flow jump, s_2 is the ending statement, (i.e., any statement which can be reached from the start statement by a consecutive sequence of code and from which a jump can be made,) and s_3 is the target statement to which execution will jump after the end statement s_2 is executed. This criterion requires that all LCSAJs be executed. Figure 2.3(d) shows the LCSAJ's and a test suite that is LCSAJ adequate. The test data -1 for the value of variable a covers the first LCSAJ $(1, 4, 8)$ and the test data 2 covers the rest of the

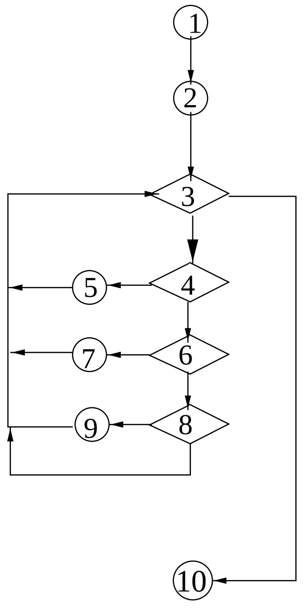


Figure 2.4: All-path Coverage Problem Illustrated

LCSAJs. When the program is executed, the LCSAJ $(1, 7, 4)$ enters the loop and makes one iteration, the LCSAJ $(4, 7, 4)$ makes the second iteration, the LCSAJ $(4, 4, 8)$ tests the loop condition and exits the loop, and the LCSAJ $(8, 9, exit)$ executes the last line of the program. LCSAJ testing tends to exercise loops more thoroughly than branch testing, and LCSAJ also exercises the case where the loop is not executed [150].

- *All-path coverage* - The all-path coverage technique generates test data to cover all possible paths of execution through a program. This is definitely the most complete coverage. However, the total number of paths quickly becomes very large for a program that contains loops because each iteration of a loop is considered to be a different path. For example, Figure 2.4 shows a program with a loop, and three *if*-statements within the loop. It is obvious that all of the following paths must be tested in order to claim that this program segment is exhaustively tested: 1-2-3-10, 1-2-3-4-5-3-10, 1-2-3-4-6-7-3-10, 1-2-3-4-6-8-9-3-10, 1-2-3-4-5-3-4-5-3-10, 1-2-3-4-5-3-4-6-7-3-10, 1-2-3-4-5-3-4-6-8-9-3-10, etc. Since each time the iteration can use any of the branches of an *if* statement within the *loop*, one can easily conclude that the total number of paths in Figure 2.4 can be represented by the formula

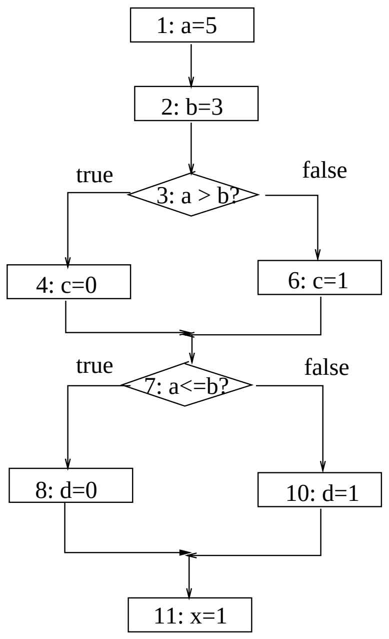
$$\sum_{i=0}^{n} m^i$$

where n is the iteration bound of the loop and m is the number of *if* statements or decisions within the loop. Using this formula, one can easily verify that if n and m in the formula are 9 and 3, respectively, this results in $29,524$ paths. It is obvious that the all-path coverage is not a practical solution since it requires all paths to be tested. It is usually included as part of the structural testing data adequacy criteria just for the completeness of test data adequacy criteria.

If a path cannot be covered due to the existence of some conflicting conditions to execute some of the statements in the path, the path is called an *infeasible* or *unrealizable path*. In Figure 2.5, the final values for c and d will never be $c = 0$, $d = 0$ or $c = 1$, $d = 1$. However, when we find test cases to cover all-statements using this control flow graph, we may generate infeasible paths such as 1-2-3-4-7-8-11 and 1-2-3-6-7-10-11, which would never occur when the program is executed. Obviously, when infeasible paths exist in a program, path coverage techniques become inaccurate. More discussion on this topic is provided later in Chapter 3. It is undecidable to statically determine whether or not the input data for exercising a given path exists [140]. That is, one cannot guarantee to be able to write a program which can statically generate the input data for exercising a given path. Nevertheless, path generation strategies aimed at reducing the effects of infeasible paths have been developed using symbolic execution and program proving [62, 156, 92] such that fewer infeasible paths will be generated. Hence, it is generally acceptable to assume

```
1.   a = 5;
2.   b = 3;
3.   if ( a > b ) then
4.     c = 0;
5.   else
6.     c = 1;
7.   if ( a <= b ) then
8.     d = 0;
9.   else
10.    d = 1;
11.  x = 1;
```

(a)



(b)

Figure 2.5: A Program with Infeasible Paths

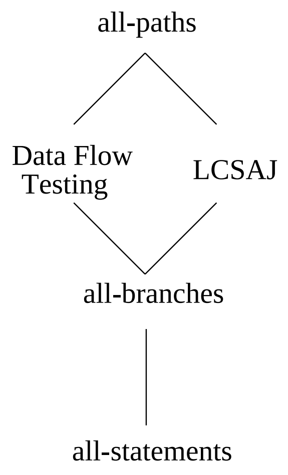


Figure 2.6: Partial Ordering of Coverage Criteria

that infeasible paths do not exist when path coverage algorithms are applied, or that infeasible paths can be dynamically identified at execution time.

2.1.3 Subsumption Hierarchy for Path Coverage Criteria

Although test data adequacy criteria are used individually, they are not independent of each other. A set of test data that is adequate with respect to one criterion sometimes implies that it is adequate for another criterion [145]. This relation is called *subsumption.* The term *subsume* is formally defined as follows:

Given two criteria A and B and a program P, we say that A subsumes B if when a set of test data T is A adequate, then it is also B adequate. In other words, if A subsumes B, then when A(P,T) is true, B(P,T) is also true [137].

As an example, the *all-branch* criterion subsumes the *all-statement* criterion, but not vice versa; if all possible branches are covered, each statement must be covered, but not vice versa. This was illustrated earlier in Figure 2.2(a), where the test set for all-branch coverage was also adequate for all-statement coverage, but the converse was not true.

The subsumption relation between two criteria can be represented as a partial ordering of these two criteria. Figure 2.6 shows the subsumption hierarchy for path coverage testing criteria. The all-path testing criterion subsumes both data flow testing and LCSAJ criteria. The data flow testing and LCSAJ criteria both subsume the all-branch testing and all-statement testing criteria. The all-branch testing criterion in turn subsumes the all-statement testing criterion. This figure reflects the fact that the data flow criteria do not subsume LCSAJ and neither does the LCSAJ criterion subsume the data flow criteria. These criteria are called *incomparable* [112]. For example, in Fig. 2.3, the test suite for all-du-path coverage is not a subset of the test suite for LCSAJ, and the test suite for LCSAJ coverage is not a subset of the test suite for all-du-path coverage.

2.1.4 Research Directions in Path Testing

This section describes the research directions in path testing including *path finding methodologies, test data generation, execution strategy, error analysis*, and *testing theory.*

Path finding methodologies constitute the first step in path coverage testing. Given a program to be tested, the ultimate goal is to find a set of paths that provides a coverage that meets a specific path coverage criterion. Since the test data generation step is usually computationally expensive, many researchers have focused on finding a minimum path coverage or selecting an optimal set of paths for the different coverage criteria [96, 20, 99, 9, 155, 140].

Test data generation is crucial for achieving the ultimate goal of automating a testing process. Although known for many years, this issue has not been satisfactorily resolved. Usually a set of paths is provided when the process of test data generation is conducted. The conditional statements included in a path constitute constraints with regard to the input space, i.e., domain of input data. Symbolic execution can be used to determine the constraints on the input domain if the constraints are represented mathematically in linear format. As an example, the constraint $i1/i2 \leq 7$ or its equivalent constraint $i1 - i2 * 7 \leq 0$ is a linear inequality due to the fact that the exponents of all the variables in the inequality equal one, whereas $i^2 \leq 8$ is a non-linear inequality since the exponent of the only variable i in the inequality is greater than one.

For constraints related to conditions that cannot be represented mathematically in linear format, symbolic execution cannot be used [20]. Nevertheless, dynamic test data generation performed at execution time has been proposed recently to generate test data for arrays and pointers [78].

Because sequential programs are deterministic, each test run with the same data will produce the same results. Thus a single test run for a given test data input is adequate. However, testing of parallel programs needs to deal with the nondeterministic nature of parallel programs. Nondeterminism implies that a given test data set may not actually force the intended path to be executed. It also makes it difficult to reproduce a test run. Considerable work has been done in the area of *reproducible testing* in the presence of nondeterminism [124, 128, 76, 126, 68]. These techniques define the execution strategy for testing.

From the early stages of testing research, researchers have focused on *error analysis*, with the goal of using the results to judge the effectiveness of various testing criteria [12]. Error types are analyzed and the fault detection capability of each path coverage testing criterion is evaluated with respect to the errors that it can uncover. Unfortunately, this work has not met with great success. Part of the reason for this failure is the difficulty in collecting actual program faults. It may be more than a fear of lost business that discourages software developers from releasing the actual data. When a project is near completion and the schedule is slipping, it is always the testing stage that suffers. Thus, unless a project is extremely well managed, it is often difficult to manually obtain reliable data [116]. Another reason for this failure is the fact that all experimental studies have different characteristics. As a result, one type of error analysis may not be useful for another type of software [90].

In an attempt to provide a common basis for developing and judging testing methodologies, some researchers have focused on developing *testing theory* similar to other scientific disciplines. Their ultimate goal is that perhaps some day testing theory could provide a means for evaluating the effectiveness of fault detection of various test methodologies [144]. However, this goal has not yet been achieved, even in the context of sequential programs [116]. Various efforts have exposed fundamental issues and led to the introduction of formal concepts regarding test methodologies. The concept of test data adequacy is a good example. A theory that assists in the development, assessment, and even actual execution of testing methods will be an invaluable asset to the testing community.

2.2 Parallel Computing

Traditionally, computing has been performed on a von Neumann computer which consists of a single processor connected to a linear organization of fixed size memory. As computer systems evolved over the years, concurrent systems were developed and various task scheduling schemes were employed to achieve concurrency through running multiple tasks on a single processor. However, the parallelism is actually achieved by the interleaving of execution supported by the operating system running on a single processor. Later, multiple stand-alone machines were connected by using high speed communication links for exchanging messages between machines. Thus, distributed systems emerged. As an example of a distributed system, a *client-server model* is usually supported by a set of processors among which one processor acts as a server with a dedicated function while clients send requests to the server asking for the service function to be performed on behalf of the clients. Many new applications have been developed based on this client-server model. However, the architectural concept of a computer is still based on the architecture of a von Neumann machine.

As higher performance was demanded by computationally intensive scientific applications while both the speed of the communication links and processors were anticipated to be approaching their physical limits, parallel computers emerged. Parallel computers range from shared memory multiprocessor workstations, to networks of workstations, to large distributed memory supercomputer machines in the thousands of processors, to distributed shared memory multiprocessors. With the use of networks of workstations as a cost-effective parallel computer, the traditional concept of parallel and distributed systems is blurring.

2.2.1 Parallel Programming Paradigms

To utilize the potential performance of parallel computing systems, users can choose between several approaches. A sequential program can be augmented with compiler directives that suggest parallelism and data distribution to the compiler which automatically creates the parallel program. A parallel program can be created using a sequential language for computation and calls to a run-time library for process creation, communication, and coordination between processes. A parallel program can be written using a parallel programming language that has parallel programming constructs for process creation, communication, and coordination.

As parallel programming has evolved, many different programming paradigms have been developed. A *paradigm* is a model of the world that is used to formulate a computer solution to some problem. Parallel programming paradigms can be classified in several ways: explicit vs. implicit parallelism, data vs. functional, and shared memory vs. message passing. These programming paradigms need not match the model of parallelism offered by the underlying parallel architecture. For example, a distributed shared memory system presents the shared memory model

```
1.  x = 10
2.  y = 3
    Parallel Sections
    Section A
3.      u = x + y
4.      v = u * 2
5.      s = u
    Section B
6.      u = x - y
7.      v = u + 1
8.      t = v
    End Parallel Sections
9.   Print 's= ', s
10. Print 't=', t
```

Figure 2.7: Explicit Parallelism with Parallel Sections

to the programmers, but the underlying memory system is distributed among the processors.

2.2.1.1 Explicit vs. Implicit Parallelism

A program written with parallel language constructs, such as *parallel loops* or *parallel sections*, expresses parallelism explicitly determined by the user. When

```
 do  i = 1 , m
   do j = 1, n
     do k = 1, p
        A(i, j) = B(i, j) + B(i, j, k) * C(i, j)
     enddo
  enddo
enddo
```

Figure 2.8: Implicit Parallelism

```
!HPF$ INDEPENDENT
        do i = 1, m
!HPF$        INDEPENDENT
              do j = 1, n
                  do k = 1, p  !inner loop not independent
                        A(i,j) = A(i,j) + B(i,j,k) * C(i,j)
                  enddo
              enddo

        enddo
```

Figure 2.9: Code Segment Using Compiler Directives

a user writes sequential code, and relies on an automatic parallelizing compiler to detect the parallelism based on data and control dependences, the approach is said to exploit *implicit parallelism.* The program in Figure 2.7 contains explicit parallelism since the semantics of the parallel sections is to state explicitly that section A and section B can be executed in parallel. In contrast, the code segment in Figure 2.8 is implicitly parallel. The compiler can detect that the do-loop iterations are independent(meaning that one iteration does not write a variable which is read or written by another iteration) and, hence, can be performed in parallel.

In general, the detection of parallelism requires sophisticated dependence analysis [148, 36]. For difficult cases in which the compiler cannot conservatively determine independence, (and therefore, parallelism) a user can provide additional information to the compiler through compiler directives. For example, a user can inform a compiler that a do-loop can be executed independently–that is, in any order or concurrently. Effectively, using compiler directives like this changes a do-loop from an implicitly parallel construct to an explicitly parallel construct. For example, in Figure 2.9, "$HPF INDEPENDENT" is inserted to indicate that the outer two loops are independent. The inner loop uses and assigns the same elements of A repeatedly and, hence, is not independent. Other compiler directives are often used to indicate suggested data distributions if in fact the loop can be parallelized.

2.2.1.2 Data Parallelism vs. Control Parallelism

Data parallelism and *control parallelism* are the two most widely used parallel programming paradigms. Data parallelism is achieved when the input data is partitioned into smaller segments, each segment of the data is distributed to a different processor, and each processor executes the same code with its share of the data segment. In contrast, control parallelism is exploited when processors execute different tasks in parallel on the same data, multiple copies of the same data, or even different data. Figure 2.9 demonstrates data parallelism, while Figure 2.7 illustrates control parallelism.

Flynn presented a parallel hardware taxonomy [35] to categorize parallel systems as either SISD, SIMD, or MIMD systems. Parallel software can also be classified in these terms. A SISD (Single-Instruction-Single-Data) program is simply a sequential program. In a SIMD (Single-Instruction-Multiple-Data) program, multiple processes execute the same single instruction with different data elements, synchronize after the parallel instruction execution, and then continue to the next parallel instruction execution. A program has a single thread of control with synchronous parallel instructions. C^* and early implementations of FORTRAN 90 were SIMD. MIMD (Multiple-instruction-multiple-data) is the most general model of parallelism. Multiple processes execute asynchronously on different instruction streams and possibly different data. The SIMD paradigm models data parallelism, whereas the MIMD paradigm can model both control and data parallelism.

A simpler style of MIMD program is often implemented as an SPMD (Single-Program-Multiple-Data) program. In an SPMD program, each process is given a unique identifier (i.e., process id), and special branch statements based on the id are used to identify the code segments intended for each process [104]. A programmer writes a single program which is loaded into each processor's memory. All processors

```
if (my_process_id == 0)
{
   /* manager task */
}
else
{
  /* worker task */
}
```

Figure 2.10: A Program Segment in an SPMD Program

execute the same program and synchronize only where synchronization and communication statements are located. A skeleton of a code segment in an SPMD program is shown in Figure 2.10.

When these models are compared from the viewpoint of static analysis, SPMD programs are more challenging to analyze than other MIMD programs. Since there is only one copy of the source code, we could naively assume that every process could execute the whole program. However, the branch statements used to identify subtasks for specific processes indicate that each process actually executes different parts of the program. Without additional information from a programmer, the static analysis may have a difficult job determining which branch statements have this special purpose.

2.2.1.3 Shared Memory vs. Message Passing Systems

When a parallel system supports a global shared address space among the processes, the parallel system is called a *shared memory system.* A shared memory model can be reflected in the architecture of the underlying machine or offered to the programmer through software, while the underlying system is physically distributed memory, i.e., a distributed shared memory. Explicitly parallel languages for the shared memory model provide synchronization mechanisms for controlling access to the shared memory from multiple processes.

The capability of creating multiple threads of execution has become widely available in many operating systems such as Solaris 2.x, OS2/2.x, Window NT, and SysVR4[110, 23]. Some programming languages also provide language features to support synchronization among multiple threads. For example, PCF FORTRAN[2] supports `post, wait`, and `advance` for synchronization, and `cobegin/coend` or `parallel sections` for creating multiple threads. Other examples are the Java language [33] and the POSIX thread libraries [69] which also support operations similar to `post`, `wait`, and `pthread_create`.

In these languages, a `wait` operation waits for a matching `post` to be executed, and a `post` operation signals the completion of the wait operation. Thus, synchronization among multiple threads can be achieved. Therefore, the `post` operation is an *asynchronous* operation; it is completed immediately independent of a matching `wait` operation. However, if a matching `post` is not completed, the `wait` operation halts. Therefore, the `wait` operation is called a *synchronous* operation. The synchronous and asynchronous operations are sometimes called *blocking* and *nonblocking* operations, respectively. Shared memory systems also provide other synchronization mechanisms such as access-locks, semaphores, and monitors for accessing shared variables exclusively [74].

Shared memory parallel programming is perhaps the easiest model to understand because of its similarity with operating systems programming and general multitasking. Shared memory systems are capable of supporting the most general form of MIMD computing. From a programmer's point of view, the need to copy variable values from one local memory owned by one processor to another local memory owned by another processor is eliminated.

The alternative to shared memory programming is for each processor to have its own local memory and to use explicit message passing statements for providing communication among processes. This type of parallel system is called a *message passing parallel system.* An explicitly parallel programming language that supports message passing provides a communication interface for sending and receiving messages in addition to process or thread creation. Each process has a unique identification name or number (process id), and processes interact by sending and receiving messages to and from named processes. For example, the MPI standard library (Message Passing Interface) [49] provides both blocking and nonblocking send and receive operations, i.e., *MPI_Send, MPI_Recv* or *MPI_Isend, MPI_Irecv*, respectively. MPI supports many other operations. *MPI_Barrier* causes all threads to synchronize before they proceed together, and *MPI_Bcast* sends one message to all processors participating in the computation.

Shared memory computers are relatively easy to program, but difficult to scale up to a large number of processors. Communication can be achieved via simple definitions and uses of shared variables. However, it is difficult to scale up to more processors without imposing high communication costs. On the contrary, message passing systems are scalable for massive parallelism, but are sometimes compared to assembly language programming in terms of their low level nature. Like most shared memory systems, the message passing model also provides the capability of creating multiple tasks dynamically. However, in practice, most message passing systems create a fixed number of tasks at program startup time and do not allow tasks to be created or destroyed during program execution. Most message passing programs follow the SPMD programming model. Hardware and software distributed shared memory systems have the goal of producing the best of both worlds - scalability and ease of programming.

```
1.  x = 10
2.  y = 3
    Parallel Sections
    Section A
3.      u = x + y
4.      v = u * 2
5.      s = u
    Section B
6.      u = x - y
7.      v = u + 1
8.      t = v
    End Parallel Sections
9.   Print 's= ', s
10. Print 't=', t
```

(a) Parallel Program

statement order	results
(1) 3,4,5,6,7,8,9,10	s=13 t=8
(2) 3,6,4,7,5,8,9,10	s=7, t=8
(3) 3,4, 6,7,5,8,9,10	s=7, t=8
(4) 3,6,7,4,5,8,9,10	s=7, t=14
etc.	etc.

(b) Possible Computing Results

Figure 2.11: Statement Orderings Illustrated

2.2.2 Memory Consistency Models for Shared Memory Systems

In shared memory systems, the memory semantics depends on the memory consistency model supported by the underlying system. Since this dissertation concentrates on the shared memory model, memory consistency models are described briefly.

A shared variable may be defined in one thread, while that same variable is used in another thread of the same parallel section. In Figure 2.11(a), the variable u is defined (in statements 3 and 6) and used (in statements 4, 5, and 7) in both threads of the parallel section. The variable v is defined (in statements 4 and 7) and used (in statement 8) also in both threads of the parallel section. At the end of the parallel section, the value stored in these variables can vary depending on many factors including the hardware architecture, language, compiler, and execution sequence.

Several *memory consistency models* have been developed to deal with the problem of memory semantics. Under *sequential consistency* [80], all processors observe actions as if they were merged into a particular global sequential order, and all observations are consistent. This condition implies that when any shared variable READ or WRITE is involved and no synchronization instructions are used to maintain consistency among processors, the execution of these program segments on various processors must maintain a consistent view as if one parallel segment on one processor is executed completely before another processor can begin executing another segment. Hence, this condition precludes any instruction interleaving; otherwise, the execution sequence observed on one processor may become inconsistent with that on another processor.

In Figure 2.11(b), several possible sequences of execution are illustrated, but only the first result is accepted under the sequential memory consistency model. All other situations violate the sequential consistency rule since these sequences of statement execution require the interleaving of execution and could not be consistently observed from all processors. Thus, the sequential consistency model is the most well-defined model; however, it is also the most restrictive model in terms of the potential performance improvement at the machine instruction level. Every machine instruction must be executed in a strict order. Hence, parallelism is severely limited. The opportunity for compiler optimization is also very limited since all instructions are required to be executed in a specific order.

When implementing a memory consistency model with the necessary condition that the correctness of accessing shared variables must be guaranteed, computer architects must consider several key factors: the efficiency of an object program, the difficulty of implementing a specific model, and the ability to maintain a consistent instruction execution order on all processors. The efficiency of an object program is improved with an increased amount of instruction level optimization, but these optimizations require flexibility in the instruction execution order.

For example, under the *weak consistency model* [27], only the synchronization instructions are required to be sequentially consistent. Ordinary READs and WRITEs that execute after a synchronization instruction in a serial program must await the completion of the synchronization instruction. Ordinary READs and WRITEs that appear before a synchronization instruction must complete before the synchronization instruction can begin. Thus, a *critical section* may be declared by a user using synchronization instructions, such as *LOCK* and *UNLOCK*. The implementation for the LOCK/UNLOCK instructions must consider possible synchronization with other LOCK/UNLOCK instructions that are executed on another processor. The implementation may be expensive. However, with these mechanisms, the weak consistency rule allows instructions outside of a critical section to be interleaved as long as they do not violate the data dependences of non-shared variables and they are not moved into the critical section. Optimization is also allowed for instructions inside each critical section as long as they are not moved out of the critical section and data dependences of non-shared variables are not violated.

Obviously, this model provides the opportunity to achieve better parallelism than the sequential consistency model; however, this model is weaker in the sense that the correctness not only relies on the implementation of the architecture, but also on the correct usage of synchronization instructions which may either be inserted by a compiler or directly coded by a user. If the LOCK/UNLOCK mechanisms are not used properly, incorrect results may still result. As an example of incorrect coding, any result listed in Figure 2.11(b) could be produced under weak consistency since there are no synchronization instructions to synchronize the access to shared variables. This is an error on the user's part. A correct program should use LOCK/UNLOCK to create a critical section within parallel sections whenever a shared variable is accessed, and use shared variable READ or WRITE only within a critical section.

Copy-in/copy-out semantics advocated by Srinivasan and Wolfe [122] is similar to the value-result parameter passing method. The values of shared variables in a parallel section are defined to be initialized to the values that they have when the parallel section is entered; any updates are made(conceptually) to the local copies of the variable. When the parallel sections are complete, the global state is updated with any modification made within any section. The update operations for all threads must maintain a consistent view when synchronization operations are completed and at the end of all parallel sections. As long as the update operations are maintained consistently in all processors, they can be completed either synchronously or asynchronously.

The copy-in/copy-out model assumes that maintaining the data independences of shared variables in all threads is the user's responsibility. This model provides a well-defined program without volatile variables, and allows optimization within a parallel section to be independent of other parallel sections. This model also provides more opportunity for computer optimization than the weak consistency model due to the use of local copies. Instructions within the same parallel section can be executed out of order when there is no data dependence among them. However, this model is not without problems. There is some performance overhead involved in making local copies of variables and atomic merging of updated variables. Also, instructions within parallel sections are supposed to be synchronized explicitly. A user is required to ensure that shared variable accesses in all parallel sections, which may be executed in parallel, are independent of each other.

Many researchers have been focusing on developing techniques to relax memory consistency rules such that more opportunities for optimization are possible. These techniques all fall into two categories: *system-centric* or *programmer-centric* models [42]. A system-centric model relaxes the consistency rule by changing the implementation of a parallel machine. The programmer-centric models require programmers to provide information regarding which code segment must be sequentially consistent and which segment can be executed out of order, while the implementation of the parallel machine provides sequential consistency of the target program (which might have been reordered). Weak consistency is an example of a system-centric model, whereas the copy-in/copy-out model is an example of a programmer-centric model.

2.3 Testing and Programming Models Targeted by this Research

This dissertation focuses on the development of program-based, structural testing methodologies since most parallel programs have been developed without a machine-readable formal specification. Furthermore, a structural testing methodology can generate a set of paths to cover a given du-pair in a shared memory program, which is particularly useful in testing the patterns of accesses to shared variables.

The targeted programs are assumed to be written explicitly as parallel programs with thread creation and event synchronization to support the MIMD programming style. While sophisticated optimization and parallelizing compiler techniques can exploit parallelism on high performance systems to improve the performance of sequential programs, these tools are limited by the underlying sequential algorithm. A parallel algorithm explicitly programmed by a user is often needed to take utmost advantage of these parallel systems.

The programs are assumed to be executed in a shared memory environment. In particular, the parallel language model used in this dissertation provides `post/wait` for synchronization and `pthread_create` for creating multiple threads. This dissertation focuses primarily on shared memory programs due to the ease of programming and the increasing attention toward distributed shared memory systems. However, in Chapter 9, the extensions needed to apply the algorithms for testing shared memory systems to concurrent programs and message passing systems are also described.

The targeted programs are assumed to be MIMD style in which only a fixed number of subtasks are created. The MIMD model is able to simulate the SIMD and SPMD models and in practice, many applications only create a fixed number of subtasks.

For this research, the analysis used during testing assumes that the du-pairs for a given program are provided. There are known techniques [51, 75] that compute the define-use pairs within a shared memory parallel program for each basic block in the program.

Chapter 3

PARALLEL PROGRAM TESTING AND ANALYSIS: CHALLENGES AND STATE OF THE ART

The evolution of parallel programming systems has created new challenges in the area of software testing. Current testing methodologies require reconsideration and redesign to achieve the ultimate goal of generating test data automatically when parallel programs are taken into account. This chapter describes (1) errors that can occur particularly in parallel programs, (2) the challenges in debugging and testing implicitly parallel and explicitly parallel programs, and (3) the state-of-the art in debugging and testing parallel programs.

3.1 Potential Errors in Parallel Programs

As early as the late seventies, studies were performed to collect and classify information about program errors [44, 119, 103]. Table 3.1, taken from Glass [44], lists program faults that occurred in two large embedded software systems. Similarly, these errors can occur in parallel programs. In addition, synchronization errors including race conditions and deadlock situations can also occur in shared memory parallel programs.

3.1.1 Race Conditions

Figure 3.1 shows a simple example of a *data race*. The value printed at the end of thread 2 could be either 3 or 5 depending on the execution sequence

Category of Fault	Percentage (approximate)
Omitted logic	31.9
Failure to reset data	12.2
Regression fault	9.0
Documentation fault	8.5
Requirement inadequate	5.9
Patch at fault	5.9
Incorrect comments	5.9
IF statement too simple	5.9
Referenced wrong variable	5.3
Data alignment fault	3.7
Timing fault causes data loss	3.2
Failure to initialize data	2.7

Table 3.1: Results of a 1981 Study of Program Faults

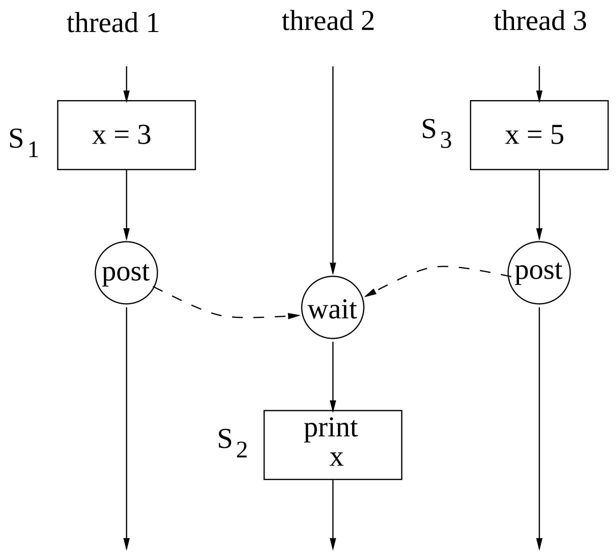


Figure 3.1: Data Race Example

```
DIMENSION A(2)
A = 0
paralleldo I = 1, 2
   A(I) = I * I
end paralleldo
```

Figure 3.2: An Internally Determinate Program

of statements S_1 and S_3 in threads 1 and 3, respectively. A race condition occurs when statements in different concurrent tasks access the same memory location, at least one of which is a *write* operation, and there is no guarantee on the order of executing these statements. Thus, a race condition can occur capriciously.

Emrath *et al.*[31] characterizes the types of various data races, provides some examples to explain these types, and discusses the techniques for detecting race conditions. Emrath argues that in most parallel programming models, deterministic execution is not guaranteed due to possible data races. Although nondeterminism may not necessarily be incorrect, in fact, some race conditions may even be intentional, most race conditions are program errors.

If a race occurs independent of the input data, it is called a *conclusive race*. If a race occurs for certain values of input but not for others, it is called an *input-dependent race*. A reported race that is not a genuine race is called a *spurious race*. Spurious races are reported due to the limited power of static analysis. Input-dependent races that cannot be detected at compile-time and spurious races are collectively called *potential races*.

A program is called *internally determinate* if the instruction execution sequence in each task and the values of each instruction's operands are guaranteed to be the same in all executions using the same input. An internally determinate parallel program will not cause race conditions, and the execution results are always

```
input N
S = 0
paralleldo I = 1, N
  LOCK(key)
  S =S + I
  UNLOCK(key)
end paralleldo
print S
```

Figure 3.3: A Non-internally Determinate Program

deterministic. However, internally determinate programs may produce a nondeterministic state of variable space during execution. For example, in Figure 3.2, the program is internally determinate because $A(I)$ always changes from 0 to I^2 after each iteration of the *paralleldo*. But the intermediate values of the two array elements $A(1)$ and $A(2)$ change either in the sequence (0, 0)$\rightarrow$(1, 0)$\rightarrow$(1, 4) or (0, 0)$\rightarrow$(0, 4)$\rightarrow$(1, 4), where the symbol (a_1, a_2) represents the value of the array elements of A and the arrow("$\rightarrow$") represents the transition of values.

A program can also have deterministic output without being internally determinate. Races exist, but the program always produces the same result. The program example shown in Figure 3.3 illustrates this situation. This program produces determinate output. However, there are races between the *read* of S in one iteration and the *write* of S in another iteration, and also between the *write* operation of S in one iteration and write in another iteration. While the iterations may be executed out of order or concurrently, the final summation result is deterministic.

3.1.2 Deadlock

Deadlock can occur in many situations. A typical situation is when a concurrent computation can no longer proceed due to task communication failure. As an example, a *circular deadlock* arises when there is a closed loop of tasks in which

```
lock(mutex);
while ( condition )
  wait(event, mutex);
unlock(mutex);
```

(a) correct

```
lock(mutex);
if ( condition )
  wait(event, mutex);
unlock(mutex);
```

(b) incorrect

Figure 3.4: A Program Example with Synchronization

each task has issued an (untimed, unconditional) entry call to the next task in the loop. Eventually, all tasks will wait indefinitely. Similarly, in a parallel program, if multiple threads all issue a *wait* for the next thread in the loop, a deadlock situation occurs. Furthermore, there are cases where deadlock occurs when a *wait* cannot be completed due to improper coding as shown in Figure 3.4 and Figure 3.5.

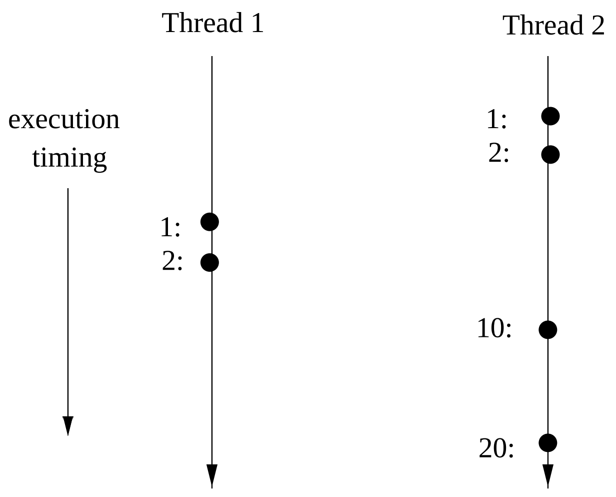
Figure 3.4(a) illustrates correct coding in conjunction with the use of synchronization primitives. If the execution of a `post` can wake up all threads that have issued a matching `wait` call and only one thread is allowed to proceed, the correct coding scheme is to use a `while`-statement for the synchronization. However, a user might incorrectly use an `if`-statement instead of a `while`-statement as illustrated in Figure 3.4(b). The incorrect code might cause multiple threads to proceed if multiple threads are awakened and the condition variable is changed immediately by one of the awakened threads. Deadlock detection may or may not detect this error since there is no circular `wait` involved.

In Figure 3.5, the program segment shows that the `post` statement in Thread 1 is executed prior to the `wait` statement in Thread 2 while the mutex lock is held by Thread 2. In some implementations, the `wait` statement must be executed prior to the matching `post` statement; otherwise, the `wait` statement might wait indefinitely. In the example shown in Figure 3.5, Thread 2 has not begun waiting

```
     Thread 1              Thread 2

1:  cond = 1;          1:  lock(mutex);
2:  post(event);       2:  while (cond == 0)
                           {      .
                                  .
                       10:    wait(event);
                                  .
                           }      .
                       20: unlock(mutex);
```

(a) program code



(b) relative thread execution timing

Figure 3.5: A Program Example with a Synchronization Error

on the condition variable and will miss the event posted by the `post` statement. When Thread 2 calls `wait`, it might never wake up. This error can be detected by delaying the execution of the `post` statements, and seeing that the program will not halt indefinitely at the `wait` statement. The correct coding scheme should enclose the two statements shown in Thread 1 with `lock` and `unlock` statements. Then either the two statements in Thread 1 will be guaranteed by the `mutex lock` to be executed later than the `wait` in Thread 2, or the condition variable `cond` will be changed while Thread 1 holds the `mutex lock` so that the `wait` in Thread 2 will not be executed. In both cases, the error of the infinite wait will not occur. This infinite wait situation also may not be detected by deadlock detection algorithms since there are no circular *wait* operations involved.

Similar to race conditions, deadlock can occur capriciously [63]. Different scheduling policies or the workload on the host machine can affect whether or not a deadlock will occur.

3.2 Detecting Errors via Debugging Tools

Both debugging and testing techniques share the common goal of detecting program errors. Many techniques used in the debugging of a program can be used for testing. Examples are techniques for reproducing an execution and trace analysis for identifying races or deadlock situations. Nonetheless, debugging and testing technologies are applied at different stages in the software life cycle. Debugging is applied to aid a programmer during the development of a program, while testing is performed after a program is completed. The ultimate goal of software testing is to expose faults by generating test data based on test data selection criteria, while debugging is the process of executing a program and identifying the exact statement where a program error occurs. Thus, a debugging tool usually provides an interactive user interface such that a program can be executed interactively, whereas testing simulates the exact environment in which a program is deployed and does not require an interactive interface for executing the program.

3.2.1 Detecting Race Conditions

Detecting unintentional data races by using static or trace analysis has been studied extensively in the context of debugging concurrent and parallel programs [31, 28, 6, 24, 94, 95, 126]. This section provides a review of the issues and approaches with the focus on the program representations and program analysis techniques.

Emrath, Ghosh, and Padua [31] proposed to use static program analysis and execution trace analysis in conjunction with an *order/conflict graph* for the detection of races. The order/conflict graph consists of nodes that represent statements in a program, and edges that represent either an order or a conflict relation between two statements. The symbol S^α represents the instance of the statement S on the α-th iteration of a loop. A directed edge exists from S to T in an order/conflict graph when it can be shown that the ordering $S^\alpha < T^\beta$ exists, that is, that an instance of the statement S is completed earlier than an instance of T. A *conflict* edge, denoted by a directed dotted line, indicates the data dependency between two statements. Figure 3.6 shows an order/conflict graph for a small segment of code. With the order/conflict graph, the static analyzer can detect all conclusive races and some input-dependent races that are detectable at compile time.

Netzer and Miller [93, 94, 95] proposed a two-phase technique, called *trace-and-reply* to detect races in shared memory systems. During the first phase, the program is executed with specific input data, and then all data races that appeared to have occurred, called *apparent data races*, are identified. These apparent data races include those that would never occur due to potential orderings of program statements exhibited by the execution of the program. During the second phase, the apparent results are validated using a temporal ordering graph for representing the run-time ordering in order to identify the apparent data races that are feasible. This

```
cobegin
  doall I=1,N
A:   A(I) = B(I) + 1
D:   post(ev(I))
  end doall
  //
  doall J=3,N
W:   wait(ev(J-2))
C:    C(J) - A(J-2) + 2
  end doall
coend
```

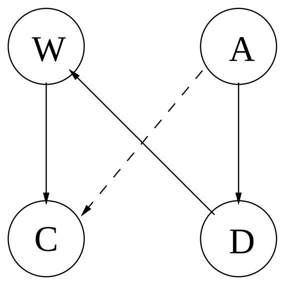


Figure 3.6: Example of Order/Conflict Graph

method is essentially a *post-mortem* approach, which analyzes trace results after the execution of a program; however, they argue that this analysis could be performed in an *on-the-fly* manner, in which trace results are analyzed during the execution of a program.

Damodaran-Kamal and Francioni [24] proposed an approach to detect races and deadlock in message passing systems by executing one thread at a time. Their run-time detection algorithm assumes that send and receive operations supported by their message passing system do not deliver messages directly to each thread or subprocess. Instead, one receiver queue per thread is used to receive messages. An algorithm is presented to control the execution of a single process for each execution run. Breakpoints can be specified by a user one thread at a time. The execution results and the receiver queue are analyzed at the breakpoint to detect program errors including races or communication deadlock, (e.g., a receiver will wait forever when the sender either waits to receive another message or has terminated abnormally).

In summary, static analysis can be cost-effective if races can be detected at compile time. However, static analysis can only detect some of the conclusive and input-dependent races; other races cannot be detected at compile time. Trace analysis can be more effective since it collects information after executing the program. However, it requires executing the program with some specified input and may report spurious race conditions. Hence, software testing methodologies are still needed to produce test data such that nonspurious race conditions can be exposed.

3.2.2 Deadlock Detection

In the area of deadlock detection for concurrent programs, both compile-time and run-time techniques have been developed. Compile-time analysis[87] does not require executing the program but cannot handle situations where tasks are dynamically created. Run-time deadlock detection[63, 157] is effective especially for programs that dynamically create tasks; however, the results are only valid with respect to specific input. The deadlock may not occur if different input values are used to execute the program. Much work has focused on the static detection of deadlock situations in the context of Ada programs. This section briefly describes some examples of these techniques which can all be incorporated into tools for the debugging of Ada programs.

Taylor and Osterweil[132] as well as Long and Clarke[86] have developed techniques to detect anomalies in Ada programs; however, space and time requirements are exponential. Masticola and Ryder [87] designed a safe, polynomial time approximation algorithm to detect deadlock situations in Ada programs. This *safe* approximation approach will always report a deadlock situation if one is present, but might also report a "false alarm," as they call it, when one might not exist. Their analysis detects possible cyclic structures in a static graphic representation of a program, and rejects cycles that cannot result in deadlock. Dwyer and Clarke[30] also developed a polynomial-time, conservative data flow analysis to verify properties of concurrent programs.

Helmbold and Luckham [63] developed a debugging tool to detect deadlock in Ada programs. Any task that cannot proceed with its computation due to waiting for another task is considered to be *blocked*. This tool essentially takes a dynamic approach. A run-time *monitor task* is created and used to interact with all other tasks in the original program. Conceptually, the monitor task keeps track of the information about a task through interacting with other tasks. The tool provides a facility to take a *snapshot* of the interaction when a global blocking situation arises. A snapshot is a textual description of a task maintained by the monitor task. The description includes the name of the task, the numeric ID, the trace status, the Ada entry statement, and the entry queue sizes reflecting the total number of pending Ada requests for the entry call, and the task being called. With this information, a deadlock situation can be determined at an execution breakpoint specified by a user.

3.2.3 Execution Replay and Nondeterminism

When nondeterminism exists in a parallel program, monitoring and controlling the execution become difficult. Without applying special techniques, two situations can occur at execution time after test cases are produced: (1) using the test data, which is generated with respect to a specific path to execute a program may not guarantee that the path will be executed, and (2) executing a program multiple times using the same input data may not cover the same path on each execution. Several methods have been developed to reproduce a particular execution of concurrent programs [124, 128, 76, 83]. Only recently have researchers applied these techniques to *testing* concurrent [68] and message passing systems [127].

Tai, Carver, and Orbid [128] developed an approach, known as *deterministic execution* (DET), to deal with nondeterministic execution during debugging and testing of concurrent programs with rendezvous communication (e.g., Ada programs). For a concurrent program P, a sequence of synchronization events in P is

```
 task T1 is          task T2;          task T3 is            task T4;
    entry E1;                             entry E3;
 end T1;                                end T3;

task body T1 is      task body T2 is    task body T3 is       task body T4 is
begin                begin              begin                 begin
 (a1) accept E1 do      T1.E1;           (a1) accept E3 do       T3.E3
       ...           end T2;                  ...             end T4;
 (a2) end E1;                            (a2) end E3;
end T1;                                 end T3;
```

(a) A Simplified Ada Program

```
     task T1              task T2            task T3            task T4

 ((a1), T2, E1, - )                    ((a1), T4, E3, - )
 ((a2), T2, E1, - )                    ((a2), T4, E3, - )
```

a1: the beginning of an execution of a successful accept
a2: the end of an execution of a successful accept

(b) Partial-Ordering Based SYN-sequences

```
((a1), T2, T1, E1, - )
((a2), T2, T1, E1, - )
((a1), T4, T3, E3, - )
((a2), T4, T3, E3, - )
```

(c) Total-Ordering Based SYN-sequences

Figure 3.7: Examples of SYN-sequences

called a *synchronization sequence* or a *SYN-sequence* of P. Tai *et. al.* defined seventeen types of synchronization events. Examples of the synchronization events are the beginning and end of the execution of an `accept` statement. Formally, a partial-ordering and a total-ordering based rendezvous event is denoted by (V, C, N, D) and (V, C, U, N, D), respectively, where V denotes the type of the rendezvous event; C represents the calling task; U represents the accepting task; N represents the name of the called *entry*; and D denotes additional information needed for some synchronization events. In Figure 3.7, a simplified Ada program is provided, and partial-ordering and total-ordering based SYN-sequences are illustrated. If a program is tested as a whole, the total-ordering based SYN-sequences are used. If only a subset of the tasks in a program is tested, the partial-ordering based SYN-sequences are used. As an example of a partial-ordering, in Figure 3.7(b), the execution of the `accept`(a1) statement in T1 and T3, respectively, is prior to the execution of the `end`(a2) in T1 and T3. But, the order of execution for the `accept` in T1 and that in T3 is not specified. Figure 3.7(c) illustrates a total-ordering SYN-sequence. In this case, the sequence of execution is ordered as follows: (1) the `accept` in T1, (2) the `end` in T1, (3) the `accept` in T3, and (4) the `end` in T3.

DET requires that the compiler insert a conceptual "request for permission" call in front of each rendezvous call. As the program is executed, a separate control task controls the order of synchronization events. The transformed program, including the control task, takes both the input data and various SYN-sequences as input and determines the sequence of execution. The control task controls program execution by synchronizing with the original tasks using the SYN-sequences specified by a tester. Various research issues center around SYN-sequences, including (1) the identification of infeasible SYN-sequences, i.e., SYN-sequences that cannot occur at run-time due to the ordering of rendezvous events, (2) selection, (3) collection, and (4) replay of SYN-sequences.

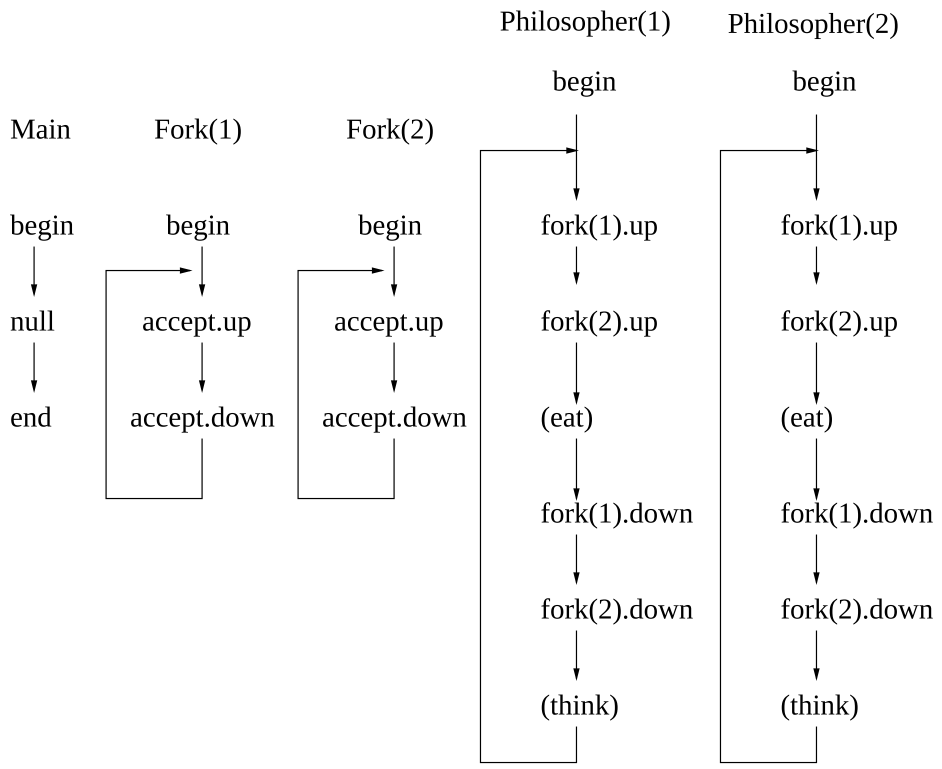
LeBlanc and Mellor-Crummey [83] developed a general trace-oriented technique to enable *replay* for shared memory as well as message passing parallel programs. Their technique involves saving the relative order of significant events as they occur during execution. The saved information can be used for a later replay of the execution with the same input. In their approach, all interactions between processes are modeled as operations on shared objects. A series of modifications to a shared object is represented as a totally ordered sequence of *versions*. Each version has a corresponding version number, which is unique with respect to a particular object. During normal program execution, the partial order of the accesses to each shared object is recorded as a sequence of version numbers for each object. During program replay, the information recorded by each process is used to ensure that the same version for the shared objects is used. Thus, the execution can be reproduced.

In summary, deterministic execution can be applied to reproduce an execution; nondeterministic execution or multiple execution can be applied to test a program. In particular, DET provides an effective approach to repeat the execution of Ada-like concurrent programs, and, LeBlanc and Mellor-Crummey's methodology provides a technique for repeating an execution of a parallel program. Hence, in this dissertation, we assume that techniques are available for reproducing the execution of a parallel program.

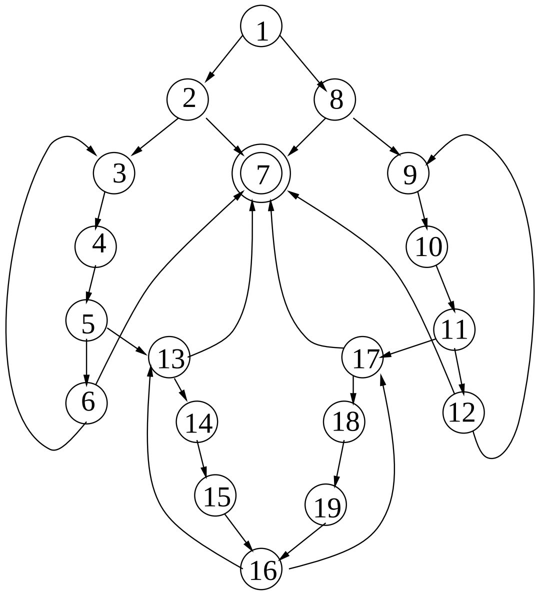
3.3 Structural Testing

This section describes several state-diagram based techniques for testing concurrent programs, a technique for tracing and measuring the testing of implicitly parallel programs, and issues to be addressed in order to perform structural testing of explicitly parallel programs with more general communication mechanisms.

3.3.1 Concurrent Program Testing



(a) Control Flow Graph



(b) Concurrency Graph

Figure 3.8: Example Representation of State Transitions for Concurrent Program

Concurrent States	Fork(1)	Fork(2)	Philosopher(1)	Philosopher(2)	Next States
1	accept up	accept up	fork(1).up	fork(2).up	2,8
2	accept down	accept up	fork(1).up	fork(2).up	3,7
3	accept down	accept down	fork(1).down	fork(2).up	4
4	accept up	accept down	fork(1).down	fork(2).up	5
5	accept up	accept up	fork(1).up	fork(2).up	6,13
6	accept down	accept up	fork(1).up	fork(2).up	3,7
7	accept down	accept down	fork(1).up	fork(2).up	
8	accept up	accept down	fork(1).up	fork(2).up	7,9
9	accept down	accept down	fork(1).up	fork(2).down	10
10	accept down	accept up	fork(1).up	fork(2).down	11
11	accept up	accept up	fork(1).up	fork(2).up	12,17
12	accept up	accept down	fork(1).up	fork(2).up	7,9
13	accept up	accept down	fork(1).up	fork(2).up	7,14
14	accept down	accept down	fork(1).up	fork(2).down	15
15	accept down	accept up	fork(1).up	fork(2).down	16
16	accept up	accept up	fork(1).up	fork(2).up	13,17
17	accept down	accept up	fork(1).up	fork(2).up	7,18
18	accept down	accept down	fork(1).down	fork(2).up	19
19	accept up	accept down	fork(1).down	fork(2).up	16

Table 3.2: Concurrent States in the Dining Philosopher Code

Several groups have developed state-based approaches to testing concurrent programs. Taylor, Levine, and Kelly [131] designed a *concurrency graph* to represent the state transitions of a concurrent program. A concurrent state corresponds to a set of synchronization events occurring in an Ada program. A legal sequence of concurrency states presents a history of synchronization events that occurred during one execution of a program. Figure 3.8 shows a control flow graph and the concurrency graph, for the well-known *dining philosopher problem* written in Ada. In a concurrency graph, a node represents a state, an edge represents the transition from one state to another, and a node denoted by a double circle represents a final state. Synchronization events corresponding to each concurrent state are illustrated in Table 3.2. Figure 3.8(b) illustrates these synchronization events and their transitions. For example, state 1 represents the event that "`accept` is *up* in Fork 1, `accept` is *up* in Fork 2, `fork(1).up` is called, and `fork(2).up` is called." When the `accept` in Fork 1 is changed to *down*, the transition from state 1 to state 2 occurs. When the `accept` in Fork 2 is changed from *up* to *down*, the current state becomes state 7, which is a final state.

Taylor *et al.* presented a series of testing criteria based on the concurrency graph, which can be derived using the formal specification of a program. These testing criteria, such as covering every state or transition once, are referred to as *concurrency analysis techniques*. However, static concurrency analysis techniques are limited in practice by the problem known as the "state explosion problem." That is, as the number of states grows linearly, the total number of possible event sequences grows exponentially. Hence, researchers have developed techniques to reduce the total number of states and derive reduced concurrency graphs based on the assumption that a formal specification is available and the concurrency graph can be derived from the specification [65, 107, 108].

Morasca and Pezzè [89] used a high level *Petri Net*, called ER net, to derive

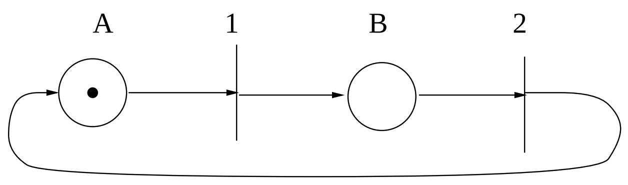


Figure 3.9: An Example Petri Net

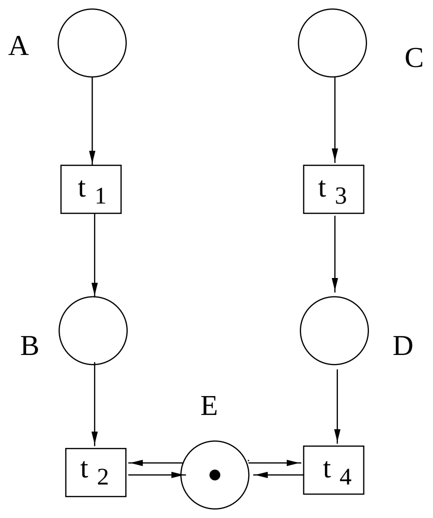


Figure 3.10: An Example ER Net

test data adequacy criteria for concurrent and real-time systems. A Petri net [25] consists of four basic elements: *places*, *transitions*, *arcs*, and *tokens*. A *place p* represents a state of a system, and is represented by a circle in a Petri net. A dot, i.e., a *token*, in the circle represents the current state of the system. A *transition t* is indicated by a vertical or a horizontal bar as shown in Figure 3.9. Each transition has zero or more *input arcs* and *output arcs*. An input arc is a directed edge (p, t) and an output arc is a directed edge (t, p), where t is a transition and p is a place. A transition is *enabled* if there is at least one input token in each of its input places. Any enabled transition can *fire* at a certain time t, removing one token from each input place and depositing a token in each output place at time t. The choice of a transition is indeterminate, which makes Petri nets useful for modeling communication protocols and concurrent events. In Figure 3.9, a Petri net with two places and two transitions is shown. The current state is state A.

An ER net is basically a high level Petri net where transitions, drawn as a *box* in an ER net, are associated with *predicates* and *actions*. The predicates associated with a transition select the tokens that enable the transitions. The action provides the relationship between the tokens removed from the input places of the transition and the tokens produced in the output places of the transition by the firing of the transition. Figure 3.10 shows an example ER net in which a predicate determines the condition when the transition t_2 occurs and the action determines the state changes, i.e., from B to E or from E to E. A *firing* of a transition is associated with the timing when the transition occurs. Figure 3.10 illustrates that the two transitions t_2 and t_4 can be fired concurrently.

Given an ER net, one can define a set of firing sequences or transitions to represent a test suite. An adequacy criterion is a predicate that selects sets of firing sequences or transition sequences. The test data adequacy criteria developed by Morasca and Pezzè include *Firing Testing*, *Firing Sequence Testing*, *Transition*

Testing, and *Transition Sequence Testing*. However, they only discussed various criteria; the applications of these criteria were not presented.

3.3.2 Implicitly Parallel Programs

Implicitly parallel programs are sequential programs with parallelism automatically detected by a compiler or through directives provided by a programmer. Harrold and Malloy [57] developed a technique to keep track of the testing paths and determine the percentage of du-path coverage achieved by each execution of a program. Their method requires a compiler to automatically insert additional statements, called *probes*, to print out trace information at proper locations in the parallelized code. When the program is executed, dynamic data flow analysis computes the define-use pairs that are actually covered.

Since this technique dynamically computes the define-use pairs that are actually covered, those that are not covered can be accurately identified. However, this technique is not designed to support the ultimate goal of generating test cases automatically. Instead, it analyzes the resulting test coverages.

3.3.3 More General Explicitly Parallel Codes

This section reviews the issues and current state of the art in analysis and testing research for explicitly parallel programs with communication features other than rendezvous style. In particular, this section focuses on the static analysis of these programs, the development of a testing framework, the applicability of path coverage testing techniques, and timing-related testing methodologies.

3.3.3.1 Static Analysis of Explicitly Parallel Programs

The static analysis of parallel programs is crucial to applying path testing methodologies to parallel programs. For example, path testing methodologies assume that static analysis can find all du-pairs in a parallel program. If static analysis cannot provide the du-pairs in a parallel program, the path coverage testing methodologies will generate inadequate path coverages. Several factors contribute to the difficulty of the static analysis of parallel programs.

The *generation of a control flow graph* is not straightforward in the presence of the SPMD programming paradigm or dynamic task creation. In an SPMD program, a naively generated control flow graph may not be accurate because *if*-statements are typically used to distinguish the program segments to be executed by each subtask. If these statements are not recognized as having a special purpose, all program segments will be included in the control flow graph for each task, and a static analysis will produce incorrect results. Furthermore, when the total number of subtasks dynamically changes, the static generation of a control flow graph is impossible without obtaining additional information.

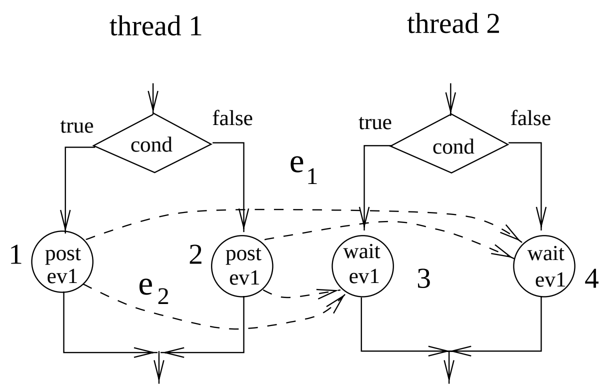
Finding precise matching synchronization calls statically can be difficult for several reasons. For example, in Figure 3.11, both `post` statements are associated with the same event id. The syntax of this code segment is valid since at execution time only one successor node of an *if*-statement will be executed. However, during the process of finding a path coverage to cover node 1 in thread 1, both synchronization edges, e.g. e_1 and e_2, will be considered. Thus, a path finding algorithm will attempt to cover both branches of an *if*-statement when, in fact, only one branch will be executed. Moreover, if the event id associated with a *post/wait* is dynamically determined, the event id cannot be used for finding the matching *post* statements of a *wait* and vice versa.

Computability issues of finding precedence and concurrency information also contribute to the difficulty of static analysis. For any two statements S_i and S_j, S_i is said to *precede* S_j if the execution of S_j implies that S_i must have already executed across all possible execution orders [28]. Concurrency analysis attempts to

```
thread 1                  thread 2

if (cond)                 if (cond)
{ ...                     {
  post (ev1);                 ...
  ...                         wait(ev1);
}                             ...
else
{                         else
  ...                      {
  post(ev1);                  ...
  ...                         wait(ev1);
}                              ...
                           }
```

(a) Control Flow Graph



(b) Code Segment

Figure 3.11: Illustration of Finding Matching Synchronization Calls

statically detect possible concurrent events in a program. Precedence and concurrency relations are important information which can be used to detect data races and compute data flow information in parallel programs [28, 6, 50]. The problem of statically determining the exact concurrent events in Ada programs has been shown to be in NP-C [130], and computing the exact precedence information is at least as hard as co-NP problems [1] for parallel programs with *parallel cases* and *post/wait* synchronization [14].

The difficulty of static analysis hinders the direct application of existing testing methodologies to parallel programs. Finding useful data flow information is difficult. Nevertheless, some heuristics have been developed to provide approximate data flow information.

Callahan, Kennedy, and Subhlok [13] developed a static analysis method that can be applied to parallel programs with event variable synchronization, such as those written in PCF FORTRAN[2]. (Recall that PCF FORTRAN supports `post, wait`, and `advance` for synchronization, and `cobegin/coend` or `parallel sections` for creating multiple threads as stated in Chapter 2.) Their focus is on how data dependent as well as synchronization statements inside loops can be used to analyze complete programs with parallel loop and parallel case style parallelism. Their goal is to eliminate false reports of data races which are seemingly caused by the dependence on private variables due to the fact that private variables do not inhibit parallelization.

Masticola and Ryder [88] developed a set of techniques for statically identifying pairs (or sets) of statements in an Ada-like concurrent program which could

[1] A language L is said to belong to the class **NP** if there is a nondeterministic Turing machine M that accepts L in polynomial time. Let A be an alphabet and set co-$\mathbf{NP} = \{L \subseteq A^* | A^* - L \in \mathbf{NP} \}$, where A^* denotes zero or more instances of the alphabet A.

never happen together. This information aids programmers in debugging and manually optimizing programs. The information can also be used to improve the precision of data flow analysis and detect anomalies in explicitly parallel programs.

Grunwald and Srinivasan [51] developed a method to derive a conservative approximation to precedence information for parallel programs with *parallel sections* and *post* and *wait* synchronization features. They argue that existing methodologies [13, 14], which compute precedence information at compile time, are expensive. They presented a more efficient algorithm that exploits the information such as dominance relations and the semantics of parallel language constructs to account for precedence relations between program statements.

Duesterwald and Soffa [28] presented a data flow based technique to identify the code segments that can potentially be executed concurrently in programs that use explicit concurrent events in the form of tasks, with synchronous intertask communication through *entry call* and *accept* statements as defined by the rendezvous mechanism of Ada. Their technique also works correctly in the presence of the interaction of procedures, tasks, sequential loops, and conditional statements. Procedures can be called recursively, and tasks can be created dynamically. They developed a system of data flow equations to express a partial execution ordering for regions in the program in a *happened before* and a *happened after* relation. After the partial execution ordering is determined, statements that can be executed in parallel are determined.

Grunwald and Srinivasan [50] developed a technique to find the reaching definitions in programs with explicitly parallel language features, e.g., parallel sections, and post/wait/advance synchronization statements. Their objective was to apply the data flow analysis results to program optimization. However, the define-use pairs can also be used to find a du-path coverage.

Knoop, Steffen, and Vollmer [75] presented static analysis algorithms for parallel programs with explicit parallelism and interleaving semantics without special synchronization statements. They also showed how to transfer the static analysis algorithms for sequential programs to the parallel program setting at almost no cost.

3.3.3.2 Theoretical Framework for Parallel Program Testing

A theoretical framework for structural testing defines the notion of a program, testing methodologies, test data selection criteria, the subsumption relations, and testing procedures. When one applies testing methodologies for sequential programs to parallel programs, nondeterminism raises several fundamental issues that affect the theoretical framework. Parallel programming features also raise the need for timing-related testing. These issues are outlined as follows:

- The notion of a program or a test is no longer well-defined without further clarification.
- The definitions of testing criteria are affected. For example, a du-pair coverage becomes meaningless if executing a program may not be able to cover the du-pair that is generated statically.
- The adequacy of a test suite based on a testing criterion becomes unclear. For example, if a set of test data covers all branches of a parallel program on the first execution run, but fails on the second run, what can we conclude about the adequacy of the test suite based on the all-branch criterion?
- A class of timing-related testing, i.e., *temporal testing*, is needed to detect errors associated with events such as thread creation, inter-task communication, and synchronization. In addition to the design issues concerning temporal testing criteria, the testing framework must be redesigned to take these additional testing criteria into consideration.

```
procedure P(Z: in INTEGER;
                X, Y: out INTEGER);
     task P0;
     W: INTEGER;
     begin
(1)   W := Z;
(2)   while Z < W + 2 do
(3)       accept E(V: out INTEGER) do
(4)            Z := Z + 1;
(5)            V := Z;
          end;
     end P0;

     task P1;
     begin
(6)   P0.E(Y);
(7)   write(Y);
      end P1;

    task P2;
     begin
(8)   P0.E(X);
(9)   write(X);
      end P2;

    begin
       cobegin P0 and P1 and P2 coend
    end P;
```

(a) Ada Program

```
stmt  task
 #
               begin
  8    P2:   P0.E(X);
  1    P0:   W := Z;
  6    P1:   P0.E(Y);
              #count := 0; #on := 1;
               while (#count <> 1 ) and ( #on > 0 ) do
  2    P0:            if (Z < W + 2) then {
  3    P0:                    accept E(V: out INTEGER) do
 4,5   P0:                    Z := Z + 1; V := Z;
                              end;
                              #count := #count + 1}
                      else
                              #on := 0;
               if (#count > 0) then
  2    P0:            if (Z < W + 2) then {
  7    P1:                  write(Y);
  3    P0:                  accept E(V:out INTEGER) do
 4,5   P0:                      Z := Z + 1; V := Z;
                            end;
  9    P2:                   write(X)}
                      else
                            { P2: write(X);  P1: write(Y)}
               else
                    { P2: write(X);  P1: write(Y) }
                end
```

(b) One Possible Serialization

Figure 3.12: Example of Weiss's Serialization Method

- A testing process is needed as a guide for a tester to execute the testing based on a specific testing criterion.

Weiss [137] proposed a method called *program serialization* for testing concurrent programs taking into account the nondeterminism. The ultimate goal was to be able to apply the program-based testing criteria originally used for testing sequential programs to testing concurrent programs. Conceptually, each different serialization of subtasks corresponds to a different possible run of the concurrent

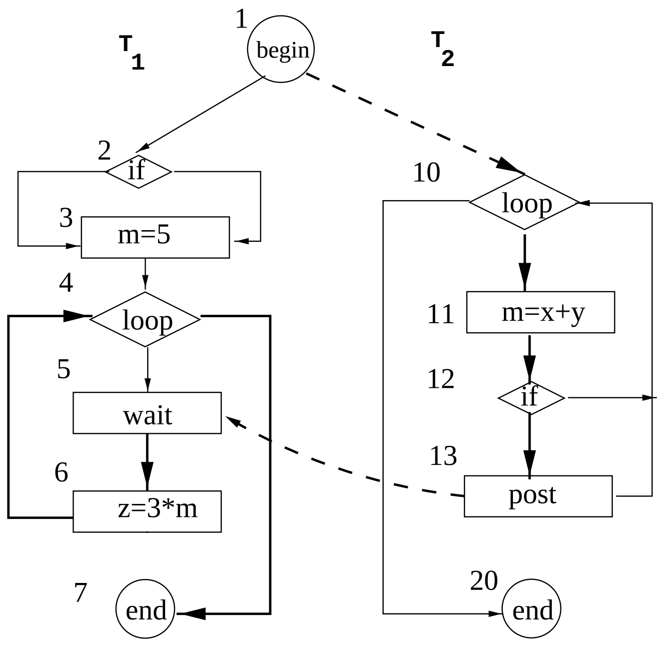


Figure 3.13: Partial Paths Found by Sequential Program Path Coverage Algorithms

program using a fixed set of input data. Figure 3.12 shows an Ada program and one possible serialization of the program.

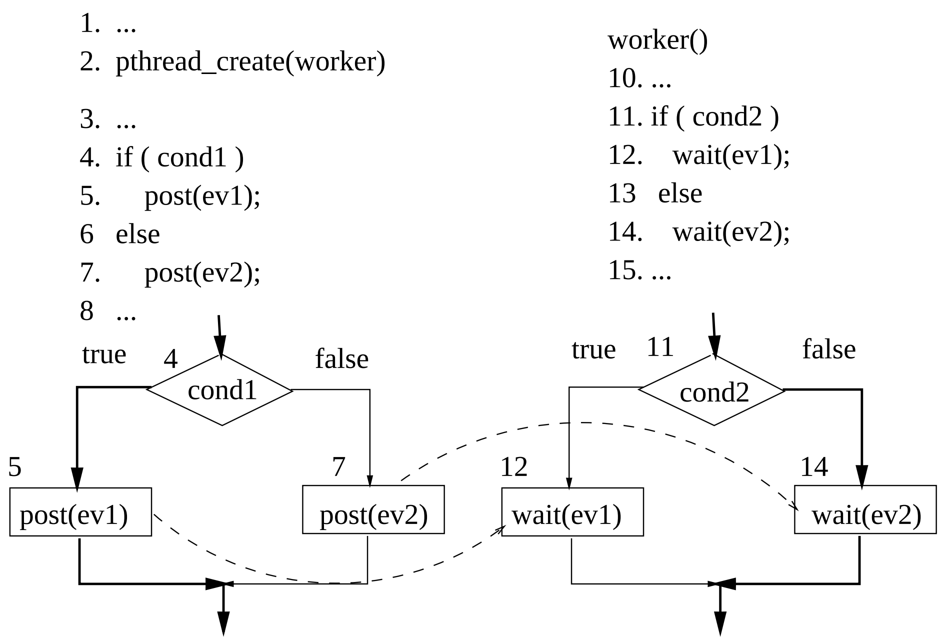
Weiss also described a hierarchy of program-based adequacy criteria derived from sequential program-based criteria. Each program *P* is essentially replaced by a set of serializations of P. However, generating serializations for a concurrent program is difficult and tedious, especially when the number of tasks is large and the execution process is lengthy [155]. This weakness also exists when applying this method to testing parallel programs with more general communication. There has been no prior work focused on building a theoretical framework for testing parallel programs with more general communication.

3.3.3.3 Applicability of Path Coverage Testing Methodologies

Finding path coverage has been a widely accepted technique for sequential program testing. However, without major modification, all sequential path testing methodologies fail when they are applied to generating path coverages for parallel programs. Figure 3.13 illustrates this problem. The variable `m` is defined at statement 11 in thread T_2, and used at statement 6 in thread T_1. If an algorithm for finding a du-path coverage for sequential programs is applied directly without any modification, we will obtain two *partial paths* connected by an edge from a `post` to a `wait`. In Figure 3.13, the original algorithm [99] for generating a du-path coverage returns the path 1-10-11-12-13-5-6-4-7 which consists of two partial paths. Current methods for symbolic execution cannot take two partial paths to generate test cases. The correct path coverage requires two paths, one in each thread. For example, a correct path coverage includes the path 1-2-3-4-5-6-4-7 in thread T_1 and the path 1-10-11-12-13-10-20 in thread T_2.

When one applies a path testing algorithm to find a path coverage in a shared memory parallel program, the algorithm cannot simply cover all statements in one thread and totally ignore statements in other threads. Matching *post* statements of a covered *wait* statement must also be covered. Otherwise, the produced path can cause an infinite wait. For example, in Figure 3.13, to cover the du-pair (m, 3, 6), the path 1-2-3-4-5-6-4-7 cannot be executed alone without executing another path 1-10-11-12-13-10-20 which covers the matching `post` at statement 13.

When static analysis is performed, infeasible paths due to program logic could be generated for parallel programs similar to sequential programs. In addition, Yang and Chung [155] also characterize paths that only cover the *wait* in one thread without a matching *post* in another thread as *infeasible paths*. For example, in Figure 3.14, the set of (partial)paths 4-5 and 11-14 is considered to be infeasible. However, in this dissertation, path coverages are characterized differently; these kinds of paths are not characterized as infeasible paths due to the fact that the execution of these paths can actually occur. Recall that the original definition of infeasible path only



```
1.  ...
2.  pthread_create(worker)
3.  ...
4.  if ( cond1 )
5.      post(ev1);
6   else
7.      post(ev2);
8   ...
```

```
worker()
10. ...
11. if ( cond2 )
12.    wait(ev1);
13   else
14.    wait(ev2);
15. ...
```

Figure 3.14: Yang and Chung's Infeasible Paths

calls a path infeasible if no execution run would ever cover that path. In Figure 3.14, if the truth values of the two conditions *cond1* and *cond2* are *true* and *false*, respectively, the set of (partial)paths 4-5 and 11-14 can indeed be covered. Software testing methodologies are designed to identify errors including these situations. The path characterization issue will be revisited later in Chapter 5.

3.3.4 Timing-Related Testing: Nondeterministic versus Deterministic Execution

Temporal testing criteria have been proposed to expose timing-related errors in concurrent programs [41, 128, 68]. There have been two major approaches to implementing temporal testing: deterministic execution(DET) and nondeterministic execution.

DET has been described previously in Section 3.2.3. Another approach to testing concurrent and parallel programs is called *nondeterministic execution.* Techniques such as multiple execution[124] and delay execution[41] both belong in this category. The objective of nondeterministic execution techniques is to change the execution timing of certain synchronization events without introducing changes in the environment such as the total number of tasks. If a program does not have synchronization errors, varying the execution timing of synchronization events should not cause unintentional results, such as abnormal program termination or deadlock. *Multiple execution* simply executes the same program multiple times using the same input data. *Delay execution* executes the program (also multiple times) after inserting some delay statements to delay the execution of certain program segments.

Gait [41] experimentally studied the effects of delay execution by inserting delay statements into concurrent programs. His test suite of concurrent programs included a multiprocessor operating system, a multiprocess debugging system that can execute on tightly- or loosely-coupled multiprocessors, and a concurrent program that communicates with a monitor process running on a separate processor. In particular, the behavior of concurrent programs with synchronization errors was studied. According to Gait's study, evidence of the *probe effect*, defined as an alteration in the frequency of run-time computational errors observed when delays are introduced into concurrent programs, is observed in the altered behavior of concurrent programs: either a non-functioning concurrent program works with inserted delays, or a functioning concurrent program stops working when previously embedded implicit delays are removed, relocated, or perhaps changed in value. His study strongly motivated the application of the delay execution approach to testing parallel programs. However, Gait's paper was an experimental study without developed testing methodology.

Hwang, Tai, and Huang [68] combined the DET and nondeterministic execution to create a technique called *reachability testing.* A SYN-sequence S is said to be *feasible* for program P with input X if S is allowed by the implementation of P with input X. A prefix of a feasible SYN-sequence of P is said to be *prefix-feasible* for P with input X. A *prefix-based testing* [126] of a program P with input X and SYN-sequence S includes the following steps:

1. Perform deterministic testing of the program P with input X and SYN-sequence S.
2. Perform nondeterministic testing immediately after the end of S is reached if the previous step was successfully completed. Otherwise, stop and check.

The reachability testing approach is essentially a variation of a prefix-based testing. A given program is executed with specific input and SYN-sequences (step 1 above). The program is immediately executed nondeterministically (step 2 above). Then, all possible prefixes of SYN-sequences are generated taking into account the partial order of statements in each thread. Deterministic execution is performed with these SYN-sequences. This approach generates feasible SYN-sequences to test all possible states that the program can reach when executed with the same input. Thus, the approach is named *reachability testing*. Recently, Tai [126] extended this approach to testing message passing systems.

There are several advantages to applying nondeterministic execution techniques to parallel program testing. Either multiple execution or delay execution can be easily implemented. Nondeterministic execution only requires the insertion of delay instructions to simulate the actual execution environment when the software is deployed. Moreover, it will be shown later in this dissertation that the number of temporal test cases can be reduced using static analysis. However, this methodology has several disadvantages. The size of a test suite is exponential if temporal testing is done exhaustively. A delayed execution is time-consuming to complete since the delaying of execution actually takes place [68].

DET's main advantage over the nondeterministic execution technique is that the execution can be cheaper since there is no delay involved [68]. However, when this method is applied to parallel program testing, several issues are raised. The additional control and synchronization mechanisms make the test execution environment very different from the actual execution environment when the software is deployed. The possible effect has never been studied. Moreover, the selection of SYN-sequences can be a tester's nightmare. If the selection cannot cover cases for which synchronization errors can occur, the effectiveness of this method becomes uncertain.

3.3.5 The Size of a Temporal Test Suite

Temporal testing involves altering the execution timing of selected program segments and observing the execution and trace results. For example, to detect race conditions, all definitions and uses of shared variables are included in the testing domain. If other synchronization errors are taken into account, all synchronization calls should be included in the testing domain.

Similar to testing all paths in a program, exhaustively testing all temporal test cases is impractical. Current methods, such as delay execution and DET, usually generate a large testing suite if temporal testing is done exhaustively. Therefore, generating an adequate temporal test suite of manageable size is an important goal.

3.4 Summary

In summary, the challenges described in this chapter include the following:

1. Developing static analysis techniques for analyzing parallel programs.

2. Detecting unintentional races in nondeterministic programs.

3. Detecting deadlock situations in nondeterministic programs.

4. Forcing a path to be executed when nondeterminism might exist.

5. Reproducing a test execution using the same input data.

6. Generating the control flow graph of a nondeterministic program.

7. Finding a proper program representation when the SPMD programming paradigm or dynamic data distribution is used.

8. Providing a testing framework as a theoretical base for applying sequential testing criteria to parallel programs.

9. Investigating the applicability of sequential testing criteria to parallel program testing.

10. Clarifying the notions of testing criteria, e.g., the definitions of test coverage criteria.

11. Redesigning sequential path finding methodologies, e.g., finding a du-path coverage, for parallel programs.

12. Developing temporal testing criteria and defining the subsumption relations with the goal of detecting synchronization errors and nonspurious unintentional races in nondeterministic programs.

13. Reducing the size of a temporal test suite.

This research focuses on a subset of issues mentioned in this chapter. The results of static analysis, e.g., all du-pairs, are assumed to be available. The detection of race conditions and deadlock situations for debugging is assumed to be addressed. The issues of creating reproducible testing and forcing a given path execution in the presence of nondeterministic execution have been addressed by other researchers [124, 76], and will not be further discussed in this dissertation. Dynamic data distribution and an unknown number of thread creations are not considered in the work of this dissertation. Thus, the generation of a control flow graph can be achieved successfully. In addition, infeasible paths due to the logic or predicate conflicts, if any, are assumed to be detectable at run-time. Errors such as missing a matching *post* of a *wait* or vice versa are assumed to be detectable at compile-time.

In summary, the research items 1-5 mentioned above are considered to be adequately addressed since they had been studied extensively; items 6 and 7 are considered future work; and this dissertation addresses research items 8-13.

Chapter 4

A STRUCTURAL TESTING FRAMEWORK FOR PARALLEL PROGRAMS

There have been several efforts led by different research groups to provide a formal foundation for software testing [46, 144, 48]. Their ultimate goals have been to provide a rigorous view on testing methodologies, establish ways of determining the effectiveness of tests in detecting program errors, provide a reasonable and useful interpretation of the notion that successful tests increase one's confidence in the correctness of a program, and provide a guide to testers for executing a test. Although these goals have not been completely achieved [137], some concepts and methodologies have been gradually assimilated and unified. For example, the notion of test data adequacy criteria and the subsumption relations among testing criteria are considered to be well-established for sequential program testing.

When concurrent or parallel program testing is considered, the testing framework for sequential programs is no longer applicable. This chapter addresses the issues and provides solutions to some of these fundamental issues with the following objectives: (1) to provide clear definitions about the notions of a test and path coverage for a parallel program, (2) to derive subsumption relations among temporal testing criteria, and (3) to describe the testing process of a parallel program in the presence of nondeterminism. The ultimate goal is to provide a testing framework for extending the sequential testing criteria to testing parallel programs.

4.1 Parallel Program Model and Notation

4.1.1 Parallel Programming Language Model

A parallel programming language, denoted by $\mathcal{L}$, is assumed to support the creation of multiple threads, a communication mechanism, and the capability of synchronization among multiple threads. A *thread* is an independent execution of a sequence of program statements within a parallel program, (i.e., a subprocess of the parallel process, where a process is a program in execution). A thread creation operation is achieved by calling the function `pthread_create()`. Communication between two threads is achieved through shared variables, which are stored in the global address space and are accessible from all threads. Threads must be able to wait for other threads to complete an activity. For example, one thread T_i might wait for a shared variable to be modified by another thread T_j. Thus, these threads are required to be synchronized. In our model, a *condition variable*, which is a statically defined program variable, can be used to handle this situation.

There are two basic operations on a condition variable: *post* and *wait*. A post operation signals the completion of an activity; a wait operation blocks the execution of the thread where the wait is located until a condition variable is posted. Since a synchronization event among threads is achieved by a number of *post* and *wait* operations that are associated with a condition variable, a condition variable is also referred to as an *event id* in this dissertation. In general, one or more threads can wait on a condition variable. While a thread is blocked, the thread is in a *sleep* state. When a condition variable is posted, one or all of the threads (as specified by the *post* operation) waiting for the condition variable to be posted are allowed to proceed. When the execution of these threads are resumed, we also say that these threads are *awakened*. A *wait* operation which has not yet been posted is called a *pending* wait. A *wait* operation associated with a post is called a *future* wait with respect to this post if the wait begins execution after the post completes.

Formally, the *post* and *wait* operations are performed by calling:

```
post(condition variable, one or all, remembered or lost);
```

```
wait(condition variable);
```

The first argument, i.e., *condition variable*, to call `post()` and `wait()` is a statically defined condition variable. The second argument in `post()` indicates that only one thread or all of the blocked threads will be awakened. When the value **"one"** is specified in a `post()` and more than one wait operation is pending, any one of the pending wait's can be awakened. When the value **"all"** is specified in a `post()`, all of the threads where each wait is located will be woke up. The third argument indicates whether or not a *post* without a pending wait can be remembered or is lost. The second and the third arguments together provide a user the choice of *including* or *excluding* pending wait operations for a post operation. If a user specifies to include only pending wait operations and to exclude future wait operations, the value of the third argument is **"lost"**; if future wait operations should also be included, the value of the third argument is **"remembered"**.

In this dissertation, a *post* and a *wait* can appear in one of the following scenarios: *single post single wait*(SPSW), *single post multiple waits*(SPMW), and *multiple posts single wait*(MPSW).

The SPSW scenario is a point-to-point style of synchronization event where one post wakes up one thread. The SPSW scenario is achieved by `post(condition variable, one, remembered)` and `wait(condition variable)`. As an example, a manager thread could initialize the value of a shared variable and then notify a worker thread that the initialization is completed. This can be achieved by issuing a `post(condition variable, one, remembered)` in the manager thread.

The SPMW scenario is a broadcast event where a single post operation wakes up multiple threads. The SPMW scenario is achieved by a `post(condition` `variable, all, remembered)` and multiple `wait(condition variable)` operations (one pending wait or one future wait in each thread). As an example, a supervisor thread can set up some sharable data and then broadcast an "order" to notify worker threads that the sharable data are ready. This event could be achieved by using the SPMW scenario.

The MPSW scenario is a notification event where the wait operation completes if any one of the post operations is completed. The MPSW scenario is achieved by multiple `post(condition variable, one, lost)` calls (one post per thread) and one `wait(condition variable)` operation. As an example of an MPSW scenario, a thread T_4 is waiting for any of the three threads T_1, T_2 and T_3 to complete an activity. The synchronization can be achieved by using one `post(condition variable, one, lost)` operation in each of the three threads $T_i (1 \leq i \leq 3)$, respectively, and a `wait(condition variable)` in thread T_4. In this scenario, any post operation wakes up the pending wait and the other two post operations are lost as specified by the third argument in `post()`.

Although only three scenarios are mentioned here, other scenarios are possible to achieve using `post()` and `wait()`. However, in the remainder of this dissertation, a synchronization event is assumed to appear in one of these three scenarios, i.e., SPSW, MPSW, or SPMW, unless otherwise specified.

It is assumed that the execution environment of $\mathcal{L}$ supports maximum parallelism. In other words, each thread is executed in parallel independently until a *wait* operation is reached. This thread is blocked until a matching *post* is executed. The execution of *post* always succeeds without waiting for any other program statements to complete.

4.1.2 Program Representation for Static Analysis

Formally, a shared memory parallel program written in $\mathcal{L}$ can be defined as follows.

Definition: A Shared Memory Parallel Program $PROG$

A shared memory parallel program $PROG$ written in $\mathcal{L}$ is defined as a set of threads, or simply $\mathcal{P}ROG = \{T_1, T_2, \ldots, T_n\}$, where T_i, $(1 \leq i \leq n)$ represents the i^{th} thread, and there are $n(\geq 2)$ threads. T_1 is called the *manager* thread created upon the start of execution of the program. All other threads are called *worker* threads, which are created by calling the routine *pthread_create()* from the manager thread.

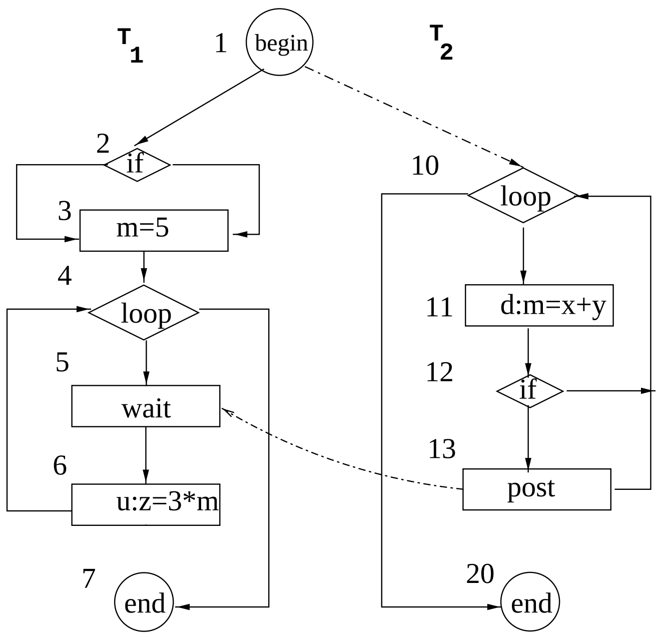


Figure 4.1: Simple Example of PPFG

Static analysis of a program is typically performed on a graphical representation of the program as it is much easier to perform the analysis on a graph that depicts the control flow of the program explicitly.

4.1.2.1 Shared Memory Programs

To represent the control flow of a parallel program written in $\mathcal{L}$, a graphical representation, called a *Parallel Program Flow Graph*($PPFG$), is defined as follows.

Definition: Parallel Program Flow Graph($PPFG$)

A PPFG is a graph $G = (V, E)$. The symbol V represents the set of nodes, representing statements in the source program. The set V can be partitioned into n sets of nodes $V_1, V_2, \ldots, V_n$, where all nodes in V_i are located in thread T_i, $1 \leq i \leq n$. The symbol E represents the set of edges E_S, E_T, and E_I. The set E_I consists of intra-thread control flow edges (m^i, n^i), where m and n are nodes in thread T_i. The set E_S consists of synchronization edges (p^i_s, w^j_t), where p^i_s is a *post* statement with s as the node id in thread T_i, w^j_t is a *wait* statement with t as the node id in thread T_j. The set E_T consists of thread creation edges (n^i, n^j), where n^i is a call in thread T_i to the *pthread_create()* routine, and n^j is the first statement in thread T_j ($i \neq j$).

As an example, Figure 4.1 illustrates a simple example of PPFG. All solid edges are intra-thread edges. Edges in E_S and E_T are represented by dotted edges. In this example, m is a shared variable. In Figure 4.1, a circle represents a *begin* or an *end* node; a diamond represents an *if*- or a *loop*-node; and a rectangle represents other program statements. Within each thread, each statement is numbered with a reverse postorder number. A postorder traversal visits the two child branches of an *if*-node i first prior to visiting the node i. Thus, a reverse postorder will visit the two incoming branches of a *join* node j prior to visiting the node j itself. This graph representation is different from traditional control flow graphs for sequential programs in two ways: (1)*if* and *loop* nodes are distinguished from other nodes, and (2) thread creation and synchronization edges are distinguished from other control transitions for several reasons: the static analysis of a shared memory parallel program requires uniquely identifying communication nodes in order to apply different analysis, known as *transition functions*, and the path finding algorithm requires different data structures for *if* and *loop* nodes. Under some situations, the reverse postorder number is used to identify the branch to take at a *loop* node.

Figure 4.2 shows the PPFG for the Producer/Consumer Problem; however, nodes other than *if* and *loop* nodes are indicated by numbered circles for space reasons.

When a shared memory parallel program is represented as a PPFG, some assumptions are implicitly made:

- The total number of tasks to be created is known at compile-time.
- The programming paradigm is assumed to be in MIMD style, and not in SPMD style.
- The potential matching *post* and *wait* nodes are assumed to be identifiable statically using the event id associated with the *post* and *wait* calls. Hence, the event id cannot be dynamically assigned.

Definition: A Path in a PPFG

Given a PPFG, a *path* $P_i(n^i_{u_1}, n^i_{u_k})$ or simply P_i, within thread T_i is defined to be an alternating sequence of nodes and intra-thread edges $n^i_{u_1}$, $e^i_{u_1}$, $n^i_{u_2}$, $e^i_{u_2}$, …, $n^i_{u_k}$ or simply a sequence of nodes $n^i_{u_1}, n^i_{u_2}, \ldots, n^i_{u_k}$, where u_w is the unique node index in a unique numbering of the nodes in the control flow graph of the thread T_i (e.g., a reverse postorder numbering).

Definition: A Complete/Partial Path in a PPFG

Given a PPFG and a *path* $P_i(n^i_{u_1}, n^i_{u_k})$, P_i is called a *complete path* if $n^i_{u_1}$ is a *begin* node and $n^i_{u_k}$ is an *end* node. If either $n^i_{u_1}$ is not a *begin* node but $n^i_{u_k}$ is an *end* node, or $n^i_{u_1}$ is a *begin* node but $n^i_{u_k}$ is not an *end* node, P_i is called a *partial path.*

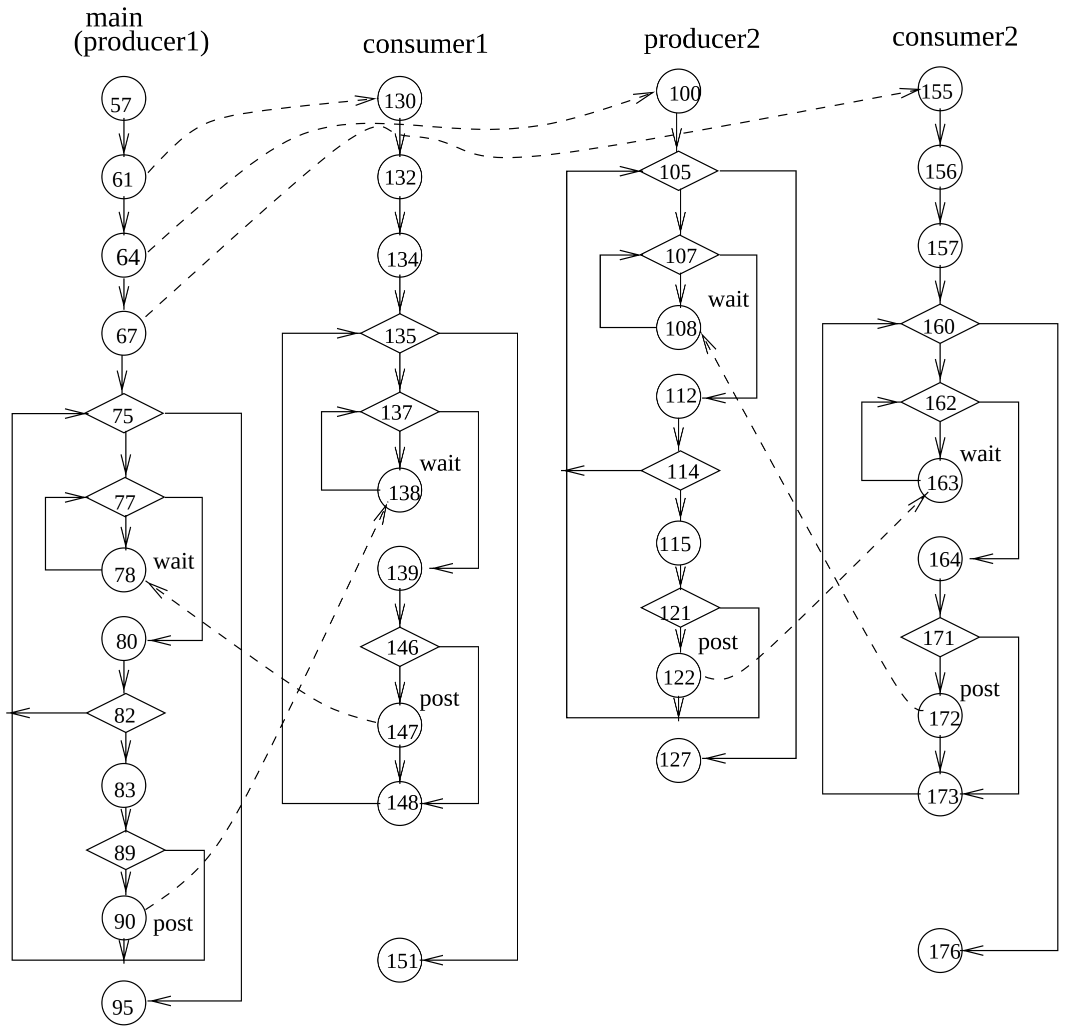


Figure 4.2: PPFG of the Producer/Consumer Problem

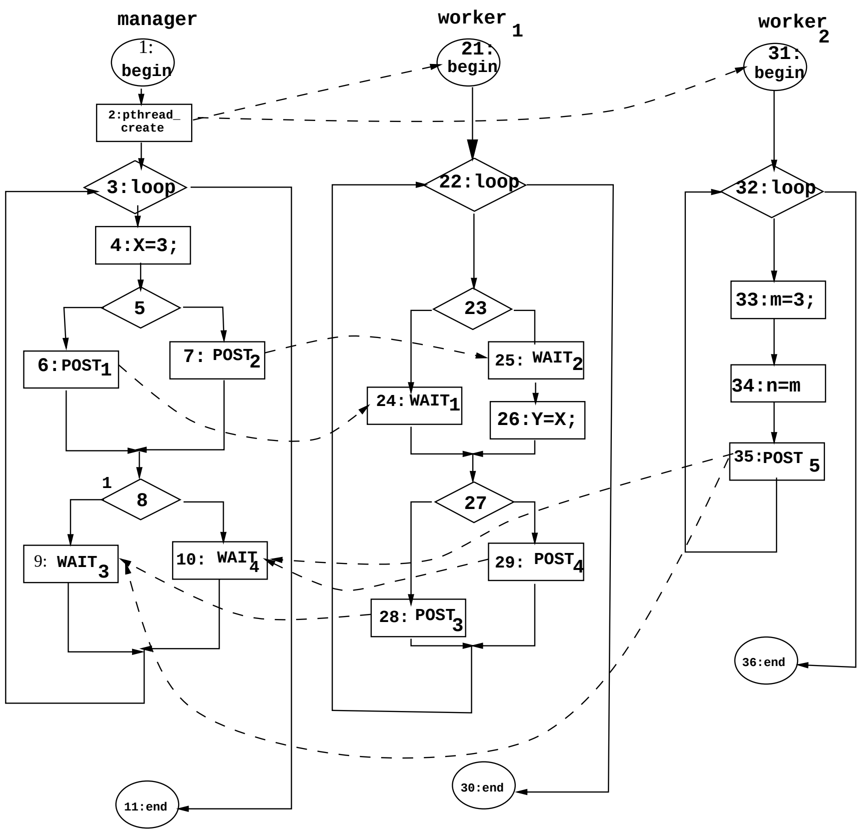


Figure 4.3: A More Complex PPFG Example

In Figure 4.3, the manager thread creates two worker threads. An example of a complete path in the *manager* thread is 1-2-3-4-5-6-8-9-3-11. An example of a complete path in $worker_1$ is 21-22-23-25-26-27-28-22-23-24-27-28-22-30. In the remainder of this dissertation, unless otherwise stated, the term *path* is assumed to mean a complete path.

Definition: An Interprocess Connection

An *interprocess connection*, represented as $IC(n^i_{u_1},\ n^j_{u_k})$, is defined to be an alternating sequence of nodes and edges, which could be intra-thread or synchronization edges from $n^i_{u_1}$ in thread T_i to $n^j_{u_k}$ in thread T_j, where u_w is the reverse post order number in the control flow graph of each thread T_i and T_j. An interprocess connection that involves only one synchronization edge is: $n^i_{u_1}$, $e^i_{u_1}$, $n^i_{u_2}$, $e^i_{u_2}$, ..., $n^i_{u_s}$, $e^{i,j}_{u_s,u_t}$, $n^j_{u_t}$, $e^j_{u_t}$, ..., $n^j_{u_k}$ or simply the sequence of nodes $n^i_{u_1}, n^i_{u_2}$, ..., $n^i_{u_s}$, $n^j_{u_t}$, ..., $n^j_{u_k}$, where $e^{i,j}_{u_s,u_t} = (n^i_{u_s},\ n^j_{u_t}) \in E_S$.

In Figure 4.1, the IC(begin, end) 1-2-3-4-5-6-4-7 is a path, while 11-12-13-5-6 is an $IC(11^2, 6^1)$. In Figure 4.3, the sequence 4-5-7-25-26 is $IC(4^1, 26^2)$. There can be more than one interprocess connection for a given pair of nodes. Since the sequence of nodes in an interprocess connection does not associate with an executable sequence of statements because it is not a complete path, the sequence of nodes is called a *connection*.

Definitions: Reach($\Rightarrow$) and Directly Reach($\Rightarrow_d$)

Given a PPFG $G = (V,\ E)$ and two different nodes, x^i and y^j, it is said that x^i *reaches* y^j, denoted as $x^i \Rightarrow y^j$, if $\exists$ an interprocess connection $IC(x^i,\ y^j)$, when $i \neq j$, or $\exists$ a path $P_i(x^i, y^j)$ when $i = j$. When $i = j$, it is said that x *directly reaches* y, denoted as $x \Rightarrow_d y$.

In Figure 4.1, the *loop* node(10) in thread T_2 directly reaches the *post* node(13), i.e., *loop* $\Rightarrow_d$ *post*; and the d node(11) in thread 2 reaches the u node(6) in thread

1, i.e., $d \Rightarrow u$.

4.1.3 Path Test Coverage for a Parallel Program

Recall that path coverage testing of a sequential program involves finding a path to cover statements in the program based on specific criteria, such as all-statements, all-branches, or all du-pairs. In this dissertation, we focus on the du-path coverage because a du-path coverage subsumes both all-branches and all-statements criteria. In a sequential program, both the definition and the use of a variable are located in the same thread of execution. In contrast, the definition and the use of a shared variable in a shared memory parallel program can be located in two different threads. Thus, the notion of a du-path coverage must be modified.

Definition: du-pair in a Parallel Program

A *du-pair* in a parallel program is a triplet (var, n^i_u, n^j_v), where var represents a program variable defined in the statement represented by node n^i_u and referenced by node n^j_v, and there is at least one path or interprocess node connection from n^i_u to n^j_v with no redefinition of var. The subscript u represents the node id for the node n^i_u in the unique numbering of the nodes in thread T_i; the subscript v represents the node id for the node n^j_v in the unique numbering of the nodes in thread T_j. If n^i_u is a definition node and n^j_v is a use node and $i = j$, the du-pair is called an *intraprocess du-pair*; if $i \neq j$, the du-pair is called an *interprocess du-pair*.

In Figure 4.3, the variable x, defined in statement 4 and used in statement 26, is a shared variable. Statement 4 reaches statement 26 through an interprocess connection 4-5-7-25-26 without redefinition of x. Hence, $(x, 4^1, 26^2)$ is a du-pair. In $worker_2$, the variable m is a local variable. The triplet $(m, 33^3, 34^3)$ is an intraprocess du-pair since both statements 33 and 34 are located in the same thread.

In this dissertation, we call the presence of a node in a path an *instance* of the node. For example, a hypothetical path 1-2-3-4-5-3-4-6-3-7 covers three instances of node 3 and two instances of node 4, etc. It is obvious that an instance of a node can be identified statically for a given path. However, the actual execution of an instance of a node in a given path cannot be determined statically [140].

Definition: $node_1$ **happens before** $node_2$ $(node_1 \prec node_2)$
In a parallel program, if an instance of the node $node_1$ completes execution before an instance of the node $node_2$ begins, it is said that $node_1$ *happens before* $node_2$, or simply $node_1 \prec node_2$.

Recall that in Section 3.3.3.1, we discussed techniques for identifying the relation of *happened before*. Grunwald and Srinivasan [51] developed a method to derive a conservative approximation to precedence information for parallel programs with *parallel sections* and *post* and *wait* synchronization features. Duesterwald and Soffa [28] developed a system of data flow equations to express a partial execution ordering in a *happened before* and a *happened after* relation for regions in Ada programs. In the rest of this dissertation, we assume that techniques are available for identifying the relation of *happens before*.

After a program is executed, a trace of the execution can reflect the execution sequence of every statement in a program. To reproduce the execution sequence of statements in a program, techniques such as deterministic execution(DET) described in Section 3.3.4 can be used. Reproducing an execution sequence of a path is often needed for debugging a program. In the replay, we need to *force a path* to be executed if a path is to be covered when the program being tested is executed. Formally, we define the term *force a path* as follows.

Definition: Forcing a Path
We say that a set of paths $PATH$ is forced to be executed, if for each *path* in $PATH$, for each node n_1 and its successor n_2 in *path*, $n_1 \prec n_2$ when the program

being tested is executed.

Definition: Path Coverage

In thread T_i, node n is *covered by a path* P, denoted by $n \in_p P$, if node n occurs in the path. The symbol $\in_p$ represents the *coverage* of a node. Node n^l is considered to be *covered by a set of paths* $PATH = (P_1, \ldots P_k)$ if $n^l \in_p P_l$ $(1 \leq l \leq k)$. This is denoted as $n^l \in_p PATH$.

Definition: A du-path Coverage

A du-path coverage for a given du-pair (var, n^i_u, n^j_v) is defined to be a set of paths $PATH$ such that $n^i_u \in_p PATH$, $n^j_v \in_p PATH$, $n^i_u \prec n^j_v$, and no other definition of var occurs between n^i_u and n^j_v.

Conceptualy, a du-path coverage must cover both the definition and the use nodes; there must be an instance of the definition node executed prior to an instance of the use node; and no other definition of the variable defined at the definition node occurs between the definition node and the use node. As an example, a du-path coverage of the du-pair $(x, 4^1, 26^2)$ in Figure 4.3 is (1-2-3-4-5-7-8-9-3-11, 21-22-23-25-26-27-28-22-30, 31-32-33-34-35-32-36). The particular path in $worker_2$ is identified since node 9 is a *wait* node and the successful execution of a *wait* node requires the matching *post* in $worker_2$ to be executed.

4.2 Software Testing Framework

This section introduces the notion of a temporal test case, temporal test coverage criteria and subsumption hierarchy, and a procedure for executing the testing process for parallel programs.

4.2.1 Parallel Program Testing Paradigm

Generally speaking, *program testing* is an approach to control the flow of execution through a program in order to detect program faults. Given a parallel program, one needs to test not only the execution flow of a program, but also the synchronization, communication, and thread creation events. Formally, a *test case* TC is a 2-tuple ($\mathcal{PROG}$, $\mathcal{I}$) where $\mathcal{I}$ is the input data to the program $\mathcal{PROG}$, whereas a *temporal test case* TTC is a 3-tuple ($\mathcal{PROG}$, $\mathcal{I}$, $\mathcal{D}$), where the third component, referred to as *timing changes*, is a parameter used for altering the execution time of program segments. For example, the third component represents the SYN-sequences [128] when DET is applied, or a positive timing delay value when the technique of delay execution is applied. Based on $\mathcal{D}$, the execution timing of certain synchronization instructions n, represented as $t(n)$, will be changed for each temporal test and the behavior of the program $\mathcal{PROG}$ will be observed. In practice, timing changes may be introduced progressively. Hence, the *level* of temporal testing is defined to be the total number of timing change statements, e.g., sleep(duration), introduced in each test case.

Conceptually, given a parallel program $\mathcal{PROG}$, one must execute $\mathcal{PROG}$ without and with timing changes. That is, for each input, the program is executed many times based on a specific testing criterion τ_l, where l is the *level* of temporal testing. If all test runs show the same execution result, this set of test data is considered τ_l adequate. If no timing changes are used, the test suite is τ_0 adequate. If the level of temporal testing is i, this test suite is called τ_i adequate. That is, if a parallel program is executed many times with a test suite and all trace results indicate that the same paths are covered, the test suite is considered τ_l adequate. For example, given a du-pair, a test suite to cover the du-pair is generated. Then each path is tested without any delay. If the same paths are covered when the program is executed many times for a du-pair, the test suite is considered τ_0 adequate with respect to that du-pair. There is no general rule for how many times a program should be executed in each test case. But it is believed that the more a program is executed, the higher the possibility of detecting synchronization errors.

4.2.2 Test Data Adequacy Criteria

In the discussion below, we use the symbol P to represent a program, t to represent a test case, T to represent a test suite, i.e., a set of test cases, and C to represent a test data adequacy criterion.

Recall that in Chapter 2.1.2, a *test data adequacy criterion* was formally defined as follows.

A test suite T is C adequate for a given sequential program P if there exist some test cases in T, such that the test data adequacy criterion C is satisfied (by the test cases) [141].

We call an *entity* of a program P the component of P, with which a test data adequacy criterion is associated. For example, an entity associated with the statement adequacy criterion is a statement in a program; an entity associated with the data flow adequacy criterion is a du-pair; and an entity associated with the path testing criterion is a path in a program.

If an entity is involved when the program is executed using a test case, the entity is *covered* by the test case. We use the symbol $COVERED_C(P, t)$ to represent the set of covered entities in P associated with the criterion C when the program P is executed using the test case t.

A test data adequacy criterion is *satisfied* by a test suite T if every entity associated with the criterion in the program is covered by a test case t in T. For example, for a given program PPFG=(V, E), the test data adequacy criterion *all statements* is *satisfied* by a set of test cases T if $\forall v \in V, \exists t \in T,$ *such that* $v \in COVERED_C(P, t)$.

For the discussion of test data adequacy criteria for parallel programs, we need to define the following terms: *consistent* and *strictly consistent* execution results.

The *execution result* of a shared memory parallel program P using the test case t is the set of values of all shared variables in the program P when the program terminates. The *execution result* for the ith execution of such a program is represented by $\mathcal{R}^P(t;i)$. Two execution results for the ith and jth execution runs are *consistent* if $\mathcal{R}^P(t;i) = \mathcal{R}^P(t;j)$. If for all i, $\mathcal{R}^P(t;i)$ stays unchanged, the execution results of program P are *strictly consistent*.

Obviously, it is impractical to try to show that all possible execution results of a program are strictly consistent. We can only expect to show that no inconsistent execution results are produced after a program is executed many times using the same input. This term is only introduced for completeness.

We define *weak test data adequacy criterion* and *test data adequacy criterion* for parallel programs as follows.

For a program P, a test suite T is weakly C adequate if there exists a test suite T such that the test data adequacy criterion C is satisfied by T after P is executed many times using a test case t in T, and execution results are always consistent. If for all possible execution runs, the results are strictly consistent, T is C adequate.

If a program P_2 is transformed from another program P_1 by inserting statements such as `sleep()` into P_1 for changing the execution timing of some statements in P_1, then the program P_2 is called a *mutant program*. The concepts of *weak test data adequacy criterion with timing changes* and *test data adequacy criterion with timing changes* are defined for parallel programs as follows.

For a program P, a test suite T is weakly C adequate with timing changes if there exist a mutant program P_m of P such that C is satisfied by T after P_m is executed many times using a test case t in T, and consistent execution results are always produced. If T is weakly C adequate with timing changes and all possible

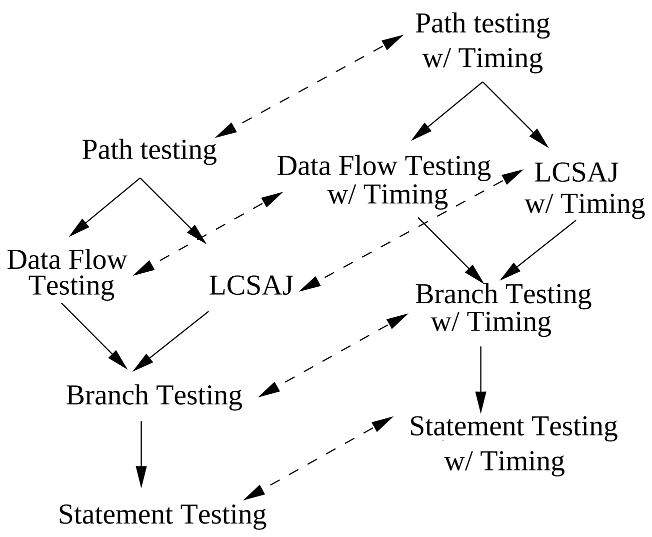


Figure 4.4: Testing Criteria Subsumption Hierarchy for Parallel Programs

execution results of the mutant program P_m are strictly consistent, then T is C adequate with timing changes for P.

4.2.3 Partial Ordering of Coverage Criteria

Recall that the concept of *subsumption* has been defined previously in Section 2.1.3 as follows.

Given a program P and two criteria A and B, we say that A subsumes B if when a test suite T is A adequate then T is also B adequate.

We formally introduce the concept of *weak subsumption* as follows.

Given a program P and two criteria A and B, we say that A weakly subsumes B if when a test suite T is weakly A adequate (with or without timing changes) then

T is also weakly B adequate.

Figure 4.4 shows the subsumption hierarchy for parallel programs. The subsumption relations between the criteria of path testing are shown with solid arrows. Weak subsumption relations are shown with dotted arrows in Figure 4.4. The temporal testing criteria hierarchy is similar to the hierarchy representing the subsumption relations among testing criteria without timing changes.

We claim without proof that a temporal testing criterion with timing changes weakly subsumes a corresponding testing criterion without timing changes, and vice versa. For example, all-du-path coverage with (without) timing changes weakly subsumes all-du-path coverage without (with) timing changes. The reason is obvious. Without the existence of synchronization errors, consistent results will be produced no matter how synchronization events are delayed. For example, if T is weakly *all statements* adequate for P with timing changes, T is also weakly *all statements* adequate for P without timing changes, and vice versa. However, this could not be true with the presence of synchronization errors. When synchronization errors exist, a temporal testing criterion could be satisfied when executing a mutant program with timing changes using a test suite T, but could fail to be satisfied when the program is executed without timing changes using T, and vice versa. For example, if T is *all statements* adequate for P with timing changes, T could fail to be *all statements* adequate without timing changes.

4.2.4 Nondeterminism and the Testing Process

Recall that the testing process for path coverage testing of sequential programs consists of the following steps.

1. Generate du-paths for a given program.

2. Generate input data using the du-paths generated previously.

3. Execute the program with the input data.

4. Evaluate the trace and execution results.

For testing parallel programs, the algorithm for generating test data is different and some additional steps are needed after executing the program with the input data. In the following, we describe a process for testing shared memory parallel programs. We use du-path testing as the testing criterion throughout the description. Any other path testing criterion could be substituted in its place.

1. Generate testing paths and corresponding test data via a static analysis. A static analysis technique is used to find all du-pairs. Then a du-path coverage is computed for each du-pair using the technique developed in this dissertation. The results can be used for generating test cases.

2. Execute the program many times without any timing changes. After test cases are generated, they are used to execute the parallel program without inserting any delay statements.

3. Examine the trace and test results manually. If a program produces different results when it is executed many times with the same input, it is a strong indication that a synchronization error could exist. The testing process should stop at this point and the debugging process must be resumed. If all execution and trace results generated by running the program with the same input multiple times stay the same, the temporal testing must still be conducted due to the fact that τ_0 adequate criteria and τ_i adequate criteria, where $i > 0$, are incomparable.

4. Generate temporal test cases with respect to the du-paths. Chapter 6 describes how to generate these test cases.

5. Perform temporal testing with various levels of delays. Again, chapter 6 describes this process in detail.

6. Evaluate the trace and execution results. The results should be evaluated by a human tester.

In the steps described above, step 1 can be automated for some cases. Steps 2, 4, and 5 can also be automated. It requires human intervention for evaluating the trace and test results. However, detecting the difference in multiple execution runs can also be automated.

During the testing process, if debugging is needed, deterministic execution must be employed to reproduce the same test scenario. Otherwise, multiple execution and nondeterministic execution can be used.

4.2.5 Summary

In this Chapter, a program model is presented to represent a shared memory parallel program. A parallel program testing framework is presented to define a test, a test case, a temporal test case, and a level of a temporal testing. Path coverage and the adequacy of a test suite in the context of a parallel program are defined. The hierarchy of subsumption relations among temporal testing criteria is provided. A procedure for executing the testing process is presented as a guide for conducting the testing of parallel programs.

Chapter 5

FINDING A DU-PATH FOR SHARED MEMORY PARALLEL PROGRAMS

In this chapter, an algorithm for finding a du-path with respect to a given du-pair in a shared memory parallel program is presented. The problem of finding a du-path coverage for testing a shared memory parallel program can be stated as follows:
Given a shared memory parallel program, $\mathcal{P}ROG = (T_1, T_2, \ldots, T_n)$, for each du-pair, (var, n_u^i, n_v^j), in $\mathcal{P}ROG$, find a set of paths $PATH = (P_1, \ldots P_k)$ in threads T_1, $T_2, \ldots T_k$, that covers the du-pair (var, n_u^i, n_v^j), such that $n_u^i \prec n_v^j$.

This dissertation only discusses finding du-pairs with the definition and use in different threads; finding du-pairs within the same thread is a subcase. The ultimate goal of finding du-paths is to be able to generate test cases automatically for testing all du-paths in shared memory parallel programs adequately. Testing a du-path in a shared memory parallel program focuses on whether or not the program being tested uses shared variables correctly by detecting the possible occurrence of synchronization errors. Finding a du-path to cover a du-pair is the first step in du-path testing. Hence, finding a du-path for a given du-pair in a shared memory parallel program is critical.

Section 5.1 discusses the issues faced in finding a du-path coverage for a given parallel program using some examples. Section 5.2 defines and characterizes a classification of du-paths in parallel programs. Section 5.3 discusses computational complexity issues concerning the algorithm for finding du-paths of the different classes. Section 5.4 describes relevant related work in finding path coverages in a sequential program. An existing path testing model for concurrent programs is also discussed. Section 5.5 illustrates issues involved in applying currently existing path finding algorithms to finding a du-path for shared memory programs. Section 5.6 presents a novel algorithm to find a du-path for a given du-pair in a shared memory program. The correctness and the running time of this algorithm are examined.

5.1 Inherent Problem Exhibited

In this section, simplified examples are used to demonstrate some of the inherent problems in finding a du-path in parallel programs. These issues are not necessarily exhaustive, but instead meant to illustrate the complexity of the problem of automatically finding a du-path for parallel programs. In addition to these issues, the algorithm must be capable of finding complete paths in each thread with nodes to be covered by the path coverage. Recall that the existing algorithm would find partial paths.

Figure 5.1 contains two threads, the `manager` thread and a `worker` thread. This figure demonstrates a path coverage that indeed covers the du-pair, but does not cover both the *post* and the *wait* of a matching *post* and *wait*. If the *post* is covered and not a matching *wait*, the program will execute to completion, despite the fact that the synchronization is not covered completely. However, if the *wait* is covered and not a matching *post*, then the program will hang with the particular test case, causing the testers to believe that there is a program error when, in fact, there may be no error.

In the same example, the `worker` thread may not complete execution, whereas the `manager` thread will terminate successfully if the generated path will cause the loop in the `manager` thread to iterate only once, while the loop in the `worker` thread

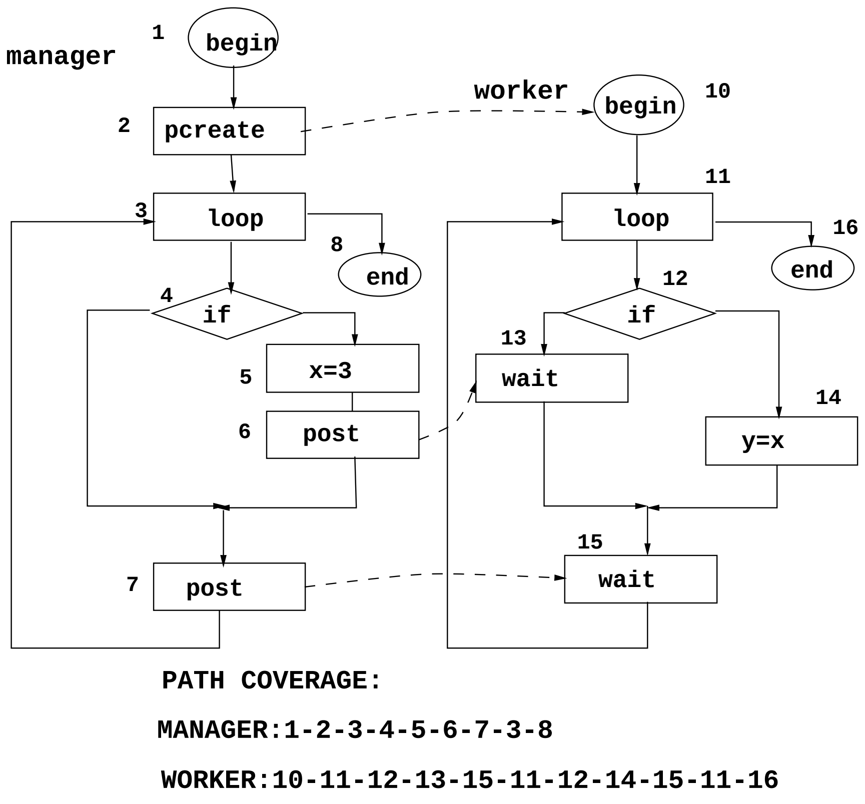
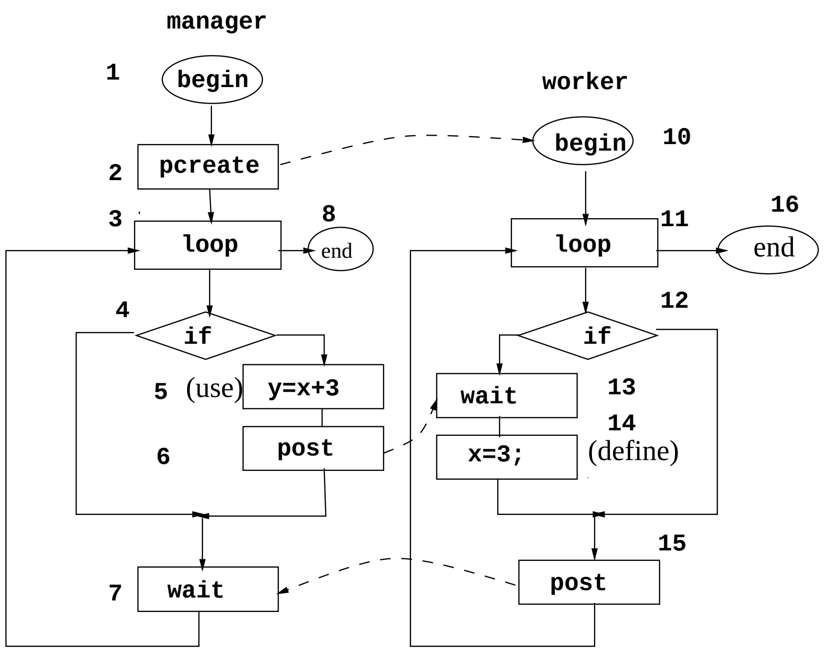


Figure 5.1: A du-path can potentially cause an infinite wait.

will iterate twice. This shows how the inconsistency in the number of loop iterations may cause one thread to wait infinitely.

In Figure 5.2, the generated paths cover both the *define* at node 14 and the *use* at node 5, but the *use* node will be reached before the *define* node, that is, $define \not\prec use$. If the data flow information reveals that the definition of x in the `worker` thread should indeed be able to reach the use of x in the `manager` thread, then we should attempt to find a path coverage that will test this du-pair. The current path coverage does not accomplish this coverage.

In Figure 5.2, if the generated paths are 1-2-3-4-7-3-8 in the `manager` thread and 10-11-12-13-15-11-16 in the `worker` thread, the `worker` thread will also hang



PATH COVERAGE:

MANAGER:1-2-3-4-5-6-7-3-8

WORKER:10-11-12-13-14-15-11-16

Figure 5.2: Du-pair is incorrectly covered.

indefinitely. This situation was caused by the selection of an incorrect branch of node 4 in the `manager` thread. Hence, the selection of a branch at an *if*-node is also important.

5.2 A Classification of du-path Coverage for Parallel Programs

For the convenience of explanation, we define the concept of *matching nodes.*

The set of matching *post* nodes of a *wait* node w is the set MP, where $MP(w) = \{p|(p, w) \in E_s\}$

The set of matching *wait* nodes of a *post* node p is the set MW, where $MW(p) =$

$\{w|(p,w) \in E_s\}$

The examples illustrated in Section 5.1 motivate a classification of du-path coverages. In particular, a du-path coverage can be classified as either *acceptable* or *unacceptable*, and as either *w-runnable* or *non-w-runnable.*

Conceptually, a path coverage $PATH$ is acceptable if the definition and the use nodes are both covered in the order such that the definition node happens before a post node; a post happens before a wait node; and a wait node happens before the use node. Recall that in Chapter 3.3.3.1, we discussed techniques to identify the relation of *happened before* [51, 28].

Furthermore, an acceptable path coverage $PATH$ is w-runnerable if the set of paths $PATH$ can be used to generate a test case that does not cause an infinite wait in any thread. To achieve this, for each instance of a wait in $PATH$, $PATH$ must also cover an instance of a matching post.

5.2.1 Acceptability of a du-path Coverage

A set of paths $PATH$ is called an *acceptable du-path coverage*, or a $PATH_a$, for the du-pair (var, $define$, use) in a parallel program free of infeasible paths if all of the following conditions are satisfied:

1. $define \in_p PATH$; $use \in_p PATH$,
2. $\forall$ *wait* nodes $w \in_p PATH$, $\exists$ a *post* node $p \in MP(w)$, such that $p \in_p PATH$,
3. if $\exists(post, wait) \in E_S$, such that $define \prec post \prec wait \prec use$, then $post, wait \in_p PATH$.
4. $\forall n^j \in_p PATH$ where $(n^i, n^j) \in E_T$, $\exists n^i \in_p PATH$.

These conditions ensure that the definition and use are included in the set of paths $PATH$ (condition 1); for each *wait* node, all matching *post* nodes are covered by $PATH$ (condition 2); any *(post,wait)* edges between the threads containing the definition and use (condition 3), and involved in the data flow from the definition to the use are included in the path; and for each head of a thread creation edge e, i.e., $H(e)$, the associated tail of e, $T(e)$, is included in the path(condition 4). If any of these conditions is not satisfied, the path coverage is considered to be *unacceptable*. For example, if only a *wait* is covered in a path coverage and none of the matching *post* nodes are covered, the path coverage is not a $PATH_a$. Figure 5.2, where the define and use are covered in reverse order, shows an instance that only satisfies the first two and last conditions, but fails to satisfy the third condition.

5.2.2 W-runnability of a du-path Coverage

Section 5.1 illustrates that a parallel program can cause infinite wait under a given path coverage, even when the du-path coverage is acceptable. More formally, a set of paths is w-runnable if it is a $PATH_a$ and all of the following additional conditions are satisfied. In the remainder of this chapter, unless specified explicitly, the subscript i of a node n_i is used to represent an instance of the node n.

1. For each instance i of a *wait* w^t, $w_i^t \in_p PATH$, (possibly represented by the same node $n^t \in_p PATH$), $\exists$ an instance u of a *post*, $p_u^s \in_p PATH$, where $p_u^s \in MP(w_i^t)$. An instance of a *wait* or *post* is one execution of the *wait* or *post*; there may be multiple instances of the same *wait* or *post* in the program.

2. The execution of the paths in $PATH$ will not cause a circular wait. That is, for a program with threads T_0, T_1, ..., and T_n, $\nexists$ threads T_{i_0}, T_{i_1}, ..., T_{i_m}, such that $(w^{i_0} \prec p^{i_0} \wedge w^{i_1} \prec p^{i_1} \wedge \ldots \wedge w^{i_m} \prec p^{i_m}) \wedge (p^{i_0} \prec w^{i_1} \wedge p^{i_1} \prec w^{i_2} \wedge \ldots \wedge p^{i_{m-1}} \prec w^{i_m}) \wedge (p^{i_m} \prec w^{i_0})$, where $1 \leq m \leq n$, $0 \leq i_x \leq n$, $\forall\ x$, $0 \leq x \leq m$.

The first condition ensures that, for each instance of a *wait* in the $PATH$, there is a matching instance of a *post*. However, it is not required that for every instance of a *post*, a matching *wait* is covered. In other words, the following condition is not required: $\forall post$ nodes $p \in_p PATH$, $\exists$ a *wait* node $w \in MW(p)$, such that $w \in_p PATH$. The second condition ensures that a deadlock will not occur due to a circular wait. For example, for a program with two threads T_i and T_j, $\nexists post$ nodes p^i, p^j, and *wait* nodes w^i, w^j, $\in_p PATH$ such that $(p^i \prec w^j) \wedge (p^j \prec w^i) \wedge (w^i \prec p^i) \wedge (w^j \prec p^j)$. However, these conditions do not guarantee the completion of execution; the classification is solely based on the semantics of synchronizations associated with *post* and *wait* but not *all* program statements.

Although all $PATH_w$ path coverages can also be classified as $PATH_a$, the term $PATH_a$ and non-$PATH_a$ will be used to strictly refer to non-$PATH_w$ $PATH_a$ and non-$PATH_w$ non-$PATH_a$ path coverages, respectively. Figure 5.1 shows an example of a $PATH_a$ path coverage due to the fact that the first necessary condition for a $PATH_w$ is violated. Figure 5.2 shows a non-$PATH_a$ for which the third necessary condition for a $PATH_a$ is violated.

From the viewpoint of software testing, it is equally desirable to find all $PATH_w$ path coverages and all $PATH_a$ path coverages because to test a program, one would use test cases for which successful results are expected as well as test cases for which the program should report an error. In other words, program testing should target both types of program behavior: (1) a program fails when successful results are expected, and (2) a program terminates successfully without reporting synchronization errors when error messages are expected. Hence, $PATH_w$ path coverages will be used to test the first type of program behavior, whereas $PATH_a$ path coverages will be used to test the second type of program behavior.

5.3 Computational Complexity Issues of du-path Testing

In this section, several computational complexity issues about du-path coverage are discussed. They are listed as follows:

1. $EXIST_a$ - Given a PPFG G=(N,E) and a du-pair (var,d, u), does a $PATH_a$ exist in G to cover (var,d, u)?
2. $CLASSIFY\_PATH$ - Given a PPFG G=(N,E) and a path coverage $PATH$, is $PATH$ a $PATH_w$?
3. $EXIST_w$ - Given a PPFG G=(N,E) and a du-pair (var,d, u), does a $PATH_w$ exist in G to cover (var,d, u)?

For a given parallel program $\mathcal{PROG}$ and a du-pair, we claim that

1. $EXIST_a$ is in **P**, and
2. $CLASSIFY\_PATH$ is in **P**,
3. $EXIST_w$ is **NP-C**.

Theorem 1: $EXIST_a$ is in **P**.
Proof:

This theorem can be proven by presenting an algorithm that is capable of finding a $PATH_a$ in deterministic polynomial time when one exists in the program.

It will be proved later in Theorem 6 that the running time of the hybrid algorithm presented in this chapter equals O($k^2 * |V|$) for a constant k. Thus, $EXIST_a$ is in **P**.
Q.E.D.

Theorem 2: $CLASSIFY\_PATH$ is in **P**.
Proof:

This theorem can be proven by presenting an algorithm that is capable of determining the classification of a path coverage in deterministic polynomial time.

```
Input: A PPFG and a set of paths path[i]
Output: TRUE if a PATH_w; FALSE otherwise.
Method:  Initialize a set of pointers, one pointer per thread;
         Initialize mark_flag to zero for each WAIT node;
         /* Simulate the execution of the parallel program */
         changed = TRUE;
         while ( changed )
         {
          changed = FALSE;
           for (i=0; i < number_of_threads; i++)
           {
             while ( (pointer[i].type != END) &&
                     not ((pointer[i].type == WAIT)&&(mark_flag  <0)) )
              {
                switch (pointer[i].type)
                 {
                   case POST: Increment the mark_flag of the matching WAIT by 1;
                              If ( mark_flag == 0) // assume the matching WAIT is in thread j
                              {     pointer[j] = next node in path[j];
                                    changed = TRUE;   }
                              pointer[i] = next node in path[i];
                              break;
                    case WAIT: decrement the mark_flag by 1;
                               if (mark_flag >= 0)
                                     pointer[i] = next node in path[i];
                               break;
                    otherwise: pointer[i] = next node in path[i];
                 } // end of switch
              } // end of while
           } //end of for
          } // end of outer while
          /* check the simulated execution results */
          /* if any END node is not reached, return not-PATH_w */
          for (i=0; i < number_of_threads; i++)
          {
            if (pointer[i].type != END node )
              return 0; // not PATH_w;
          }
          return 1; // PATH_w;
```

Figure 5.3: Path Classification Algorithm

The algorithm for determining the path classification is given in Figure 5.3. If the path is a $PATH_w$, this routine returns a 1; otherwise, it returns a 0.

This algorithm contains two parts. The first part of the algorithm simulates the execution of all threads in parallel by using a for loop and the inner *while* loop to iterate through all paths in the path coverage until either all paths reach the end of the thread or a *wait* node with no matching *post* is found. The second part of the algorithm checks whether any thread halts in the middle of a path. If there is any thread whose current node pointer points to a node other than an *end* node, this path coverage is a $PATH_a$. Otherwise, the path coverage is a $PATH_w$. The running time of the second part is generally smaller than that of the first part. Hence, we can ignore the running time of the second part.

In the rest of proof, we use the scenario of *multiple posts single wait(MPSW)* to analyze the running time. This scenario is a worst case in which a *wait* node is visited most often.

We describe the algorithm first and the running time later. Conceptually, to determine the classification of a path coverage, we need to check whether an instance of a *post* node appears in the set of paths for each instance of a *wait* node. Also, the sequence of execution will cause each instance of the *post* node to be executed for each instance of the *wait* node. The checking whether or not there exists an instance of *post* for an instance of *wait* is achieved by using the following strategy. In the body of the inner *while* loop, the type of a node is checked. If it is a *post* node, the `mark_flag` of each of the matching *wait* nodes is incremented by 1. (If the value of its `mark_flag` equals zero, the current pointer of the thread j where the *wait* locates must be changed to point to the next node in the path j. In addition, the `changed` flag is set to $TRUE$.) If the type of a node is a *wait* node, the value of the `mark_flag` is decremented by 1. At the end of each iteration of the *for* loop, if there is any *wait* node whose `mark_flag` contains a negative value, it indicates that a matching *post* was not reached.

We start the discussion of the running time by analyzing the upper bound on the total number of nodes in a $PATH_w$ or $PATH_a$. Based on part of the proofs for Theorem 1 previously and Theorem 6 later, we conclude that the maximum TRN in all nodes when running the du-path finding algorithm equals k^2, where k is the total number of *post* and *wait* nodes in the PPFG. Thus, we argue that the upper bound on the total number of times a *post* node or a *wait* node can be visited when finding a path to cover the *post* or *wait*, respectively, is at most O(k^2). Hence, the total number of nodes in a path coverage equals O($k^2 * |V|$).

The running time of the algorithm in Figure 5.3 is bounded by the number of times the body of the `switch` statement is executed. This upper bound is in turn bounded by the total number of nodes in PPFG multiplied by the number of times each node in the PPFG is visited. In our worst case MPSW scenario, an instance of a wait node will be visited as many times as the number of matching post nodes. All other instances in the set of paths will be visited exactly once. This is due to the following reason. An array of pointers is used to remember where the traversal stops every time a thread i is traversed. But the `pointer[i]` for the thread i is not reinitialized every time to point to the first node in `path[i]`. Instead, the traversal of the remaining nodes in the set of paths continues from the point it left off until either a wait node with negative `mark_flag` or an *end* node is reached in each thread.

In summary, the outer `while` loop, the `for` loop and the inner `while` loop cause all instances of nodes except *wait* node to be visited exactly once. An instance of a *wait* node will be visited once when each instance of a matching *post* is visited. Thus, each instance of a *wait* node will be visited at most $m+1$ times, where m is the maximum number of matching *post* nodes of all *wait* nodes. In total, all instances of nodes in the set of paths can be visited at most O($m * k^2 * |V|$) times. Each visit takes O(1) time to finish. In total, this algorithm takes time O($m * k^2 * |V|$)

to complete.

Thus, $CLASSIFY\_PATH$ is in **P**.

Q.E.D.

Theorem 3: $EXIST_w$ is **NP-C**.

Proof:

To show that $EXIST_w$ is **NP-C**, we need to show that

1. $EXIST_w$ is in **NP**, and
2. $EXIST_w$ can be reduced in polynomial time from a decision problem in **NP-C**.

First, we show that $EXIST_w \in$ **NP**. We can guess a path coverage $PATH$ and verify the *w-runnability* in polynomial time(Figure 5.3). Thus, the problem $EXIST_w$ is in **NP**.

We now polynomially transform the problem of 3-satisfiability($3SAT$) to $EXIST_w$.

First, we define the terms used in this chapter as follows. Let $V = \{v_1, v_2, \ldots, v_k\}$ be a set of boolean *variables*. A *truth assignment* for V is a function f_T:$V \rightarrow$ {true, false}. If v is a variable in V, then v and $\bar{v}$ are *literals* over V. In the rest of this chapter, we use the alphabets x, y, or z to represent literals and the alphabets, such as u or v, to represent variables. The literal x ($x = v$) is $true$ under f_T if and only if the variable v is $true$ under f_T; the literal x ($x = \bar{v}$ is $true$ if and only if the variable v is $false$. A *clause* over V is a set of literals over V. A clause is represented by the disjunction of those literals and is *satisfied* by a truth assignment if and only if at least one of its literals is true under that assignment. A collection C of clauses over V is *satisfiable* if and only if there exists some truth assignment for V that simultaneously satisfies all the clauses in C. Such a truth assignment is called a *satisfying truth assignment* for C. At the bottom of Figure 5.8, a boolean expression and a satisfying truth assignment are illustrated. Based on the definition of the function f_T, truth assignments, such as $v = false$ and $v = true$ together, cannot be a valid truth assignment with respect to a boolean variable v. We call them *conflicting truth assignments* with respect to a boolean variable v under f_T.

The **NP-C** problem 3SAT is formally defined as follows.

PROBLEM: 3-SATISFIABILITY (3SAT)
INSTANCE: A collection of clauses $C = C_1, C_2, \ldots, C_m$ on a finite set V of boolean variables $v_1, v_2, \ldots, v_k$ such that $|C_i| = 3$ for all i, where $1 \leq i \leq m$, and the boolean expression B in Conjunctive Normal Form(CNF), $B = C_1 \wedge C_2 \wedge \ldots C_m$.
QUESTION: Is there a truth assignment that satisfies all the clauses(i.e., is the boolean expression B satisfiable)?

We now show that an instance of **3SAT** can be reduced to an instance of $EXIST_w$ in polynomial time in three steps.

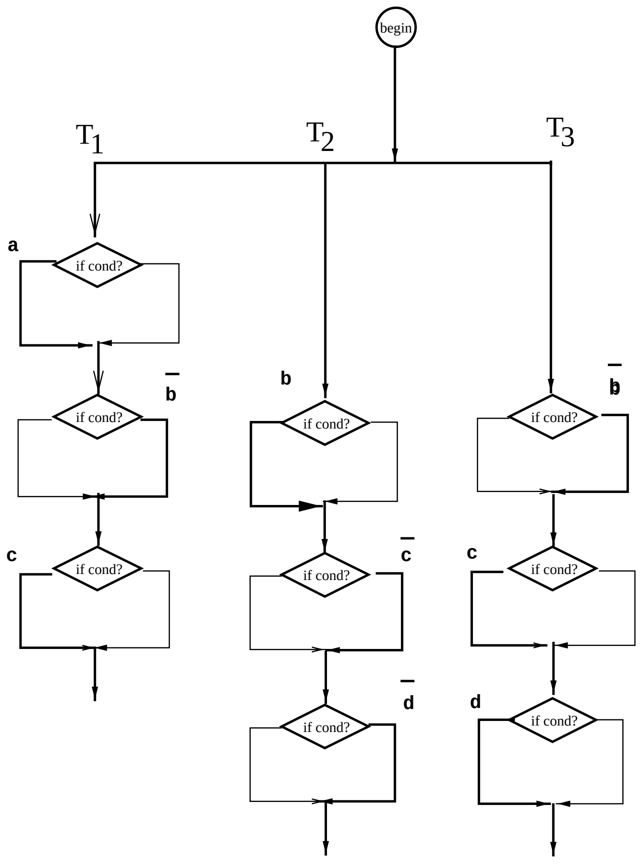
1. Describe a function f for transforming an instance of **3SAT** to an instance of $EXIST_w$.

2. Prove that the transformation can be done in polynomial time.

3. Prove that $x \in$ **3SAT** if and only if $f(x) \in EXIST_w$.

For each boolean expression B with m clauses in Conjunctive Normal Form(CNF), a parallel program $G = (N, E)$ with m threads can be constructed.

For each clause C_i where $1 \leq i \leq m$, a thread T_i can be constructed as follows. For each boolean variable v_{ij} in the clause C_i, where $1 \leq j \leq 3$, one *if-statement* is constructed for which the *then* part represents v_{ij}, whereas the *else* part represents $\bar{v}_{ij}$. The branch of an if-statement corresponding to $v_{ij}(\bar{v}_{ij})$ is called a(an) *then(else)* edge. Formally, the threads T_1, T_2, ..., T_m are created. In each thread T_i, three *if* nodes, represented by $IF^i_j (1 \leq j \leq 3)$ are created. The *then* part and the *else* part



$$B=\underbrace{(a+\bar{b}+c)}_{C_1}\underbrace{(b+\bar{c}+\bar{d})}_{C_2}\underbrace{(\bar{b}+c+d)}_{C_3}$$

Figure 5.4: Three Threads Reduced from Three Clauses

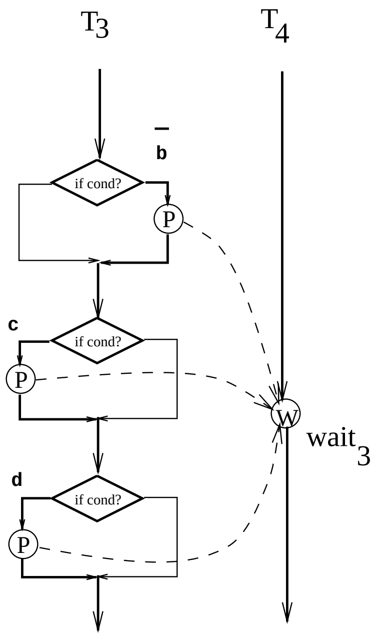


Figure 5.5: Vertical Constraint

of the *if* statement IF_j^i are represented as $THEN_j^i$ and $ELSE_j^i$, respectively. For example, in Figure 5.4, the clause $a+\bar{b}+c$ will be represented by three *if-statements* with two *then* parts representing a and c, and one *else* part representing the literal $\bar{b}$.

Next, we add one *post* node in each thread T_i $(1 \leq i \leq m)$ corresponding to each literal x_{ij} $(1 \leq j \leq 3)$ and m number of *wait* nodes in an additional thread T_{m+1} to the graph. These constraints are referred to as *horizontal* and *vertical constraints*:

- Vertical Constraint: A satisfying truth assignment must satisfy all clauses simultaneously by the definition of a satisfiable truth assignment. Hence, we claim that the truth value of B is *true* (i.e. $f_T(B) = true$) if and only if the final truth value of each clause C_i is *true*.

- Horizontal Constraint: With respect to each variable v_{ij}, we must assign the same truth value in different clauses. For example, we cannot assign *true* to v_{ij}

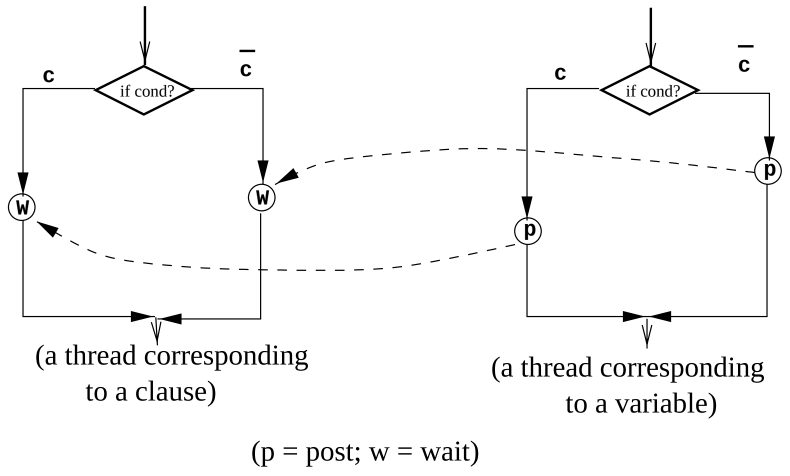

c
if cond?
$\bar{c}$
w
w
(a thread corresponding to a clause)
c
if cond?
$\bar{c}$
p
p
(a thread corresponding to a variable)
(p = post; w = wait)

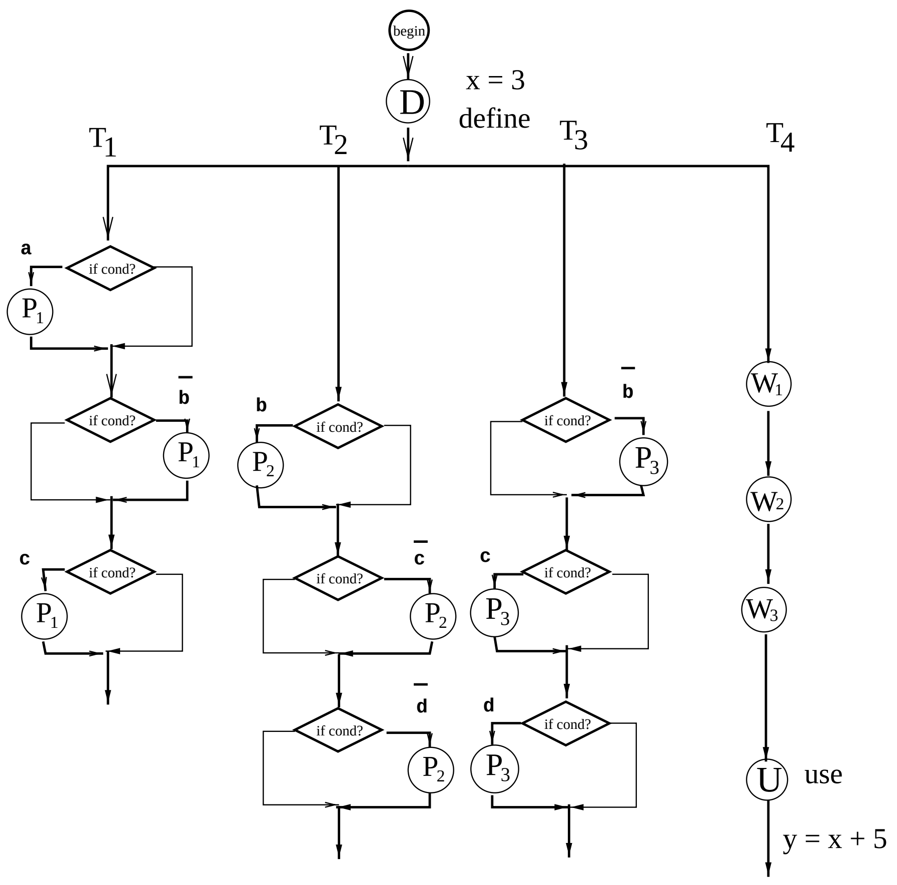
Figure 5.6: Horizontal Constraint

in one clause and $\bar{v_{ij}}$ in another clause. This result is obvious by the definition of the function f_T.

Corresponding to the vertical constraint, we need to construct thread T_{m+1} and some synchronization edges as shown in Figure 5.5. On the branch corresponding to the literal appearing in the clause, a *post* statement is inserted. Also, corresponding to each clause, a *wait* statement is added to the thread T_{m+1}. For example, a *post* operation is added on each of the edges marked as $\bar{b}$, c, and d, in thread T_3; a $wait_3$ is added to the thread T_{m+1}.

Formally, if a literal $x_{ij}(=u_{ij})$ appears in the clause C_i, a *post* node $POST_j^i$ is added to the edge $THEN_j^i$; if a literal $x_{ij}(=\bar{u}_{ij})$ appears in the clause C_i, a *post* node $POST_j^i$ is added to the edge $ELSE_j^i$. In addition, a matching wait node $WAIT_i^{m+1}$ of these *post* nodes is added to the thread T_{m+1}.

Furthermore, we need to add some nodes, some synchronization edges, and several additional threads with respect to the horizontal constraints. For each variable v_p appearing as a literal x_{ij}, e.g., either $x_{ij} = v_p$ or $x_{ij} = \bar{v}_p$, in some clause $C_i(1 \leq i \leq m,\ 1 \leq j \leq 3$, and $1 \leq p \leq k)$, an additional thread T_{m+1+p} is



$$B=(\underbrace{a+\bar{b}+c}_{C_1})(\underbrace{b+\bar{c}+\bar{d}}_{C_2})(\underbrace{\bar{b}+c+d}_{C_3})$$

Figure 5.7: A Parallel Program Reduced from the Boolean Expression B

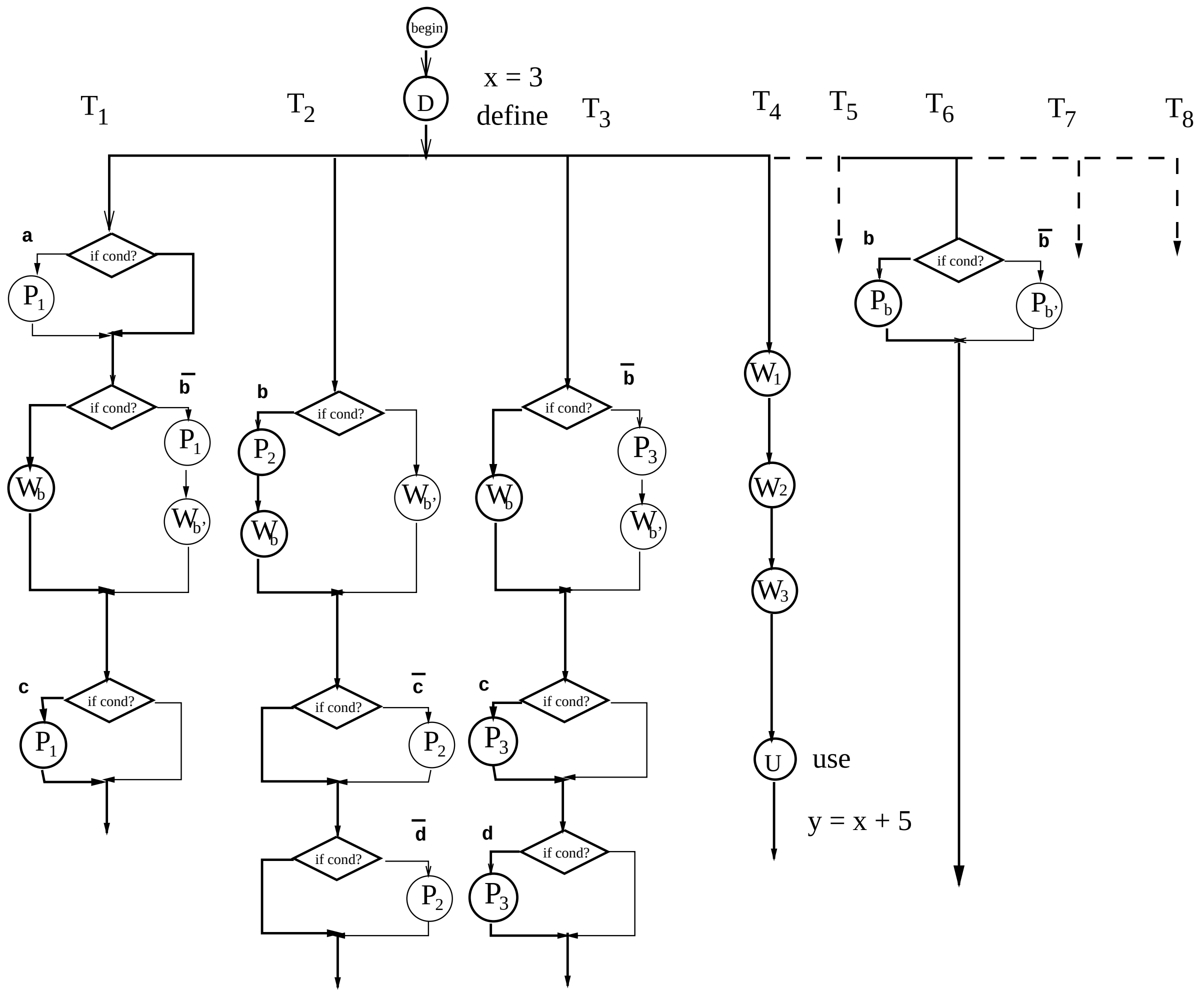
constructed. In thread T_{m+1+p}, an *if* statement is constructed with the *then* part represented by a *then* edge $THEN^{m+1+p}$ and the *else* part represented by an *else* edge $ELSE^{m+1+p}$. A *post* node $POST^{m+1+p}$ is added to the edge $THEN^{m+1+p}$; a matching wait node $WAIT_j^i$ is added to the *then* edge corresponding to each instance of the variable v_p associated with the literal x_{ij}, e.g., either $x_{ij} = v_p$ or $x_{ij} = \bar{v}_p$, in clause C_i. Another *post* node $POST^{m+1+p}$ is added to the edge $ELSE^{m+1+p}$; a matching wait node $WAIT_j^i$ is added to the *else* edge corresponding to each instance of the variable v_p associated with the literal x_{ij} in the clause C_i. As shown in Figure 5.6, an additional thread is added with respect to the variable c, and two *post* and two *wait* statements are added. The number of *wait* nodes is dependent on the number of instances of a particular variable in the boolean expression; for each instance of a variable associated with a literal, two *wait* nodes are added–one incident with the *then* edge and another incident with the *else* edge.

In this proof, we use the *single post multiple waits (MPMW)* scenario. With minor modification, the proof holds for the other scenarios. For example, *single post single wait (SPSW)* requires simply adding to each thread representing a variable one *post* in the *if* edge and one *post* in the *else* edge for each occurrence of a variable in a clause instead of one *post* for a variable in the whole boolean expression.

Finally, we need to construct a *du-pair*. This can be done by adding the *define* node at the beginning of the program and the *use* node at the end of the thread T_{m+1} as shown in and Figure 5.7 and Figure 5.8. In Figure 5.8, some threads are shown partially, and the synchronization calls corresponding to horizontal constraints are shown only for the variable b.

In summary, the reduction from $3SAT$ to $EXIST_w$ includes the following tasks:

1. Create one thread per clause and add one *if*-statements for each variable in each clause. This creates m threads.



V={a, b, c, d}

B=$(a + \bar{b} + c)(b + \bar{c} + \bar{d})(\bar{b} + c + d)$

A satisfying truth assignment: a=0,b=1,c=1,d=1

Figure 5.8: Final PPFG (some threads and synchronizations shown partially)

2. Add an additional thread T_{m+1}.

3. Add three *post* statements in each clause, and m *wait* statements in the thread T_{m+1} accordingly.

4. Add one thread for each variable. This creates k more threads.

5. Add two *post* statements in each of these newly created threads. For each instance of a literal associated with a variable in the threads corresponding to each clause, add two *wait* statements–one incident with the *then* edge and another incident with the *else* edge.

6. Add a *define* node and a *use* node.

It is clear that the above construction can be accomplished in polynomial time.

The next step is to show that there exists a $PATH_w$ in G that covers the du-pair (x, define, use) if and only if there exists a truth assignment for the boolean expression B.

ONLY IF: If there exists a $PATH_w$ in G to cover the du-pair, then there exists a truth assignment for the boolean expression B.

Given a parallel program represented by the graph G with the construct described previously, we need to show that there exists a truth assignment which satisfies the boolean expression B.

First, we can construct the truth assignment for the boolean expression from the $PATH_w$ as follows. If the path in every thread T_i, where $1 \leq i \leq m$, covers the *then* edge of the jth literal, we assign $true$ to the variable u_{ij}; if the path in every thread T_i, where $1 \leq i \leq m$, covers the *else* edge, we assign $false$ to the variable u_{ij}; For example, Figure 5.8 uses dark edges and nodes in threads T_1, T_2, and T_3 to show the truth assignment $a = 0, b = 1, c = 1, d = 1$ derived from the paths illustrated by dark edges and nodes.

Then, we have to show that the truth assignment thus constructed satisfies the boolean expression. We claim that since all threads T_p, where $(m+1+1) \leq p \leq (m+1+k)$, are executed to the end, either all $THEN$ edges in all clauses associated with a variable or all $ELSE$ edges will be executed. For example, in Figure 5.8, all THEN edges corresponding to the variable b is covered. The *wait* nodes marked as W_b will be posted by the *post* node P_b in thread T_6. Since either all $THEN$ edges associated with a variable or all $ELSE$ edges will be executed, conflicting truth assignments won't occur.

Since the path coverage is a $PATH_w$, all threads will be executed to the end without causing an infinite wait. If thread T_{m+1} is executed to the end, at least one of the three matching *post* nodes of the *wait* node $WAIT_i^{m+1}$ must be covered. Hence, one of the three literals in the clause C_i will be assigned the value *true*. Hence, the truth assignment is a valid truth assignment which satisfies the boolean expression B.

IF: If there is a truth assignment for the boolean expression B, there exists a $PATH_w$ that covers the du-pair (x, define, use).

For the graph G constructed using the reduction described previously, we need (1) to show how the path coverage PATH is derived, and (2) to prove that PATH is a $PATH_w$.

A path coverage must cover three sets of threads: (1) threads $T_i (1 \leq i \leq m)$, (2) thread T_{m+1}, and (3) threads $T_p (1 \leq p \leq k)$.

First, we construct the paths in threads T_i $(1 \leq i \leq m)$. For an *if* node IF_j^i in thread T_i, if the truth value of a variable u_p is *true*, the *then* edge is included in the path coverage; otherwise, the *else* edge is included in the path coverage.

Second, the thread T_{m+1} has only one path which covers the *use* node. We use this path to cover every node in thread T_{m+1}.

Third, for the threads corresponding to each variable $T_{m+1+p}(1 \leq p \leq k)$, the branch selection associated with an *if* statement is determined by the truth value of the variable u_p. If u_p is *true*, the *then* part is included in the path coverage; otherwise, the *else* part is included in the path coverage.

To show that the path coverage PATH is a $PATH_w$, we must show that PATH satisfies the definition defined in Section 5.2.2. That is, (1) for each instance j of a *wait* w^i, $w^i_j \in_p PATH$, $\exists$ an instance of a *post*, $p^r_s \in_p PATH$, where $p^r_s \in MP(w^i_j)$; and (2) the execution of the path $PATH$ will not cause a circular wait.

It is obvious that the execution of all paths in PATH will not cause a circular wait because (1) threads $T_p(1 \leq p \leq k)$ and thread T_{m+1} are not interconnected via any synchronization edge, and (2) synchronization edges between any thread T_i $(1 \leq i \leq m)$ and thread T_{m+1} does not cause a circular wait.

We only need to show that for each instance j of a *wait* w^i, $w^i_j \in_p PATH$, $\exists$ an instance of a *post*, $p^r_s \in_p PATH$. For those threads containing no *wait* nodes, we consider them to be finished normally at run-time. For example, all threads $T_{m+1+p}(1 \leq p \leq k)$ corresponding to boolean variables u_p will be executed to the end without being halted because there is only one *if* statement and one *post* node which is incident with a branch of the *if* statement.

First, we check threads $T_i(1 \leq i \leq m)$. Since we know that the truth assignment is a valid truth assignment, either all *then* edges associated with a variable or all *else* edges associated with a variable in all clauses will be covered. Thus, the *wait* nodes $WAIT^i_j$ incident with either the *then* edges or the *else* edges associated with the jth literal x_{ij} in each thread T_i will be posted by the *post* statement in thread T_{m+1+p}.

Then, we check thread T_{m+1}. Thread T_{m+1} can only be executed to the end if all *wait* nodes $WAIT^{m+1}_i$ are posted. As part of the reduction, a *post* is added to the *then* edge if the literal x_{ij} is $u_p(1 \leq i \leq m,\ 1 \leq j \leq 3,$ and $1 \leq p \leq k)$;

a *post* is added to the *else* edge if the literal x_{ij} is $\bar{u}_p$. Since at least one literal in each clause is assigned $true$, at least one of the three matching *post* nodes of the *wait* node $WAIT_i^{m+1}$ will be covered by the path coverage generated by using the satisfiable truth assignment.

Thus, if there exists a valid truth assignment for B, there exists a $PATH_w$ that covers the du-pair (x, define, use).

Therefore, we have shown that $EXIST_w$ is in **NP** and $3SAT$ can be reduced to $EXIST_w$ in polynomial time. This implies that $EXIST_w$ is **NP-C**.
Q.E.D.

In conclusion, one should only expect to find $PATH_a$ in polynomial time since determining whether or not a $PATH_w$ exists is **NP-C**. Hence, our research focuses on finding $PATH_a$ path coverages for a given du-pair.

5.4 State of the Art in Path Testing Methodologies

5.4.1 Path Testing of Sequential Programs

In general, the path coverage algorithms for sequential programs fall into the following categories: (1) detecting infeasible paths or reducing the effect of infeasible paths, (2) finding a path coverage based on a specific test data selection criterion, or (3) finding a set of paths based on a path coverage criterion such that the number of paths in the path set is minimal.

This section focuses on (2) and (3) as they are most closely related to the du-path finding problem.

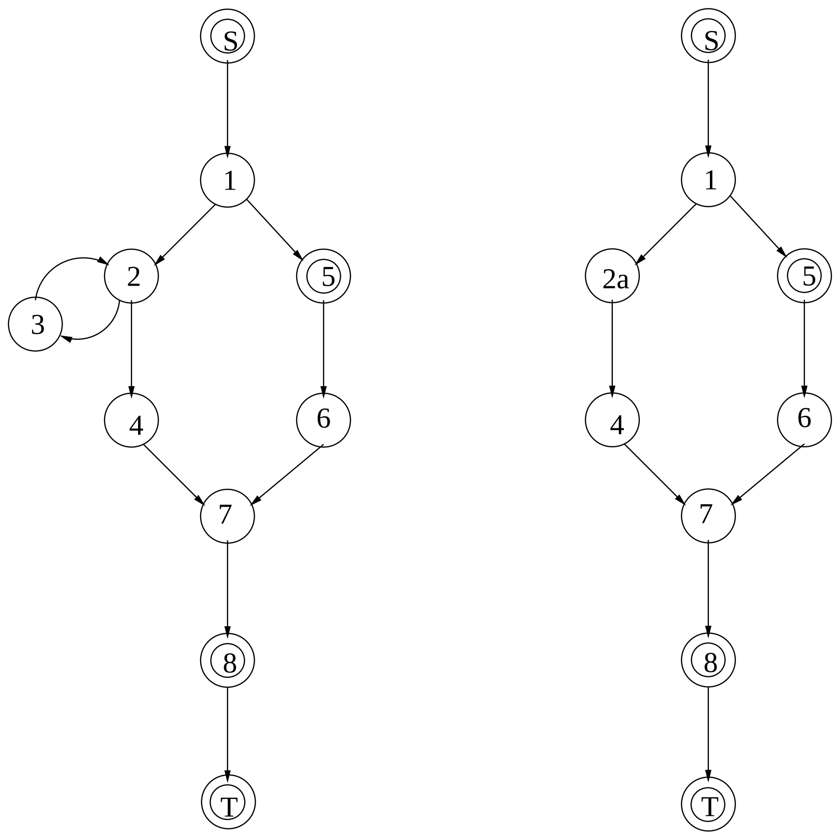
Ntafos and Hakimi[96] showed that the minimum path finding problem can be solved by one of two methods: (1) transforming the problem into a minimum flow problem, and (2) transforming the problem into a maximum matching problem. They also showed that finding a minimum path cover for a set of required pairs, i.e., a pair of vertices that should appear together in at least one test path, is NP-hard.

Their result implies that one should not expect to obtain a solution in polynomial time for finding a minimum set of path coverages for a given set of du-pairs.

Clarke [20] developed a system to generate test data and symbolically execute FORTRAN programs. This system has four functions: (1) generating test data to drive execution down a given program path, (2) detecting nonexecutable program paths, (3) creating symbolic representations of the program's output variables as functions of the program's input variables, and (4) detecting program errors that can be detected by executing a given set of paths, e.g., array index out of bounds. The system can generate symbolic representations of an input variable involved in a path assuming that the predicates involved in the *if*-statements within the given path can be represented by linear inequalities. After these inequalities are derived, an inequality solver is invoked to solve these inequalities. If no solution can be found, this path is an infeasible path. Otherwise, a test suite can be produced to force this path to be executed. The result of Clark's research provides a possible methodology to automate the process of test data generation such that the provided paths will be covered.

Since the du-path finding algorithm developed in this dissertation builds on the next two techniques, these techniques are described in detail here. Gabow, Maheshwari, and Osterweil [40] studied two path finding problems: (1) identifying a path that passes through a specific set of program statements in a sequential program, and (2) determining whether infeasible paths exist in a sequential program. They showed that these problems are graph theoretic in nature, and that they can be efficiently solved by applying existing techniques and results concerning graphs.

Gabow et al. called the set of nodes which are required to be included in the path from the `begin` (the node s) to the `end` (the node t), *constrained nodes*, represented by V^*. A path that covers them, if one exists, is called a *multiple node constrained path*. For an arbitrary control flow graph G of a sequential program, they



(a) An Arbitrary Flow Graph (b) An Acyclic Flow Graph

Figure 5.9: Reduction of an Arbitrary Graph to an Acyclic Graph

```
      procedure CONSTRAINTPATH;
           begin
               SET1 <- {s};
               SET2 <- {};
TRAVERSE:      while SET1 is nonempty and |SET2| is less than or equal to 1 do
                begin
                    delete a node x from SET1;
                    for each node w adjacent to x do
                     begin
                        INDEG(w) <- INDEG(w) - 1;
                        if INDEG(w) = 0 then
                          begin
                            PARENT(w) <- x;
                            if w is a constrained node then
                              add w to SET2
                            else
                              add w to SET1;
                          end
                     end
                end;

              if |SET2| > 1 then output "no path exists"
              else
                if t is not in SET2 then
                  begin
                    delete the node from SET2 and add it to SET1;
                    goto TRAVERSE:
                  end
                else
                   output PARENT;
          end;
```

Figure 5.10: Gabow et al's Algorithm for Existence of a Multiple Node Constrained Path

reduce G to G_r, an acyclic graph by replacing the set of vertices in each strongly-connected component in G with a single node. Figure 5.9(a) is an arbitrary flow graph G and Figure 5.9(b) is the reduced acyclic graph of G. The loop (containing the nodes 2 and 3) is represented by a single node $2a$. The problem of determining the existence of a multiple node constrained path P in an arbitrary graph can be reduced to the problem of determining the existence of P in an acyclic graph G_r. They developed a polynomial time algorithm (Figure 5.10) to find all the nodes(in G_r) that must be included in the final multiple node constrained path P. The parent node of each node in the path P is also determined in the process of finding all the required nodes in P.

Figure 5.9 is used to illustrate finding a multiple node constrained path from S to T to cover the nodes 5 and 8 using the algorithm shown in Figure 5.10. For example, in Figure 5.9, the doubly circled nodes (except the S and T node) are constrained nodes. The algorithm in Figure 5.10 starts by initializing the set $SET1$ to $\{S\}$ and the set $SET2$ to an empty set. This algorithm uses $SET1$ and $SET2$ to store the sources of the current subdigraph which are nonconstraining and constraining nodes, respectively. As the algorithm is executed, the parent of 1 is set to s, the parent of $2a$ is set to 1, the parent of 4 is set to $2a$, the parent of 5 is set to 1. At this time, $SET1$ becomes empty and $SET2$ contains 5. The `if`-statement immediately after the `while` loop is executed. The node in $SET2$, i.e., node 5, is deleted from $SET2$ and stored into $SET1$. The execution of the `while` is resumed. The parent of 6 is set to 5, the parent of 7 is set to 6, etc. At the end of execution, the existence of the multiple node constrained path in G can be determined. After the existence of the path is determined, a representation of the multiple node constrained path in the original program G can be constructed by replacing each node n in the representation of the path in G_r, by an arbitrary permutation of the nodes in the corresponding strongly-connected component of n. To obtain the actual path in

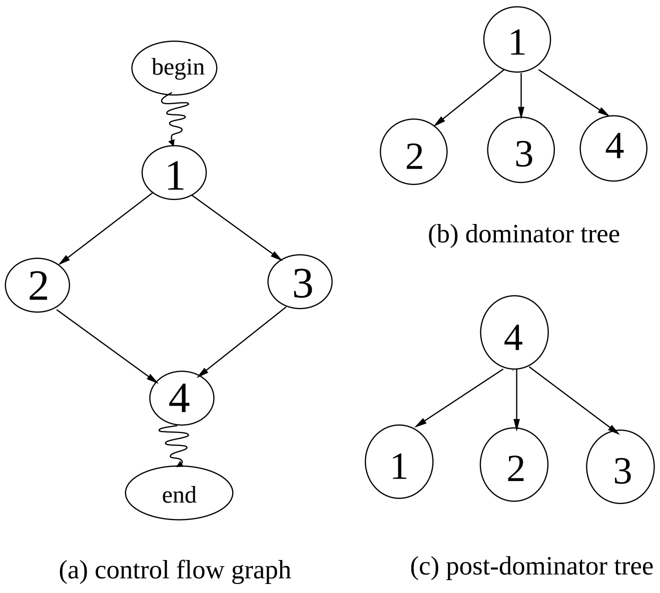


Figure 5.11: An Example of Dominator Trees and Implied Trees

G, they use a depth-first search to build subpaths between each pair of consecutive nodes in the representation [99]. Thus, in this dissertation, this approach is called the DFS approach.

Bertolino and Marrè [9] developed an algorithm, referred to as the DT-IT approach in this dissertation, that uses dominator trees (DT) and implied trees (IT) (i.e., post-dominator trees) to find an all-branch coverage for a sequential program. A *dominator* of a node n in a control flow graph(CFG) representation of a program is a node that appears in every path in the CFG from the *begin* node to n. A *post-dominator* of node n is a node that appears in every path from node n to the *end* node. In Figure 5.11(a), node 1 dominates nodes 2, 3, and 4, but nodes 2 and 3 do not dominate node 4 because there exists a path from *begin* to 4 that does not pass through 2 and similarly for node 3. Node 4 post-dominates nodes 1, 2, and 3, but, nodes 2 and 3 do not post-dominate node 1.

A dominator tree (DT) for a program is a tree representation of the domination relations. An ancestor of a node n in the DT is a dominator of the node n. The *immediate* dominator d of n is the parent node of n in a DT. A path from the root of a DT to a node in the DT is called a *tree path.* The definition of implied-tree(IT) is analogous to that of DT for the post-dominator relation.

The intuition behind the DT-IT approach to all-branch coverage is simple. *Unconstrained edges* are CFG edges e_u such that e_u dominates no other CFG edges and is not post-dominated by any other CFG edge. The unconstrained edges of a program CFG can be computed by finding the intersection S of the set of leaf nodes in DT and the set of leaf nodes in IT. Unconstrained nodes are defined similarly to unconstrained edges except that CFG nodes are used instead of CFG edges. In general, the members in the set S are called unconstrained elements. In Figure 5.11, the unconstrained elements are nodes 2 and 3. Bertolino and Marrè showed that if one path is found to cover each unconstrained element, the set of paths is an all-statement path coverage. For example, in Figure 5.11, if a path is found to cover node 2 and a path is found to cover node 3, then this set of paths is an all-statement coverage for the program in Figure 5.11(a).

Before the detailed algorithm for finding an all-branch coverage can be explained, some notation must be introduced. The head and the tail of a directed edge e are represented by $H(e)$ and $T(e)$, respectively. If there is a path in the CFG leading from $H(e_1)$ to $T(e_2)$, it is said that the edge e_1 can *reach* the edge e_2 or e_2 is reachable from e_1. A program is represented as a *decision-to-decision graph*, or *ddgraph.* A ddgraph is a digraph $G = (N, E)$ with two *distinguished edges* e_0 and e_k, i.e., the unique *entry edge* e_0 and the *unique exit* edge e_k, such that e_0 can reach any edge in E, e_k is reachable from any edge in E, and for every node $n \in V$ where $n \neq T(e_0)$ and $n \neq H(e_k)$, the following are true: (1) (indegree(n) + outdegree(n)) > 2, (2) indegree($T(e_0)$) $= 0$ and outdegree($T(e_0)$) $= 1$, and (3)

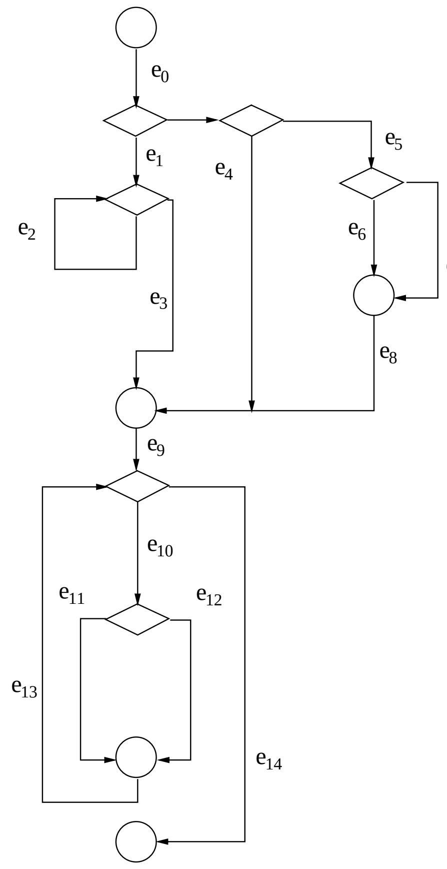
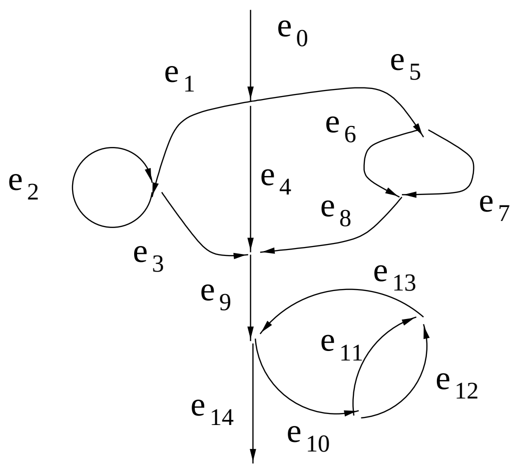
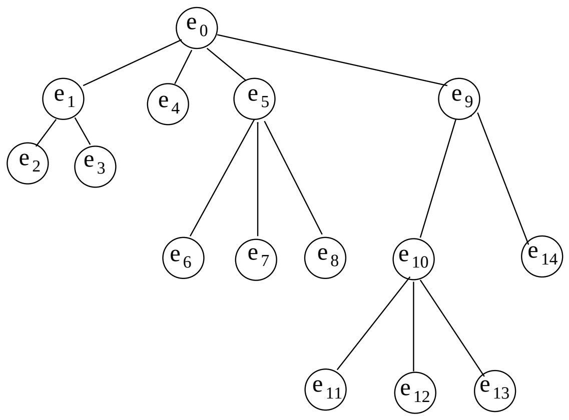


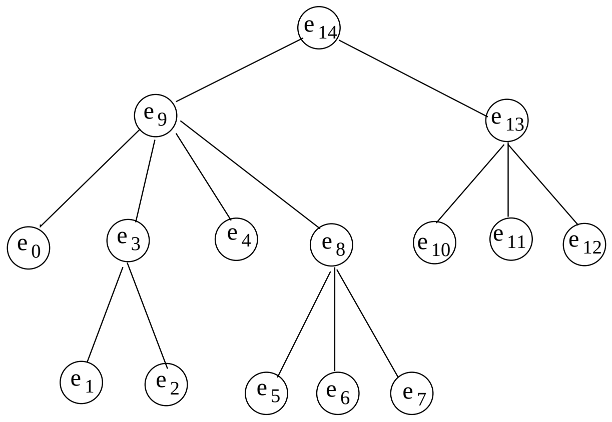
Figure 5.12: The CFG G of a Sequential Program P



(a) A decision-to-desion graph(ddgraph) of program P



(b) DT(P)



(c) IT(P)

Figure 5.13: A ddgraph, dominator, postdominator trees of the program P

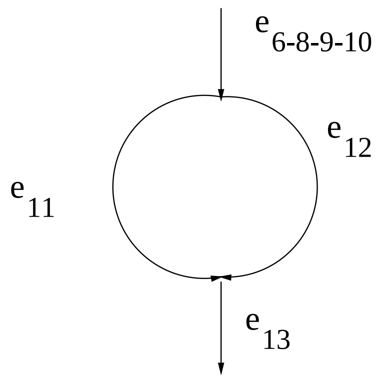


Figure 5.14: A Sub-ddgraph (G, e_6, e_{13})

indegree($H(e_k)$) $= 1$ and outdegree($H(e_k)$) $= 0$. A ddgraph is essentially a CFG without showing the vertices in the CFG. Multiple consecutive *if*-statements in a CFG are combined into one node in a ddgraph to represent a *switch*-statement. An example ddgraph, and the corresponding DT and IT of the program in Figure 5.12 are given in Figure 5.13(a), 5.13(b), and 5.13(c), respectively.

The actual procedure for finding a path to cover a given CFG edge e is called *FIND-A-PATH()*. First, the DT and IT of a program represented by a CFG G, i.e., DT(G) and IT(G), are generated, respectively. Thus, a node in the DT(G) and IT(G) in Figure 5.13(b) represents an edge in the ddgraph shown in Figure 5.13(a). For the given node representing the CFG edge e in DT(G) and IT(G), there exists a tree path (represented by $PATHDT(e)$) in DT(G) from the root of DT(G)(which represents the CFG edge e_0) to the node representing the CFG edge e in $DT(G)$, and there exists another tree path (represented by $PATHIT(e)$) in IT(G) from the node representing e to the root in IT(G)(which represents the CFG edge e_k). The two tree paths can be concatenated together "back-to-back" without repeating the edge e twice to form the temporary working path for covering the edge e.

The process of finding a final path requires a check to determine whether the working path is actually a genuine path in G. For example, the simplest case for finding a path is shown by trying to cover e_6 in Figure 5.13(a). In the DT and IT, the tree paths $e_0 - e_5 - e_6$ and $e_{14} - e_9 - e_8 - e_6$ are found, respectively. By concatenating these two partial paths without repeating e_6, a complete path covering e_6 is generated. To cover e_{13}, the sub-paths found in the DT and IT are $e_0 - e_9 - e_{10} - e_{13}$ and $e_{14} - e_{13}$, respectively. After the concatenation is done, the temporary working path $e_0 - e_9 - e_{10} - e_{13} - e_{14}$ is generated. There are two instances of discontinuity in the temporary working path: (1) between e_0 and e_9, and (2) between e_{10} and e_{13}.

If the immediate predecessor e_p of e_i in the DT or IT is not the immediate predecessor of e_i in the CFG, then the procedure *FIND-A-PATH()* is recursively called to find a path that connects e_p and e_i. To achieve this, an algorithm is used to reduce the original graph to a reduced ddgraph, or *sub-ddgraph.* Conceptually, a sub-ddgraph from an edge e_i to e_j is another ddgraph constructed according to the following two rules: If there is only one successor edge e_s for an edge e_t, a new edge e_{s-t} is constructed by merging e_s and e_t; if the $H(e)$ of the edge e in the CFG G is an *if* statement, it is kept in the sub-ddgraph. The sub-ddgraph G_n generated from two distinguished edges e_i and e_j is represented by $G_n = (G, e_i, e_j)$. The recursive call for connecting two disconnected edges e_i and e_j in G is achieved using the sub-ddgraph (G, e_i, e_j). As an example, Figure 5.14 shows the sub-ddgraph (G, e_6, e_{13}), where G is the CFG in Figure 5.12. In the example of covering e_{13}, *FIND-A-PATH()* will be called for each instance of discontinuity, between e_0 and e_9 and between e_{10} and e_{13}.

The heuristic algorithm for finding an all-branch path coverage consists of calling the procedure *FIND-A-PATH()* repeatedly until all unconstrained edges are covered. For flexibility, they allow different criteria for the selection of next unconstrained edge. Within the procedure *FIND-A-PATH()*, another procedure *SELECT-AN-ARC()* is called. This procedure call allows a user to decide which criterion to use for selecting the next unconstrained edge. They compare two strategies. One selects the "so far uncovered" unconstrained edge, and the other selects the next unconstrained edge such that the number of decision nodes in each path is minimized. The first strategy derives a near minimum number of paths, whereas the second strategy is less likely to find infeasible paths since it is experimentally shown that the more decision nodes included in a path, the more likely the path is infeasible [156].

In the example of covering e_{13}, prior to calling the routines, $G_1 = (G, e_0, e_9)$ and $G_2 = (G, e_{10}, e_{13})$ are generated. The $DT(G_1)$, $IT(G_1)$, $DT(G_2)$, $IT(G_2)$ are generated. To find a path to connect e_0 and e_9, there are three choices, and a different solution is found depending on the policy that a user chooses. If a user chooses to find the *minimum number of if-nodes* in all the paths from e_0 to e_9 to cover a given unconstrained edge e, the total number of *if-nodes* in $PATHDT(e)$ and $PATHIT(e)$ will be computed. This will provide an estimation, but not the exact number, of *if-nodes* in the path covering e. A user can also choose to find the "so far uncovered" unconstrained edge for finding a path coverage that is near the minimum path coverage. The recursion continues until all unconstrained edges are covered.

5.4.2 Path Analysis of Concurrent Programs

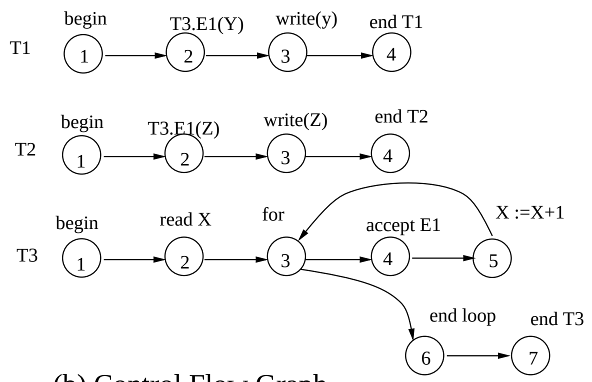
Yang and Chung [155] developed a path analysis model for concurrent program testing. They defined two kinds of paths: C-paths and C-routes. A C-path is defined on the control flow graph of a concurrent program, while a C-route is defined on the rendezvous graph of the program. For each task, a rendezvous graph contains nodes that represent statements and edges that represent transfer of execution. Only certain statements are included in a rendezvous graph: the *begin* statement, the *end* statement, *loop* statements, *if* statements, synchronization calls, and *accept*

```
   task body T1 is
     Y: INTEGER;
1. begin
2.   T3.E1(Y);
3.   write (Y);
4. end T1;
```

```
   task body T2 is
     Z:INTEGER;
1. begin
2.   T3.E1(Z);
3.   write(Z);
4. end T2;
```

```
   task body T3 is
     X:INTEGER;
1. begin
2.   read X;
3.   for i in 1..2 loop
4.     accept E1(T:out INTEGER) do
5.       X := X + 1;
       end E1;
6.   end loop
7. end T3;
```

(a) An ADA Program P



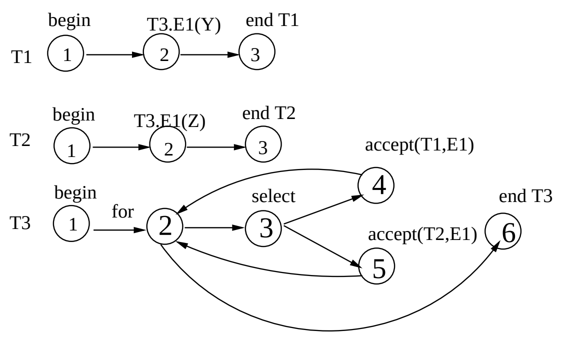
A set of C-Paths:

T1: 1-2-3-4

T2: 1-2-3-4

T3: 1-2-3-4-5-3-6-7

(b) Control Flow Graph



A set of C-routes:

T1: 1-2-3

T2: 1-2-3

T3: 1-2-3-4-2-3-5-2-6

(c) Rendezvous Graph

Figure 5.15: Illustrating C-paths and C-routes in concurrent programs

statements. Each *accept* statement and its body are replicated once for each *request* call related to the *accept* statement. A *select* node is added to be the common predecessor of all the replicated copies of the *accept* node. The edges represent the possible execution paths in each task.

C-paths and C-routes are used to model the static behavior and the execution behavior of a concurrent program, respectively. In Figure 5.15(b) and (c), a control flow graph, a rendezvous graph, a set of C-paths, and a set of C-routes are shown for the program shown in Figure 5.15(a).

Yang and Chung also described test execution procedures and a reliability analysis of the path testing model based on the control and rendezvous graphs. They claim that they are currently in the process of implementing a testing environment to support test activities such as C-path generation in accordance with coverage criteria such as branch coverage and rendezvous coverage. But, they did not present the actual algorithm for generating C-paths and C-routes. In one of their earlier papers [154], they mentioned that the generation of C-paths was under development and they presented the description of a heuristic approach to generate C-routes. For a given C-path, they suggest finding all syntactically possible combinations of synchronization events in the C-path to produce all possible C-routes. Infeasible C-routes are eliminated. The actual algorithms for generating C-routes and eliminating infeasible C-routes were not presented; only an example of eliminating infeasible C-routes was presented after generating all possible C-routes from a given C-path.

5.5 Applying Path Finding Algorithms to Parallel Programs

All of the existing path testing methodologies for finding actual paths focus on programs without parallel programming features and, therefore, cannot be applied directly to finding a du-path coverage for parallel programs. The next section presents a hybrid approach that uses DFS and DT-IT together with an extension to provide a du-path coverage for parallel programs. In this section, the weakness of each approach, when used in isolation to find a du-path coverage for a given parallel program, is described. For simplicity of illustrating the algorithm, the subscripts associated with *post* or *wait* nodes in the figures in this chapter are only used for distinguishing these nodes. They are not the unique node id described in Chapter 4.

When applying the DFS approach [40] to parallel programs in isolation, it is not appropriate even for finding $PATH_a$, not to mention $PATH_w$. The reason is that although DFS can be applied to find a set of paths for covering a du-pair, this approach does not cope well with providing coverage for any intervening *wait*'s, and the corresponding coverage of their matching *post*'s as required to find $PATH_a$. For example, consider a situation where there are more *wait* nodes to be included while completing the partial path for covering the *use* node. Since the first path is completed and a matching *post* is not included in the original path, the first path must be modified to include the *post*. This is not a straightforward task, and becomes a downfall of using DFS in isolation for providing a du-path coverage for parallel programs.

If DT-IT is applied to finding a du-path coverage for parallel programs, the goal is first changed to finding a path coverage for a du-pair instead of an edge, which is a minor modification. However, this approach will also run into the same problem as in DFS. That is, if some *post* or *wait* is reached when we are completing a path, we need to adjust the path just found to include the matching *posts*. In addition, we will run into another problem regarding the order in which the *define* and *use* nodes are covered in the final path. For instance, in Figure 5.2, an incorrect path coverage will be generated using the DT-IT approach alone. The final path will have $define \not\prec use$. Thus, using this method alone cannot guarantee that we find a $PATH_a$.

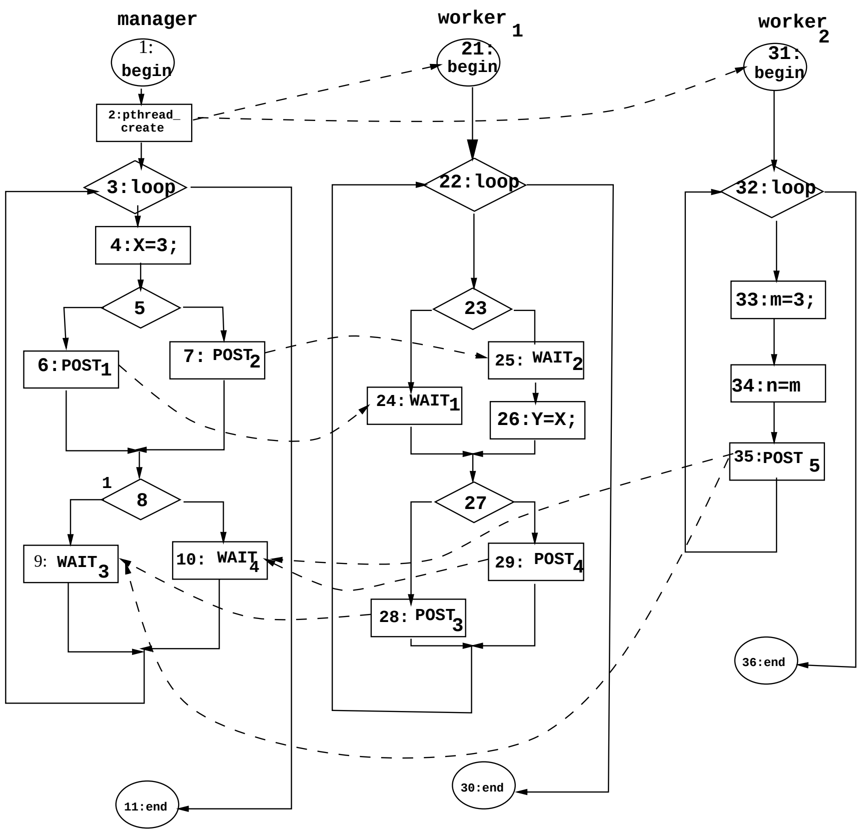


Figure 5.16: Example of the path finding algorithm

5.6 A Hybrid Approach

In this section, an extended "hybrid" approach is presented to find a path coverage for a particular du-pair in a parallel program.

There are actually two disjoint sets of nodes in a path used to cover a du-pair in a parallel program: *required* nodes and *optional* nodes. The set of *required* nodes includes the *pthread_create()* calls as well as the *define* node and *use* node to be covered, and the associated *post* and *wait* with which the partial order $define \prec use$ is guaranteed. All other nodes on the path are *optional* nodes for which partial orders among them are not set by the requirements for a $PATH_a$. However, if a *wait* is covered by the path, a matching *post* must be covered. For instance, in Figure 5.16, the nodes 2, 4, 7, 25, and 26 are required nodes for finding the du-pair $(x, 4, 26)$, whereas all other synchronization nodes are optional. Among the required nodes, the partial orders are uniquely identified, whereas the partial orders among the optional nodes are not. For example, it is acceptable to include either $post_3$ or $post_4$ first in a path coverage. The node $wait_1$ can be covered later than $post_4$ in a $PATH_a$.

The hybrid algorithm consists of two phases. During the first phase, called the *annotate phase*, the depth-first search (DFS) approach is employed to cover the required nodes in the PPFG. Then, a modified DT-IT approach, which will be explained later, is used to cover the optional nodes. As a path to cover a node is found, nodes in the path are annotated with a *traversal control number* (TRN). In the second phase, called the *path generation* phase, the actual path coverage is generated using the traversal control annotations. The data structures utilized in the du-pair path finding algorithm are described first, and then the details of the algorithm are presented.

During the first phase, a modified DT-IT approach is used to cover the optional nodes. The modified DT-IT is different from the original DT-IT in two ways. First, unlike the original DT-IT, which is designed for finding all-branch coverage, the modified DT-IT is designed to find a du-path coverage. Thus, the DT and IT are associated with nodes instead of edges in the PPFG. Second, the criterion for connecting two nodes in a DT or IT is modified. When applying the original DT-IT to find an all-branch coverage, if the immediate predecessor e_p of e_i in the DT or IT is not the immediate predecessor of e_i in the CFG, then the procedure *FIND-A-PATH()* is recursively called to find a path for connecting e_p and e_i. Within the procedure *FIND-A-PATH()*, a procedure, i.e., *SELECT-AN-ARC()* lets a user decide which criterion to use for selecting the next edge. When applying our modified DT-IT algorithm to find a path coverage for a du-pair, the total number of *post* or *wait* nodes in all partial paths between n_p and n_i is recursively computed. The partial path with the highest number of *post* and *wait* nodes is selected to connect n_p and n_i. This criterion generates a path coverage which covers the highest possible number of synchronization edges. It is expected that a path coverage could have a higher possibility of detecting synchronization errors if the path coverage covers more synchronization edges.

5.6.1 Data Structures

For a given parallel program represented by a PPFG, the main data structures used in the hybrid algorithm are described for the thread level and the node level, respectively.

1. At thread level:

    - one working queue per thread is used to store the *post* nodes that are required in the final path coverage,
    - one path queue per thread is used to store the resulting path.

2. At node level:

```
Algorithm  annotate_the_graph()
Input:    A DU-pair, and a PPFG
Output:   Annotated PPFG
Method:
  1. Initialize TRN's, decision queues, and working queues;
  2. Find a path to cover pthread_create and define nodes using dfs;
     Find a path to connect the define to a post;
     From the matching wait node, search for the use node using dfs.
  3. Complete the two sub-paths using modified DT-IT.
  4. For each node in the complete paths:
     Increment TRN by one;
     If node is a WAIT,
       Add matching post nodes into appropriate working queues,
     If node is an if-node,
       Add the successor node in the path into decision queue;
  5. /* process the synchronization nodes  */
     while ( any working queue not empty )
     {  For each thread, if working queue not empty
        {  Remove one node from the working queue;
           if the node's TRN is zero
           {
              Find a path to cover this node
              For each node in the complete path:
                  Increment TRN by one;
              If node is a WAIT,
                  Add matching post nodes into appropriate working queues,
              If node is an if-node,
                  Add the successor node in the path into decision queue;
           }
        }
     }
```

Figure 5.17: Phase 1: Annotate the PPFG.

```
Algorithm    traverse_the_graph()

Input:       An annotated PPFG
Output:      A DU-path
Method:
For all threads
{
 1.   current = begin node of the thread;
 2.   while ( current node's TRN > 0 and
              current is not the end node )
      {
 3.       add the current node to the result DU-path;
 4.       decrement TRN of current node by one;
 5.           if ( current is an if-node )
                  current = first node from decision queue;
                  delete the first node in the queue;
              else
 6.           if ( current is a loop node )
                  current = successor with smallest non-zero RPO;
              else
                      current = successor node of current;
      }
 }
```

Figure 5.18: Phase 2: Generate the du-path coverage

- a traversal control number(TRN) is used to decide which node must be included in the final path coverage and how many iterations are required for a path through a loop,
- a reverse post-order number (RPO) is used for selecting a path at loop nodes,
- a decision queue per *if-node* is used for determining which branch to take.

5.6.2 The du-path Finding Algorithm

In this section, the du-path finding algorithm is described with respect to finding du-pairs in which the define and use are located in different threads. The handling of du-pairs with the define and use in the same thread is a simplification of this algorithm. Figure 5.17 contains the *annotate_the_graph()* algorithm, which accomplishes the annotate phase. The *traverse_the_graph()* algorithm, shown in Figure 5.18, traverses the PPFG and generates the final du-path coverage. The steps of these algorithms are described in more detail here. As a simple example, the steps to find a path coverage for the du-pair with the define of x at node 4 and the use of the variable x at node 26 in Figure 5.16 is shown.

Phase 1: Annotating the PPFG.

Step 1. Initialize the working and decision queues to empty, and set the traversal control number(TRN) of each node to zero. The reverse post-order number(RPO) of each node in the PPFG is precomputed.

Step 2. Use DFS to find a path from the *pthread_create* of the thread containing the *define* node to the *define* node, and then from the *define* node to the *use* node. When a *post* node is encountered in the path, a matching *wait* that is located in the same thread as the *use* node is selected as the next node to be traversed, and the search for the *use* node continues. Upon returning from each DFS() call after a *wait* is traversed, the traversal returns to the matching *post* before continuing the search for the *use* node if not yet found.

After this step is completed, the required nodes, including the $begin$, $define$, $post_2$, $wait_2$, and the *use* nodes, are included in the partial path 1-2-3-4-5-7-25-26, which is actually the concatenation of two partial paths 1-2-3-4-5-7 and 25-26.

Step 3. Apply modified DT-IT to complete the partial paths found in Step 2. Since a partial path is not an executable path, we must "complete" the partial paths such that the final paths become executable. For example, to complete the partial paths 1-2-3-4-5-7(*begin-define-*$post_2$) and 25-26($wait_2$-*use*), respectively, we must find partial paths to connect the $post_2$ node(node 7) with the `end` node, the `begin` node to the $wait_2$ node(node 25), and the *use* node to the *end* node, respectively. To complete the partial path in the thread containing the *define* node, we use the dominator tree of the *define* node and the post-dominator tree of the *post* node that occurs after the *define* node in the partial path just found. Similarly, to complete the partial path in the thread containing the *use* node, we use the dominator tree of the matching *wait* of the *post* node associated with the *define* node and the post-dominator tree of the *use* node.

In the simple example, after the third step is completed, the two partial paths are completed, using the DT-IT approach. These two paths are 1-2-3-4-5-7-8-9-3-11 for *manager* and 21-22-23-25-26-27-28-22-30 for $worker_1$. The node 9 instead of 10 has been included in the path only due to the node returned from the routine $SELECT - AN - ARC()$. Either 9 or 10 could have been returned. As mentioned previously, the path with the maximum number of synchronization calls is selected. If this does not result in a unique path, then the partial path containing the node with the smaller node id is chosen.

Step 4. For each node along the paths covering the *define* and the *use* nodes, respectively, (1) increment the node's TRN by one to indicate that the node should be traversed at least once. (2) If the node is a *wait*, add a matching *post* into the working queue of the thread where the *post* is located. (3) If the node is an *if-node*, add the reverse post-order Number(RPO) of the successor node within the path into the *if-node*'s decision queue to ensure that the branch selected in phase 2 will be included in the right sequence. Hence, the selection of a branch at an *if* node must be determined on a first-in-first-out basis to ensure the coverage of required nodes. Decision *queues* are used for storing the RPO of the child node at an *if* node.

After this step, the TRNs of all nodes along the paths for covering required nodes have been incremented. In the simple example, the TRN for every node along the two paths equals 1 after step 4 except the loop node 22 for which the TRN is 2. When node 9 was reached during this traversal, nodes 28 and 35 were put into the working queues for $worker_1$ and $worker_2$, respectively.

Step 5. While any thread's working queue is not empty, remove one *post* node from a thread's working queue, and apply the modified DT-IT algorithm to find a path to cover the node. Increment the TRN of the nodes in that path. In this way, the TRN identifies the number of instances of each node to be covered. This is particularly important in finding a path coverage for nodes inside loops, where it might be necessary to traverse some loop body nodes several times to ensure that branches inside the loop are covered appropriately. Process *wait* and *if*-nodes in this path as in Step 4.

After this step is completed, the TRN's of all necessary nodes along the paths that are needed for covering the required nodes and all necessary *post* nodes have been annotated; all decision queues associated with all annotated *if* nodes have been constructed; and all path queues have been initialized. In the simple example, when node 28 is taken out of the working queue in step 5, it is found to have a nonzero TRN, and thus no more paths are added. When node 35 is taken out of the working queue, the TRN is zero. Hence, the path 31-32-33-34-35-32-36 is found to cover node 35.

Phase 2: Generating a du-path. For each thread, perform the following steps:

Step 1. Let n be the *begin* node of the thread.

Step 2. While n's TRN > 0 and n is not the *end* node, add n to the thread's path queue, which contains the resulting path coverage, and decrement n's TRN. If n is an *if-node*, then let the new n be the node removed from n's decision queue. Otherwise, if n is a *loop* node, the successor with the smallest non-zero TRN is chosen to be the new n. Thus, the nodes within a loop body will be covered prior to covering the nodes after a loop. If the children have the same TRN, a child is arbitrarily chosen. In our implementation, the child node with the smallest RPO is chosen. Otherwise, if n is not an *if-node* or *loop* node, let the successor of n be the new n.

In the simple example, with the annotated PPFG as input, the second phase finds a final path of 1-2-3-4-5-7-8-9-3-11 for $manager$, 21-22-23-25-26-27-28-22-30 for $worker_1$, and 31-32-33-34-35-32-36 for $worker_2$.

5.6.3 Example of Generating a $PATH_a$

In this section, an example is presented to illustrate the procedure of generating a $PATH_a$. Like the earlier simple example, this example also covers the du-pair with the define of x at node 4 and the use of the variable x at node 26 in Figure 5.16.

During the second step of the annotating phase, the required nodes, including the *pthread_create*, $define$, $post_2$, $wait_2$, and the *use* nodes, are included in a partial path identified as 2-3-4-5-7-25-26. During the third step of the annotating phase, the two partial paths are completed, and found to be 1-2-3-4-5-7-8-9-3-11 for $manager$ and 21-22-23-25-26-27-29-22-30 for $worker_1$. The TRN for every node along the two paths except node 22 becomes 1 after step 4; the TRN of node 22 becomes 2.

When node 9 was reached during this traversal, nodes 28 and 35 were put into the working queues. Similarly, during the fifth step of the annotate phase, two paths 21-22-23-25-26-27-28-22-30 and 31-32-33-34-35-32-36, are found to cover nodes 28 and 35, respectively. The final TRN's for this example label each node in Figure 5.16. The path generation phase finds final paths of 1-2-3-4-5-7-8-9-3-11 for *manager*, 21-22-23-25-26-27-29-22-23-25-26-27-28-22-30 for $worker_1$ and 31-32-33-34-35-32-36 for $worker_2$. This set of paths is not *w-runnable* because the execution will cause $worker_1$ to wait infinitely at node 25, but it is a $PATH_a$.

5.6.4 Correctness and Complexity

Given a du-pair in a parallel program where the $define$ node and the *use* node are located in two different threads, it can be shown that this algorithm indeed will terminate and find a $PATH_a$. Some lemmas are introduced first before the final proof is shown.

Lemma 1: *TRN preserves the number of required traversals of each node within a loop body.*

During the first phase, the TRN of a node is incremented by one each time a path is generated that includes that node. Therefore, the number of traversals of each node in paths found during the first phase is preserved by the TRN. Although the number of traversals during the first phase is preserved, we are not claiming that these nodes will indeed be traversed during the second phase that same number of times. For nodes outside of a loop body, each node will be traversed at most as many times as its TRN. But a node may not need to be traversed that many times because the path generation phase may reach the *End* node before the TRN of all nodes becomes zero. Moreover, if there is no loop node in a program, only required nodes will be traversed as many times as the TRN indicates.

Lemma 2: *The decision queue and TRN of an* if-node *guarantee that the same sequence of branches selected during the first phase will be selected during the second phase.*

When an *if-node* is found in a path during the first phase, one branch is stored into the decision queue at that time. Hence, the number of branches in the decision queue of a given *if-node* is equal to the TRN of that *if-node*. Each time the *if-node* is traversed during the second phase, one node is taken out of the decision queue and the TRN of the *if-node* is decremented by one. Therefore, the sequence that a branch is selected is preserved.

Lemma 3: *DFS used during the first phase ensures $define \prec post \prec wait \prec use$ in the final generated path.*

During the first phase, the required nodes will be marked by DFS prior to any other nodes in the graph. This ensures that necessary branches are stored in the decision queues first. By Lemma 2, these branches will be traversed first during the second phase. Hence, these nodes will be traversed in the correct order as given by the relationships above. Therefore, Lemma 3 is valid.

Lemma 4: *The working queues and TRN together guarantee the termination of the du-path Finding Algorithm.*

We must show that both phases terminate.

Phase 1 Terminates. We use mathematical induction on m, where m represents the total number of pairs of synchronization nodes covered in a path coverage.

Base case: $m = 1$. Since there is only one pair of synchronization calls, the required ones, they will be included in the path generated by the DFS. The completion of the two partial paths will automatically terminate since there are no extra *post* or

wait's involved. Now, assume Lemma 4 is true when $m = k$ where k is an integer greater than 1. We need to show that Lemma 4 is also true when $m = k + 1$. If the *post* and *wait* have been traversed previously, the TRN of these nodes will be greater than zero. Hence, they will not be included again during the first phase. When we generate a new path to cover this pair of *post* and *wait* nodes, if they currently have TRN=0, all other pairs of synchronization nodes will have been covered (by the induction step). Hence, this new pair of synchronization calls will not trigger an unlimited number of actions. Therefore, the annotation phase will terminate.
Phase 2 Terminates. Since the TRN for each node is a finite integer, TRN is decremented each time it is traversed during phase 2, and when a node with zero TRN or the *End* node is reached, the path generation phase terminates, the traversal during phase 2 will not iterate forever.

Finally, we show the proof of the following theorem.
Theorem 5: *Given a du-pair in a shared memory, parallel program, the hybrid approach terminates and finds a* $PATH_a$.
Proof:

By Lemma 4, the hybrid approach terminates. To show that a $PATH_a$ is generated, we must show that the conditions described in the definition of $PATH_a$ are satisfied. By Lemma 1, Lemma 2, and Lemma 3, we can conclude that the $define$, *use*, the required *post*, and *wait* nodes will be covered in the correct order. Step 1 of Phase 1 ensures that all appropriate *pthread_create* calls are covered. Step 5 ensures that a matching *post* node for each *wait* node included in the path is also covered. Therefore, all conditions for a $PATH_a$ are satisfied for the path stored in the path queue for each thread. Q.E.D.

The running time of the hybrid approach includes the time spent searching for the required nodes and time spent generating the final path coverage. We assume that the dominator trees, the implied trees and the du-pairs have been provided by an optimizing compiler.

Theorem 6: For a given $G = (V, E)$, and a du-pair (variable, d, u), the total running time of the du-path finding algorithm is equal to O($k^2 * |V|$), where the total number of *post* and *wait* calls is denoted by k.

Proof:

Step 1 takes running time O($|V|$) to initialize information for all nodes. Since the DFS is used when searching for the required nodes, the running time for step 2 is equal to O($|V|$). To complete the two partial paths using the modified DT-IT in step 3, the running time is O($k * (|V|)$), where k is the total number of *post* and *wait* calls. Finally, the total number of required iterations in phase two is less than or equal to the maximum TRN in all nodes. When finding a path to cover a *post* p, we may need to find one additional path for each *wait* covered in the resulting path for covering p. Thus, we need to invoke the modified DT-IT at most O(k) times for covering node p. If this is true for every *post* node, a *post* node can be visited at most O(k^2) times when the du-path finding algorithm terminates. This is true due to the fact that once we start finding a path to cover a node, we will never start again with the same node. Otherwise, an infinite recursion will occur. Hence, the maximum value of a TRN is O(k^2) when the du-path finding algorithm terminates. The second phase takes time O($k^2 * |V|$) to finish. In total, the running time is O($k^2 * |V|$). Q.E.D.

5.7 Summary and Contributions

In this chapter, we presented a number of contributions concerning the detection of a du-path for a given du-pair in a shared memory parallel program. The issues of finding a du-path for a given parallel program were investigated, a classification of du-path coverages was presented, an algorithm was developed to find an acceptable du-path coverage for a given parallel program, and the correctness and the running time were examined. The intractability issues of finding an acceptable and a w-runnable du-path coverage were addressed, respectively.

Chapter 6

TEMPORAL TESTING

The objective of temporal testing is to expose synchronization errors in a parallel program without introducing environmental changes such as an additional control thread. The two major approaches to temporal testing of concurrent programs are deterministic execution (DET) and delay execution (DEL). Tai et al.[128] developed DET for reproducing a test of concurrent programs. However, the method changes the execution environment by adding an extra control task and some additional synchronization events between the original task and the new control task. Gait [41] introduced the approach of instrumenting delays into Ada programs to alter the execution time of synchronization events. The delay execution approach does not require an additional task to control and monitor the execution. Thus, in our research, the delay execution approach is used. However, the delay execution approach can generate an impractical number of test cases. Hence, the research described in this chapter focuses on identifying and reducing redundant test cases using static analysis techniques.

The remainder of this chapter is organized as follows. Section 6.1 describes the location of delay statements and illustrates the inherent problem of applying delay execution to testing. Section 6.2 defines the redundancy of delay points, presents the Redundancy Delay Theorem, develops a static analysis technique for identifying redundant delays in a given parallel program, and demonstrates the usefulness of identifying redundant delays. Section 6.3 describes the data structure and implementation of test case generation and redundant delay reduction. Section 6.4 summarizes the contributions.

6.1 Delay Execution Approach

The crux of the delay execution technique is to determine where to insert delay statements for the most benefit in temporal testing without requiring an impractical number of temporal test cases. The potential locations where delay statements can be inserted are referred to as *delay points.* Delay execution is used in conjunction with path testing. For example, du-path testing can be implemented by locating delay points along the du-paths being tested and inserting delay statements at the delay points. The purpose of these delay statements is to alter the execution time of all process creation and synchronization events along these du-paths. A testing tool is used to automatically generate and execute temporal test cases.

6.1.1 Execution Timing Chart

To illustrate the execution timing of a parallel program, an execution timing chart, or simply timing chart [53], is used in this chapter. Since temporal testing is conducted after a testing path (e.g., du-path) is determined and tested, the branch taken at an *if*-statement is determined statically during du-path detection. Thus, the execution timing of each thread can be represented by a vertical time line. The execution timing proceeds from top to bottom of the vertical line. In a path with timing changes, there are several possible kinds of statements that can occur: delay points, *post/wait/pthread_create* statements, and computation statements. Hence, in a timing chart, a dot represents a delay point; a directed edge that points from a *post* to a *wait* represents a *post/wait* statement; an edge leading from a process creation node to the first statement in the child process represents a process creation; a short horizontal bar marked on a vertical line is used to represent a computation statement. Figure 6.1 shows several examples of timing charts. For simplicity, the subscripts associated with a *post* or *wait* used in these graphs are used for identifying each *post* or *wait*, respectively.

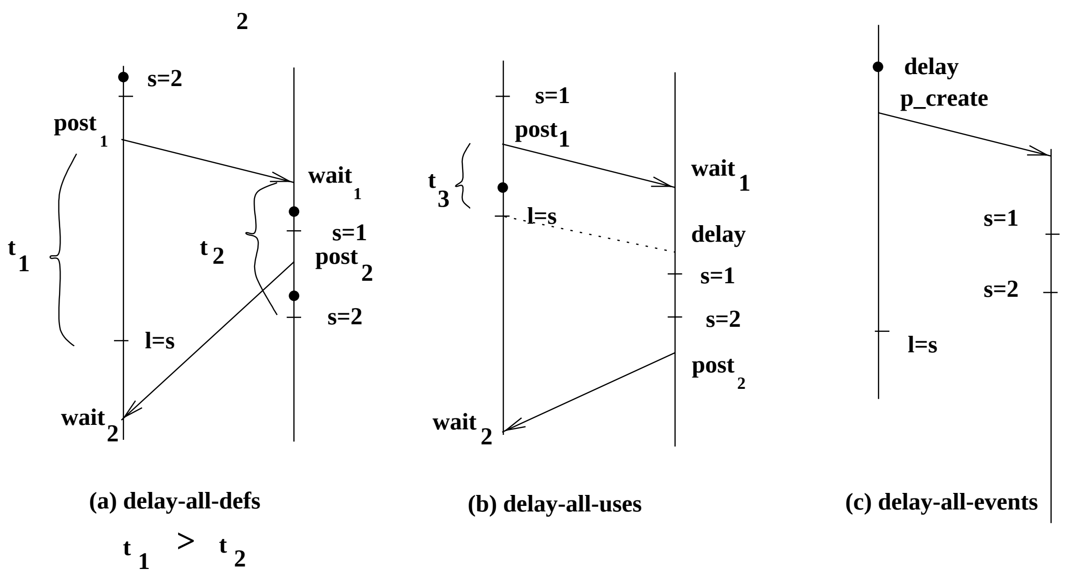


Figure 6.1: Temporal Testing Criteria

6.1.2 Locating Delay Points

Given a particular structural testing criterion, one of several methods can be used for locating delay points on the testing paths of interest. This chapter focuses on three different methods for locating delay points along du-paths to enable du-path testing with timing differences.

1. Delay-all-definitions(*delay-all-defs*) - Locate a delay point prior to each definition of a shared variable along the testing paths. The number of these delay points can be reduced by locating only one delay point prior to the first definition of any shared variable in a basic block that contains no synchronization statements or thread creation statements. In addition, if a basic block contains one of these statements, then it is only necessary to place a delay point before

the first definition of any shared variable in each segment of the basic block, where the basic block is segmented by the synchronization and thread creation statements. Thus, a single delay point is needed only for the first definition of shared variables in a code segment, where a *code segment* is defined to be a basic block if the basic block contains no synchronization statement or thread creation statement, or a sequence of instructions in a basic block such that a synchronization event, thread creation, or *end* statement is the last statement of the sequence of instructions.

2. Delay-all-uses(*delay-all-uses*) - Locate a delay point prior to each use of a shared variable along the testing paths. The number of delay points can be reduced similarly to the delay-all-defs method.

3. Delay-all-events(*delay-all-events*) - Locate a delay point prior to each thread creation, communication, and synchronization event along the testing paths.

Each of the three methods for locating delay points can be viewed as one instantiation of the *X-testing with timing* criteria in Figure 4.4. Three examples are presented in Figure 6.1 to show that these temporal testing criteria can achieve different types of temporal testing coverages. Figure 6.1(a) illustrates a situation of applying delay-all-defs criterion for which three delay points are inserted for delaying the definitions of the shared variable s, respectively. Figure 6.1(b) illustrates a situation of delay-all-uses for which one delay point is inserted to delay the code segment containing the assignment statement "$l = s$". Figure 6.1(c) illustrates a situation of applying delay-all-events criterion.

In Figure 6.1(a), the race condition can be exposed during the testing process by delaying the code segment after the $post_2$ containing the statement "$s = 2$", based on the *delay-all-defs* criterion. In this case, delay-all-uses criterion may not expose the race condition. In Figure 6.1(b), the race condition can be exposed during testing by delaying the code segment containing the statement "$l = s$", based on the *delay-all-uses* criterion. In this case, delay-all-defs criterion cannot expose the race condition because the delay-all-definitions criterion only applies to the definitions of shared variables; the statement "$l = s$" will not be included. In Figure 6.1(c), the race condition can be exposed by delaying the thread creation statement based on the *delay-all-events* criterion. In this case, delay-all-uses criterion cannot expose this race condition.

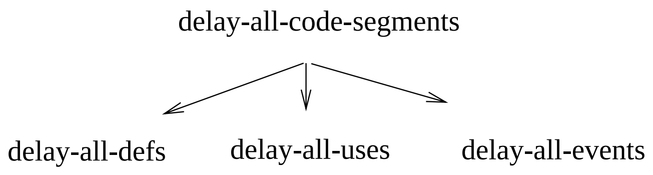


Figure 6.2: Temporal Testing Hierarchy

Although these three temporal testing criteria seem independent of each other in terms of detecting various race conditions, they are related in the way that inserting delay statements at the beginning of a basic block or segment of a basic block ending with the *end*, *post*, *wait*, or *pthread_create*, can actually delay *all* statements within the same code segment. To achieve all three criteria, one could insert one delay statement for each code segment. Figure 6.2 shows the hierarchy of temporal testing criteria just described. However, this would create an impractical number of delay points and temporal test cases.

In the rest of this chapter, we focus on the *delay-all-events* criterion.

6.1.3 Inherent Problem of Delay Insertion

For a particular parallel program, the total number of possible temporal test cases per input data set can be formulated as follows.

$$\text{No. of tests} = \begin{pmatrix} n \\ 1 \end{pmatrix} + \begin{pmatrix} n \\ 2 \end{pmatrix} + \cdots + \begin{pmatrix} n \\ n \end{pmatrix} = 2^n - 1,$$

where n is the total number of delay points in a set of paths $PATH$ of the parallel program based on a particular path coverage criterion. This formula reflects the fact that delay statements can be inserted progressively. For example, one delay is inserted at each delay point one at a time, followed by two delays in each test case, etc., until all delay points are instrumented with delay statements. For a typical number of delay points, the total number of temporal test cases can grow exponentially.

Part of the resolution to this issue is to perform the process of temporal testing semi-automatically; the process of inserting delays, compiling the program, executing the tests as well as collecting the results, should be done with the aid of a testing tool. In addition, the number of temporal test cases can be reduced by using any combination of the following techniques.

- Fix the testing level, that is, the number of delay statements that are actually inserted simultaneously in a temporal test case to some level *level*. This approach reduces the total number of temporal test cases to $\mathcal{O}(n^{level})$ where n is the total number of delay points for small testing levels.
- Use operational profiling to find critical paths and perform temporal testing more thoroughly for those paths than less critical paths.
- Use static analysis to find equivalent classes of delay points and then eliminate redundant delays and the corresponding test cases.

Since the first two approaches can be implemented easily, the rest of this chapter focuses on eliminating redundant delays and the corresponding test cases by using the static analysis commonly employed in optimizing compilers.

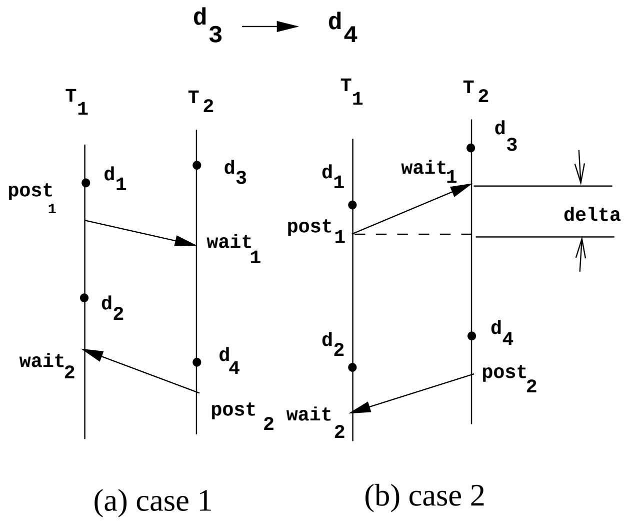


Figure 6.3: Observation of Delay Correlations

6.2 Redundant Delays

To represent the execution time of a statement, we define the following notation.

Definition: execution time $t(n)$

The execution time of a node n is defined to be the clock time when the node n is executed, and is represented by $t(n)$.

Prior to introducing the notion of a *redundant delay*, this section discusses an important observation that helps to explain the possible correlations between different delay points.

In Figure 6.3(a) and (b), if the duration of the delay at d_3 is progressively increased, eventually the statements after $wait_1$ in thread T_2 will also be delayed. However, the statements between $post_1$ and $wait_2$ in thread T_1 will not be influenced. The effect of the delay at d_4 is to delay the statements after d_4 in thread T_2, and the completion of $wait_2$ and the statements following $wait_2$ in thread T_1. Hence, informally, *the effect of delaying* d_4 *is potentially achieved when the delay* d_3 is inserted. This observation can be informally explained by looking at the two possible cases :(1) when $t(post_1) < t(wait_1)$ originally, and (2) when $t(post_1) \geq t(wait_1)$ originally. The two cases are illustrated in Figure 6.3(a) and 6.3(b), respectively.

When the original scheduled execution time of $post_1$ is earlier than that of $wait_1$, delaying d_3 will delay both $wait_1$ and $post_2$, and thus the completion of $wait_2$ and the statements following $wait_2$ in thread T_1. The scheduled time to start executing $wait_2$, $t(wait_2)$, is not influenced. In the second case, assuming that $delta = t(post_1)$ - $t(wait_1)$, only when $d_3 \leq delta$ will $t(post_2)$ not be influenced. Otherwise, delaying d_3 will also delay $post_2$, and thus the completion of $wait_2$ and the statements following $wait_2$ in thread T_1. Thus, in both cases, delaying at d_3 will potentially delay only $post_2$ but not the start of $wait_2$; hence, the claim.

Unlike the situation just mentioned, *delaying d_1 will not potentially achieve the effect of delaying at d_2.* Note that the effect of d_2 is to delay the statements after d_2 in thread T_1 only. This can also be informally explained for the two possible cases: (1) when $t(post_1) < t(wait_1)$ and (2) when $t(post_1) \geq t(wait_1)$. In the first case, only when the delay d_1 is less than $delta = t(wait_1) - t(post_1)$ will $wait_2$ be delayed and not $post_2$. If d_1 keeps increasing, eventually both $post_2$ and $wait_2$ will be delayed for about the same time. In the second case, delaying d_1 will delay both $post_2$ and $wait_2$. It can be concluded that *delaying d_1 will potentially delay both $post_2$ and $wait_2$*; hence, the claim.

This observation motivates the definition of *redundant delays* defined in this section. The symbol $\mathcal{V}_d(s)$ is a set of tuples where each tuple contains a variable dependent on a definition of the shared variable s and the set of possible final values for that variable, during execution with the delay d inserted. For example, in Figure 6.4, $\mathcal{V}_d(a) = \mathcal{V}_{d'}(a) = \{(v, \{1, 2\})\}$, and $\mathcal{V}_d(b) = \mathcal{V}_{d'}(b) = \{(w, \{1, 2\})\}$. For simplicity of notation, when there exists only one variable dependent on a particular shared variable as in our example, we use simple set notation, i.e., $\mathcal{V}_d(a) = \mathcal{V}_{d'}(a) =$

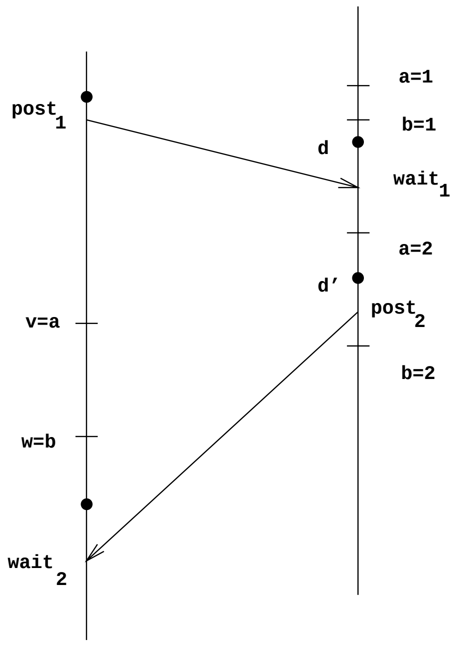


Figure 6.4: Illustrating Delay Redundancy

$\{1, 2\}$, and $\mathcal{V}_d(b) = \mathcal{V}_{d'}(b) = \{1, 2\}$.

Def: Redundant Delay.

For <u>all</u> shared variables s, if $\exists$ delays d_1 and d_2, such that d_1 reaches d_2 (i.e., $d_1 \Rightarrow d_2$) and $\mathcal{V}_{d_2}(s) \subseteq \mathcal{V}_{d_1}(s)$, the delay d_2 is redundant.

An important observation is that for generating test cases to expose synchronization errors, it is not necessary to create an execution such that every possible combination of values for all variables can be produced. To explain this, we use the notation $\mathcal{V}_{d_2}(\langle s_1, \ldots, s_n \rangle)$ to represent the set of all possible tuples of final values for variables dependent on shared variables s_1 through s_n, where each tuple represents the final values for these variables on a single execution. It is likely that for a redundant delay d_1, $\mathcal{V}_{d_1}(s_1) \subseteq \mathcal{V}_{d_2}(s_1)$, ..., $\mathcal{V}_{d_1}(s_n) \subseteq \mathcal{V}_{d_2}(s_n)$, but $\mathcal{V}_{d_1}(\langle s_1, \ldots, s_n \rangle) \not\subseteq \mathcal{V}_{d_2}(\langle s_1, \ldots, s_n \rangle)$. For example, in Figure 6.4, $\mathcal{V}_{d'}(a) = \mathcal{V}_d(a) = \{1, 2\}$, and

$\mathcal{V}_{d'}(b) = \mathcal{V}_d(b) = \{1, 2\}$, but, $\mathcal{V}_d(\langle a, b\rangle) = \{\langle 2, 2\rangle, \langle 1, 2\rangle, \langle 1, 1\rangle\}$, $\mathcal{V}_{d'}(\langle a, b\rangle) = \{\langle 2, 1\rangle\}$. This example illustrates the notion of redundancy just defined. If for all shared variables s, all of the possible values of s can be exposed by delaying both the delay d and d', then one of them is redundant. But it is not necessary to expose all *combinations* of values of all shared variables.

6.2.1 Redundant Delay Theorem

In order to describe the algorithm for identifying redundant delays, the concepts of an immediately preceding delay and the intrusiveness of a delay are first defined, and then a theorem that lies at the base of the algorithm is presented. For each du-path coverage $PATH$, redundant delay points are identified along the set of paths in $PATH$ covered by the nontemporal test case in order to reduce the number of temporal test cases associated with that set of input data. Thus, the computation of the set of redundant delays is done with respect to a set of paths $PATH$ through the parallel program, not the entire PPFG.

Def: Incidence with an Edge

For an edge $e = (n_1, n_2)$, we say that nodes n_1 and n_2 are *incident with* e, denoted by $n_1 : e$ and $n_2 : e$, respectively.

Def: Immediately Preceding Delay

A delay d is called the immediately preceding delay of a post *or* wait *node n in* $PATH$*, or IPD(n), if* $\exists$ *a path P(d, n), and* $\nexists$ *no other delay in P(d, n).*

Def: Intrusiveness of a Delay

A delay d in $PATH$ *is* intrusive *to an edge* $e = (n_1, n_2)$ *in* $PATH$*, if* $d \Rightarrow n_1 \bigwedge d \not\Rightarrow n_2$*, or* $d \not\Rightarrow n_1 \bigwedge d \Rightarrow n_2$*. This is denoted as* $d \mapsto e$*. The delay d is called an intrusive source node and the edge e is called the target edge of the intrusion.*

REDUNDANT DELAY THEOREM

Given a parallel program $\mathcal{P}$ represented as a PPFG and a set of testing paths $PATH$, where each post and wait n has $IPD(n) \neq 0$, and given delays d and d' in $PATH$, d' is redundant and we say that the delay d **kills** *d', denoted as $d \rightarrow d'$ if the following conditions are satisfied:*

(1)$d \mapsto e$, where $e = (n_i, n_j) \in E_S$,

(2)$d \Rightarrow d'$, and

(3)$d' = IPD(n_i)$ or $d' = IPD(n_j)$.

PROOF:

The crux of this proof is to show that whenever the set of conditions above are satisfied for all shared variables s, if $\exists\ a \in \mathcal{V}_{d'}(s)$, then $a \in \mathcal{V}_d(s)$. Without loss of generality, we use *two* threads to prove this theorem. Let $length(d)$ be a particular duration of a delay d. Let $t_1(l = s)$ and $t_2(l = s)$ represent the scheduled execution times of a statement using the shared variables s, say l=s, *before* and *after* inserting a delay with $length(d) > 0$ at the beginning of the corresponding code segment, respectively. Let $TH(l = s)$ represent the thread in which l=s is located. It can be claimed that for each a in $\mathcal{V}_{d'}(s)$, exactly one of the following is true: (1)$a \in \mathcal{V}_{d'}(s)$ even when $length(d') = 0$, or (2)$a \in \mathcal{V}_{d'}(s)$ only if $length(d') > 0$. The theorem is trivially true for the first case.

Assume that the conditions in the Redundant Delay Theorem hold, and that $a \in \mathcal{V}_{d'}(s)$ for some arbitrary shared variable s.

Case (1):

$t_1(s = a) < t_1(s = a') < t_1(l = s)$, and

$t_2(s = a) < t_2(l = s) < t_2(s = a')$.

(i.e., d' delays $s = a'$ past $l = s$ to expose older value a. For d' to delay $s = a'$ w.r.t. $l = s$, $TH(d') = TH(s = a') \neq TH(l = s)$, and $d' \Rightarrow (s = a')$).

Case (2):

$t_1(s = a') < t_1(l = s) < t_1(s = a)$, and

$t_2(s = a') < t_2(s = a) < t_2(l = s)$.
(i.e., d' delays $l = s$ past $s = a$ to cover old value a'. For d' to delay $l = s$ w.r.t. $s = a$, $TH(d') = TH(l = s) \neq TH(s = a)$, and $d' \Rightarrow (l = s)$).

Some instantiations of Case (1) are illustrated in Figure 6.5(a)-(e). Some case (2) instantiations are shown in Figure 6.5(f)-(j). In cases 6.5(a)-(e), the instruction $s = a$ must be executed prior to the execution time of the delay d', i.e., $t_1(d')$.

The two cases, shown in Figure 6.5(e) and (j) are invalid cases. In Figure 6.5(e), since it is impossible to delay instructions in $TH(s = a')$ without delaying instructions in $TH(l = s)$, there is no non-trivial case such that $a \in \mathcal{V}_{d'}(s)$. Similarly, in Figure 6.5(j), since it is impossible to delay instructions in $TH(l = s)$ without delaying instructions in $TH(s = a)$, there is no non-trivial case such that $a \in \mathcal{V}_{d'}(s)$. In summary, when $d' = IPD(n_i)$, $t_1(n_j) > t_1(n_i)$.
In the following, the proof for Case (1) is shown. The proof for Case (2) is very similar.
There are two subcases: $d' =$ IPD(n_i), or $d' =$ IPD(n_j) where $e = (n_i, n_j) \in E_S$. Let $d' =$ IPD(n_i). $d \Rightarrow d'$ and $d \mapsto e$ imply that $d \Rightarrow n_i$ and $d \not\Rightarrow n_j$. In addition, $d \Rightarrow d'$ implies $d \Rightarrow (s = a')$.

Let $\Delta = t_1(l = s) - t_1(s = a')$. Let $\Delta' = t_1(wait_2) - t_1(post_2)$. Since $a \in \mathcal{V}_{d'}(s)$, $t_1(l = s) - t_1(s = a') \leq \Delta'$ and we also know that $\Delta' > 0$. This is important because, otherwise, delaying d' cannot cause $t_2(s = a')$ to become greater than $t_2(l = s)$.

Now we can increase $length(d)$ by the amount greater than Δ. Since $d' = IPD(post_2)$ and $t_1(wait_2) > t_1(post_2)$, the increase in delay length is also limited to less than Δ' such that $t_2(l = s) = t_1(l = s)$ regardless of whether $t_1(l = s)$ is greater or less than $t_1(wait_2)$. For all instructions i in $TH(post_2)$, such that $t_1(i) > t_1(post_2)$, $t_2(i) - t_1(i) > \Delta$.
Hence, $t_2(s = a') > t_1(s = a') + \Delta \quad (**)$.

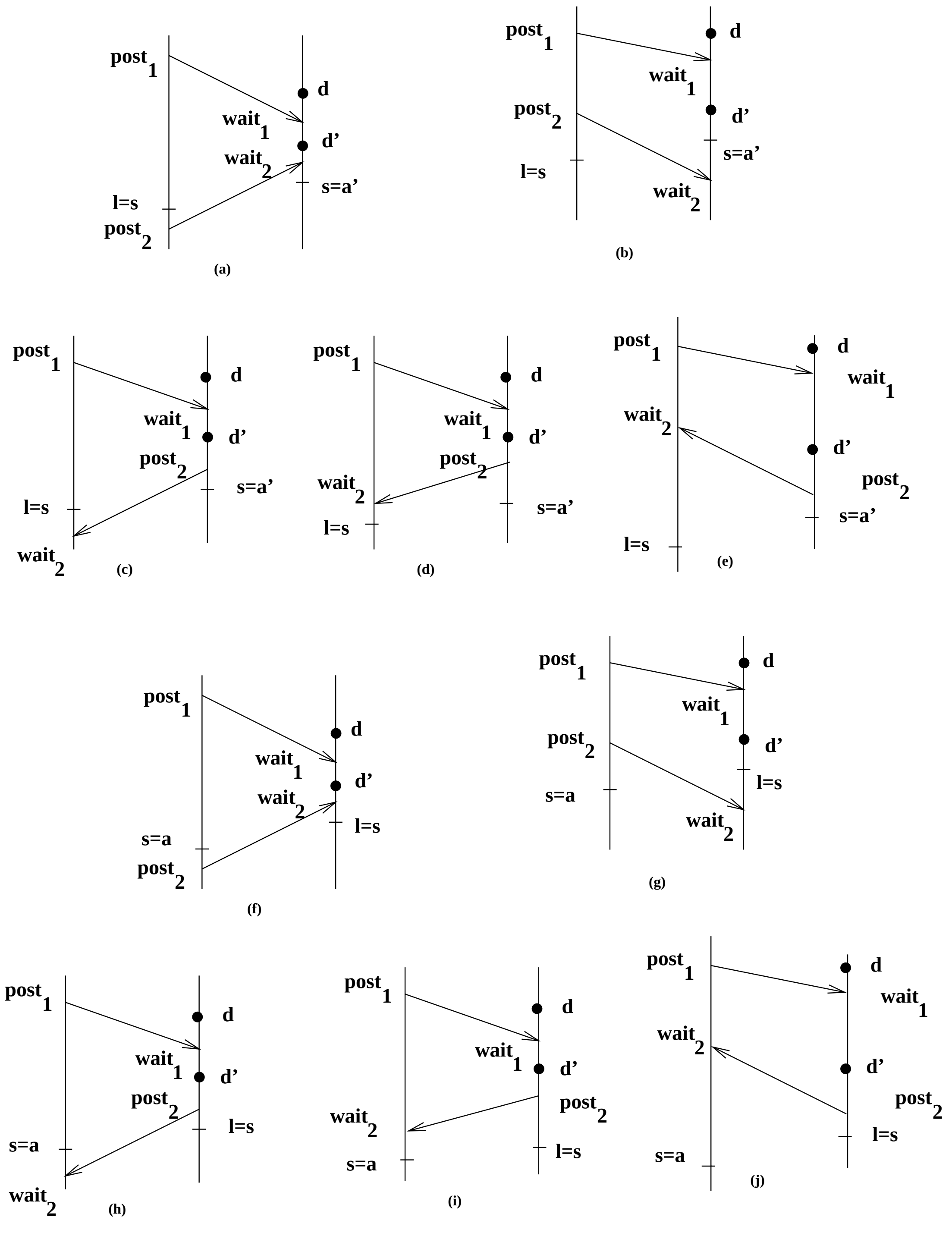


Figure 6.5: Possible Memory Accesses

Moreover, $t_2(l = s) = t_1(l = s)$. $\qquad (*)$

Using $(*) - (**)$, we get the following:

$t_2(l = s) - t_2(s = a')$

$< t_1(l = s) - (t_1(s = a') + \Delta)$

$= [t_1(l = s) - t_1(s = a')] - \Delta = 0$

That is, $t_2(l = s) - t_2(s = a') < 0$. Hence, $a \in \mathcal{V}_d(s)$. So $\mathcal{V}_{d'}(s) \subseteq \mathcal{V}_d(s)$. The proof for subcase $d' = \text{IPD}(n_i)$ is similar.

The proof is valid for all shared variables.

Q.E.D.

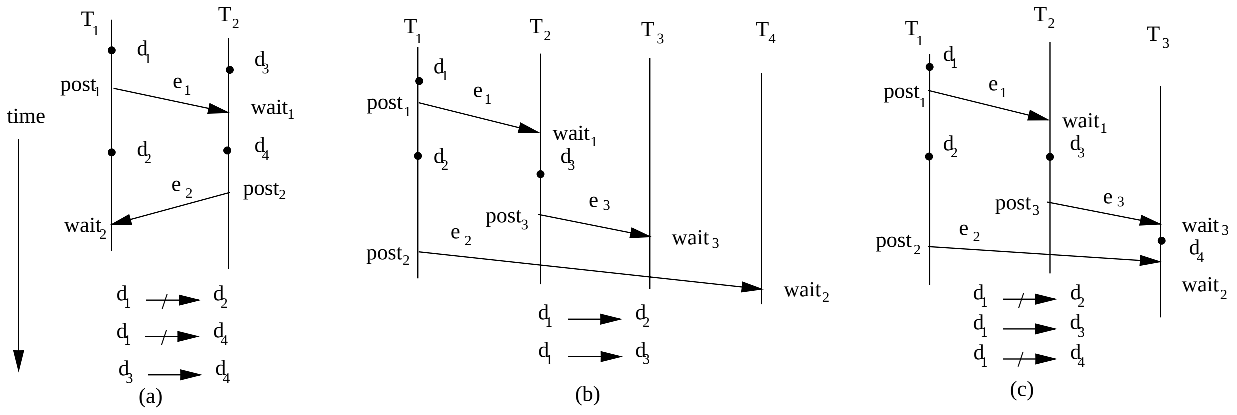


Figure 6.6: Illustrating Redundant Delays

As an example of applying the Redundant Delay Theorem, the delay point d_4 in Figure 6.6(a) is redundant, i.e., $d_3 \rightarrow d_4$ because the following conditions are satisfied: $d_4 = IPD(post_2)$, and $post_2 : e_2$, and $d_3 \mapsto e_2$, and $d_3 \Rightarrow d_4$. Other redundant delays include d_2 and d_3 in Figure 6.6(b) and d_3 in Figure 6.6(c). Not all redundant delay points are included in these diagrams. For instance, d_4 would also be redundant in Figure 6.6(c) due to the intrusion of a delay point not shown in the diagram in front of $wait_3$, if the delay point existed.

6.2.2 Detecting Redundant Delays

Algorithm: Identifying Redundant Delays
Input: A PPFG
Output: A list of redundant delay points
Method:

1. Compute Reach(n) for all post and wait nodes n.
2. Compute the intrusion set Intrusion(n) for each post and wait node n in PPFG using Reach(n).
3. Determining the redundant delay points using Intrusion(post) and Intrusion(wait).

Figure 6.7: Identifying Redundant Delays

Based on the Redundant Delay Theorem, an algorithm for detecting redundant delay points is given in Figure 6.7. The detailed steps in computing the redundancy of delay points are briefly discussed in this section. In subsequent sections, the steps are described in more detail, with an example.

Step 1. Compute $Reach(n)$ - The *reaching delay* of the node n is defined as the set of delay points in $PATH$ that can reach a *post* or a *wait* n in $PATH$ through either a path or interprocess connection in $PATH$. That is, $Reach(n) = \{d' | d' \textit{ a delay point}, \ d' \Rightarrow n\}$. For every *post* or *wait* n, we need to compute $Reach(n)$. Thus, the computation of $Reach(n)$ can be formulated as a data flow problem over $PATH$.

Step 2. Compute $Intrusion(n)$ - For each *post* and *wait* n in $PATH$, the set $Intrusion(n)$ can be represented as follows:
$Intrusion(n) = \{d | d \textit{ is a delay in } \text{PATH}, \ d \mapsto \text{e} \bigwedge n : e \bigwedge d \Rightarrow n\}$.
The intrusion set consists of all delays d such that d reaches a node n which is incident with an edge e, and d is intrusive to the edge e. The definition of Intrusion(n) implies that the set of reaching delays to the nodes incident with the synchronization edge e can be used to determine which delay points are intrusive to e=(n_i, n_j). That is, if d can reach either n_i or n_j but not both, this delay d is added to the set $Intrusion(n_i)$ or $Intrusion(n_j)$, respectively. Based on this principle, two equations for computing $Intrusion(n)$ are formulated. They are shown in Section 6.2.4. Applying these formulas, $Intrusion$ information can be computed for each *post* and *wait* n in $PATH$ accordingly

and the information is then stored in the data structure associated with each *post* p and *wait* w. For a target synchronization edge e=(p, w), the set of all intrusive source nodes for e, computed by the union of $Intrusion(p)$ and $Intrusion(w)$, can be stored in the data structure associated with the edge e.

Step 3. Determining redundancy of delay points - Intuitively, if $Intrusion(post_i)$ includes any delays other than the delay $IPD(post_i)$, then the delay point $IPD(post_i)$ is redundant, and can be removed. The same holds for determining whether a delay point prior to a *wait* is redundant.

6.2.3 Computing Reach(n)

To characterize this problem within a data flow framework, the following three aspects must be considered: (1) the direction of propagation of information, (2) the transfer functions which characterize how information propagates through a node(i.e., statement), and (3) the confluence operator which characterizes how information is handled at points in the program where paths are joining together. First, this problem propagates information in a forward direction, that is, all delays which can reach a particular statement from previous statements in the program are taken into account, whereas delays that occur after this particular point in the program are irrelevant. Second, the transfer functions are defined as follows:

$$Reach_{out}(delay) = Reach_{in}(delay) \bigcup \{delay\} \tag{6.1}$$

$$Reach_{out}(wait) = \bigcup_{matching\, posts\, p} Reach_{out}(p) \bigcup Reach_{in}(wait) \tag{6.2}$$

$$\forall\ other\ statements\ s,\ Reach_{out}(s) \ = \ Reach_{in}(s) \tag{6.3}$$

where $Reach_{in}(n)$ is the set of delays that reach the point just prior to n in the program and $Reach_{out}(n)$ is the set of delays that reach the point just after n in the program. The confluence operator is *set union* ($\bigcup$), representing the fact that at a convergence of paths, the set of delays that reach this point is the set of delays that can reach from any of the paths to this point.

Because the problem of computing *Reach* can be formulated as a monotonic data flow problem, an iterative algorithm to compute the *Reach* information over the relevant subgraph of PPFG[5] can be used. The algorithm begins by initializing all $Reach_{out}$ and $Reach_{in}$ sets to empty, representing the conservative view that a delay point d_i should never be added to the set of reaching delays at a node n, i.e., $Reach(n)$, unless $d_i \Rightarrow n$.

6.2.4 Computing Intrusion and Redundancy

After the set of reaching delays is computed, this information is used to identify delay redundancy. The set of intrusive delays to a node n, i.e., $Intrusion(n)$, associated with a synchronization edge can be determined by using the following formulas where the *post* and the *wait* are associated with the same synchronization edge e =(p : *post*, w : *wait*):

$$Reach_{both} = Reach_{in}(p) \cap Reach_{in}(w); \tag{6.4}$$

$$Intrusion(p) = Reach_{in}(p) - Reach_{both}; \tag{6.5}$$

$$Intrusion(w) = Reach_{in}(w) - Reach_{both}; \tag{6.6}$$

In the above equations, the intersection of the set $Reach_{in}(p)$ and $Reach_{in}(w)$ represents the reaching delays that reach both threads with which the synchronization edge $e = (p, w)$ is associated. In other words, delays in this set can kill neither IPD(p) nor IPD(w). Since the intrusive source nodes n associated with e are those that are only in $Reach(p)$ or only in $Reach(w)$, respectively, the intrusive reaching delays for a *post* and a *wait* can be computed by Equations 6.5 and 6.6, respectively.

Now, d_p, i.e., IPD(p), can be removed because it is redundant if $Intrusion(p) - d_p$ is nonempty. Similarly, we can remove d_w, i.e., IPD(w), because it is redundant if $Intrusion(w) - d_w$ is nonempty. That is, when there exists at least one delay d_i in the set $Intrusion(n)$ at the node n (i.e., a *post* or a *wait*) and d_i is not the IPD(n), d_n is redundant.

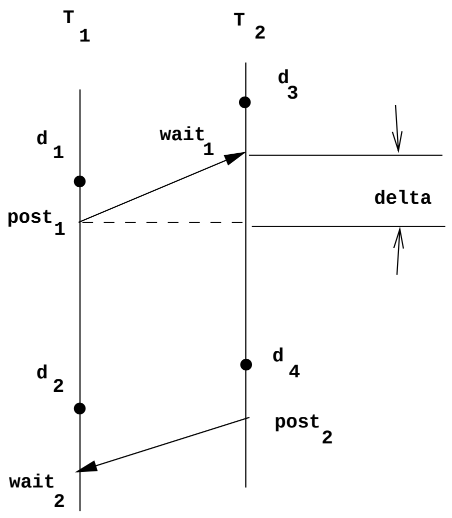


Figure 6.8: Redundant Delays

6.2.5 An Example

As a simple example of identifying redundant delays, consider Figure 6.8. The set of reaching delays, i.e., $Reach(n)$ for all nodes n, is first computed. After the data flow analysis reaches a fixed point, the results are

$$Reach_{in}(post_2) = \{d_1, d_3, d_4\}$$

$$Reach_{in}(wait_2) = \{d_1, d_2\}$$

The second step is to compute $Intrusion(post_2)$ and $Intrusion(wait_2)$. The results are

$$Intrusion(post_2) = \{d_3, d_4\}$$

$$Intrusion(wait_2) = \{d_2\}$$

From the results, d_4 can be identified to be a redundant delay because $Intrusion(post_2) - \{d_4\} = \{d_3\} \neq \{\}$. Intuitively, this implies that the delay inserted at d_3 can delay the execution of instructions in T_2 after $wait_1$, but not the instructions in T_1 before $wait_2$. Because d_3 is intrusive to ($post_2$, $wait_2$), the delay point d_4 is redundant. It can be concluded that d_3 kills d_4. Note that since $Intrusive(wait_2) - \{d_2\} = \{\}$, d_2 is not redundant and cannot be removed.

6.2.6 Proof of Correctness

In this section, we demonstrate the correctness of the algorithm for finding redundant delays. We prove that by using the static analysis for determining redundant delays, a delay can be removed if and only if it satisfies the conditions (1) (2) and (3) stated in the Redundant Delay Theorem.

COROLLARY. The Redundant Delay Identification algorithm correctly identifies and removes a delay if and only if the delay point satisfies the three conditions stated in the Redundant Delay Theorem.

IF. If a delay point satisfies the three conditions, it will be removed.

Assume that d_{post} satisfies the conditions in the
  Redundant Delay Theorem, we know that $\exists$ a delay
  point $d \neq d_{post}$ such that $d \in Reach(post)$ and
  $d \notin Reach(wait)$.
Hence, $d \notin (Reach(post) \cap Reach(wait))$.
But we also know that $d \in Reach(post)$.
So $\exists\, d(\neq d_{post}) \in Intrusion(post)$, i.e.,
  $Reach(post) - (Reach(post) \cap Reach(wait))$
So the set $Intrusion(post)$ is nonempty.
This implies that the delay point d_{post} will be
  removed by the rules of the algorithm.
Similarly, if d_{wait} satisfies the three conditions,
  it will be removed.

ONLY IF. If a delay is removed, it satisfies the three conditions
stated in the Redundant Delay Theorem.

Assume that a delay point d_{post} is removed.
By the algorithm, we know that the set
$Intrusion(post) - d_{post}$ is nonempty.
That is, $Reach(post) - (Reach(post) \bigcap Reach(wait)) - d_{post}$ is nonempty.
So $\exists$ at least one delay point $d \neq d_{post}$
and $d \in Intrusion(post)$.
Hence, $\exists$ at least one delay point $d(\neq d_{post})$
such that $d \in Reach(post)$ and
$d \notin Reach(wait)$.

Similarly, we can show that if d_{wait} is
removed, it satisfies the three conditions.

Q.E.D.

6.2.7 Complexity of the Algorithm for Identifying Redundant Delays

The running time of the algorithm for identifying redundant delay points consists of three parts. First, the running time for computing the set $Reach(n)$ is $O(n^2)$, where n is the total number of nodes in the PPFG. Second, the running time for computing $Intrusion(n')$, where n' is the total number of *post* and *wait* nodes, is $O(n')$. Third, the running time for the final step to identify redundant delay points is also $O(n')$. Since $n' \leq n$, the total running time is $O(n^2)$.

6.2.8 Eliminating Test Cases By Identifying Redundant Delays

test cases	killed by	test cases	killed by
1		2,4	2,3
2		3,4	3
3		1,2,3	
4	3	1,2,4	1,2,3
1,2		1,3,4	1,3
1,3		2,3,4	2,3
1,4	1,3	1,2,3,4	1,2,3
2,3			

Table 6.1: Reducing Test Cases Using $d_3 \rightarrow d_4$

An example of how a redundant delay can assist in eliminating test cases is shown in Table 6.1[1]. Because d_4 is killed by d_3, eight out of the possible fifteen test cases do not need to be included in the test suite.

In order to further investigate the effectiveness of computing and using redundant delay information to reduce the test suite, one of the most common parallel program examples, i.e., matrix multiplication, from the *Solaris Multithreaded Programming Guide* [123] was examined. The reaching delays and intrusion information were computed using the technique described in this chapter. In this program, multiple threads are created for computing the multiplication of a row and a column in parallel. The results are then returned to the master thread. There are two delay points in the master thread and three delay points in each worker thread. Using the redundant delay approach, one delay point is eliminated in each worker thread. Therefore, the total number of delay points is changed from $2 + 3n$ to $2 + 2n$, where n is the number of worker thread. Hence, the total number of temporal test cases is changed from $2^{(2+3n)} - 1$ to $2^{(2+2n)} - 1$. In the case that n equals 1, the total number of test cases is changed from 31 to 15. This result indicates that using redundant delay points to eliminate temporal test cases can be effective.

[1] In Table 6.1, the numbers are the indices of the delay points in Figure 6.8.

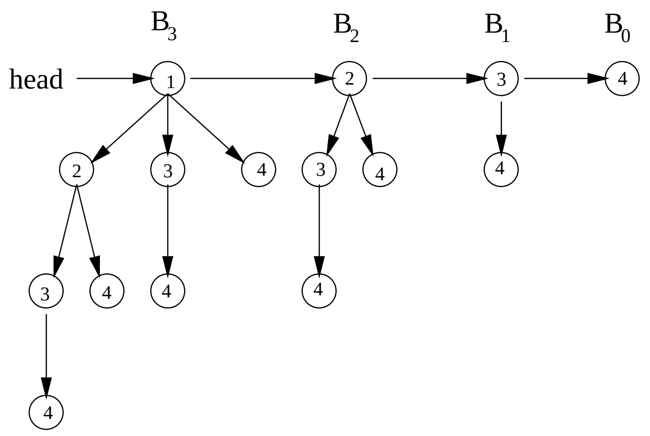


Figure 6.9: A Binomial Heap for n = 4

6.3 Generating and Tracking Temporal Test Cases

In this section, an algorithm is presented to illustrate how to generate temporal test cases automatically using redundant delay information. A data structure for representing temporal test case suites is described. Algorithms that use this data structure to track and remove redundant test cases are presented.

6.3.1 Representing A Temporal Test Case Suite

For the purpose of developing an automated testing tool, a binomial heap is used to represent combinations of delay points in which each node represents a test case. By definition, a binomial heap is a collection of binomial trees [22]. Each binomial tree is identified by using the degree of the root node, i.e., the number of children for the root.

Within a binomial heap representing an n delay point test suite, there are exactly n binomial trees with their root at the top level. As an example, the binomial heap in figure 6.9 contains binomial trees B_3, B_2, B_1, and B_0.

For the purpose of creating a binomial heap with respect to a du-path coverage, the total number of delay points in the path coverage must be computed first. Then, a binomial heap with the number of delay points is created using a binomial heap creation algorithm [22]. Thus, the nodes included in the path from the root of each binomial tree to the node represent the delay points associated with each test case.

The main reason for choosing a binomial heap to represent the temporal test case suite is the desirable property that there are exactly C(n,i) nodes at depth i, where the symbol C(n,i) represents all possible combinations of i elements taken from n elements. Each test case can be represented by a path from the root node of a binomial tree to an internal node. The number of delay points included in each test case is equal to the depth of this internal node. In general, the total number of test cases in an n delay point test suite is equal to $2^n - 1$. As an example, the test suite for a program with four delay points can be represented by the binomial heap illustrated in Figure 6.9 which can be used to represent all possible combinations of test cases generated by using four delay points. For example, the node 2 in the binomial tree B_3 represents the set of delay points $\{1, 2\}$, and is identified by the subpath 1-2; the two leaf nodes in the subtree B_2 represent the set of delay points $\{2, 3, 4\}$ and $\{2, 4\}$, respectively. If n represents the total number of delay points in a du-path coverage, a binomial heap of n nodes will be used to represented the temporal test cases associated with this du-path coverage.

After a binomial heap is created, the test suite can be generated by traversing all nodes on the first level and then all nodes on the second level, etc. When each node is traversed, a bit map is used to represent the test case. The key value is translated into a bit map in which each bit position corresponds to the index of a delay point. For example, the test case $\{d_1, d_3\}$ is represented by the path from node 1 to node 3. Hence, the bit map for the node 3 on the second level under the node 1 on the first level is equal to "0101". In other words, if the *third* bit position counting from the right has a 1, a delay statement is inserted at the delay point d_3.

A complete representation of the temporal test case suite can be constructed by a recursive algorithm illustrated in Figure 6.10. The algorithm generates the *head* node of the binomial heap structure first. It then recursively calls a routine for generating all other nodes. Test cases are generated by traversing this structure as described above.

6.3.2 Using the Test Case Suite Representation

The binomial heap representation of the test case suite is useful in several ways.

- Eliminating redundant tests - If the redundant delays are detected after test cases are produced, i.e., the binomial heap is available, we need to traverse the binomial heap and eliminate redundant test cases. Figure 6.11 provides an algorithm to achieve this goal. The main idea behind this algorithm is that when a delay point d_n becomes redundant due to any other node, we only need to delete all nodes in the heap rooted at the node with n as the key. As an example, if $d_1 \rightarrow d_3$, we only need to delete all nodes in the heap rooted at those nodes with 3 as the key value. This algorithm assumes that there are no *cyclic redundant delay relations*, i.e., redundant delay relations such as $a \rightarrow b$, $b \rightarrow c$, and $c \rightarrow a$. If the redundancy delay relations provided for a program form a cycle, this algorithm may erroneously eliminate all nodes involved in the cycle. This problem can be easily resolved by detecting the cyclic redundancy first and then throwing away one redundancy relation within the cycle to eliminate a cyclic redundancy.

- Tracking test cases - The fact that each path from the root of a binomial tree within the heap to a node represents a test case can be used to track the progress of testing. For example, we can annotate each node with a flag indicating the current status of the test case, i.e., tested or not tested. A

```
Algorithm:    Generate test suite representation
Input:        n - maximum number of delay points
Output:       A binomial heap
Method:
1. Generate the Head Node of the binomial heap;
2. Construct a root node and store the address in the left pointer
     field of the Head node;
3. Recursively calling gen() to generate a child node or a sibling node;
   If a child node is generated, the left pointer of its parent node
     will be used to store the address of the newly created node;
   If a sibling node is generated, the right pointer of its parent node
     will be used to store the address of the newly generated node.

   gen(pt1, pt2, max, key)
   /* pt1 - a parent node pointer
      pt2 - a child node or a sibling node pointer
      max - the maximum number of delay points
      key - the key value of the node being created */
   {
      Store the key value into the child node or the sibling node
        whose address is stored in pt2;
      Record that the previous node accessed prior to pt2 is pt1;
      If ( (key < max) and (pt2->left is NULL) )
          Recursively call gen() to generate a child node and
            store the new key value in the child node;
      If ( (key < max) and (pt2->right is NULL) )
          Recursively call gen() to generate a sibling node and
            store the new key value in the sibling node;
   }
```

Figure 6.10: Generating the Test Suite Representation

```
Algorithm:  Eliminate redundant tests from full test suite representation
Input:      pt - address of a node N in the binomial heap;
            key - the key value of a redundant delay point
            lev - maximum level of nodes traversed
Output:     A binomial heap free of redundant tests.
Method:
reduce( pt, key, lev )
{
   if ( pt.key == key and pt.level <= lev )
   {
      if ( pt.parent == pt.prev )
            Eliminate the node N from its parent node
       else
            Eliminate the node N from its sibling nodes
     delete the subtree rooted at node N;
   }
   If ( pt->left != NULL and pt->left->key <= key and current level < lev )
       Call reduce(pt->left, key. lev);
   If ( pt->right != NULL and pt->right->key <= key and current level < lev )
       Call reduce(pt->right, key, lev);
}
```

Figure 6.11: Eliminating Redundant Tests

testing tool can then traverse the heap providing information on what has been tested and what still remains to be tested.

6.4 Summary and Contributions

In summary, temporal testing is used after a path testing is conducted to further enhance the possibility of exposing timing-related synchronization errors. Thus, the delay execution approach provides a seamless merger of sequential testing criteria with temporal testing.

The contributions of this chapter include details of the use of delay execution in conjunction with path testing of shared memory parallel programs, methods for locating delay points, the Redundant Delay Theorem, a static analysis technique for automatically identifying redundant delays, and a data structure and algorithms for generating temporal test cases and eliminating redundant test cases.

Chapter 7

THE DESIGN AND IMPLEMENTATION OF DELLA PASTA

This chapter describes the prototype testing tool DELaware PArallel Structural Testing Aid(**della pasta**). The objective is to demonstrate that the process of test data generation for parallel programs can be partially automated, and that the same tool can provide valuable information in response to a programmer's queries with regard to testing. The tool **della pasta** consists of three major components: the *static analyzer* that takes as input a parallel program and finds all du-pairs as well as a du-path coverage for every du-pair; a *path visualizer* that provides users with a graphic user interface for displaying the program and du-paths; and a *path handler* that accepts commands for modifying the covered nodes in a path.

7.1 The Static Analyzer

The *static analyzer* achieves four functions: PPFG construction, du-pair computation, du-path coverage identification, and path coverage classification. The analyzer was built using Stanford University's SUIF compiler infrastructure [79]. The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format. The passes are implemented as separate programs that link with the kernel contained in the SUIF library. SUIF provides a compiler infrastructure that allows researchers to easily separate passes for the implementation and study of new compiler optimization techniques. The nsharlit tool provides an infrastructure for implementing static program analyses. For the implementation work described in this dissertation, SUIF was modified to enable parsing and PPFG construction for shared memory parallel programs, and the nsharlit tool was used to create a data flow analyzer that computes du-pairs over the PPFG.

7.1.1 PPFG Construction

The PPFG construction routine receives as input a source program written in the *C* language with parallel language features `pthread_create`, `post`, and `wait`. It parses the source program and creates an equivalent representation in the SUIF intermediate format. The rest of this subsection describes the SUIF intermediate format. In particular, the SUIF representation hierarchy, symbolic information, and annotation mechanism are discussed. The construction of a PPFG with SUIF will also be described.

7.1.1.1 The SUIF Representation Hierarchy

To support interprocedural analysis with separate compilation, the root of the `SUIF` hierarchy is a *file set* that contains an entry for each program file as well as a global symbol table, which is shared by all files. This shared symbol table is used in interprocedural analysis when accessing global variables (e.g., symbols and types). Each separately compiled program file is a *file set entry* that contains a symbol table for representing items private to the file. This level of the SUIF hierarchy is the top shaded region of Figure 7.1. The symbol tables, which contain a symbol for each procedure, link this file level to the next level of the hierarchy.

The representation of procedures in SUIF includes both high-level and low-level information in order to accommodate a variety of uses. Tree data structures are used to represent both forms. High-level program structure is represented in a language-independent form of an abstract syntax tree (AST). However, because some analyses operate on lists of instructions, a low-level program structure is also

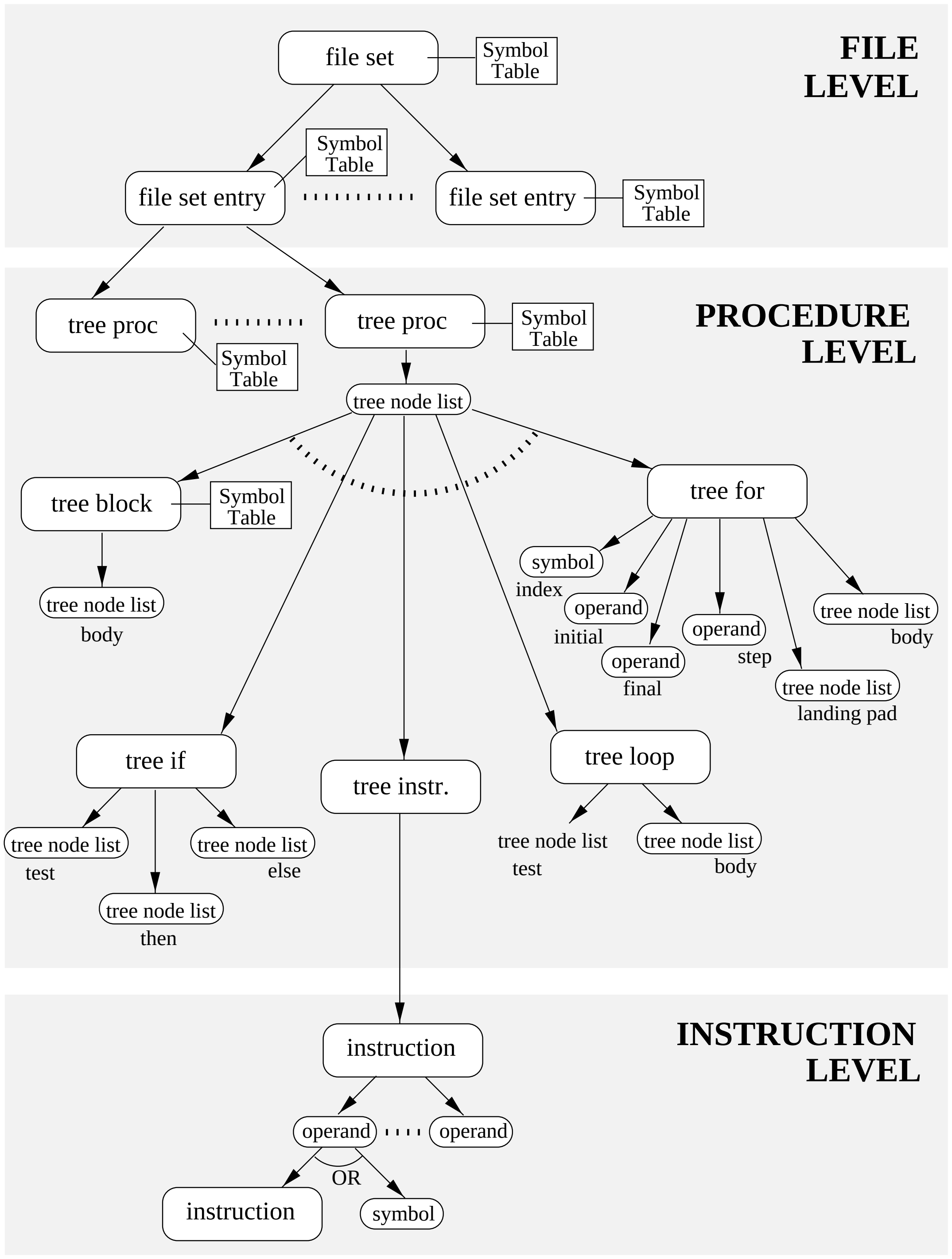


Figure 7.1: SUIF Intermediate Representation

available. This low-level representation dismantles the AST, but the list of instructions is maintained using the same tree data structures that encapsulate the high-level AST. A procedure can contain a mixture of high-level and low-level representations. There are five types of nodes within the SUIF tree data structures, which are shown in the middle shaded region of Figure 7.1: `block` nodes, `if` nodes, `for` nodes, `loop` nodes, and `instruction` nodes.

Block nodes represent nested scopes. Each `block` node contains a symbol table and a list of AST nodes. A procedure is actually a special form of a `block` node.

If nodes model high-level conditional statements. An `if` node consists of three lists of AST nodes. One list contains the nodes representing the code that evaluates the condition and branches to the appropriate program location. Another list contains nodes making up the code of the "then" part. The last list similarly comprises the code of the "else" part of the conditional.

For nodes in SUIF model well-behaved types of loops. A `for` node is a loop that uses scalar indices varying from their initial to final values being incremented or decremented on each iteration. The `for` node is intended to capture loops like the typical Fortran `DO` loop. These types of loops are distinguished because many optimizations require these characteristics. `For` nodes contain a list of AST nodes representing the code of the loop body, expressions for the initial, final, and step calculations, the index variable symbol, and the comparison operator. A *landing pad* list of AST nodes is also maintained as a place for loop-invariant code.

Loop nodes represent any type of loop that fails to meet the conditions for a `for` node. Each `loop` node contains two lists of AST nodes, one holding the code that evaluates the loop test and another for the code of the loop body. SUIF `loop` nodes represent bottom-tested loops. This means that a top-tested loop in the input program is transformed into a conditional with the loop body as part of the conditional.

Instruction nodes are the leaves of the SUIF trees. An instruction tree node serves as the link to the next level of the SUIF hierarchy. This last level is the bottom shaded region of Figure 7.1.

Most SUIF instructions use a quadruple format with an opcode, destination operand and two source operands. However, there are a few instructions with specialized formats, which capture high-level information. For example, the SUIF call instruction (`cal`) encapsulates a list of parameters in order to hide particular, machine-dependent linkage conventions, and the `array` instruction computes the address of an element in an array. It contains the array name, a list of index values, and known dimension bounds. An array instruction eventually expands into a series of individual add and multiply instructions, which compute the element address.

7.1.1.2 Symbolic information

A symbol table is attached to each element of the SUIF hierarchy that defines a scope, and the symbol tables themselves form a mirrored tree hierarchy. The global symbol table is at the root of this hierarchy; it is attached to the root of the main SUIF hierarchy, the file set. Each file in the file set has a symbol table whose parent is the global symbol table. The file symbol tables contain child links to procedure symbol tables that contain child links to block symbol tables, which may be nested to any level.

A symbol table contains a list of symbols and a list of types that are defined within the corresponding scope. Variable symbols have a name and a type, which is just a pointer to an element of a symbol table type list. Several pre-defined flags are associated with variable symbols to quickly identify variables used as formal parameters or variables that may be accessed by their memory address (e.g., through a pointer containing the memory address of the variable). The SUIF type system is capable of representing high-level user-defined types.

7.1.1.3 Annotations

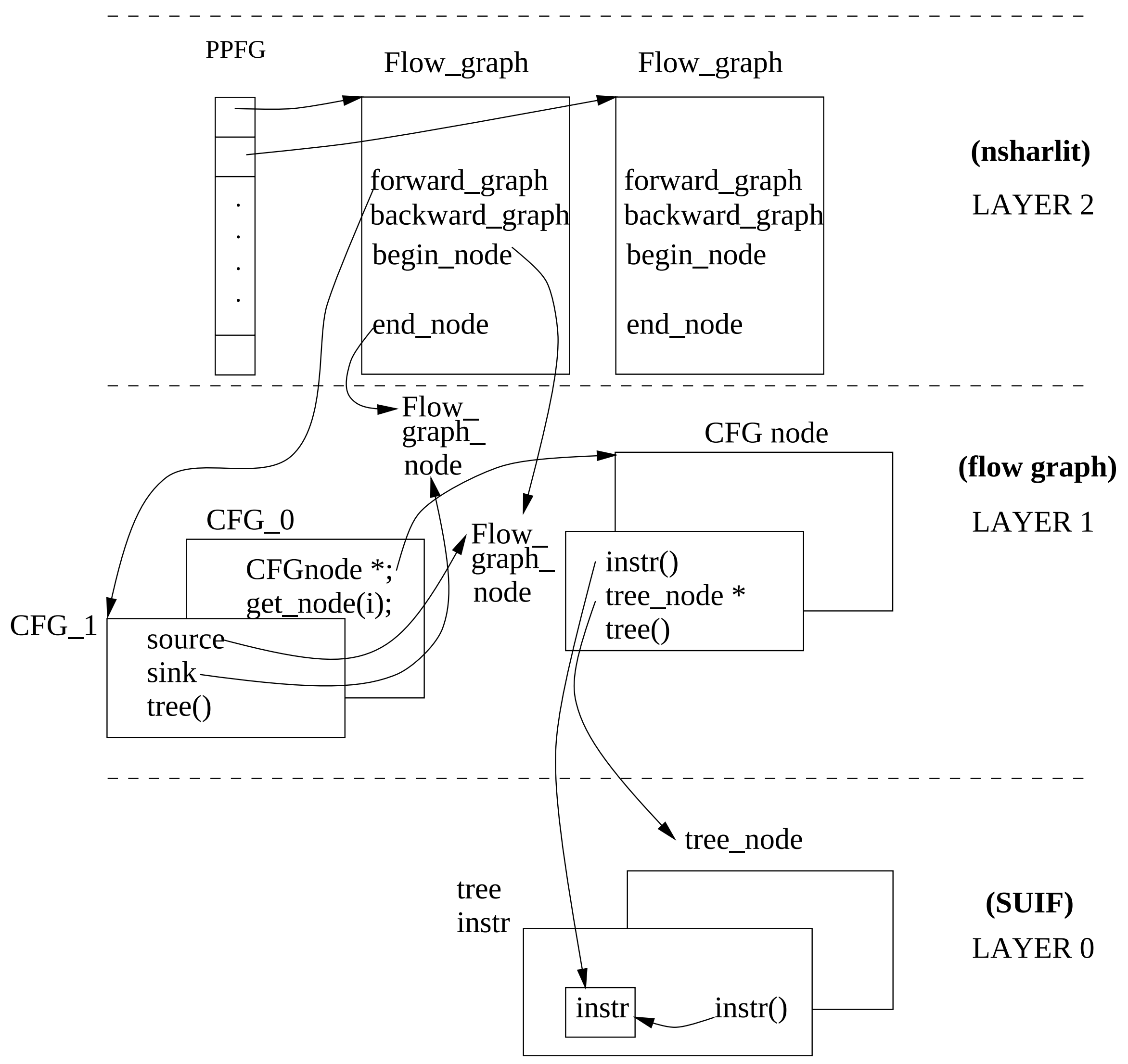
Providing for extensibility, the SUIF system allows new analyses to encode analysis-specific information directly within the SUIF intermediate representation. Such information is generically called an annotation. Annotations can be attached to any object in the SUIF intermediate representation. Tree nodes, instructions, symbols, types, symbol tables, file set entries, and the file set itself can all be annotated. Furthermore, an object can have any number of annotations. An annotation consists of a name and some type of user data. A *structured* annotation contains a user-defined data structure. A *flat* annotation contains of list of data elements, and the elements can be data of different types including character strings, integers, SUIF objects, etc.

7.1.1.4 PPFG based on SUIF

The PPFG is constructed by first constructing a control flow graph for each process. Then, the set of matching synchronization nodes is stored in a table along with the PPFG. Hence, a PPFG is not actually constructed. Only the control flow graphs and the table for storing information concerning synchronization edges are constructed.

Figure 7.2 illustrates the layered structure of the SUIF infrastructure. In this figure, a rectangle R_j that overlaps on top of another rectangle R_i represents a child class. That is, R_j is a child of R_i. For example, a `Flow_graph_node` in Layer 1 is a child class of a `CFGnode`, and a `tree_instr` in Layer 0 is a child class of the `tree_node`.

At the bottom of this infrastructure, the SUIF run-time library provides the basic function of creating an AST and computing the dominator and post-dominator trees. In the middle, a control flow graph is built on top of the AST. At the top of the infrastructure, the user can use nsharlit to implement data flow analysis techniques. At this level, a node in the CFG is referenced by a high level name such



Legend: A rectangle represents a class definition.
A variable name within a class is a data member.
A procedure name within a class is a method.

Figure 7.2: SUIF Layered Structure

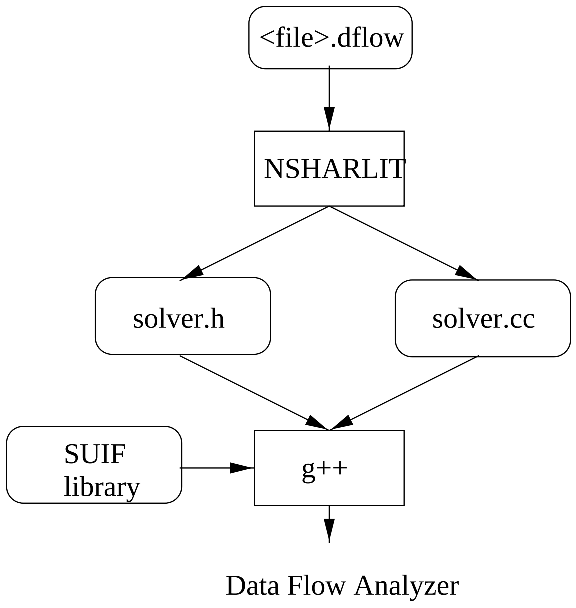


Figure 7.3: Generating a Static Analyzer Using Nsharlit

as a BEGIN_NODE, END_NODE, IF_NODE, etc. To create a PPFG, an array of pointers is created in Layer 2. Each element in the array points to a flow graph for each thread.

7.1.2 Du-pair Computation

The static analyzer performs a reaching definition analysis to find du-pairs using an iterative approach [5]. The static analyzer applies intraprocedural data flow analysis iteratively until all reaching definitions reach a fixed point. Then, another pass will iteratively propagate reaching definition information across synchronization edges, if any, to another process until all reaching definitions reach a fixed point again. Grunwald and Srinivasan describe details of the data flow equations and algorithms of this methodology [50].

Figure 7.3 shows an example of using nsharlit to generate a data flow analyzer. An input file containing the information such as the transfer function, the meet operator, the direction of information propagation, the IN and OUT sets are passed to the nsharlit tool. A header file "`solver.h`" and a C++ program file "`solver.cc`" are generated automatically by nsharlit. The C++ program `solver.cc` is then compiled and linked with the SUIF run-time library; the final executable is a data flow analyzer.

The nsharlit framework is illustrated in Figure 7.4. To start the propagation of data flow information, a user program calls the function `n_solve(num)` from a procedure level routine. The `n_solve()` routine will then iteratively propagate data flow information throughout all nodes in a control flow graph for each procedure until either the value of data flow information comes to a fixed point, i.e., all values in all nodes stay unchanged, or the maximum number of iterations `num` is reached.

At the beginning of the routine `n_solve()`, the internal data structures are initialized. For each instruction, an instance of the set of flow function class is created. For each instruction, an instance of the base flow function, is created. Later, for each instruction, an actual flow function is created using this base flow function as the parent class.

The next step is the iteration stage. During the iteration stage, the flow function defined in the nsharlit input file will be executed to compute the *OUT* set using the *IN*, *GEN*, and *KILL* information [5]. Then, the routine `copy_var()` will be invoked to check whether or not the data flow value(dfv) in the *new variable* is the same as that in the output flow variable. If there is any change, a `changed` flag is set; the data flow value in `dfv` will be copied into the output flow variable.

At the end of the iteration, the routine `n_propagate()` is called to allow several user action routines to be invoked after the data flow information reaches a fixed point.

In our implementation of du-path computation, the reaching definitions are then copied across threads via synchronization edges, if any. The `n_solve()` routine

Input: A CFG of a thread and the maximum number of iterations num.
Output: reaching definitions.
Method:

```
 // Initialize internal information
  Initialize the IN, OUT, and the address of
      flow() for each node in the CFG;
 // n_iterate()
  Create a place holder, called dfv for storing data flow value when
      propagating data flow information through each node;
  While ( changed and number of iterations < num )
  {
      for (i = 0; i < totalnodes; i++)
      {
          if the node[i] is a normal CFG node
             if (current.IN != dfv.IN) // call copy_var()
                set changed to true and copy dfv.IN to current.IN;
          else if node[i] is a Header node
              {
                  call meet();
                   set changed to true and copy dfv.IN to current.IN;
              }
          call the transfer function for this node;
       }
        increment the number of iterations;
   }
 // n_propagate()
    for (i = 0; i < total nodes; i++)
     {
         if node[i] is a Header node
            meet();
         if node[i] is a normal CFG node
            in_action();
        call the transfer function for this node;
        call out_action();
     }
```

Figure 7.4: nsharlit Framework **n_solve()**

is then called recursively to propagate reaching definitions within the CFG of each thread until the reaching definition information reaches a fixed point again or the total number of iterations specified by a user program is reached.

7.1.3 Du-path Coverage and Classification

After all du-pairs are computed, the analyzer applies the algorithm presented in Chapter 5. to find a path coverage for a given du-pair. The tool **della pasta** also provides the function of determining the path type for a given du-path coverage using the algorithm illustrated in Figure 5.3.

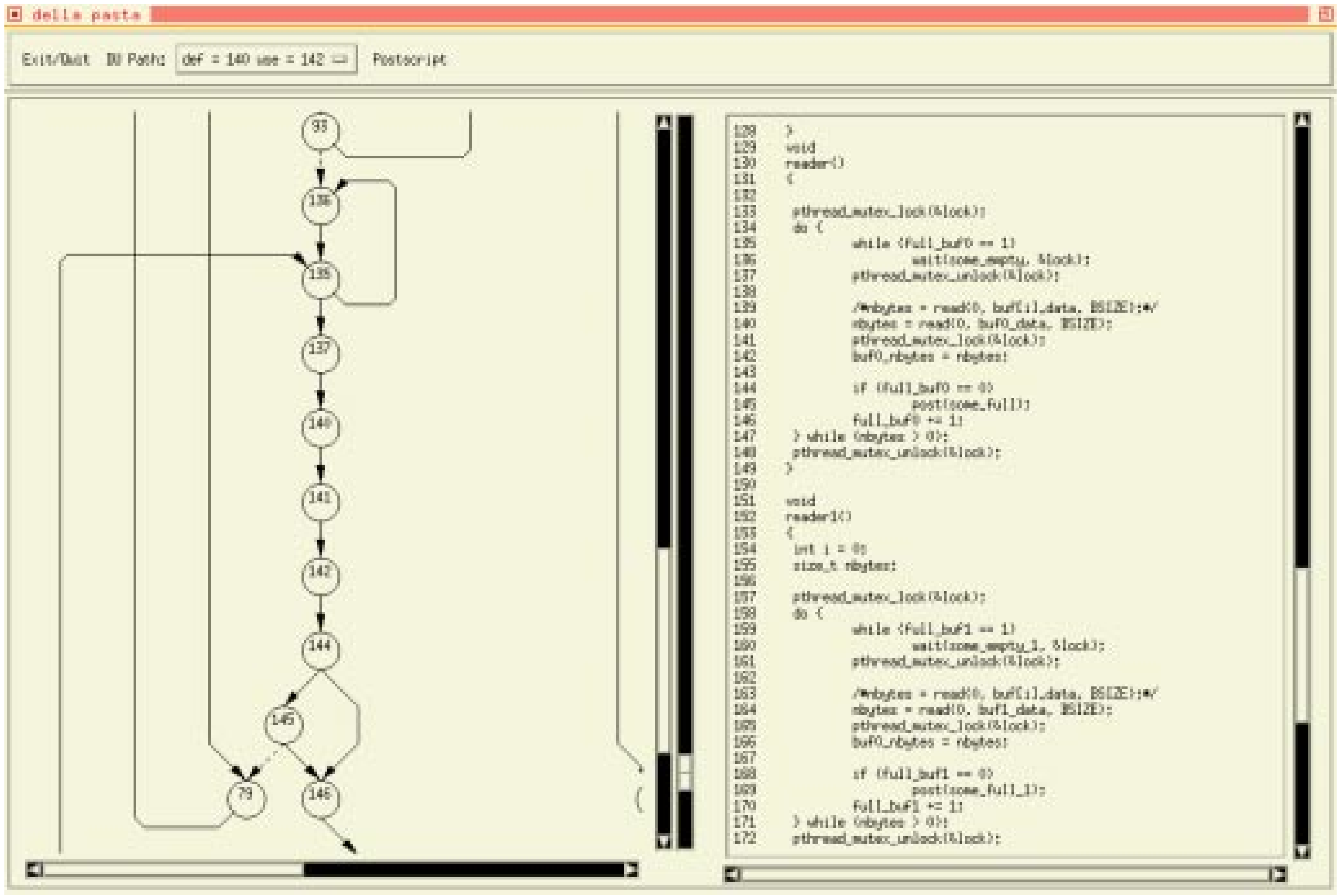
7.2 Path Visualizer Illustrated

The Path Visualizer takes as input two data files generated by the static analyzer for displaying information including the program, the du-pairs, and the du-paths. The two files contain (1) the source level PPFG, and (2) the du-pairs and du-paths in the PPFG.

The path visualizer is built on top of *dflo* which is a data flow equation visualization tool developed at Oregon Graduate Institute [1]. In our implementation, *dflo* is only used for displaying the PPFG and a path coverage; *dflo* is not used for the visualization of data flow equations.

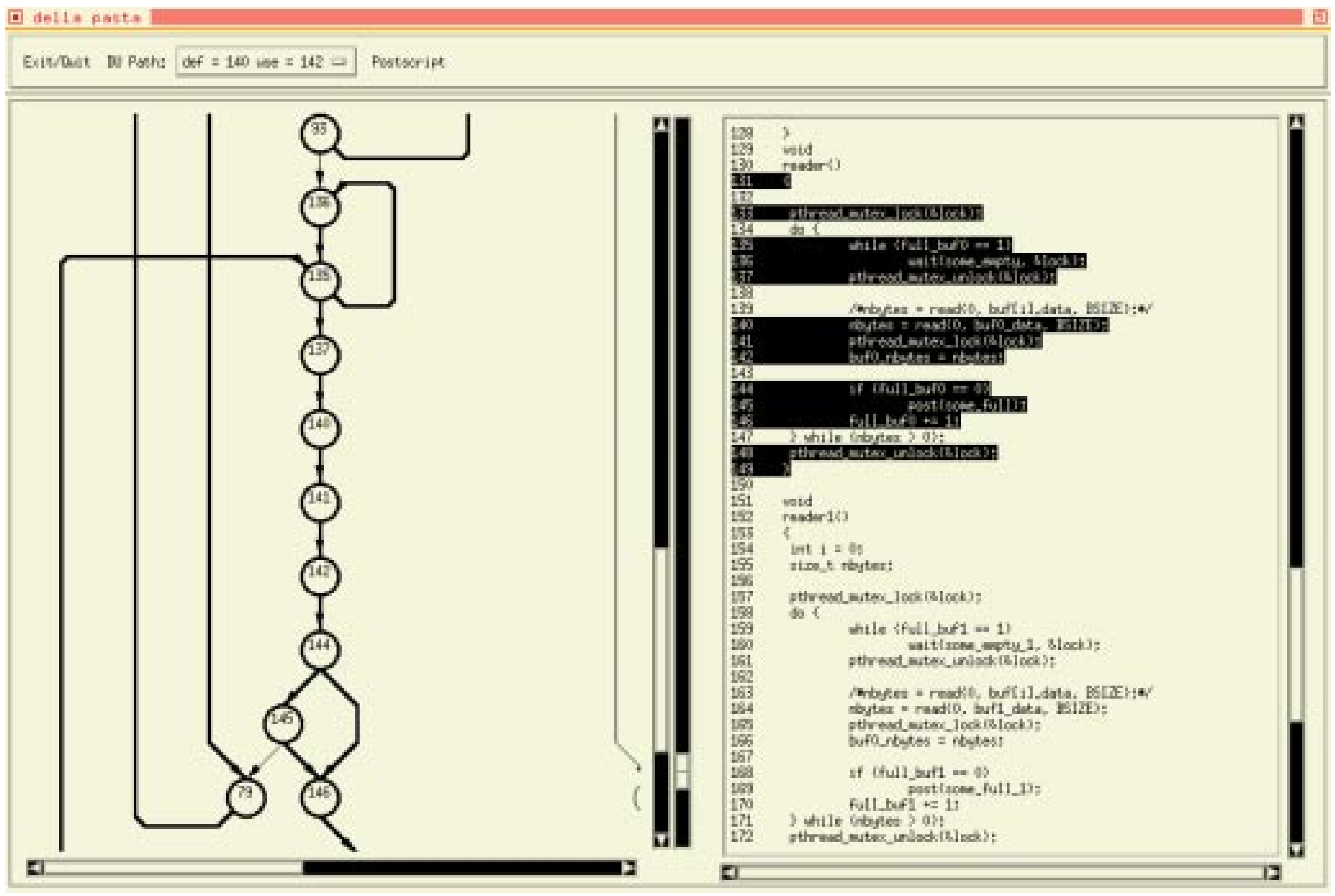
The user interface of **della pasta** is illustrated in Figures 7.5, 7.6, and 7.7. In these figures, a reader/writer program is illustrated in which the main thread creates three additional threads: two readers and one writer. The main thread then acts as one writer itself and communicates with one of the two readers just created. These two pairs of reader/writer threads will work independently in parallel. The du-pair coverage shown in this example only involves two of the 4 threads in the program. In the example program, some system level calls, e.g., `pthread_mutex_lock()` and

[1] This tool can be downloaded from the Internet. Refer to the web site http://www.cse.ogi.edu:80/Sparse/dflo.html for details.



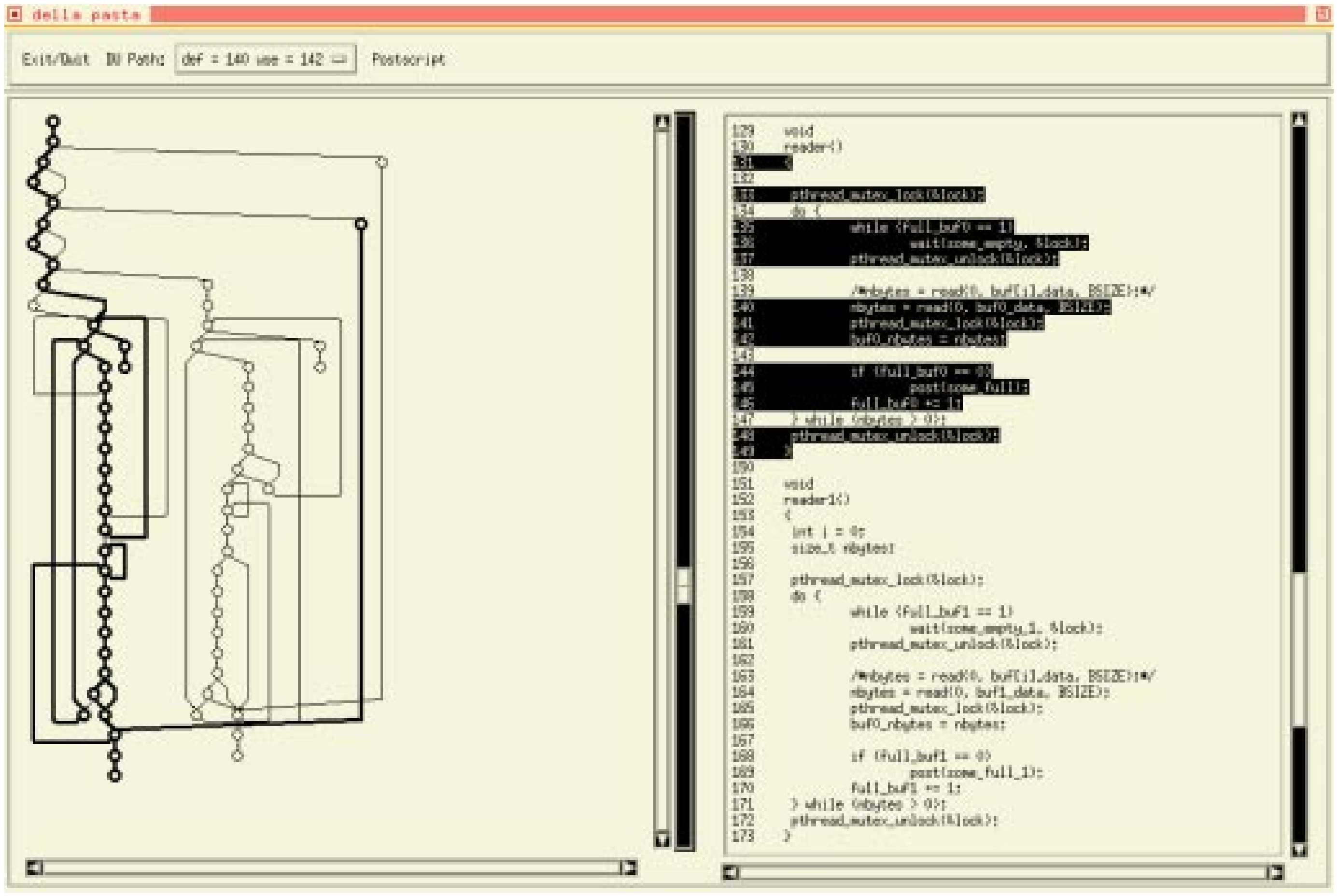
```
128   }
129   void
130   reader()
131   {
132
133    pthread_mutex_lock(&lock);
134    do {
135            while (full_buf0 == 1)
136                    wait(some_empty, &lock);
137            pthread_mutex_unlock(&lock);
138
139            /*nbytes = read(0, buf[i].data, BSIZE);*/
140            nbytes = read(0, buf0_data, BSIZE);
141            pthread_mutex_lock(&lock);
142            buf0_nbytes = nbytes;
143
144            if (full_buf0 == 0)
145                    post(some_full);
146            full_buf0 += 1;
147    } while (nbytes > 0);
148    pthread_mutex_unlock(&lock);
149   }
150
151   void
152   reader1()
153   {
154    int i = 0;
155    size_t nbytes;
156
157    pthread_mutex_lock(&lock);
158    do {
159            while (full_buf1 == 1)
160                    wait(some_empty_1, &lock);
161            pthread_mutex_unlock(&lock);
162
163            /*nbytes = read(0, buf[i].data, BSIZE);*/
164            nbytes = read(0, buf1_data, BSIZE);
165            pthread_mutex_lock(&lock);
166            buf0_nbytes = nbytes;
167
168            if (full_buf1 == 0)
169                    post(some_full_1);
170            full_buf1 += 1;
171    } while (nbytes > 0);
172    pthread_mutex_unlock(&lock);
```

Figure 7.5: della pasta Path Visualizer User Interface



```
128    }
129    void
130    reader()
131    {
132
133     pthread_mutex_lock(&lock);
134     do {
135             while (full_buf0 == 1)
136                     wait(some_empty, &lock);
137             pthread_mutex_unlock(&lock);
138
139             /*nbytes = read(0, buf[i].data, BSIZE);*/
140             nbytes = read(0, buf0_data, BSIZE);
141             pthread_mutex_lock(&lock);
142             buf0_nbytes = nbytes;
143
144             if (full_buf0 == 0)
145                     post(some_full);
146             full_buf0 += 1;
147     } while (nbytes > 0);
148     pthread_mutex_unlock(&lock);
149    }
150
151    void
152    reader1()
153    {
154     int i = 0;
155     size_t nbytes;
156
157     pthread_mutex_lock(&lock);
158     do {
159             while (full_buf1 == 1)
160                     wait(some_empty_1, &lock);
161             pthread_mutex_unlock(&lock);
162
163             /*nbytes = read(0, buf[i].data, BSIZE);*/
164             nbytes = read(0, buf1_data, BSIZE);
165             pthread_mutex_lock(&lock);
166             buf0_nbytes = nbytes;
167
168             if (full_buf1 == 0)
169                     post(some_full_1);
170             full_buf1 += 1;
171     } while (nbytes > 0);
172     pthread_mutex_unlock(&lock);
```

Figure 7.6: Highlighted Du-Path Illustrated



```
void
reader()
{

 pthread_mutex_lock(&lock);
 do {
         while (full_buf0 == 1)
                 wait(some_empty, &lock);
         pthread_mutex_unlock(&lock);

         /*nbytes = read(0, buf[i].data, BSIZE);*/
         nbytes = read(0, buf0_data, BSIZE);
         pthread_mutex_lock(&lock);
         buf0_nbytes = nbytes;

         if (full_buf0 == 0)
                 post(some_full);
         full_buf0 += 1;
 } while (nbytes > 0);
 pthread_mutex_unlock(&lock);
}

void
reader1()
{
 int i = 0;
 size_t nbytes;

 pthread_mutex_lock(&lock);
 do {
         while (full_buf1 == 1)
                 wait(some_empty_1, &lock);
         pthread_mutex_unlock(&lock);

         /*nbytes = read(0, buf[i].data, BSIZE);*/
         nbytes = read(0, buf1_data, BSIZE);
         pthread_mutex_lock(&lock);
         buf0_nbytes = nbytes;

         if (full_buf1 == 0)
                 post(some_full_1);
         full_buf1 += 1;
 } while (nbytes > 0);
 pthread_mutex_unlock(&lock);
}
```

Figure 7.7: Resizing PPFG with Highlighting Illustrated

`pthread_mutex_unlock()`, are needed to ensure the mutual exclusiive accessing of the conditional variables associated with a *post* or *wait* statement as well as variables shared among threads. The execution of these statements in multiple threads could impact the actual execution sequence of statements in different threads. Since temporal testing will be conducted after du-path testing is completed to simulate various sequences of execution, these statements are considered as regular statements in a PPFG.

The command line input of **della pasta** includes three parameters: the source file name, the PPFG file name, and the du-path coverage file name. The tool then displays the PPFG in graphic mode and text mode on two adjacent sub-windows. Figure 7.5 illustrates this window display. On the left of the screen, the PPFG is shown in graphic mode; on the right, the corresponding source code is shown. A user can move both program displays up and down or left and right.

A user can click on the `DU Path` button at the top of the window to select the du-pair to be covered. After the du-pair is selected, the du-pair and and the du-path coverage are highlighted on both sub-windows. Figure 7.6 shows the screen display after the du-pair (140, 142) is selected. A hard copy of the complete PPFG can be printed by clicking the `Postscript` button. As shown in Figure 7.7, a user can resize the data flow graph as desired. The currently selected def-use pair is shown at the top of the screen, and again, the corresponding du-pair path coverage is depicted in the PPFG as well as in the text as highlighted nodes and statements, respectively.

7.3 Path Handler

The third component of **della pasta** is a path handler which will allow a user to adjust a path coverage. In case a $PATH_a$ is generated and a $PATH_w$ exists, a user can make changes to the path coverage. The main reasons for a path coverage to be classified as a $PATH_a$ (and not a $PATH_w$) are unmatching number of iterations of loop statements which are located in multiple threads, respectively, or incorrect branch selected for an *if*-statement. Hence, a user may need to change the total number of iterations of loop statements or select a different branch of an *if*-node. Thus, the Path Handler allows a user to click on a node to delete or add a node to the current path coverage. To change the path coverage associated with a *loop* node, a user will be prompted for the choice of changing the total number of iterations. After a path coverage is finalized, the path analyzer will be invoked automatically to verify the classification of the new path coverage.

These functions are summarized as follows:

- **COVER A NODE** - click on a node to find a path to cover this node. The DT-IT approach will be applied to achieve this functionality. During the process of constructing the path, if there is a possibility that more than one branch can be taken, a user can choose to be prompted for making a choice or let the tool make an arbitrary choice.

- **BEGIN THE ADJUSTMENT OF A DU-PATH COVERAGE** - click on the `BEGIN` button to acknowledge that the current path coverage is the base for later adjustment. Actions including adjusting a branch, adding a node and its post-dominators to the current path (to save some typing and clicking), deleting a node and its successors from the current path, and changing the number of a loop's iterations. All require initiation by clicking on this button. The rationale behind this design is to make the adjustment be an action which can be committed or aborted later. All actions can be disregarded later if a user decides to abort the adjustment.

- **ADJUST THE BRANCH** - change the branch of an if-node to another branch. For a given du-pair, a du-path coverage will be highlighted on the screen first. If the path coverage requires taking a different branch of an *if*-node, a user can change the branch by clicking on the uncovered child node of the *if*-node. The original path will be adjusted to take the user-clicked branch. During the adjustment, if there is a possibility that more than one branch can be taken, a user will be prompted to make a choice.

- **DELETE A NODE** - remove a node and all post-dominating nodes from the current path.
- **ADD A NODE** - add a node and its post-dominators to the current path. Users will be prompted to make a choice at each *if*-node.
- **ADJUST AN ITERATION** - change the total number of iterations associated with a while-node. To achieve this, a sub-window will be displayed after a *loop*-node is clicked. The sub-window will include necessary prompts for modifying the path in which the *loop*-node is included.
- **ABORT THE ADJUSTMENT OF A DU-PATH COVERAGE** - click on the `ABORT` button to abort the adjustment of a path coverage. A user will be prompted to confirm the decision. If confirmed, the original path will be restored. Further adjustment must be initiated by clicking on the *BEGIN* button again.
- **COMMIT THE DU-PATH COVERAGE** - click on the `COMMIT` button to acknowledge that the du-path is the final path coverage for the selected du-pair. After this choice is made, the path type of the new path coverage will be determined and displayed on the screen.

7.4 Summary and Current Status

In this chapter, the design and implementation of the *della pasta* testing tool were described. Currently, the implementation of the path handler is still in progress. Temporal testing will also be incorporated into this tool so that part of the testing process, e.g., inserting delay statements and the compilation of the modified program, can be automated.

Chapter 8

EMPIRICAL STUDIES OF TESTING METHODOLOGIES FOR SHARED MEMORY PROGRAMS

Experimental studies have been conducted by many researchers to investigate the fault detecting capabilities of structural testing methodologies in the context of sequential programs [37, 142, 29, 39, 38]. Also the effectiveness of temporal testing methodologies has been studied in the context of concurrent programs [41]. The major goal of the experimental studies described in this dissertation is to understand the effectiveness of the algorithms for finding a du-path coverage and redundant delay points in shared menory parallel program as well as the characterization of du-paths and redundant delay points in real program. In particular, the empirical studies are designed to investigate the following questions:

1. For a given du-pair, what program characteristics cause a particular path coverage to be classified as $PATH_a$?

2. When the hybrid path coverage finding algorithm returns a path coverage of the type $PATH_a$, are there $PATH_w$ path coverages for the given du-pair that the algorithm did not find?

3. When a $PATH_w$ path coverage is found, are there $PATH_a$ path coverages for the given du-pair, i.e., does the algorithm find a $PATH_w$ when in fact $PATH_a$ path coverages exist?

4. How many delay points are there for each path coverage with respect to each du-pair?

The answer to question one gives insight into possible causes for synchronization errors in a shared memory parallel program. Questions two and three provide clues on what test suites we should expect to generate when the problem of determining whether or not a $PATH_w$ exists has been proven to be NP-C. The answer to question four provides insight into the number of delay points and thus the cost of temporal testing and the need for eliminating redundant delays.

The remainder of this chapter is organized as follows. Section 8.1 describes the state of the art in experimental studies of testing methodologies for sequential and concurrent programs. Section 8.2 describes the experiments conducted to study the questions posed above. Section 8.3 summarizes lessons learned from the experimental studies.

8.1 Others' Experimental Results on Testing Methodologies

8.1.1 Sequential Programs

Many researchers have studied the fault detection capabilities of structural testing methodologies by experimentally comparing various methods [37, 142, 39, 38]. Frankl and Weiss [37] performed an experimental study to compare the effectiveness of the all-uses and all-branches adequacy criteria. A large number of test sets were randomly generated for each of nine Pascal programs with subtle errors introduced. For each test set, the percentage of executable paths for covering an edge and a du-pair, respectively, were measured. Their results indicated that all-uses was significantly more effective than all-branches for five of the programs, and appeared to guarantee to detect the error in four of them. Further analysis showed that in four of these target programs, test suites that are all-uses adequate were more effective than all-statements test suites of similar size. The error exposing capability was shown to be strongly related to the percentage of covered du-pairs in only four of the nine programs. The error exposing capability was also shown to be strongly related to the percentage of covered branches in four (different) programs.

Weyuker [142] performed an experimental study on the cost and effectiveness of data flow testing criteria. Fifteen Pascal programs taken from a suite of numerical programs were used for the study. These programs appeared originally in several issues of the *Communications of the ACM*, and are available as a separate entity known as the *Collected Algorithms from ACM* [1]. The experimental results indicated that 71 percent of the known faults were exposed by all-du-paths, and 67 percent were exposed by all-uses. These results provide empirical evidence for supporting the conclusion that data flow testing techniques are effective for finding program defects.

Frankl and Weyuker [39, 38] compared the relative fault detecting capabilities of data flow testing, mutation testing, and branch testing. Their results showed that most of the criteria were guaranteed to be better than branch testing according to two different probability measures.

In summary, data flow testing criteria including all-uses and all-du-paths have been shown to be effective in many cases. In cases where these testing criteria are not sufficient, other structural testing techniques such as cost-effective analysis and fault-based testing techniques can be used [142].

8.1.2 Temporal Testing of Concurrent Programs

As described in Section 3.3.4, Gait [41] experimentally studied the effects of inserting delay statements into concurrent programs. According to Gait's study, either a non-functioning concurrent program may work with inserted delays, or a functioning concurrent program may stop working when previously embedded implicit delays are removed, relocated, or perhaps changed in value. In particular, a small timing change can expose a large number of errors until the timing change reaches a saturation level. If the saturation level is reached, errors might remain undetected regardless of how the timing is changed. In conclusion, temporal testing can be effective although not conclusive.

8.2 Experimental Studies Targeting Four Research Questions

One of the main issues that a programmer needs to address when writing a shared memory parallel program is the *synchronization problem*. That is, when multiple threads are accessing the same memory locations, some synchronization events are needed to prevent errors such as data race. In general, there are two classes of synchronization problems [74]. The first class of synchronization problems requires the serialization of activities, and the second class requires that one thread must wait for another thread to complete an activity. For example, the dining philosopher problem falls into the first category, while the reader/writer problem(or the producer/consumer problem) falls into the second category. For coding the first class of problems, mutual exclusive locks are adequate, whereas for coding the second class of problems, mutex locks alone are not enough. The algorithms presented in this dissertation, i.e., du-path finding and identifying redundant delay points, mainly deal with the testing of programs addressing the second class of synchronization problems due to the parallel programming paradigm chosen as the focus, i.e., *post* and *wait*.

The following efforts were made to collect unbiased, representative programs for experimental study.

- Requests were posted to the `comp.software.testing` newsgroup on Usenet.
- Email was sent to a well-known researcher in program analysis of parallel programs at IBM T.J. Watson Research Center requesting to use her parallel FORTRAN programs.

- Published research papers, and commercially available articles and programming books were used as a source for program codes also used for their studies.

- A search of Internet archives was performed.

These efforts resulted in three *representative* programs for conducting the experimental study: a producer/consumer program, a pipelining program, and a TCP and UDP name caching service program [74]. These three programs vary in size, number of synchronization calls, and number of condition variables. In short, the producer/consumer program is a medium size(177 lines) program mainly used for testing redundant delays; the pipelining program is a small program(95 lines) used mainly for experimenting with the effectiveness of finding redundant delays; and the name caching program(320 lines) is a complicated program with many synchronization calls for the study of both the du-path finding algorithm and the redundant delay algorithm. Although the number of programs used in the experimental studies is small, these programs represent typical applications of using the mutual exclusion locks and the *post/wait* synchronization operations for controlling the execution sequence of synchronization events.

Since the current prototype **Della Pasta** does not address aliasing and interprocedural analysis, slight modifications to the benchmark programs were made. For example, procedure calls were eliminated by inlining the bodies of the procedures at call sites; some aliases were eliminated by removing some assignments; and the initial assignment of each data definition was also removed due to a possible bug in the SUIF system which was discovered during the empirical studies. However, these modifications should not impact the empirical results because the synchronization constructs were not changed.

8.2.1 Producer/Consumer

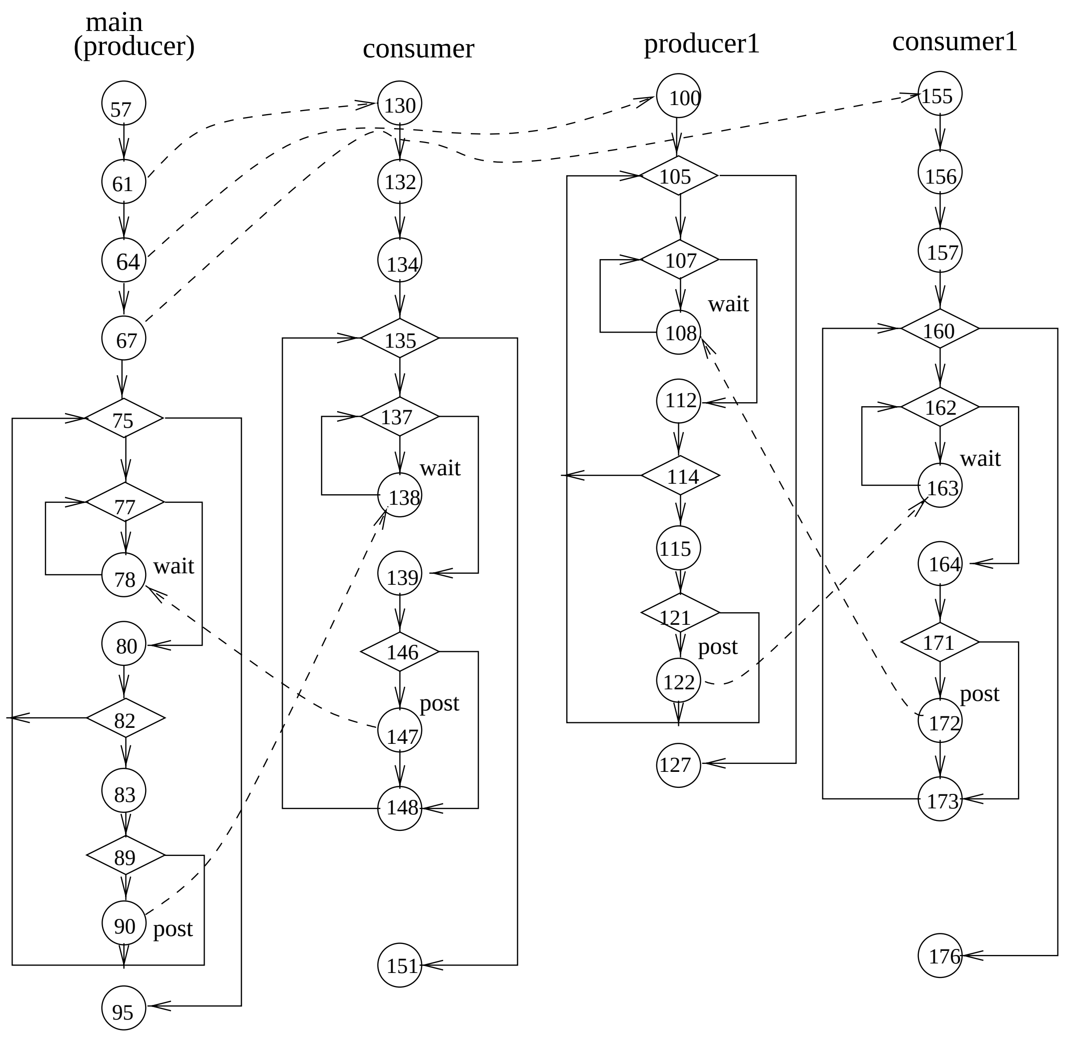


Figure 8.1: A Producer/Consumer Program

```
 thread 1:                          thread 2:

...                                ...
while (full_buf == 0)              if (full_buf ==0)
  wait(some_full, &lock)             post(some_full);
...                                ...
```

Figure 8.2: Pattern of a Matching *post* And *wait* Causing $non - PATH_w$

8.2.1.1 Problem Description

Figure 8.1 illustrates the PPFG of this program. The producer/consumer program generates three worker threads from the producer thread. The producer thread and one worker thread jointly work as a "team" and another two worker threads as another "team" for processing input data. The input data will be randomly received by one of the two producer threads, and the consumer thread associated with the producer thread will then perform computation activities using the data. The producer/consumer program structure has been found extensively in many shared memory parallel applications. For example, a client/server paradigm is a producer/consumer problem. Even the paradigm of software pipelining can be considered another example of the producer/consumer problem. In general, both data parallelism and function parallelism can be achieved using the producer/consumer paradigm. Data parallelism can be achieved with one producer and multiple consumers, whereas function parallelism can be achieved with equal number of producers and consumers. Pipelining is simply a special combination of data parallelism and function parallelism [74].

8.2.1.2 Finding a du-path

In the producer/consumer program, the du-path finding algorithm returns 19 $PATH_w$ and 16 $PATH_a$ path coverages in total with respect to 35 du-pairs. In this benchmark program, the reason that a path coverage becomes $PATH_a$ is mainly due to the coding style of several *wait* events. Figure 8.2 illustrates one matching *post* and *wait* event in this program. If the *wait* is executed twice and the *post* is only executed once, the path coverage becomes a non-$PATH_w$ path coverage. In this case, the path coverage finding algorithm reports a $PATH_a$ path coverage which could potentially cause a possible program fault, while there are indeed no coding errors. In general, if a $PATH_a$ path coverage is found, a synchronization error might exist in the program. However, in this case, the empirical result indeed illustrates a possible "false alarm." Although this programming style caused a non-$PATH_w$ path coverage to be reported in this program, this is definitely not the only possible cause of the du-path finding algorithm finding a non-$PATH_w$ path coverage. Resolving the handling of this situation will not guarantee that the algorithm finds $PATH_w$ path coverages in all programs.

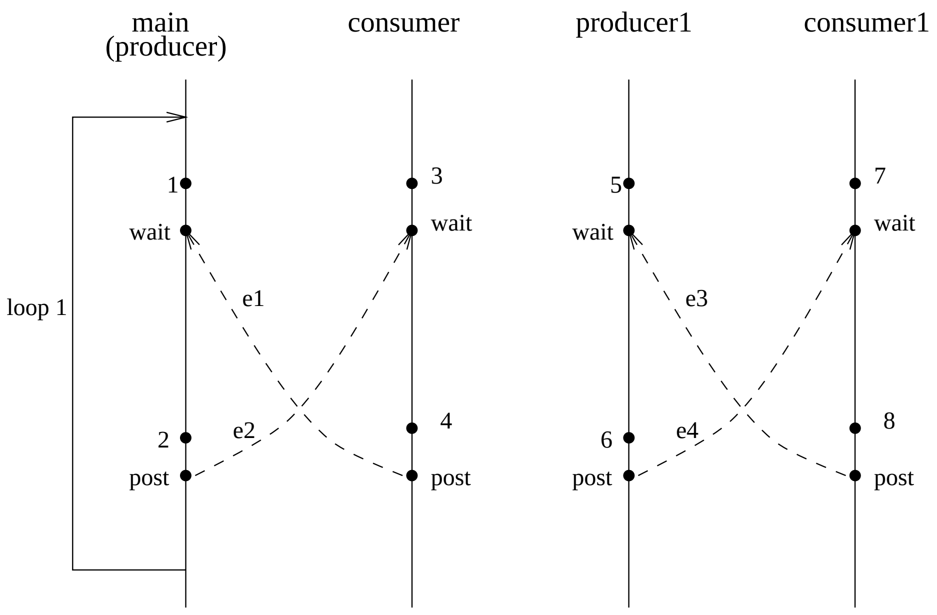
The same path coverages used previously were evaluated again for answers to the second and the third questions listed at the beginning of this chapter. Obviously, the answers to these two questions are both "yes." That is, for a given du-pair, the algorithm could report $PATH_w$ path coverages when $PATH_a$ path coverages exist, and vice versa. For example, to cover the du-pair (x, 83, 139) in Figure 8.1, it is obvious that the two paths 57-61-64-67-75-77-78-77-80-82-83-89-90-75-95 and 130-132-134-135-137-138-137-139-146-147-148-135-151 in the main program and the consumer, respectively, can be classified as part of a $PATH_w$. However, after the define node(83) is covered, a path is also found to cover the *post* node(147). This path in the consumer thread could have covered the *use* node(139). However, another path will have to be found to cover the *use* node(139) despite that the TRN of the *use* node is non-zero. The reason is to ensure that an instance of the *use* node will be covered after an instance of the *define* node(83). The number of iterations of the loop node(137) becomes two. Hence, the TRN of the *wait* node(138) equals two. But the TRN of the matching *post* remains one. A $PATH_a$ is generated.

Ideally, one would like the path finding algorithm to find the expected $PATH_w$ path coverages when they exist. However, the effort to find a $PATH_w$, if one exists, greatly increases the complexity of the du-path finding algorithm due to the need to backtrack on paths already found. For example, in Figure 8.1, if the *post* statement(node 90) is covered twice to match the number of *wait* statement instances(node 138), another *wait* statement (node 78) will also have to be included twice. The matching *post*(node 147) will have to be checked. In our case, the TRN happens to be equal to what we need. However, this may not be the case. Hence, another path must be found to cover the node 147. As a result, the TRN's of other nodes may become unequal to that of their matching nodes. Hence, the effort to maintain an equal number of *post's* and *wait's* complicates the algorithm of du-path finding.

8.2.1.3 Redundant Delay Points

The study of redundant delays leads us to claim two observations. First, since $PATH_a$ path coverages can cause an infinite wait upon execution, they are not considered as targets for finding redundant delay points. Second, for testing each du-pair, a path coverage must be executed at least as many times as the number of delay points. Thus, eliminating any delay point from any path coverage can result in savings in testing time.

In order to compare the results generated by the redundant delay elimination algorithm with the results acquired manually, our study of potential delay points in the producer/consumer program only focused on testing synchronization events; the process creation events, all-uses and all-definitions are not included. For testing synchronization events, 8 delay points are manually determined to be in front of nodes 77 (for testing a *wait*), 89 (for testing a *post*), 137 (for testing a *wait*), 146(for testing a *post*), 107 (for testing a *wait*), 121 (for testing a *post*), 162(for testing a *wait*), and 171 (for testing a *post*).



Note: delay point 1 kills delay point 2, but delay point 3 does not kill 4
since delay point 3 reaches both 1 and 4.

Figure 8.3: A Timing Chart of the Producer/Consumer Program

A timing chart is shown in Figure 8.3. Of the 19 $PATH_w$ path coverages, 18 of them derive 4 redundant delays each. Of the 19 cases, only one derives 3 redundant delay points. The reason is that the execution of the loop edge marked as *loop* 1 is required for covering du-pairs for which *loop* 1 is needed. For example, for covering a variable definition near the delay point 4 and a use near delay point 1, iteration of *loop* 1 is needed. Hence, the timing chart only includes one loop.

In fact, the number of test cases corresponding to the original 8 delay points is 256, the number of test cases corresponding to 5 delay points (after three delay points are eliminated) is 32, and the number of test cases corresponding to 4 delay points (after four delay points are eliminated) is 16. Hence, we concluded that 93% of the temporal test cases can be eliminated for 18 of the 19 du-pairs and 87.5% of the temporal test cases can be eliminated for testing the remaining du-pair.

8.2.2 Pipelining

8.2.2.1 Problem Description

Figure 8.4 shows a PPFG of a simplified software pipelining program. There are three stages in this pipeline. Each stage is represented and performed by one thread. Many applications use software pipelining to achieve higher parallelism. For example, protocol processing on a host machine has been proposed to achieve higher performance using software pipelining [129, 19, 70].

8.2.2.2 Finding a du-path

The result of the du-path finding algorithm indicated that two $PATH_a$ path coverages and six $PATH_w$ path coverages are found. The $PATH_a$ path coverages are generated due to a situation similar to the producer/consumer program. That is, the *loop* 1 shown in Figure 8.5 is executed twice and causes the unbalance of matching *post* and *wait* statements in two different threads.

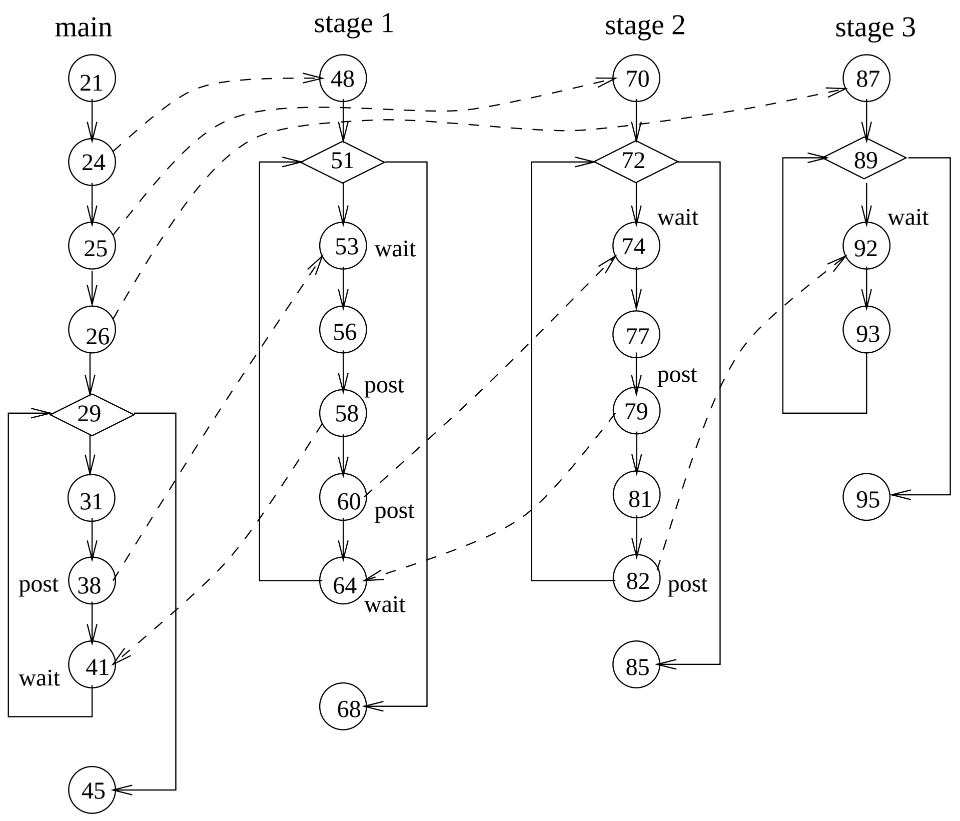


Figure 8.4: A Pipelining Program

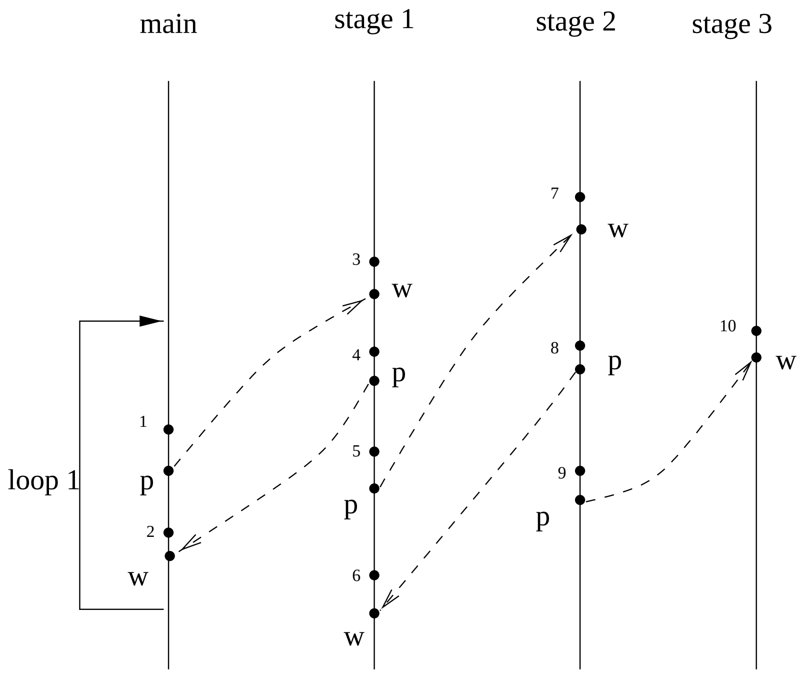


Figure 8.5: A Timing Chart for the Pipelining Program

8.2.2.3 Redundant Delay Points

Figure 8.5 illustrates the relations among delay points in this program. In total, there are ten delay points. Four of these delay points are found to be redundant, and can be deleted. In particular, delay points 4 and 5 are redundant because delay point 3 kills delay point 4, and delay point 4 kills delay point 5; delay points 8 and 9 are redundant due to the relations that delay point 7 kills delay point 8, and delay point 8 kills delay point 9. Hence, the total number of delay points is reduced from 10 to 6. If each temporal test case with various combinations of delay points is executed once, the total number of test cases equals at least $2^n - 1$, where n is the total number of non-redundant delay points. Hence, the total number of test cases in this example is changed from $1023(2^{10} - 1)$ to $63(2^6 - 1)$; the percentage of reduction is 93%.

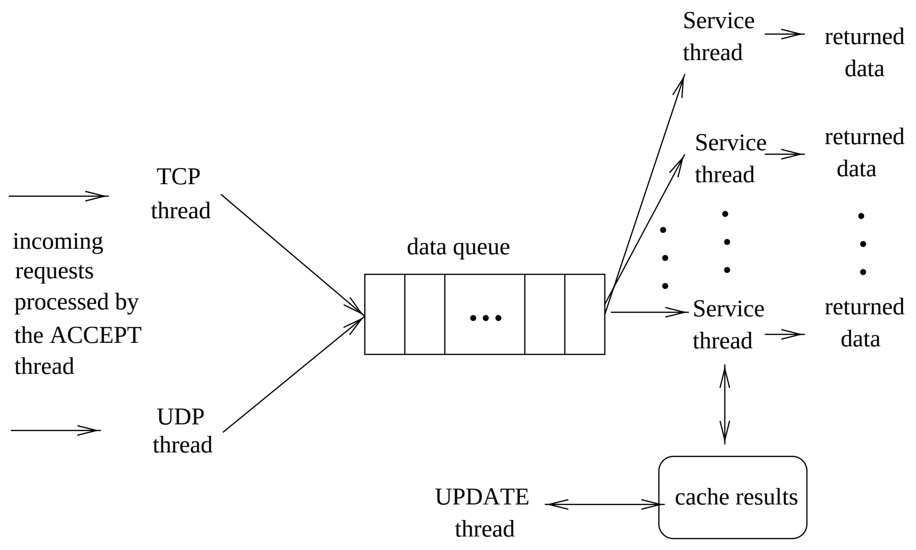


Figure 8.6: A TCP and UDP Name Caching Program

8.2.3 TCP and UDP Name Caching Service

8.2.3.1 Problem Description

Figure 8.6 illustrates the program structure of the TCP and UDP name caching service. When TCP or UDP protocol processing takes place, an IP address, such as "128.175.1.9" is used to identify a location, i.e., a host, in a network; a domain name such as "eecis.udel.edu" must be translated into an IP address for the purpose of processing. Thus, domain names used once are usually stored in a high speed cache memory so that the speed of translating the same domain name can be expedited. In general, the name cache service provided by the TCP or UDP protocol server is usually provided by running a name caching system as a daemon, which is simply a process that will never terminate as long as the machine is up and running. For each domain name translation, there is a service request associated with it. In real practice, multiple translations can trigger more than one thread to be created dynamically. However, in our experiment, only one service thread was created. In general, the name cache daemon has the following design features:

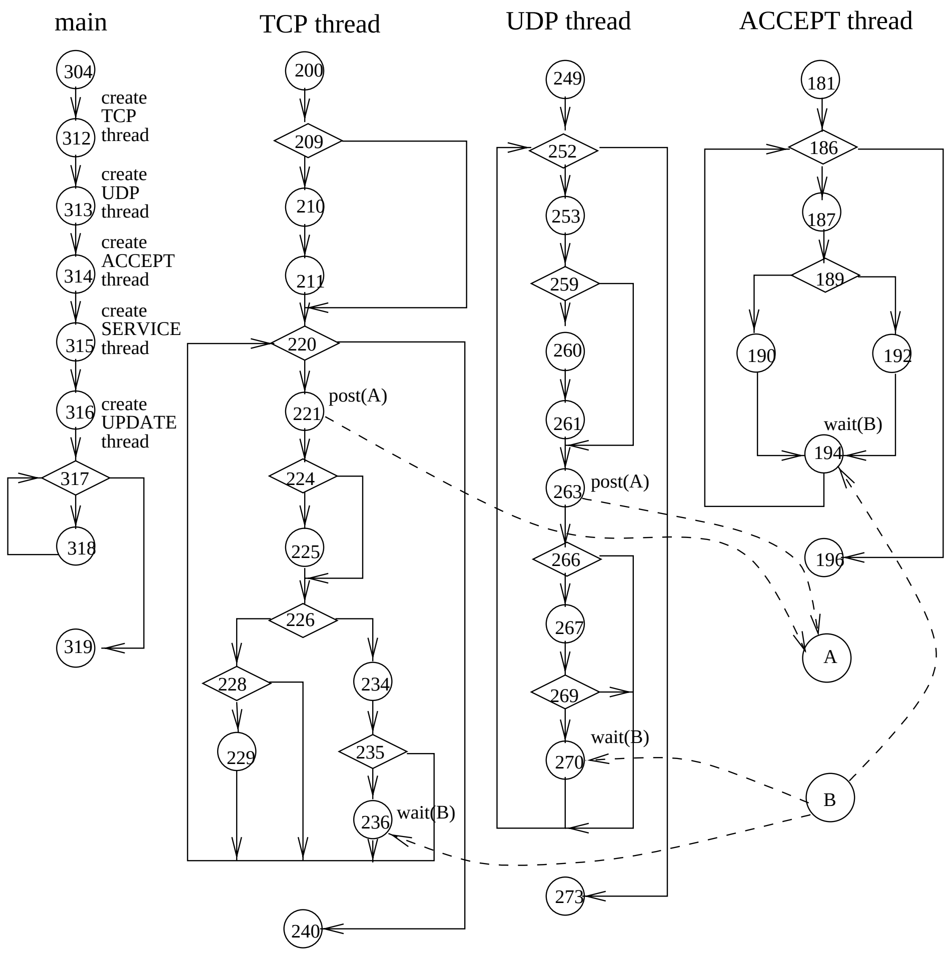


Figure 8.7: The PPFG of the TCP and UDP Name Caching Service Program

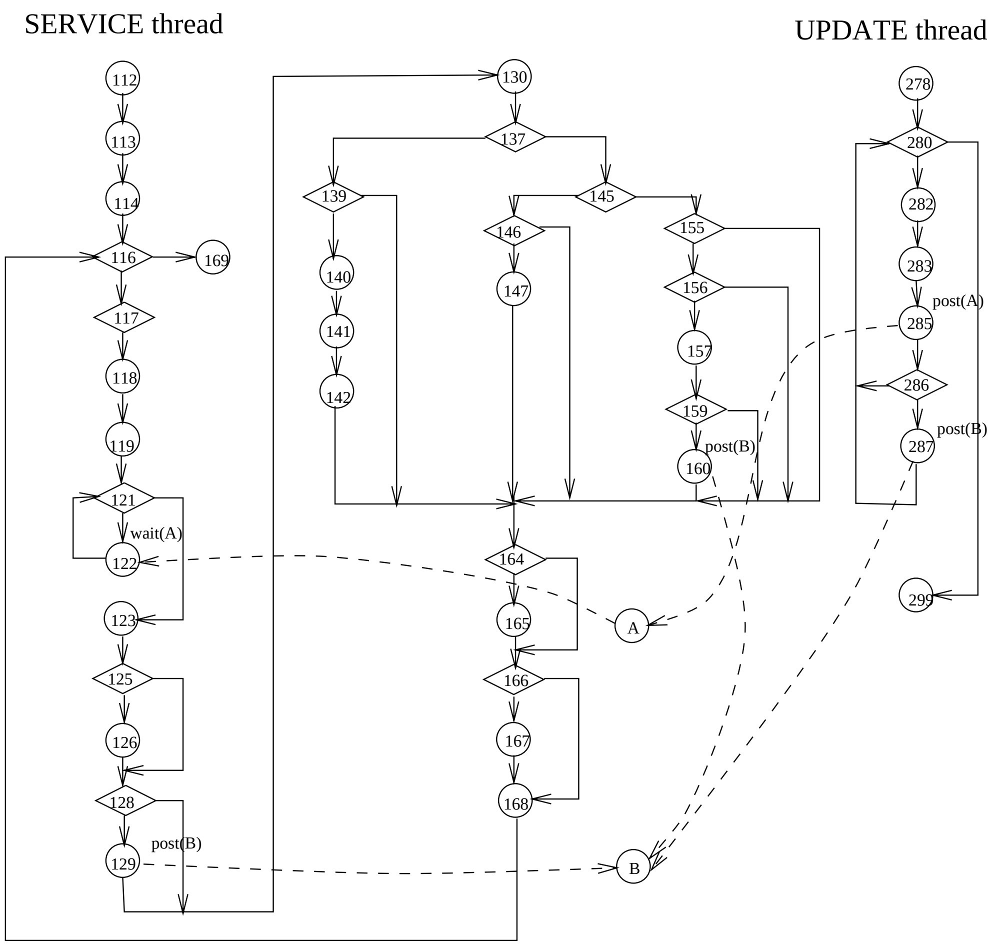


Figure 8.8: (Cont.)The PPFG of the TCP and UDP Name Caching Service Program

- It needs to support multiple types of connections. In this case, both the TCP and UDP sockets are supported.
- It is assumed that the name caching service takes a long time (∼1 second) to complete, so it is important to provide a multithreaded service.

There are many other features and ideas that might be incorporated into a realistic cache daemon [74]. The name caching service is implemented as a multi-threaded daemon for caching the results of service requests associated with many host names. In total, five threads are created by the manager thread including TCP, UDP, ACCEPT, SERVICE, and UPDATE thread. Various SERVICE threads use a single data queue to order service requests. The SERVICE thread performs all the lookup, update, and delete functionalities. The UPDATE thread periodically scans the hash table and refreshes the most commonly used host names and discards any whose time-to-live, i.e., the life-time of a data item, has expired.

Condition Variables	#posts	#waits
sleeper_cv	3	3
wait_cv	3	1

Table 8.1: Number of *post/wait* Nodes

This program is more complicated than the other two benchmark programs in its number of *post* and *wait* statements, and the number of condition variables. Table 8.1 shows the total number of *post* statements and *wait* statements associated with each condition variable used in this program. Figure 8.7 and Figure 8.8 together illustrate the PPFG of this program.

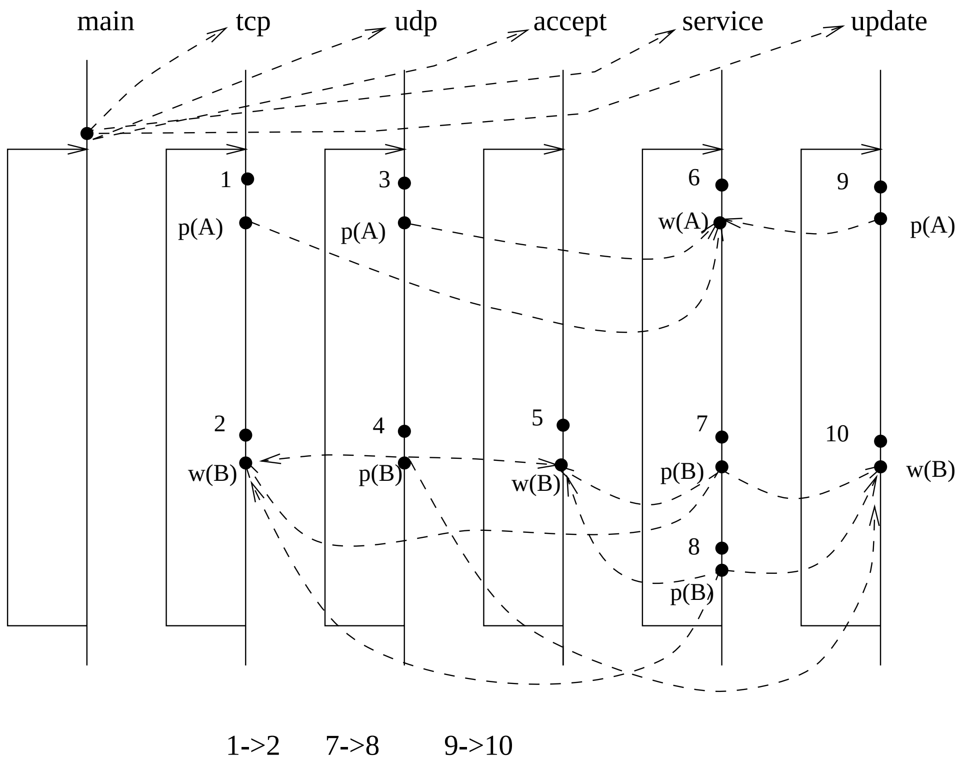


Figure 8.9: A Timing Chart for the Name Caching Program

8.2.3.2 Finding a du-path

The experimental results for du-path finding showed similar behavior to the other two representative programs. Of the 32 du-paths, 22 are $PATH_w$ path coverages. That is, some reported path coverages are $PATH_a$ due to the fact that a *wait* statement was included twice and the matching *post* statement was included only once in the $PATH_a$ path coverage. For some path coverages, there exist $PATH_w$ path coverages when a $PATH_a$ path coverage was reported, and vice versa. The reasons are explained in previous sections.

8.2.3.3 Redundant Delay Points

In the timing chart shown in Figure 8.9, the delay points 2, 8, and 10 can be killed by 1, 7, and 9, respectively. The number of delay points is reduced from 10 to 7; the total number of test cases is reduced from $1023(2^{10}-1)$ to $127(2^7-1)$. Hence, the percentage of reduction in the total number of temporal test cases is 87%.

8.3 Summary

Program	$\#PATH_w$	$\#PATH_a$
Consumer/Producer	19	16
Pipelining	6	2
Name Caching	22	10

Table 8.2: Results of the Path Finding Algorithm

In summary, Table 8.2 lists the number of $PATH_w$ and $PATH_a$ paths found in each of the three programs; Table 8.3 summarizes the experimental results of finding redundant delay points for $PATH_w$ path coverages.

Some conclusions can be made from conducting these experiments. They are described as follows:

Program	#du-paths ($PATH_w$)	#delay points (original)	#delay points (Reduced Set)	% of reduction (#test cases)
Consumer/Producer	18	8	4	94%
Consumer/Producer	1	8	5	87.5%
Pipelining	6	10	6	93%
Name Caching	22	10	7	87%

Table 8.3: Redundant Delay Point Results

1. For a given du-pair, synchronization errors can cause a $PATH_a$ to be generated. However, some coding styles can also cause $PATH_a$ path coverages to be generated.

2. The du-path finding algorithm presented in this dissertation can find a $PATH_a$ path coverage. However, it may return a $PATH_a$ when a $PATH_w$ exists. This result is consistent with the theoretical result. Since we have proved that determining whether or not a $PATH_w$ path coverage exists for covering a du-pair is **NP-C**, we only developed a polynomial time algorithm to find $PATH_a$ path coverages even when $PATH_w$ path coverages might exist. Ideally, when $PATH_w$ path coverages exist, a polynomial time algorithm should be able to report a $PATH_w$ path coverage as specified by a user. However, unless $\mathbf{P} = \mathbf{NP}$, it is impossible. One should only expect to find $PATH_a$ path coverages and manually modify the paths if necessary.

3. The du-path finding algorithm can report a $PATH_w$, if one exists, when a $PATH_a$ also exists. The reason is due to the fact that the algorithm only tries to find a $PATH_a$ and returns the type of the path coverage without making an effort to find a $PATH_w$. The fact that a $PATH_w$ is returned does not guarantee that a $PATH_a$ does not exist.

4. The redundant delay algorithm can aid testers in saving the execution time of temporal testing. The algorithm resulted in more than 85% of reduction on

the total number of temporal test cases in all three benchmark programs.

Chapter 9

CONCLUSIONS AND FUTURE DIRECTIONS

The main focus of this dissertation has been to examine methodology by which structural testing criteria for sequential programs can be extended to shared memory parallel programs. This chapter presents a summary of the dissertation, contributions, and a description of future directions, including extension to other parallel programming paradigms and incremental temporal testing.

9.1 Summary of the Work

Chapter 2 covered background on software testing and parallel programming languages, paradigms, and execution models. The parallel programming model targeted throughout this research was described.

Chapter 3 outlined the specific challenges for testing shared memory parallel programs. Related research in the area of concurrent program testing was described in detail.

Chapter 4 defined the program representation used in this dissertation, and presented the testing framework that lies at the base of the testing algorithms presented in this dissertation.

Chapter 5 examined the problem of finding a du-path coverage for shared memory parallel programs. Two types of path coverages were characterized, and intractability issues were addressed. An algorithm for finding a du-path coverage was presented and analyzed.

Chapter 6 focused on temporal testing for detecting synchronization errors. A method was presented to automatically change the scheduled execution time of synchronization or communication events along paths identified for path testing. The major part of this chapter focused on a method to automatically identify redundant test cases in order to reduce the size of a temporal test suite.

Chapter 7 described the design and implementation of the **PA**rallel **S**oftware **T**esting **A**id(*della pasta*) which allows a user to view a parallel program in a graphic or text mode, and incorporates the algorithms presented in this dissertation.

Chapter 8 described the experiments conducted for exploring the effectiveness of the algorithms presented in this dissertation.

9.2 Contributions

To our knowledge, this is the first effort to extend the notion of du-path coverage to shared memory parallel programs. The key contributions presented in this dissertation include the following:

- Identified the limitations of current path coverage techniques in the context of parallel programs [151].
- Identified the issues to be addressed in order to find a du-path coverage in the context of parallel programs [152].
- Developed an algorithm that successfully finds an acceptable du-path coverage for a given du-pair in a shared memory parallel program [153].
- Defined and characterized the types of du-path coverages.
- Investigated intractability issues related to finding a du-path coverage: proved that the problem of determining the existence of an acceptable du-path coverage is in **P**, and that the problem of determining the existence of a w-runnable du-path coverage is in **NP-C**.

- Outlined a method for generating temporal test cases automatically using redundant delay information.

- Proposed and proved the Redundant Delay Theorem.

- Developed and implemented a static program analysis technique to compute redundant delays in a shared memory parallel program.

- Conducted experimental studies to evaluate the effectiveness of the algorithms presented in this dissertation.

- Designed and developed a software testing tool to demonstrate the techniques developed in this dissertation.

- Demonstrated that the structural testing techniques developed in this dissertation can be modified for message passing and rendezvous communication.

9.3 Future Directions

The du-path coverage algorithm presented in this paper has some limitations. The algorithm requires that a PPFG be constructed statically. Thus, the number of `worker` threads is currently assumed to be known at static analysis time. It is assumed that there are no *infeasible paths* of any kind. Currently, it is assumed that matching *post* and *wait* operations both appear in a program. If a program contains a *post* and no matching *wait* or vice versa, the compiler will report a warning message prior to the execution of our algorithm. In the case where a *clear* operation is used to clear an event before or after the *wait* is issued, our analysis will report more du-pairs than needed. In testing, this only implies that we indicate more test cases than really needed.

Future research work in the area of structural testing of parallel programs can be summarized as follows:

- Extensions to the present methodology should be studied with the goal of eliminating some of the limitations just mentioned. For example, the parallel language model can be expanded to include more language features, such as *parallel do*, *barrier*, or *clear*.

- Extensions to cover def-use pair that involve procedure calls along the paths. In particular, investigate the use of interprocedural program representations, e.g., interprocedural control flow graphs or system dependence graphs [76, 67, 10], in du-path testing work.

- Methods to enable other path coverages using traditional path testing criteria, e.g., all-du-path coverages, should be studied.

- The effectiveness of the structural testing approach should be studied more extensively. Experimental work should be conducted to further understand the effectiveness of the redundant delay identification algorithm.

- Other parallel systems, e.g., concurrent or message passing systems should be considered as targets for structural testing techniques described in this dissertation.

- To effectively deal with source code changes, incremental or regression testing becomes an important area for further investigation. The generation of du-path coverages is generally an expensive task. An incremental test case generation approach could possibly save the computational cost of generating a full test suite again.

In the following two sections, the preliminary design for the extension of the testing methodologies described in this dissertation to other parallel programming paradigms and a closer look at the problem of incremental testing are further described.

9.3.1 Extension to Other Parallel Programming Paradigms

9.3.1.1 Rendezvous Communication

Among other researchers, Long and Clarke developed a data flow analysis technique for concurrent programs with rendezvous communication[86]. Their technique can be applied to compute du-pairs in a concurrent program. After their data flow analysis is performed, a modified version of our du-path finding algorithm can be applied to find a du-path coverage for a concurrent program with rendezvous communication. In particular, modifications are required in the: (1) construction of the PPFG, (2) definition of path acceptability, and (3) du-path finding algorithm.

To accommodate the *request* and *accept* operations in concurrent programs to achieve rendezvous communication, the PPFG needs to include a doubly-directed edge to connect a *request* and an *accept*.

Unlike the *post/wait* shared memory model, if either an *accept* or a *request* is included in a du-path, the matching synchronization calls must be included. This change impacts the definitions of $PATH_a$ and the algorithm to find a $PATH_a$. The definition of a $PATH_a$ must be modified by replacing the second condition in the original definition with the following two conditions:

2a. $\forall$*accept* nodes $a \in_p PATH$, $\exists$ a *request* node $r \in MR(a)$, such that $r \in_p PATH$, and

2b. $\forall$*request* nodes $r \in_p PATH$, $\exists$ a *accept* node $a \in MA(r)$, such that $a \in_p PATH$. where the set $MR(a)$ is defined to be the set of *matching requests for the accept node a*; the set $MA(r)$ is defined as the set of *matching accepts for the request node r*.

The algorithm for finding a $PATH_a$ must be modified only slightly. That is, during the first phase of the algorithm, whenever a *request* **or** an *accept* is found, the matching node must be added into the working queue.

Figure 9.1 shows an Ada program that was used by Weiss [137] to illustrate a

```
procedure P(Z: INTEGER; X, Y: out INTEGER);
        task P0;
        W: INTEGER;
        begin
(1)         W := Z;
(2)         While (Z < W + 2) do
(3)             accept E(V: out INTEGER) do
(4)                  Z := Z + 1;
(5)                  V := Z;
               end;
        end P0;

       task P1;
       begin
(6)         P0.E(Y);
(7)         write (Y);
       end P1;

       task P2;
(8)          P0.E(X);
(9)          write(X);
       end P2;

       begin
          cobegin P0 and P1 and P2;
       end;
```

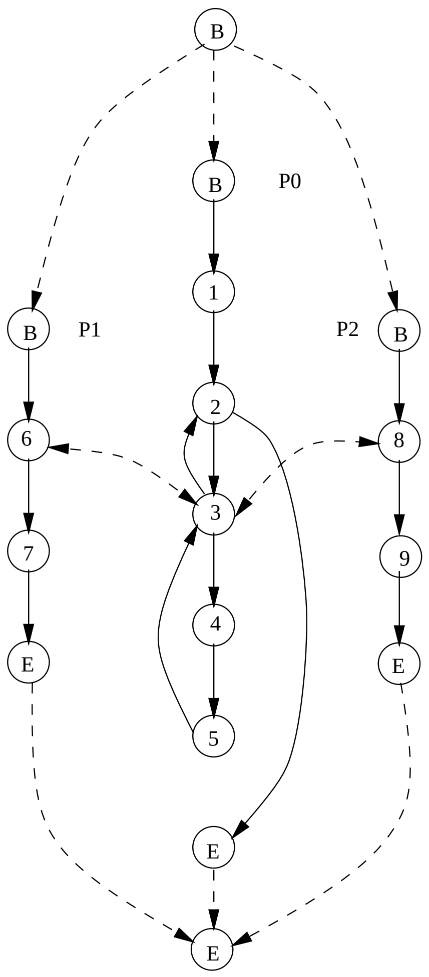


Figure 9.1: An Ada Program

framework for testing Ada programs. The PPFG is also shown. The static analysis technique must be able to find du-pairs including $(V, 5, 3)$, or $(V/Y,\ 3,\ 7)$, using techniques such as interprocedural analysis and alias analysis. The du-path coverage algorithm can then be applied to find a du-path coverage.

The static analysis for finding redundant delay points will find that all delay points can reach both the source and the sink of a synchronization edge since these synchronization edges are always double-directed. As a result, there are no redundant delay points in concurrent programs with rendezvous communication under the notion of delay redundancy defined in this dissertation. Other techniques such as identifying infeasible event sequences [155] could be used to eliminate invalid test cases.

9.3.1.2 Message Passing Programs

To analyze message passing programs, a data flow analysis similar to interprocedural analysis for sequential programs is needed to compute the du-pairs across processes. Several researchers have developed interprocedural reaching definitions data flow analysis techniques in the presence of aliases in C programs [105, 60]. Although this analysis could find du-pairs that may not actually occur during each execution of the program, the reaching definition information is sufficient for program testing. After this information is computed, the algorithm for finding a du-path coverage for a given message passing program can be applied.

To find a du-path coverage for message passing programs, the type of *send* or *receive* operations, i.e., synchronous or asynchronous, must be identified. For example, the Message Passing Interface(MPI)[49] standard library of routines achieves various types of inter-process communication, i.e., synchronous or asynchronous *send* and synchronous *receive* operations. If the *send* operation is synchronous, the definition of a $PATH_a$ must be modified to include both the *send* and the matching

```
        main()
        {
(1)       x = 5;
          ...
(9)       if ( my_process_rank == 0)
(10)        MPI_Send(&x,...);
          else
           {
(20)        MPI_Recv(&x,...);
(21)        y = x + 10;
            }
           ...
        }
```

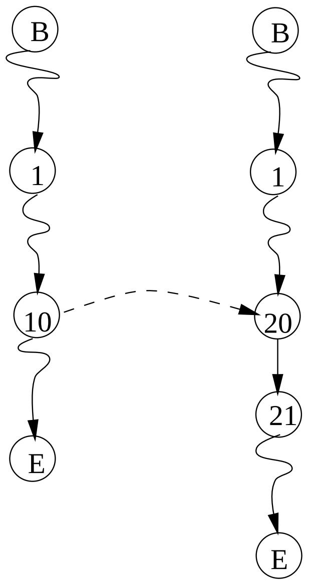


Figure 9.2: An MPI Program

receive in the path coverage similar to the change made for supporting rendezvous-communication parallel programs. If the *send* is asynchronous, the only change is to replace *post* by *send* in this dissertation. For each synchronous *receive* operation, the modification is to replace the *wait* by a *receive* in our algorithms and definitions.

Figure 9.2 illustrates an example MPI program using asynchronous *send* and synchronous *receive* operations. The PPFG needs to include a directed inter-task edge such as $e = (10, 20)$. The SPMD programming paradigm is used to write MPI programs as shown in this example. Thus programs contain special *if*-nodes, i.e., statement 9, that must be distinguished in order to create a PPFG or other representation. In addition, interprocedural analysis and alias analysis must be used to find du-pairs such as $(x, 10, 21)$.

The temporal testing approach for testing shared memory systems can be trivially applied to testing message passing programs if asynchronous *send* and synchronous *receive* operations are used. However, if synchronous *send* and *receive* operations are used, other approaches [155] must be used to find infeasible synchronous events for reducing temporal test cases.

9.3.2 Future Directions in Incremental Testing of Parallel Programs

When a programmer makes program changes, it is very costly to recompute all information required for computing du-pairs, du-paths, and redundant delays. Hence, future work includes investigating an incremental approach to du-path testing and redundant delay computation. There have been two different focuses of research toward generating test cases incrementally to test sequential programs and concurrent programs. The first group focuses on various methods to determine the impact of code changes on the test data that had been used before the programmer's modification [59, 84, 58, 52, 81, 146, 10, 4, 147, 8, 82, 118, 117, 16]. The main target of these research efforts has been the testing of sequential programs. The second group of research has focused on concurrent programs [76]. Instead of generating the test suite incrementally, they generate the program representation incrementally. Their goal is to be able to perform deterministic execution *hierarchically*, that is, to perform deterministic execution of the routines in a bottom-up fashion starting at the lowest level of the calling hierarchy of a program and incrementally testing the program.

With regard to incremental redundant delay computation, the first question to be answered is what possible program changes can a programmer make for which the redundant delay information may need to be adjusted? The following is a complete list of these code changes.

- Insert a Delay - At a program location, a statement such as *sleep(time)* may be inserted to delay the execution of instructions. This action includes creating a delay point after a *post/wait* pair is inserted.

- Delete a Delay - At a program location, a statement such as *sleep(time)* may be deleted to cancel the delay. This action includes deleting a delay point after a *post/wait* pair is deleted.

- Insert a Post - This is for emphasizing the insertion of a *post* within a *post/wait* pair. When inserting a *post*, a delay point is *not* automatically created.

- Insert a Wait - This is for emphasizing the insertion of a *wait* within a *post/wait* pair. When inserting a *wait*, a delay point is *not* automatically created.

- Delete a Post - This is for emphasizing the deletion of a *post* within a *post/wait* pair. When deleting a *post*, a delay point is *not* automatically deleted.

- Delete a Wait - This is for emphasizing the deletion of a *wait* within a *post/wait* pair. When deleting a *wait*, a delay point is *not* automatically deleted.

- Insert an If Statement - An *if statement* may be added to form the *then* part. If the *else* part is added later by inserting other statements inside the alternative to the *then* part, including some *delay statments* or *post/wait* calls, it is *not* referred to as *inserting an if statement.* Instead, it is considered inserting other kind of statements.

- Delete an If Statement - The key words *if-then* are deleted. This makes the original if-then become a simple sequence of statements with no alternative around any of them. Again, the deletion of the *else* part is *not* referred to as the *deletion of an if statement.*

- Insert a Loop - The insertion of the back edge to form a loop is referred to as the *insertion of a loop.* That is, the instructions in the *body* already exist, and just the looping mechanism is added.

- Delete a Loop - The deletion of the back edge connecting the *endloop* to the loop header in a control flow graph is referred to as *deleting a loop.* After the loop is removed, the original body will be executed once with no loop mechanism to repeat the instructions in that sequence.

In order to recompute redundant delays to adjust the test suite, the reaching delays and intrusion sets, i.e., *Reach*() and *Intrusion*(), must be adjusted after the source code is modified. The goal of incremental data flow analysis is not to recompute data flow information all over again from scratch, if not needed [59]. In some cases, we only need to propagate the adjusted information starting from where the code is modified along the control flow edges and synchronization edges, whereas other cases require an initialization phase prior to the propagation of information.

In summary, the future work in the area of incremental testing includes

- the design and implementation of algorithms using a single phase update, as well as a two phase update–the initialization phase and the information propagation phase–for the incremental computation of redundant delay points.
- the design of incremental testing methodology for message passing systems as well as programs using rendezvous.
- the design and study of incremental analysis to support programs with inter-procedural calls, that is, when a subroutine is changed, the sets of reaching delays in the called routine may need to be modified in response to the code change.
- an empirical study of the effectiveness of the incremental approach for testing parallel programs with various flow control statements.

BIBLIOGRAPHY

[1] *Collected Algorithms for ACM, Vol. 1.* Association for Computing Machinery, 1980.

[2] PCF parallel FORTRAN extensions. *FORTRAN Forum*, 10(3):1–57, Sept. 1991.

[3] A. F. Ackerman, P. J. Fouler, and R. G. Ebenau. Software inspection and the industrial production of software. In *Software Validation.* Hans-Ludwig Hausen, 1983.

[4] H. Agrawal, J. R. Horgan, E. K. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 348–357, Montreal, Quebec, Canada, Sept. 1993.

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

[6] V. Balasundaram and K. Kennedy. Compile-time detection of race conditions in a parallel program. In *1989 International Conference on Supercomputing*, 1989.

[7] M. J. Balcer, W. M. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 210–218, Dec. 1989.

[8] S. Bates and S. Horiwitz. Incremental program testing using program dependency graphs. In *Twentieth Annual Symposium on Principles of Programming Languages*, pages 384–396, South Carolina, USA, 1993.

[9] A. Bertolino and M. Marrè. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, Dec. 1994.

[10] D. Binkley. Using semantic differencing to reduce the cost of regression test. In *Proceedings of the International Conference on Software Maintenance*, pages 41–50, Orlando, Florida, Nov. 1992.

[11] G. v. Bochmann and A. Petrenko. Protocol testing: Review of methods and relevance for software testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 109–124, 1994.

[12] J. B. Bowen. Standard errors classification to support software reliability assessment. In *Proceedings of the National Computing Conference*, volume 49, pages 697–705, 1980.

[13] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[14] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, 1988.

[15] R. H. Carver and K. C. Tai. A semantic-based approach to analyzing concurrent programs. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 132–133, July 1988.

[16] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube:a system for selective regression testing. In *International Conference on Software Engineering*, pages 211–220, Sorrento, Italy, 1994.

[17] S. C. Cheung and J. Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, Aug. 1994.

[18] T. Chusho. Coverage measurement for path testing based on the concept of essential branches. *Journal of Information Processing*, 6(4):199–205, 1983.

[19] D. D. Clark and D. L. Tennenhouse. Architecture considerations for a new generation of protocols. In *ACM SIGCOMM*, pages 200–208, Philadelphia, PA,USA, 1990.

[20] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, Sept. 1976.

[21] J. C. Corbett and G. D. Avrunin. A practical technique for bounding the time between events in concurrent real-time systems. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 110–116, Cambridge, Massachusetts, June 1993.

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* McGraw-Hill Book Company, 1990.

[23] H. Custer. *Inside WindowsNT.* Microsoft Press, 1993.

[24] S. K. Damodaran-Kamal and J. M. Francioni. Testing races in parallel programs with an OtOt strategy. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 216–227, Aug. 1994.

[25] A. A. S. Danthine. Petri nets for protocol modelling and verification. In *Computer Networks and Teleprocess Symposium*, pages 663–685, 1977.

[26] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sept. 1991.

[27] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessor systems. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–442, 1986.

[28] E. Duesterwald and M. L. Soffa. Concurrent analysis in the presence of procedures using a data-flow framework. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 36–48, Oct. 1991.

[29] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[30] M. B. Dwyer and L. A. Clarke. A flexible architecture for building data flow analyzers. In *International Conference on Software Engineering*, pages 554–564, Berlin, Germany, 1996.

[31] P. A. Emrath, S. Ghosh, and D. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, pages 69–77, Jan. 1992.

[32] M. E. Fagan. Design and code inspection to reduce errors in program development. *IBM System Journal*, 15(3), Mar. 1976.

[33] D. Flanagan. *Java in a nutshell.* O'Reilly & Associates, Inc., 1996.

[34] V. N. Fleyshgakker and S. N. Weiss. Efficient mutation analysis: A new approach. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 185–195, Seattle, Washington, Aug. 1994.

[35] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept. 1972.

[36] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.

[37] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, Aug. 1993.

[38] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, Mar. 1993.

[39] P. G. Frankl and E. J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, Oct. 1993.

[40] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, SE-2(3):227–231, Sept. 1976.

[41] J. Gait. A probe effect in concurrent programs. *Software-Practice and Experience*, 16(3):225–233, Mar. 1986.

[42] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the International Conference on Parallel Processing*, pages 1355–1364, 1991.

[43] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *International Conference on Software Engineering*, pages 313–319, Imperial College, London, UK, 1985.

[44] R. L. Glass. Persistent software errors. *IEEE Transactions on Software Engineering*, 7(2):162–168, Feb. 1981.

[45] A. GoldBerg, T. C. Wang, and D. Zimmerman. Application of feasible path analysis to program testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 80–94, Seattle, Washington, Aug. 1994.

[46] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[47] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 171–181, Cambridge, Massachusetts, June 1993.

[48] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, Nov. 1983.

[49] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[50] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, California, USA, 1993.

[51] D. Grunwald and H. Srinivasan. Efficient computation of precedence information in parallel programs. In *Languages and Compilers for Parallel Computing*, pages 502–616. Springer-Verlag, 1993.

[52] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression test using slicing. In *Proceedings of the International Conference on Software Maintenance*, pages 299–308, Orlando, Florida, Nov. 1992.

[53] R. Gupta and M. Spezialetti. Towards a non-intrusive approach for monitoring distributed computations through perturbation analysis. In *Languages and Compilers for Parallel Computing*, pages 586–501. Springer-Verlag, 1993.

[54] F. Halsall. *Data Communications, Computer Networks and Open Systems.* Addison Wesley, 1992.

[55] D. Hamlet, B. Gifford, and B. Nikolik. Exploring dataflow testing of arrays. In *International Conference on Software Engineering*, pages 118–129, Baltimore, Maryland, 1993.

[56] M. J. Harrold. *An Approach to Incremental Testing.* PhD thesis, University of Pittsburgh, 1989.

[57] M. J. Harrold and B. A. Malloy. Data flow testing of parallelized code. In *Proceedings of the International Conference on Software Maintenance*, pages 272–281, Orlando, Florida, Nov. 1992.

[58] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *International Conference on Software Engineering*, pages 68–80, Melbourne, Australia, 1992.

[59] M. J. Harrold and M. L. Soffa. An incremental data flow testing tool. In *International Conference on Testing Computer Software*, pages 1–9, Silver Spring, Maryland, 1989.

[60] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2), Mar. 1994.

[61] H.-L. Hausen, editor. *Software Verification.* Hans-Ludwig Hausen, 1983.

[62] D. Hedley and M. A. Hennell. The cause and effect of infeasible paths in computer programs. In *International Conference on Software Engineering*, pages 259–266, Imperial College, London, UK, 1985.

[63] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, Feb. 1985.

[64] D. Hoffman and C. Brealey. Module test case generator. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 97–102, Dec. 1989.

[65] G. J. Holzmann and D. Peled. The state of SPIN. In R. Alur and T. A. Henzinger, editors, *Lecture Notes in Computer Science*, pages 386–389. Springer-Verlag, 1996.

[66] J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 87–97, Oct. 1991.

[67] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[68] G. H. Hwang, K. C. Tai, and T. L. Huang. Reachability testing: An approach to testing concurrent software. In *Asia-Pacific Software Engineering Conference*, page 246255, 1994.

[69] IEEE. *Information Technology - Portable Operating Systems Interface (POSIX) - Part 1:System Application Program Interface (API) - Ammendment 2: Threads Extension.* IEEE Standard 1003.1c, IEEE, New York, N.Y., 1995.

[70] N. Jain, M. Schwartz, and T. R. Bashkow. Transport protocol processing at gbps rates. In *ACM SIGCOMM*, pages 188–199, Philadelphia, PA,USA, 1990.

[71] R. Jasper, M. Brennan, K. Williamson, and B. Currier. Test data generation and feasible path analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 95–107, Seattle, Washington, Aug. 1994.

[72] B. Jeng and E. J. Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology*, 3(3):254–270, July 1994.

[73] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, 1976.

[74] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. SunSoft Press, 1996.

[75] J. Knoop, B. Stedden, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analysis for parallel programs. *ACM Transaction on Programming Languages and Systems*, 18(3):268–299, May 1996.

[76] P. V. Koppol and K. C. Tai. An incremental approach to structural testing of concurrent software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–23, San Diego, California, June 1996.

[77] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, SE-16(8):870–879, Aug. 1990.

[78] B. Korel. A dynamic approach of test data generation. In *Proceedings of the International Conference on Software Maintenance*, pages 311–317, San Diego, California, Nov. 1990.

[79] M. S. Lam. Introduction to the SUIF compiler system. In *First SUIF Compiler Workshop*, pages 1–6, Jan. 1996.

[80] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[81] J. Laski and W. Szermer. Identificaiton of program modifications and its applications in software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 282–290, Orlando, Florida, Nov. 1992.

[82] J. Laski, W. Szermer, and P. Luczycki. Dynamic mutation test in integrated regression analysis. In *International Conference on Software Engineering*, pages 108–117, Baltimore, Maryland, 1993.

[83] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4), Apr. 1987.

[84] H. K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the International Conference on Software Maintenance*, pages 290–301, San Diego, California, Nov. 1990.

[85] R. London. A view of program verification. In *Proceedings of International Conference on Reliable Software*, pages 534–545, 1975.

[86] D. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 21–35, Oct. 1991.

[87] S. P. Masticola and B. G. Ryder. A model of ADA programs for static deadlock detection in polynomial time. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, May 1991.

[88] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[89] S. Morasca and M. Pezzè. Using high-level petri nets for testing concurrent and real-time systems. In H. Zedan, editor, *Real-Time Systems: Theory and Applications, Proceedings of the conference organized by the British Computer Society*, pages 119–131. Elsevier Science Publishings, 1990.

[90] G. J. Myers. *The Art of Software Testing.* John Wiley and Sons, Jan. 1979.

[91] T. Nakajo and H. Kume. A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, 17(8):830–838, Aug. 1991.

[92] K. Naoi and N. Takashashi. Detection of infeasible paths with a path dependence flow graph. *Systems and Computers in Japan*, 25(10):1–14, Oct. 1994.

[93] R. Netzer and B. Miller. Detecting data race in parallel program executions. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 109–129. The MIT Press, 1991.

[94] R. H. Netzer and B. P. Miller. Optimal tracing and replay for debugging a message-passing parallel program. In *Supercomputing*, pages 502–511, Nov. 1992.

[95] R. H. B. Netzer. Trace size vs. parallelism in trace-and-replay debugging of shared-memory programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 616–632. Springer-Verlag, 1993.

[96] S. C. Ntafos and S. L. Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, 5(5):520–529, Sept. 1979.

[97] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, Jan. 1992.

[98] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*, pages 100–107, Baltimore, Maryland, 1993.

[99] L. Osterweil. Depth first search techniques and efficient methods for creating test paths. Technical Report CU-CS-077-75, Department of Computer Science, University of Colorado, Boulder, Aug. 1973.

[100] L. J. Osterweil and L. D. Fosdick. DAVE - a validation error detection and documentation system for FORTRAN programs. *Software Practice and Experience*, pages 473–486, Oct. 1976.

[101] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[102] T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 73–86, Oct. 1991.

[103] T. J. Ostrand and J. E. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4):289–300, Apr. 1984.

[104] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[105] H. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations in C programs. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 139–153, Oct. 1991.

[106] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[107] D. Peled. Combining partial order reductions with on-the-fly model-checking. In R. Alur and T. A. Henzinger, editors, *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, 1994.

[108] D. Peled. Partial order reduction: Model-checking using representatives. In R. Alur and T. A. Henzinger, editors, *Lecture Notes in Computer Science*, pages 93–112. Springer-Verlag, 1996.

[109] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement during function test. In *International Conference on Software Engineering*, pages 287–301, Baltimore, Maryland, 1993.

[110] M. Powell, S. Kleiman, S. Barton, D. Shah, and M. Weeks. SunOS multithread architecture. In *the 1991 USNIX Conference*, 1991.

[111] J. Ramsey and V. R. Basli. Analyzing the test process using structural coverage. In *International Conference on Software Engineering*, pages 306–312, Imperial College, London, UK, 1985.

[112] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, Apr. 1985.

[113] P. T. Revanbu, D. S. Rosenblum, and A. L. Wolf. Automated construction of testing and analysis tools. In *International Conference on Software Engineering*, pages 241–250, Sorrento, Italy, 1994.

[114] D. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 138–153, Seattle, Washington, Aug. 1994.

[115] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, 1992.

[116] M. Roper. *Software Testing.* McGraw-Hill Book Company, 1994.

[117] G. Rothermel and M. J. Harrold. A comparison of regression test selection techniques. Technical report, Clemson University, Oct. 1994.

[118] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 169–184, Seattle, Washington, Aug. 1994.

[119] N. F. Schneidewind and M. Hoffman. An experiment on software error data collection and analysis. *IEEE Transactions on Software Engineering*, 5(3):276–286, Mar. 1979.

[120] M. Spezialetti. *A generalized approach to monitoring distributed computations for event occurrences.* PhD thesis, University of Pittsburgh, 1989.

[121] M. Spezialetti and P. Kearns. Simultaneous regions: A framework for the consistent monitoring of distributed computations. In *Proceedings of the Ninth IEEE International Conference on Distributed Computing Systems*, pages 61–68, 1989.

[122] H. Srinivasan and M. Wolfe. Analyzing programs with explicit parallelism. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 405–419. Springer-Verlag, 1992.

[123] *Solaris Multithreaded Programming Guide.* SunSoft Press, A Prentice Hall Title, 1995.

[124] K. C. Tai. Testing of concurrent software. In *Proceedings of the 4th Annual International Computer Software and Applications Conference*, pages 62–66, Sept. 1989.

[125] K. C. Tai. Predicate-based test generation. In *International Conference on Software Engineering*, pages 267–276, Baltimore, Maryland, 1993.

[126] K. C. Tai. Reachability testing of asynchronous message-passing programs. In *2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 50–61, 1996.

[127] K. C. Tai and S. Ahuja. Reproducible testing of communication software. In *Proceedings of the International Computer Software and Applications Conference*, pages 331–337, Oct. 1987.

[128] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, SE-17(1):45–63, Jan. 1991.

[129] A. N. Tantawy. *High Performance Networks.* Kluwer Academic Publishers, 1994.

[130] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–83, 1983.

[131] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, Mar. 1992.

[132] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–278, May 1980.

[133] M. C. Thompson. An information flow model of fault detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 182–192, Cambridge, Massachusetts, June 1993.

[134] R. H. Untch and A. J. Offutt. Mutation analysis using mutant schemata. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 139–148, Cambridge, Massachusetts, June 1993.

[135] H. Ural and B. Yang. Modeling software for accurate data flow representation. In *International Conference on Software Engineering*, pages 277–286, Baltimore, Maryland, 1993.

[136] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, Feb. 1995.

[137] S. N. Weiss. A formal framework for studying concurrent program testing. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 106–113, July 1988.

[138] S. N. Weiss and V. N. Fleyshgakker. Improved serial algorithm for mutation analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 149–158, Cambridge, Massachusetts, June 1993.

[139] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[140] E. J. Weyuker. The applicability of program schema results to programs. *International Journal of Computer Information Science*, 8(10):387–403, Oct. 1979.

[141] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[142] E. J. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, Sept. 1993.

[143] E. J. Weyuker and B. Jeng. Analyzing partition testing strategy. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[144] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

[145] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 1–10, Oct. 1991.

[146] L. J. White and H. K. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the International Conference on Software Maintenance*, pages 262–271, Orlando, Florida, Nov. 1992.

[147] L. J. White, V. Narayanswamy, T. Friedman, P. Piwowarski, and M. Oha. Test manager:Firewall. In *Proceedings of the International Conference on Software Maintenance*, pages 338–347, Montreal, Quebec, Canada, Sept. 1993.

[148] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.

[149] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? an analysis of some mutation testing issues. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 152–158, July 1988.

[150] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, 6(3):278–286, May 1980.

[151] C.-S. D. Yang and L. L. Pollock. An algorithm for all-du-path testing coverage of shared memory parallel programs. In *Sixth Asian Test Symposium*, Nov. 1997.

[152] C.-S. D. Yang and L. L. Pollock. The challenges in automated testing of multithreaded programs. In *the 14th International Conference on Testing Computer Software*, pages 157–166, June 1997.

[153] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, Mar. 1998.

[154] R.-D. Yang and C.-G. Chung. A path analysis approach to concurrent program testing. In *9th Annual Phoenix Conference on Computers and Communications*, pages 425–432, 1990.

[155] R.-D. Yang and C.-G. Chung. Path analysis testing of concurrent programs. *Information and Software Technology*, 34(1):43–56, Jan. 1992.

[156] D. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 48–54, Dec. 1989.

[157] M. Young, R. N. Tylor, K. Forester, and D. Brodbeck. Integrated concurrent analysis in a software development environment. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 200–209, Dec. 1989.

[158] S. J. Zeil, F. H. Afifi, and L. J. White. Detection of linear errors via domain testing. *ACM Transactions on Software Engineering and Methodology*, 1(4):422–451, Oct. 1992.