A Knowledge Based Tool for Regression Testing

by

Captain T.F. Middelveen, B.Sc. (RRMC), CD

Canadian Armed Forces

A thesis submitted to the school of Graduate Studies in the Department of Electrical and Computer Engineering Royal Military College of Canada Kingston, Ontario

In partial fulfillment of the requirements for the degree Master of Engineering

May 1997

Copyright © 1997 by T.F. Middelveen
This Thesis may be used within the Department of National Defence but copyright for open publication remains the property of the author.
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

0-612-22775-8
ABSTRACT

Much of the total life cycle of software is spent in the maintenance phase, during which a software system may undergo modifications in order to rectify discovered errors or to satisfy new or changing specifications. After these modifications have been implemented, the software is re-tested through the re-execution of predefined software test cases that were generated during earlier phases and the execution of new tests that correspond to the new specifications or new software structure. This regression testing seeks to achieve a degree of confidence that the software continues to perform in accordance with the desired behavior and that the modifications have not introduced errors into previously working portions of the software (regressions).

The approach to regression testing should not simply involve the re-execution of all previous test cases, since not all of the test cases will have been affected and new test cases may also be required. Also, such an approach is not desirable as there is usually a very limited amount of time available to perform the regression test and analyze the results. What is preferred is that the portion of code that is affected by the modifications is sufficiently tested. To achieve this, test cases must be selected from existing ones in accordance with some test coverage adequacy criterion and new test cases developed where required.

A prototype for an automated tool to reduce the effort required for software regression test planning is presented. It is a knowledge based tool which selects test cases for a regression test based on structural coverage criteria and identifies structural elements that remain untested.
Index Terms: Software maintenance, regression testing, knowledge-based systems, structural testing, control flow, data flow, test coverage measures, test case selection, test plan update, test suite size, object-oriented testing.
ACKNOWLEDGEMENTS

Before I acknowledge those who assisted with this thesis, I must first mention Dr. Peter Smart of Royal Roads Military College (RRMC), whose assistance and support during undergraduate studies had much to do with my presence here now.

I would like to thank Dr. Terry Shepard of the Royal Military College (RMC) of Canada, who suggested the initial idea for this thesis, for his support and guidance. I feel I must mention Dr. Fugère (also of RMC) as well, who’s enthusiasm while teaching my introductory course to Artificial Intelligence provided the inspiration to pursue such a topic. A few of my classmates who deserve credit for their assistance are Capt Dan Ferguson for acting as a sounding board as well as for pointing out a few key references relating to the topic, and Lt (N) Claude Bernard, my office roommate, for his editorial assistance with the layout of this document.
DEDICATION

In appreciation of the patience of
my wife Sharon and
my sons Alexander and Robbie.
# TABLE OF CONTENTS

ABSTRACT.......................................................................................................................... ii
ACKNOWLEDGEMENTS...................................................................................................... iv
DEDICATION....................................................................................................................... iv
VITA .................................................................................................................................. v
TABLE OF CONTENTS ........................................................................................................ vi
LIST OF FIGURES ............................................................................................................... vii
LIST OF TABLES ................................................................................................................ ix
ABBREVIATIONS ............................................................................................................. x
TRADEMARKS .................................................................................................................. x

1. Introduction .................................................................................................................. 1
   1.1 Software Testing ........................................................................................................ 1
   1.2 Functional Testing .................................................................................................... 1
   1.3 Structural Testing ..................................................................................................... 2
       1.3.1 Structural Test Coverage Criteria .................................................................... 3
   1.4 Regression Testing ................................................................................................... 5
       1.4.1 The Regression Test Problem ........................................................................ 6
   1.5 Knowledge Based Systems ..................................................................................... 12
   1.6 Thesis Objectives .................................................................................................... 13
   1.7 Thesis Organization ............................................................................................... 14

2. A Knowledge Based System ......................................................................................... 15
   2.1 General .................................................................................................................... 15
   2.2 A Regression Test Knowledge Based Tool ............................................................... 16
       2.2.1 The Prolog Language .................................................................................... 18
       2.2.2 Syntax of Prolog Facts and Rules ................................................................... 19
       2.2.3 Representation of Software Structure ............................................................ 19
       2.2.4 Representation of Test Cases ........................................................................ 20
       2.2.5 Classification of Test Cases .......................................................................... 21
       2.2.6 Analysis of Structural Element Coverage Requirements ............................. 21
       2.2.7 Test Suite Update and Regression Test Planning ........................................... 22
       2.2.8 The RTKBT User Interface .......................................................................... 22
   2.3 Summary .................................................................................................................. 24

3. Control Flow Based Test Coverage Criteria ................................................................ 25
   3.1 McCabe’s Structured Testing Criterion ................................................................... 25
   3.2 The Simple Basis Paths Criterion .......................................................................... 29
   3.3 Comparison of the McCabe and SBP Criteria ......................................................... 32
   3.4 Determining Testcase Coverage of Simple Paths .................................................... 34
       3.4.1 Branch coverage ............................................................................................ 35
   3.5 A SBP Method Demonstration .............................................................................. 35
   3.6 Strengthening the SBP criterion ........................................................................... 48
       3.6.1 Chaining Requirements .............................................................................. 49
       3.6.2 Isolating Requirements ............................................................................... 49
       3.6.3 Demonstration of the Strengthened SBP Method ........................................... 50
3.7 Summary ........................................................................................................... 52
4. Data Flow Based Test Coverage Criteria .......................................................... 53
  4.1 The All Du-paths Criterion ............................................................................ 55
    4.1.1 Determining Testcase Coverage of du-paths ....................................... 60
    4.1.2 Branch Coverage .................................................................................. 61
  4.2 A Data Flow Example ................................................................................... 62
  4.3 Strengthening the All Du-paths Criterion ..................................................... 74
    4.3.1 Chaining Du-path Requirements .......................................................... 75
  4.4 Summary ........................................................................................................ 77
5. Controlling the Size of the Test Suite ............................................................... 79
  5.1 A Test Cases Approach ................................................................................ 80
  5.2 A Requirements Approach .......................................................................... 83
  5.3 Knowledge Base Implementation ................................................................. 84
    5.3.1 Enhancement of the Tie Resolution Strategy ........................................ 86
    5.3.2 Test Suite Update ............................................................................... 87
    5.3.3 Regression Test Set ............................................................................. 88
  5.4 Summary ........................................................................................................ 89
6. A Regression Testing Model Using the RTKBT ................................................ 90
  6.1 Regression Test Process .............................................................................. 90
  6.2 An Example .................................................................................................. 93
    6.2.1 Control Flow ....................................................................................... 94
    6.2.2 Data Flow ......................................................................................... 100
7. A Potential Application of the RTKBT to OO Software Testing ....................... 105
  7.1 State-Based Paradigm for Object-Oriented Software Testing ....................... 105
  7.2 FSM Modeling of Dynamic Behavior ............................................................ 106
  7.3 FSM Diagram Test Coverage ..................................................................... 107
  7.4 Message Sequence Chart Analysis ............................................................... 112
  7.5 Determining Test Coverage Adequacy ........................................................ 115
  7.6 Test Automation .......................................................................................... 116
  7.7 Summary ....................................................................................................... 117
8. Conclusions and Future Work .......................................................................... 118
  8.1 Conclusions ................................................................................................. 118
  8.2 Recommendations for Future Research ........................................................ 120
    8.2.1 Integration with Automated Analysis and Test Tools ......................... 120
    8.2.2 Analysis for Different Test Coverage Adequacy Criteria ..................... 121
    8.2.3 Automated Identification of Infeasible Paths ....................................... 122
    8.2.4 Inter-Module Analysis ........................................................................ 122
    8.2.5 Combination with Functional Testing ............................................... 122
    8.2.6 Implementation of OO Test Coverage Analysis .................................. 123
REFERENCES .......................................................................................................... 124
APPENDICES ........................................................................................................ 127
  Appendix 1 - FORTRAN Source Code for Blackjack ..................................... 127
  Appendix 2 - Power Example Source Code and Test Scripts ......................... 130
  Appendix 3 - Parser for ObjecTime Linear Form ............................................ 145
  Appendix 4 - Device Class Behavior Specification ......................................... 150
List of Figures

Figure 1. Evolution of a Test Suite [Leun89] ................................................................. 11
Figure 2. RTKBT Model .................................................................................................. 16
Figure 3. SBP clause hierarchy ....................................................................................... 31
Figure 4. SBP criterion - best and worst cases ............................................................... 33
Figure 5. Control Flow example - Black Jack ................................................................. 37
Figure 6. Modified portion of blackjack control flow graph ........................................ 42
Figure 7. The all-du-paths clause hierarchy .................................................................. 59
Figure 8. Data flow example - source code .................................................................... 62
Figure 9. Data flow example - flow graph ...................................................................... 63
Figure 10. Data flow graph modification ....................................................................... 70
Figure 11. Hitting set clause hierarchy ........................................................................... 86
Figure 12. Regression Testing Model .............................................................................. 91
Figure 13. Power control flow example .......................................................................... 95
Figure 14. Modified Power example .............................................................................. 97
Figure 15. Power data flow graph ................................................................................ 100
Figure 16. Behavior model for the “device” actor class ............................................... 109
Figure 17. Message Sequence Chart ........................................................................... 114

List of Tables

Table 1. Hitting set example 1 - a test cases approach ................................................... 81
Table 2. Hitting set example 2 - a test cases approach ................................................... 82
Table 3. Hitting set example 2 - a requirements approach .............................................. 84
Table 4. Hitting set example 3 - tie resolution enhancement .......................................... 87
Table 5. Power example test cases ............................................................................... 94
ABBREVIATIONS

AI    Artificial Intelligence
   c-use  compute use
cdu   definition compute use set
def   definition
def-use definition-use
dpu   definition predicate use set
du-path  definition-use path
FSM   Finite State Machine
MSC   Message Sequence Chart
OO    Object-Oriented
PGMGEN Test Program Generation
Prolog a programming language name derived from “programming in logic”
p-use predicate use
ROOM Real-time Object-Oriented Modeling
RTKBT Regression Testing Knowledge Based Tool
SBP   Simple Basis Paths

TRADEMARKS

MS-DOS is a registered trademark of Microsoft, Inc.
ObjecTime is a registered trademark of ObjecTime Limited.
SWI-Prolog is Copyright © 1993-1996 by The University of Amsterdam.
UNIX is a registered trademark licensed through X/Open Company, Ltd.
Chapter 1

1. Introduction

1.1 Software Testing

When a new software system enters the operational (in-service) phase of its life cycle, it should have been thoroughly tested during the previous phases in the software development by a series of test cases called a test suite. The types of test cases contained in a test suite are generally categorized as functional or structural. The functional or ‘black box’ tests check for correct input/output relations of all functional requirements in accordance with the software specification, usually without regard for the internal implementation. Structural or ‘white box’ tests attempt to exercise all structural elements of the internal implementation of the software that have not been exercised by the functional tests. The output produced by the test cases of the test suite are compared with the results expected in accordance with the specification. The expected outputs are usually stored together with the test cases in the test suite. The successful execution of all test cases in the test suite provides the level of confidence that the software performs in accordance with its specification.

1.2 Functional Testing

Functional testing, often called specification-based testing, involves the identification of functional criteria in terms of sets of inputs and outputs that represent the
intended function of a program as described in the requirements specification for the software. Testing of this type is generally referred to as black-box testing (sometimes called opaque-box testing) as the tests are developed without regard for the internal structure of the code. A summary of specification-based testing techniques such as equivalence partitioning, boundary value analysis, cause-effect graphing and design-based testing can be found in [Mack96]. Specification-based test criteria are not explicitly dealt with in this thesis.

1.3 Structural Testing

A prominent structural testing method for procedural programming languages involves the identification of the control structure of a program, described in terms of program language statements, blocks, logical branches or paths. A block is a sequence of statements of which only the first statement may be the destination of a logical branch and only the last statement may be a branching statement. A path is a sequence of statements, blocks or branches. The structure of a program is often represented by a directed graph where each node in the graph corresponds to a block and each arc (edge) corresponds to a possible branch. Such a graph is commonly referred to as a control flow graph.

The development of structural-based test cases involves the derivation of input test data through the examination of the program control structure. A set of test cases is generated such that the test execution paths cover the control structure according to some adequacy criterion. Testing of this type is generally referred to as white-box testing.
(sometimes called clear-box testing) as the tests are developed by examining the internal structure and logic of the program under test [Hart88].

1.3.1 Structural Test Coverage Criteria

The structural test coverage criteria is used to determine if enough test cases have been selected. The determination of the thoroughness of the criteria used will depend on the level of risk involved or the acceptable reliability (e.g. critical control or safety systems will require more thorough test coverage), the time allotted to conduct the regression testing (e.g. in order to deliver the product to market on time or to meet customer demands) and the cost of performing the regression test.

Several control flow based structural test coverage criteria are summarized by [Mill94] as follows:

a. statement (or line) coverage shows which statements in a program have been exercised, but in the case of a line with multiple actions (such as a conditional statement) it may not ensure that all actions have been exercised;

b. branch coverage measures the number of times logical branches have been exercised for each condition; it is a stronger criterion than statement coverage and is used most often for detailed unit testing, but may not test all possible paths (and may not test all possible truth values of compound predicates);

c. call-pair coverage measures the number of times function calls have been exercised and is used for system interface testing; and
d. path coverage measures the number of times each path or path class in a module was exercised; it is more time consuming and generally used for more critical modules but is often unachievable due to the existence of infeasible paths.

Before attempting to test a program in accordance with some structural coverage criterion, an assumption is often made regarding the syntactic structure of the program. A program is considered to be a ‘proper program’ [IBM92] if it has a single entrance node, a single exit node, no unreachable node sets and no unleavable node sets. If all nodes of a program are syntactically reachable and the program contains no syntactically endless loops, then all nodes (and all branches) will appear on some path from the entrance node to the exit node, although possibly an unexecutable one [Rapp85]. Without these assumptions regarding a program’s syntactic structure, it would not be possible to satisfy even the weakest criterion of statement coverage. In this thesis, it is assumed that these types of program anomalies can be detected and reported by a static analysis tool prior to regression testing.

Statement coverage, often called the all-nodes criterion, arises from the idea that no estimate of reliability is possible for any part of a program that is not tested. The stronger all branches (or all-edges) criterion is intended to improve confidence by ensuring that all possible transfers of control associated with conditional statements are executed. The all-edges criterion however, is known to be weak, in that it can be satisfied without executing all possible paths of a program. Often, however, not all paths in a program are feasible, and in the case of programs with loops, there may be an infinite number of
possible paths [Rapp85], so the all-paths criterion is not considered to be a practical testing goal. Many test coverage criteria have been developed to "bridge the gap" [Rapp85] between the relatively weak all-branches criterion and the often unattainable all-paths criterion. In this thesis, two coverage criteria based on Control Flow and Data Flow methods that are intended to bridge the all-branches versus all-paths gap are presented.

The all-branches criterion will be the minimum coverage criterion in this thesis. It requires that every logical branch for each condition is exercised at least once. Regression testing will then require that every modified branch (as determined by static analysis) is exercised at least once.

1.4 Regression Testing

It has been estimated that fifty to seventy-five percent of the total software life cycle is spent in the maintenance (operational) phase [Harr88]. During this phase, a software system may undergo modifications in order to rectify discovered errors or to satisfy new or changing specifications. As these modifications are implemented, the software is re-tested through the re-execution of some of the pre-defined software test cases that were generated during earlier development phases and the execution of new tests that correspond to the new specifications or new software structure. This testing seeks to regain a degree of confidence that the software continues to perform in accordance with the desired behavior (normally described in a specification) and that the
modifications have not introduced errors into previously working functions. Such errors are called regressions [Hump95], hence the name regression testing.

"Most people assumed that regression testing is simply repeating all the tests in a test plan and re-testing all the features of the program." [Leun89] This practice has quickly fallen out of favor since not all of the test cases will have been affected and "in the case of large software systems, the cost of such re-executions would be extremely high in terms of time and resources." [Hart88] Also, such exhaustive re-testing is not desirable since there is usually a very limited amount of time available to perform the regression tests. "In many cases, software development and software fixing take the majority of time in the schedule, so often this means there is little time devoted to testing." [Mill94] It is preferred that only the portion of code that was actually modified or that may be affected is sufficiently tested. To achieve this, test cases must be selected from existing ones (and new test cases developed where required). "Difficulties arise in this approach due to the maintenance analysts' limited knowledge of the system and the rippling effects of the changes in the system." [Harr88]

1.4.1 The Regression Test Problem

The problem of regression testing may be broken down into two sub-problems: the regression test selection problem and the test suite update problem [Leun89]. The regression test selection problem involves the selection of test cases from the existing test suite and the development of new test cases that will adequately test only the affected
functional and structural elements of a modified program. The test suite update problem refers to the management of the test suite through successive software modifications. Certain test cases that become obsolete will be removed and new test cases required to test new functional or structural elements will be added to the test suite. The updated test suite contains test cases that will adequately cover all functional and structural test requirements of a program. The solution to the regression test problem begins with the classification of test cases according to how they have been affected by the software modifications.

The first step in classifying test cases involves differentiating between affected and unaffected test cases. Test cases that have been affected by a modification to the program specification are those that tested an input/output relation that has changed or those that tested a required functional element (e.g. equivalence partition or boundary value) that has changed. Test cases that have been affected by a modification to a program construct may be identified using stored dynamic analysis and static analysis information [Leun89]. The structure of the modified source code as determined by static analysis can be compared with that of the original source code in order to identify program constructs that are unchanged, changed, deleted or new. The paths traversed by test cases (trajectories) obtained from previous dynamic analysis may be compared with the static analysis results in order to determine if they have been affected by a structural modification. Unaffected test cases are those that test unmodified portions of the specification and unmodified program constructs. The more detailed classifications of test cases taken from [Leun89] follow.
Unaffected test cases are classified as ‘reusable’. They do not test any functional or structural modifications and will give the same results as previous tests. Affected test cases are considered unclassified at this step and will eventually be classified as either ‘obsolete’ or ‘re-testable’.

If there has been a modification made to the specification, there will be a requirement to develop new specification-based test cases to exercise new or changed input-output relations or boundary conditions (constraints) required by the new specifications. The ‘new specification’ test cases will have to be executed by a dynamic analyzer to determine their contribution to the structural coverage. The addition of these new specification test cases may cause some structural test cases to become redundant.

Next, the unclassified test cases are analyzed. Test cases which no longer reflect valid input-output relations or no longer test a required functional element are classified as ‘obsolete’. Test cases that test a program construct that has changed may no longer test the same construct due to the modification of the program. These test cases may be obsolete or they may instead be ‘re-testable’ but provide different structural coverage. Test cases cannot be classified as re-testable or obsolete using static analysis and stored dynamic analysis information alone. They will have to be subjected to dynamic analysis where they are executed to determine their trajectory through the modified program structure [Leun89]. Specification-based test cases are normally dynamically analyzed first, so structural-based test cases are only analyzed to in an effort to increase the
structural coverage. If a structural test case does not contribute to the structural coverage of the test suite, it is considered to be redundant with respect to the chosen structural test coverage criterion and will be classified as obsolete. If the chosen structural test coverage criterion is satisfied during the dynamic analysis of test cases, any remaining unclassified test cases may be classified as obsolete.

Finally, new structural test cases are developed as required to test new structure that is not adequately covered by new specification or re-testable test cases.

The updated test suite contains all of the reusable, new-specification re-testable, and new-structural test cases ready for use during the next regression test cycle. The set of test cases for the regression test (the regression test set) consists only of new-specification, re-testable and new-structural test cases. Since the new-specification and re-testable classifications of test cases are executed by a dynamic analyzer during the test case classification step, the new structural test cases are the only ones remaining to complete the regression testing. The new structural test cases are also executed by a dynamic analyzer to confirm the adequacy of structural coverage. Obsolete test cases can be discarded to prevent the size of the test suite from becoming unmanageable.

The two types of regression testing required as a result of software modifications are viewed as either progressive (in which there is a specification that has been modified) or corrective (in which there is no reference to any change in a specification) as shown in Figure 1 [Leun89]. The test case classifications in accordance with how they are related
to the modifications to the structure or specification are shown below. All classifications of test cases apply to progressive regression testing. All except the fourth classification apply to corrective regression testing:

1. Reusable - unaffected test cases that test functional requirements pertaining to unmodified portions of the specification and unmodified program constructs;
2. Re-testable - affected test cases that test modified program constructs and continue to specify correct input/output relations;
3. Obsolete - test cases that no longer specify a correct input/output relation, no longer test a required functional element (e.g. equivalence partition or boundary value) or no longer contribute to the structural coverage of the program;
4. New Specification - new specification-based test cases that test new input/output relations or new required functional elements related to a change in the specification; and
5. New Structural - new structural-based test cases (developed after any new specification test cases) that are required to test new or modified program constructs.
Chapter 1

Introduction

The selection of test cases for regression testing involves a compromise between conflicting goals. On one hand, the selection of test cases is aimed at ensuring that those which are required to satisfy certain test coverage adequacy criteria are not excluded. On the other hand, an attempt is made to minimize the number of regression test cases selected by eliminating those that are redundant. It is hoped that the regression testing
effort can be reduced by automating the procedure of selecting necessary test cases and identifying redundant cases for removal.

1.5 Knowledge Based Systems

A knowledge based system comprises knowledge that is specific to a domain of application, including such things as simple facts about the domain, rules that describe relations or phenomena in the domain and possibly also methods, heuristics and ideas for solving problems in this domain [Brat90].

Knowledge based systems can be implemented relatively easily using goal-oriented programming languages as compared with other languages. Computer languages for goal-oriented programming are an evolution away from lower level procedural languages in which a programmer specifies how something is to be done, toward higher level declarative languages in which a programmer simply specifies what is to be done [Brat90]. Declarative symbolic languages such as Prolog are particularly well suited for solving problems that involve objects and relations between objects.

The application of knowledge based techniques may be used to select test cases according to a chosen test coverage criterion and minimize the number of test cases

---

1 Not every knowledge-based system can be considered an expert system; an expert system also has to be capable, in some way, of explaining its behavior and its decisions to the user [Brat90]. In domains in which the problem solving process has been well documented and is considered to be standard procedure, such an explanation feature is not necessary.
needed for regression testing by facilitating the analysis of the affects of the changes on structural elements and their relations to test cases.

1.6 Thesis Objectives

I propose to investigate the use of knowledge based techniques to identify regression test requirements in terms of structural elements using information from static analysis and to classify test cases as reusable, re-testable or unclassified using information from both static and dynamic analysis. A static analysis tool will be needed that can provide information regarding new and modified structural elements as well as information describing the structure of the source code. A dynamic analysis tool will be needed that can provide information regarding the trajectories of test cases in a form that can be compared with the output of the static analysis tool. The associations between classified test cases and the structural test requirements they satisfy will be inferred using the stored static and dynamic analysis information. An algorithm to identify redundant test cases will be used to produce a minimal regression test set and updated test suite. The requirement for new-structural test cases will be inferred from the associations between classified test cases and the structural test requirements they satisfy.

The scope of this thesis is restricted to structural testing; functional testing will not be addressed. The scope of this thesis is also restricted to unit testing, where the control structure of a program unit can be represented by a control flow graph with one component.
1.7 Thesis Organization

The remainder of this thesis is organized in accordance with the following outline:

In Chapter 2, an overview of some aspects of Knowledge Based systems is given, followed by a description of a prototype Regression Test Knowledge Based Tool;

In Chapter 3, the knowledge based implementation of a method of identifying structural coverage requirements for a control flow method is described;

In Chapter 4, the knowledge based implementation of a method of identifying structural coverage requirements for a data flow method is described;

In Chapter 5, the knowledge based implementation of selecting test cases to reduce the size of the regression test set and updated test suite is described;

In Chapter 6, a process model for the use of the Regression Test Knowledge Based Tool is described with an example;

In Chapter 7, a potential application of the Regression Test Knowledge Based Tool to aid object oriented software testing in conjunction with the use of an Object-Oriented modeling tool (ObjecTime) is presented; and

Finally, Chapter 8 provides a summary and recommendations for further study.
Chapter 2

2. A Knowledge Based System

2.1 General

A knowledge based system consists of a knowledge base, a domain base, an inference engine and a user interface (except for autonomous systems). The inference engine, accessed through the user interface, controls the execution of rules in the knowledge base in order to solve problems or answer questions by manipulating the facts contained in the domain base [Brat90].

A knowledge base comprises the knowledge that is specific to the domain of application in terms of rules that describe relations or phenomena in the domain and also methods and heuristics for solving problems in this domain.

A domain base consists of a collection of facts which describe a particular instance of a problem or situation to be analyzed or from which inferences may be made using the rules contained in the knowledge base.

A operation of a knowledge based system may vary from a data driven process to a query driven process. A query driven system solves problems by asking the user questions about possible symptoms of the problem. Facts are solicited from the user during the problem solving process. The answers provided by the user determine
subsequent questions that eventually lead to a solution. In a data-driven system, the set of facts describing the problem or situation are available at the start. The problem solving process will not require user input, or at least the input required will be limited.

One of the aims of developing a regression test tool is reduce the level of effort and expertise required of a user, so the Regression Test Knowledge Based Tool was developed as a data driven type of system.

### 2.2 A Regression Test Knowledge Based Tool

The Regression Test Knowledge Based Tool consists of the following components:
- Domain Base: contains facts representing the control flow or data flow graph of the source code including an indication of modified nodes and branches and facts representing the trajectories of existing test cases.

- Knowledge Base: contains rules about how to identify the structural test coverage requirements according to the coverage criterion chosen for regression test, how to classify test cases and how to select test cases for the updated test suite and the regression test set.

- The Shell consists of the User Interface and the Inference Engine:
  - The User Interface contains the production rules (actions to be taken) regarding:
    - loading the domain base with static and dynamic analysis data;
    - consulting the knowledge base to infer facts regarding the classification of test cases from the static and dynamic analysis data;
    - consulting the knowledge base to infer facts describing the structural coverage requirements from the static analysis data;
    - consulting the knowledge base to infer facts regarding the association between each testing requirement and the test cases that satisfy the requirement from the dynamic analysis data and the inferred requirements facts;
    - consulting the knowledge base to infer facts describing the reduced regression test set (non-redundant re-testable test cases), the updated test suite (non-redundant re-testable and
reusable test cases) and the structural coverage requirements which are not covered by the classified test cases.

- The Inference Engine knows how to use the rules in the knowledge base.

In this thesis, the application of knowledge based techniques to support regression testing depends upon the symbolic representation of the structure of the software as determined by static analysis techniques and the trajectory (path traversal through the flow graph) of each test case determined through dynamic analysis techniques. This information is used to determine the structural coverage requirements and the existing coverage of these requirements to assist in the automation of the regression test planning process.

2.2.1 The Prolog Language

The RTKBT was implemented as a prototype in Prolog, a goal-oriented programming language centered around a small set of basic mechanisms including pattern matching, tree-based data structuring and automatic backtracking [Brat90]. Prolog stands for "programming in logic", based on the idea of using logic as a programming language that emerged in the early 1970s. Early developers of Prolog were R. Kowalski and M. van Emden at Edinburgh and A. Colmerauer at Marseilles. The recent rise in the popularity of the Prolog language is attributed to the efficient implementation of a Prolog interpreter by David Warren from Edinburgh. The RTKBT has been developed using the
SWI-Prolog interpreter, an implementation based on the Warren Abstract Machine, which uses a popular Prolog dialect called Edinburgh syntax.

2.2.2 Syntax of Prolog Facts and Rules

A fact is a clause that describes an object or a relation between objects and is always, unconditionally true. Rules specify things that are true if some condition is satisfied. Rules have a condition part (the right-hand side of the rule) and a conclusion part (the left-hand side of the rule). The conclusion part is also called the head of a clause and condition part the body of a clause. The body of fact clauses are null. The Edinburgh syntax requires that clauses and constants begin with a lower case letter or be enclosed in single quotes. Variables must begin with an uppercase letter or the underscore character. All clauses are terminated with a period.

2.2.3 Representation of Software Structure

The software structure is represented by a directed graph consisting of sets of nodes and arcs. In the representation of procedural programs, the nodes are blocks of sequential code and the arcs are the logical branches. In the representation of object-oriented programs, the nodes are the object states and the arcs are the state transitions triggered by message passing events. The syntax of a digraph fact is as follows:

\[
\text{digraph( Nodes, Arcs ).}
\]

where 'Nodes' is a set of the form \([\text{Node}_1, \text{Node}_2, \ldots]\) and Node names are constants;
'Arcs' is a set of the form [ Arc1, ... ] and each arc is of the form a( Source, Destination) where 'Source' and 'Destination' are nodes.

Note that the ellipsis is not actually part of the syntax, but is used here in the usual manner to indicate that any number of additional nodes or arcs may be included in the respective sets. As a minimum, the digraph shall include a start node and an exit node with an arc connecting the start node to the exit node.

The digraph data to be interpreted by the RTKBT must also include information that will indicate whether or not a node or arc has been modified. This 'change' information associated with each node and arc in the digraph is appended following a forward slash '/' after each node and arc. Unmodified elements are indicated by a change value of 0 and new or modified elements are indicated by a change value of 1. The new digraph fact containing the change information will have a syntax as shown in the following example (where a modification of the second node and the arc is indicated):

\[
\text{cdigraph( [ node1/0, node2/1 ] , [ a( node1, node2 )/1 ] )}.
\]

2.2.4 Representation of Test Cases

Test case trajectories through the digraph are represented by paths consisting of the sequence of nodes visited when the test case is executed. The syntax of a test case fact is as follows:

\[
\text{testcase( Identifier, Path )}.
\]
The ‘Identifier’ is a constant and is unique; and
the ‘Path’ is of the form [node1, node2, ...].

2.2.5 Classification of Test Cases

The cdigraph and testcase facts as described above are sufficient for the
classification of test cases as re-testable, reusable or unclassified (possibly obsolete). A
test case is classified as reusable if all the nodes contained in the trajectory of the testcase
fact and the arcs that connect them exist in the cdigraph fact and the sum of the change
values of those nodes and arcs equals 0. A test case is classified as re-testable if all the
nodes contained in the trajectory of the testcase fact and the arcs that connect them exist
in the nodes and arcs sets of the cdigraph fact and the sum of the change values of those
nodes and arcs is greater than 0. A test case is deemed to be unclassified if any of the
nodes contained in the trajectory of the testcase fact or any of the arcs that connect them
do not exist in the nodes or arcs sets of the cdigraph fact.

2.2.6 Analysis of Structural Element Coverage Requirements

The analysis of the information contained in the cdigraph fact to identify structural
elements requiring test coverage in accordance with two possible coverage criteria is
described in Chapters 3 and 4. The structural element facts inferred through the control
flow and data flow analysis methods used in this thesis as described in these chapters are
simple basis paths and definition-use paths respectively. These facts will contain a unique
identifier and a set of nodes to represent the sub-path of the structural element. By
The user interface for the RTKBT is menu driven. The commands listed in the menu are shown on the following page.
* Regression Test Knowledge Based Tool *

legal commands are:

load - load flow graph and testcase data
classify - classify test cases
analyze - analyze flow graph for requirements
chain - chain several requirements together
isolate - isolate a requirement from another
plan - determine test plans
list - list testcases/requirements/coverage
quit - exit

A brief description of the commands follows:

a. load - loads facts describing the flow graph and test case trajectories from a disk file into the domain base;

b. classify - infers the classification of test cases from the cdigraph and testcase facts;

c. analyze - prompts the user to enter the desired coverage analysis method (either control_flow or data_flow) and identifies the coverage requirements in terms of paths in the flow graph;

d. chain - asserts an additional coverage requirement fact that two or more coverage requirements identified by the user must be tested together;

e. isolate - asserts an additional coverage requirement fact that one coverage requirement must be tested independently from another as identified by the user;

f. plan - produces the regression test set and updated test suite and identifies any required structural elements that are not covered by any test case; and

g. quit - exits the RTKBT shell.
2.3 Summary

The RTKBT is a prototype knowledge based regression test tool implemented in the Prolog language. The knowledge base of the RTKBT contains rules that describe the relationships between test case trajectories and flow graphs. These rules, invoked by commands through the user interface, can be used to classify test cases, identify paths in a flow graph that represent structural test coverage requirements for a control flow or data flow method and produce the minimized regression test set and updated test suite that satisfies the coverage adequacy criterion for the selected analysis method.


Chapter 3

3. Control Flow Based Test Coverage Criteria

The aim of this chapter is to demonstrate how the RTKBT can select test cases for regression testing based on a control flow test coverage adequacy criterion. A prominent control flow based test coverage criterion developed by McCabe is called the Structured Testing criterion [McCa83]. It is a form of path coverage that is based on McCabe’s cyclomatic complexity metric. Since this method involves the identification of a path representing a primary function of a program that may need more than control flow graph information (i.e. McCabe’s method requires human intervention), an alternative method I have developed called the Simple Basis Paths or SBP method is presented. This method provides a similar level of coverage also based on the cyclomatic complexity number, but where required paths can be identified algorithmically.

First each method is described, then the two methods are compared. Next, the selection of test cases for regression testing using the SBP criterion is demonstrated. Finally, a method of strengthening the SBP method to equal that of McCabe’s method is presented.

3.1 McCabe’s Structured Testing Criterion

McCabe’s Structured Testing criterion requires that a program is represented by a control flow graph in which it is assumed that each node can be reached from the entry
node and each node can reach the exit node (an artificial exit node may be added if needed to model embedded return statements as having an arc to the artificial exit node in order that the graph will have only one exit node).

The following graph theoretic definitions used in this thesis to describe the Structured Testing criterion resolves some conflicting definitions taken from [Wils79], [Rapp85] and [McCa83]. A graph $G$ can be represented by two sets such that $G = (N,E)$ where $N$ is the set of vertices (nodes) in the graph and $E$ is the set of edges in $N \times N$. A graph (not a directed graph) in which any two vertices are connected by a path is called a connected graph; otherwise it is disconnected. A finite disconnected graph can be expressed as the union of a finite number of connected graphs. In order to be consistent with [McCa83], a path is defined less rigidly in this thesis than in [Wils79] as a finite sequence of edges in which vertices and edges may be repeated. A complete path is a path in a control flow graph in which the initial node is the entrance node and the final node is the exit node. A path is closed if its initial vertex is equal to its final vertex. A cycle is a closed path in a graph. A simple cycle is a closed path in a graph in which all vertices are distinct with the exception of the first and last vertices. A graph is a directed graph if there is a direction associated with the edges. The directed edges are represented as an ordered pair of vertices called an arc. A directed graph (digraph) is strongly connected if there is a directed path joining any pair of arbitrary distinct vertices. A basis set of a strongly connected digraph is a set of linearly independent cycles in the digraph such that any arbitrary path through the graph can be expressed as a linear combination of cycles in
the basis set. A cycle is linearly independent with respect to a set of cycles if it cannot be expressed as a linear combination of other cycles in the set.

The cyclomatic number ‘v’ of a graph ‘g’ with ‘e’ edges, ‘n’ vertices and ‘p’ connected components is calculated with the formula \( v(g) = e - n + p \). Since this thesis is restricted to unit testing in which there will only be one component in the flow graph, the cyclomatic formula becomes \( v(g) = e - n + 1 \). In a strongly connected digraph, the cyclomatic number is equal to the maximum number of linearly independent cycles [McCa83]. In order to make the control flow graph a strongly connected graph, an extra arc connecting the single exit node to the single entrance node is added. Now a basis set with a size equal to the cyclomatic number may be formed from linearly independent basis paths (the extra arc is added to each basis path to form a basis set of cycles) such that any arbitrary path through the control graph can be represented by a linear combination of basis paths. Since a control flow graph does not contain the extra arc connecting the exit node to the entrance node, the formula for the cyclomatic number is modified to reflect this:

\[
\begin{align*}
\text{v}(g) &= (e + 1) - n + 1 \quad \text{or} \quad \text{v}(g) = e - n + 2.
\end{align*}
\]

While the above formula gives the size of a basis set, it does not identify which specific cycles a basis set contains. Also, since the cycles in the basis set are not necessarily simple cycles, there may exist many possible cycles corresponding to basis paths from which different basis sets may be formed.
Since the path through a control flow graph that corresponds to the execution of a test case is a complete path, the Structured Testing criterion is described in terms of complete paths. In this case, a basis set is a set of linearly independent complete paths through the control flow graph such that any arbitrary path through the graph can be expressed as a linear combination of complete paths in the basis set. The complete paths of the basis set are called basis paths.

The McCabe Structured Testing Method describes a procedure for selecting complete paths to form a basis set for the control flow graph. First a path is identified which represents a primary or ‘baseline’ function of the program. Next, the first decision along the baseline path is flipped while simultaneously holding the maximum number of the original baseline decisions the same as on the baseline path to form a new path. Each subsequent decision along the baseline is flipped to form new paths in a similar manner. If fewer paths than the complexity number have been generated by flipping decisions along the baseline, more paths are required. In this case, decisions along paths other than the baseline are flipped to form new paths in a similar manner as was done for the baseline. This process continues until the number of paths generated is equal to the complexity number and all edges have been covered.

A set of test cases that will satisfy Structured Testing criterion for a program with cyclomatic complexity $v$ must satisfy the following:
a. every arc in the control flow graph is traversed at least once; and

b. at least ‘v’ distinct independent complete paths are executed.

This Structured Testing criterion will be met if a test case exists that will exercise each basis path as identified in the procedure described above. Since the selection of a baseline path is somewhat arbitrary, there may be several sets of test data that satisfy the Structured Testing criterion.

3.2 The Simple Basis Paths Criterion

The Simple Basis Paths or SBP criterion provides a level of test coverage based on the cyclomatic complexity number where the paths that represent the required structural elements can be identified algorithmically.

The flow graph theoretic terminology used to describe the SBP method follows. A simple path is a path in which all nodes are distinct with the possible exception of the first and last nodes [Rapp85]. A simple complete path is a path in which the initial node is the entrance node, the final node is the exit node and all nodes are distinct. A self loop is a simple path, containing a single node, formed by an arc whose initial and final nodes are identical. A simple cycle is a simple path consisting of more than two distinct nodes (i.e. not a self loop) in which the initial node has two or more entrance arcs (an extra arc is connected to the single entrance node of the control flow graph to represent the entrance to the program) and the initial node is identical to the final node.
There will be a set of simple paths greater than or equal to the cyclomatic complexity number, consisting of the sets of simple complete paths, self loops and simple cycles in a graph. A basis set of simple paths can be formed by selecting a number of paths equal to the cyclomatic complexity number from the set of simple paths such that all arcs in the graph are included in some simple path. Any arbitrary path through the graph may be expressed as a linear combination of these simple basis paths.

The structural elements required to satisfy the SBP criterion for a program with cyclomatic complexity $v$ are:

a. every arc in the control flow graph is traversed at least once; and

b. at least `$v$' distinct simple paths must be executed.

The combination of requirements a and b above ensures that a basis set of simple paths is covered. In the event of a software modification, test cases which cover modified simple basis paths will be classified as re-testable and become candidates for regression testing.

The requirement that the basis paths used with the SBP method are selected from the sets of simple complete paths, simple cycles and self loops means that the basis paths are all simple paths as compared to other possible basis paths that may contain one or more cycles.
A portion of the hierarchy of clauses used to determine the set of simple paths from which the basis set is selected is shown as an And-Or graph in Figure 3. The ‘list’ clauses at the lowest level of the figure are rules (sub-goals are not shown, for greater detail, refer to [Midd97]) that describe each type of simple path as per the definitions given above. These clauses make use of the automatic backtracking feature of Prolog to generate the sets of each type of simple path. The ‘enumerate’ sub-goal clauses of the main clause provide a unique identifier to enumerate each simple path and assert facts in the domain base that describe all simple paths from which the basis set is selected.

```
Figure 3. SBP clause hierarchy
```

The facts asserted are of the following form:

```
simplepath(Identifier, Type, Path)
```

where the unique Identifier is number starting at 1, Type is ‘path’, ‘loop’ or ‘cycle’ and Path is the set of nodes that the path represents.
3.3 Comparison of the McCabe and SBP Criteria

Both criteria provide for the coverage of basis paths, however when using McCabe’s method, all basis paths are complete paths. This is important, since it is known from mathematical analysis that the cyclomatic complexity gives the exact number of tests necessary to test each decision outcome independently [McC94], and the McCabe’s method will produce such a satisfactory set of test paths. On the other hand, the SBP method, though it will ensure cyclomatic coverage adequacy in that it will ensure that a basis set of paths is covered, it will not ensure that each simple basis path is tested independently. Instead, it favors test cases that cover a greater number of basis paths in order to reduce the size of the test suite.

The SBP criterion is most rigorous (will require that all basis paths are tested independently) for control flow graphs in which nodes with more than one exit arc (splitting nodes) are not associated with any self loops or cycles. In this case, all basis paths, which are linearly independent, are simple complete paths. The SBP criterion is less rigorous for control flow graphs in which splitting nodes may be associated with self loops or cycles since cycles and loops are dependent on a complete path in order to be reachable. In essence, each simple complete path requires a separate test case while one or more simple cycles or self loops can be covered by a test case that covers a simple complete path.
When there are no cycles or self loops, the SBP method will require the same number of test cases as McCabe’s method. For example, graph A of Figure 4 contains four simple complete paths:

\[ [1,2,4,5,7], [1,3,4,5,7], [1,2,4,6,7] \text{ and } [1,3,4,6,7] \]

any three of which provide coverage that satisfies both the SBP and McCabe criteria.

![Graph A and Graph B](image)

**Figure 4. SBP criterion - best and worst cases**

When all splitting nodes are associated with a cycle or self loop, the SBP could be satisfied by a single test case. For example, graph B of Figure 4 (which has the same complexity as graph A) contains one simple complete path and two simple cycles giving four possible SBP combinations:

\[ [1,2,4,5,7], [1,2,3,2,4,5,7], [1,2,4,5,6,5,7] \text{ and } [1,2,3,2,4,5,6,5,7] \]

The SBP criterion could be satisfied by two test cases covering the second and third paths or by a single test case covering the last path. In the latter case, coverage is only equivalent to the all-branches criterion.
As a result of the above, the simple basis paths criterion is weaker than and is subsumed by McCabe’s criterion in that a set of test cases that satisfies McCabe’s criterion also satisfies the SBP criterion, while a smaller set of test cases may satisfy the SBP criterion. A method of ‘strengthening’ the SBP method to equal that of McCabe’s Structured Testing criteria is described later in this chapter.

3.4 Determining Testcase Coverage of Simple Paths

A test case can be said to cover a simple path if the path is traced completely and in the same sequence by the execution path, called the trajectory, for the test case. It can be determined that a test case covers a simple path if the set of nodes that represents the simple path is a sub-list of the set of nodes that represents the trajectory of the test case. Determining this association is more difficult when the trajectory contains loops and/or cycles. A test case’s coverage of a simple complete path or a simple cycle becomes obscured when iterations or combinations of cycles and loops are embedded within the test case trajectory.

This problem is solved by first extracting all self loops from the trajectory and asserting the corresponding association between the test case and the self loops covered as new facts, thereby making simple cycles visible as sets of contiguous nodes contained within the remaining portion of the trajectory. Next, all simple cycles are extracted from the trajectory and the corresponding association between the test case and the simple
cycles covered are asserted as new facts. This recursive process continues until a single simple complete path remains. The association between the test case and the simple complete path covered is asserted as the last fact of coverage associations between the test case and the simple paths. A test case trajectory may cover zero or more self loops, zero or more simple cycles and one and only one simple complete path.

3.4.1 Branch coverage

As mentioned previously, the SBP method subsumes branch coverage as a component. In order to confirm that a set of test cases covers a combination of simple paths that form a basis set, the branch coverage achieved by the classified test cases is checked. This is performed by a Prolog clause that converts the sets of nodes representing the trajectories of all classified testcases into a set of arcs, which is subtracted from the set of arcs in the digraph to produce the set of arcs that are not covered by any testcase. If this set is not null, the SBP criterion has not been met. This set of uncovered arcs is displayed to the user to guide the development of new structural test cases.

3.5 A SBP Method Demonstration

This section describes a control flow example taken from Section 2, Appendix B of [McCa83]. The program is a FORTRAN implementation of a Blackjack game. Blackjack was chosen as it should be familiar to most people. The specification for the implementation is given in [McCa83] and part of the FORTRAN code for this example is included in Appendix 1. Analysis is performed on the “HAND” routine, which contains
the logic for a one-hand session of Blackjack. The HAND routine contains calls to a subroutine "HIT", which determines the next card in the deck (a DEBUG mode allows card values to be specified for testing) and handles the Ace card taking on two values.

The program contains an intentional design error, which is used in [McCa83] to compare the error revealing capability of several coverage adequacy criteria. The error causes the player to automatically win with a 3-card hand of 21 before the dealer is allowed a chance to hit and match 21 to win.

The change digraph fact for the flow graph shown in Figure 5 follows. The change value associated with all nodes and arcs is zero, indicating that there have been no modifications made to the software.

cdigraph [deal/0, p_eq_21/0, d_eq_21/0, p_hit/0, win_eq_1/0, p_eq_21_2/0, p_gt_21/0, push/0, d_wins_bj/0, count_ge_5/0, p_wins_21_or_5/0, d_le_16/0, p_gt_d/0, d_gt_21/0, d_wins/0, p_wins/0, p_busts/0, d_busts/0, exit/0], % nodes
[a(deal,p_eq_21)/0, a(p_eq_21,d_eq_21)/0, a(p_eq_21,p_hit)/0,
 a(d_eq_21,win_eq_1)/0, a(d_eq_21,p_eq_21_2)/0, a(p_hit,d_eq_21)/0, a(p_hit,p_gt_21)/0,
 a(win_eq_1,push)/0, a(win_eq_1,d_wins_bj)/0, a(p_eq_21_2,p_wins_21_or_5)/0,
 a(p_eq_21_2,count_ge_5)/0, a(p_gt_21,p_hit)/0, a(p_gt_21,p_busts)/0, a(push, exit)/0,
 a(d_wins_bj,exit)/0, a(count_ge_5,p_wins_21_or_5)/0, a(count_ge_5,d_le_16)/0, a(p_busts,
 exit)/0, a(p_wins_21_or_5,exit)/0, a(d_le_16,p_gt_d)/0, a(d_le_16,d_gt_21)/0, a(p_gt_d,
 d_wins)/0, a(p_gt_d, p_wins)/0, a(d_gt_21,d_le_16)/0, a(d_gt_21,d_busts)/0, a(d_wins, exit)/0,
 a(p_wins, exit)/0, a(d_busts, exit)/0] % arcs
).
The flow graph in Figure 5 differs from that in [McCa83] in that the compound predicate \((P \text{ .EQ. } 21) \text{ .OR. } (\text{COUNT } \text{ .GE. } 5)\) has been modeled differently. In [McCa83], each condition in the compound predicate is modeled as a separate arc which is named by the condition it represents. This allows for the testing of all possible truth values of predicates using the cyclomatic adequacy criterion. The Blackjack example in [McCa83] models the compound predicate as a single node “p_eq_21_or_count_ge_5”
with two arcs to the node "p_wins_21_or_5"; one arc is named p_eq_21 and the other is named count_ge_5.

The flow graph representation used in the RTKBT does not provide for the naming of arcs to distinguish between two otherwise identical arcs. Instead, the compound predicate is modeled in Figure 5 as nested IF-THEN-ELSE statements as follows:

\[
\text{IF (P .EQ. 21) THEN p_wins_21_or_5} \\
\text{ELSE IF (COUNT .GE. 5) THEN p_wins_21_or_5} \\
\text{ELSE d_le_16}
\]

The above statement is equivalent and also has the same complexity [McCa83]. This representation of compound predicates can be constructed by a static analysis tool.

The set of test cases from [McCa83] and their trajectories are represented for use by the RTKBT as shown below. These test cases will be referred to throughout the SBP demonstration. Note that the ‘%’ symbol is used to indicate a comment in Prolog. The card values selected for testing and the final hand values are indicated as comments above each test case.

\[
\begin{align*}
\text{% P:10, D:10, P:5, D:4, P:4, D:6; P=19, D=20} \\
\text{testcase(t01,[deal, p_eq_21, p_hit, p_gt_21, p_hit, d_eq_21, p_eq_21_2, count_ge_5, d_le_16, d_gt_21, d_le_16, p_gt_d, d_wins, exit]). % baseline}
\end{align*}
\]

\[
\begin{align*}
\text{% P:10, D:10, P:11, D:6; P=21, D=16} \\
\text{testcase(t02,[deal, p_eq_21, d_eq_21, p_eq_21_2, p_wins_21_or_5, exit]).}
\end{align*}
\]
The first step in the regression testing process after the domain base has been loaded with the facts describing the software control flow and the test case trajectories is to classify the test cases. The Blackjack example will be used to demonstrate the regression test process to compare the SBP method with that of McCabe's structured
testing method in the absence of program modifications. As expected, all test cases are classified as reusable.

The next step is the analysis of the test coverage requirements and the coverage of these requirements provided by the test cases. First the test coverage requirements obtained using the SBP method are shown below. Notice that simple paths 1 to 15 are complete paths and simple paths 16 and 17 are cycles.

Next, the coverage of the simple paths by the test cases is shown. Although many of the simple paths are not covered by any test cases, coverage will be deemed adequate in accordance with the SBP criterion if a basis set is covered.
Chapter 3  Control Flow Based Test Coverage Criteria

Finding Classified test cases...

Checking simple path coverage...

Building structural requirement coverage sets...

Path 1 is covered by test cases [ ]
Path 2 is covered by test cases [ ]
Path 3 is covered by test cases [ ]
Path 4 is covered by test cases [ ]
Path 5 is covered by test cases [t02]
Path 6 is covered by test cases [ ]
Path 7 is covered by test cases [t11]
Path 8 is covered by test cases [t09]
Path 9 is covered by test cases [t01, t03, t08]
Path 10 is covered by test cases [t10]
Path 11 is covered by test cases [t07]
Path 12 is covered by test cases [t06]
Path 13 is covered by test cases [t05]
Path 14 is covered by test cases [ ]
Path 15 is covered by test cases [t04]
Path 16 is covered by test cases [t01, t03, t10]
Path 17 is covered by test cases [t01, t05, t06, t07, t08, t09, t10]

Press <Enter> to continue.

The final step is to determine the minimum set of test cases required to meet the SBP criterion. Although the regression test set is of primary concern during regression testing, test cases that are redundant and therefore obsolete are identified in order to produce a minimum updated structural test set. The algorithm used to identify the redundant test cases is described in a later chapter. In this example, no changes have been made yet, so the regression test set is expected to be empty. Of interest is the comparison to the test cases provided by McCabe’s structured testing method. Notice that test cases ‘t03’ and ‘t08’ are redundant for the SBP method. This is due to the fact that test case ‘t01’ covers the same simple complete path (path 9) as test cases ‘t03’ and ‘t08’ and also covers both simple cycles (paths 16 and 17), whereas test cases ‘t03’ and ‘t08’ each only cover one of the simple cycles; hence test case ‘t01’ is selected in favor of either of the other two.

---

2 It is encouraging to note that the simple basis paths method did not exclude test case ‘t06’ which reveals the intentional design error, in which the player automatically wins with a 3-card hand before the dealer is allowed a hit. This however does not imply that the set of test cases selected using this method is as error revealing as for McCabe’s method.
To demonstrate the use of the RTKBT with the SBP method in a regression testing scenario, the modifications required to correct the error in the program are simulated by modifying the control flow graph of the blackjack program to include the appropriate logic. The affected portion of the control graph is shown in Figure 6. The node ‘count_lt_3’, the arc entering it and the two arcs leaving it are identified as ‘new’ by assigning them a change value of 1 in the cdigraph fact.
As before, the first step in the regression testing process after the domain base has been loaded with the new facts describing the modified software structure is to re-classify the test cases. Test cases whose trajectories cannot traverse the new graph due to the addition or removal of nodes or arcs are unclassified.

```
RTKBT> classify
Classifying test cases...
The test cases are ['t01', 't02', 't03', 't04', 't05', 't06', 't07', 't08', 't09', 't10', 't11']
Test case 't01' is Reusable.
Test case 't02' is Unclassified.
Test case 't03' is Reusable.
Test case 't04' is Reusable.
Test case 't05' is Reusable.
Test case 't06' is Unclassified.
Test case 't07' is Reusable.
Test case 't08' is Reusable.
Test case 't09' is Reusable.
Test case 't10' is Reusable.
Test case 't11' is Reusable.
Press <Enter> to continue.
```

Test cases ‘t02’ and ‘t06’ are unclassified since their trajectories depend on the arc from node ‘p_eq_21_2’ to node ‘p_wins_21_or_5’, which no longer exists. The analyze step is repeated to demonstrate that the RTKBT will identify the arcs which are not covered according to the old test case trajectory facts.

```
RTKBT> analyze
Purging old facts...
Select the analysis method to be used; Enter "control_flow" or "data_flow" : control_flow
The classified test cases are: ['t01', 't03', 't04', 't05', 't07', 't08', 't09', 't10', 't11']
Identifying simple paths...
The simple paths are ['1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23']
Simple path 1 is path: ['deal', 'p_eq_21', 'd_eq_21', 'p_eq_21_2', 'count_ge_5', 'd_le_16', 'd_gt_21', 'd_busts', 'exit'].
```
Chapter 3  Control Flow Based Test Coverage Criteria

<table>
<thead>
<tr>
<th>Simple path 2 is path</th>
<th>Simple path 3 is path</th>
<th>Simple path 4 is path</th>
<th>Simple path 5 is path</th>
<th>Simple path 6 is path</th>
<th>Simple path 7 is path</th>
<th>Simple path 8 is path</th>
<th>Simple path 9 is path</th>
<th>Simple path 10 is path</th>
<th>Simple path 11 is path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple path 11 is path</td>
<td>Simple path 12 is path</td>
<td>Simple path 13 is path</td>
<td>Simple path 14 is path</td>
<td>Simple path 15 is path</td>
<td>Simple path 16 is path</td>
<td>Simple path 17 is path</td>
<td>Simple path 18 is path</td>
<td>Simple path 19 is path</td>
<td>Simple path 20 is path</td>
</tr>
<tr>
<td>Simple path 21 is path</td>
<td>Simple path 22 is cycle</td>
<td>Simple path 23 is cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice that there are four new simple complete paths (paths 15 to 18) containing the new node.

Finding Classified test cases...

Checking basis path coverage...

Building structural requirement coverage sets...

Path 1 is covered by test cases []
Path 2 is covered by test cases []
Path 3 is covered by test cases []
Path 4 is covered by test cases []
Path 5 is covered by test cases []
Path 6 is covered by test cases []
Path 7 is covered by test cases []
Path 8 is covered by test cases []
Path 9 is covered by test cases []
Path 10 is covered by test cases [t11]
Path 11 is covered by test cases [t09]
Path 12 is covered by test cases [t01, t03, t08]
Path 13 is covered by test cases [t10]
Path 14 is covered by test cases [t07]
Path 15 is covered by test cases []
Path 16 is covered by test cases []

44
The coverage inadequacy is revealed in the ‘plan’ step below. Since there were no test cases classified as re-testable, none are available to select for the regression test set, so it is indicated as a null set. The cyclomatic complexity has increased to 12. Coverage is known to be inadequate before checking the number of simple paths covered, since there are uncovered arcs.

To identify test cases that will satisfy the outstanding structural coverage requirements, the unclassified test cases can be processed by a dynamic analysis tool to determine their new trajectories. This has been done manually for test cases ‘t02’ and ‘t06’:

```
% P:10, D:10, P:11, D:6; P=21, D=16
testcase(t02,[deal, p_eq_21, d_eq_21, p_eq_21_2, count_lt_3, p_wins_21_or_5, exit]).
```
NOW, test cases 't02' and 't06' are classified as re-testable since they both traverse the modified node and arcs:

```
RTKBT> classify
Classifying test cases...
The test cases are [t01, t02, t03, t04, t05, t06, t07, t08, t09, t10, t11]
Test case t01 is Reusable.
Test case t02 is Re-testable.
Test case t03 is Reusable.
Test case t04 is Reusable.
Test case t05 is Reusable.
Test case t06 is Re-testable.
Test case t07 is Reusable.
Test case t08 is Reusable.
Test case t09 is Reusable.
Test case t10 is Reusable.
Test case t11 is Reusable.
Press <Enter> to continue.
```

```
Checking simple path coverage...
Building structural requirement coverage sets...
Path 8 is covered by test cases [t02]
Path 10 is covered by test cases [t11]
Path 11 is covered by test cases [t09]
Path 12 is covered by test cases [t01, t03, t08]
Path 13 is covered by test cases [t10]
Path 14 is covered by test cases [t07]
Path 17 is covered by test cases [t06]
Path 19 is covered by test cases [t05]
Path 21 is covered by test cases [t04]
Path 22 is covered by test cases [t01, t03, t10]
Path 23 is covered by test cases [t01, t05, t06, t07, t08, t09, t10]
```

The simple paths are unchanged, but test cases 't02' and 't06' now show up in the structural requirement coverage sets as shown above. When the 'plan' step is performed to determine the structural test set and regression test set, this time coverage is determined to be inadequate as the number of simple paths covered by test cases is less than the
complexity number (all arcs have been covered however). A list of uncovered simple
paths is presented to the user to assist with the development of new test cases.

```
RTKBT> plan
Determining Structural Test Set
and Regression Test Set...

The Structural Test Set is:
[t02, t04, t05, t06, t07, t09, t10, t11, t01]

The hitting set algorithm has found
the following test cases to be redundant:
[t03, t08]

The Regression Test Set is:
[t02, t06]

The graph cyclomatic complexity is 12.
All arcs have been covered but
1 more simple path(s) must be covered in order to cover a basis set.
Up to 1 of the following simple paths should be covered:
[1, 2, 3, 4, 5, 6, 7, 9, 15, 16, 18, 20]
Press <Enter> to continue.
```

A logical place to begin when developing new test cases is in the area of the
modification: the new logic to permit the dealer a chance to match a 3-card 21 hand, such
as path 16. For test case 't06', the control flow reaches the node in which the decision to
allow the dealer to take a hit is made, but since the dealer has a hand of 17, the dealer
must stand as required by the specification.

A new test case is added in which the dealer takes a hit and matches 21 to win the
hand:

```
% P:10, D:6, P:6, D:10, P:5, D:5; P=21, D=21
testcase(t12,[deal, p_eq_21, p_hit, p_gt_21, p_hit, d_eq_21,
             p_eq_21_2, count_lt_3, d_le_16, d_gt_21, d_le_16, p_gt_d,
             d_wins, exit]).
```

When the classify and analyze steps are repeated with the new test case 't12'
included, it is shown as a re-testable test case covering simple paths 16, 22 and 23. The
test planning phase now produces a structural test set which meets the SBP coverage criterion and a regression test set containing the three re-testable test cases:

```
RTKBT> plan
Determining Structural Test Set
and Regression Test Set...

The Structural Test Set is:
[t02, t04, t05, t06, t07, t09, t10, t11, t12, t01]

The hitting set algorithm has found
the following test cases to be redundant:
[t03, t08]

The Regression Test Set is:
[t02, t06, t12]

The graph cyclomatic complexity is 12.
All arcs have been covered and
simple path coverage is equal to the complexity number.

Press <Enter> to continue.
```

3.6 Strengthening the SBP criterion

The SBP method does not take into consideration the possibility of interactions between structural elements. It simply attempts to ensure that a basis set of simple paths is covered by the test cases in the test set. It is possible that a fault may exist that will not be observed if a number of simple paths are not exercised either in conjunction with or separate from each other. Given that the control flow structure of a program is known and the simple paths to be covered have been identified, a method I have developed to manually assert that particular simple paths are to be tested either together or separately is presented (further research is required to automate this process).
3.6.1 Chaining Requirements

The McCabe structured testing method will test a basis set of linearly independent complete paths, whereas the SBP method will test a basis set consisting of simple complete paths, simple cycles and self loops, but not necessarily independently of each other. The SBP method favors test cases that cover a greater number of basis paths over those that cover a single basis path. The SBP method can be strengthened to that of McCabe’s method by representing the McCabe basis paths as combinations of simple paths. This is enabled by ‘chaining’ requirements together to form a new ‘chain’ requirement. The new chain requirement is satisfied only by test cases that satisfy all of the requirements in the chain. The chain requirement is considered to be a modified requirement (a candidate for regression testing) if any of the requirements (simple paths) in the chain have been modified.

3.6.2 Isolating Requirements

Chaining simple paths will ensure the selection of a test case that will test the path combination chain that represents the McCabe basis path, but it is not sufficient to ensure the selection of a test case that will test this basis path independently from some other simple path. This is enabled by ‘isolating’ a chain requirement representing a McCabe basis path from other simple paths to form a new ‘isolation’ requirement. An ‘isolation’ is satisfied by test cases that satisfy the isolated requirement and do not satisfy any of the requirements that it is to be isolated from. An isolation is considered to be a modified
requirement (a candidate for regression testing) if the isolated requirement has been modified.

3.6.3 Demonstration of the Strengthened SBP Method

To demonstrate, the original blackjack example from page 40 will be used in which basis paths containing cycles 16 (the cycle $[d\_le\_16, d\_gt\_21, d\_le\_16]$ in which the dealer takes a hit) and 17 (the cycle $[p\_hit, p\_gt\_21, p\_hit]$ in which the player takes a hit) are to be isolated from each other. First these simple cycles are chained to path 9 (a simple complete path) in order to form the complete basis paths intended by McCabe’s method. In this example, it is also desired that the McCabe basis path in which both the player and the dealer take a hit is tested, so a requirement chaining both cycles 16 and 17 to path 9 is created as well. Once the chains representing the McCabe basis paths are created, they can be isolated from the cycle they are to be tested independently from.

```
RTKBST> chain
Add a requirement chain:
Enter the first requirement to chain: 16
Enter the second requirement to chain: 9
Add more requirements to chain? (y/n) n
[18, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
RTKBST> chain
Add a requirement chain:
Enter the first requirement to chain: 17
Enter the second requirement to chain: 9
Add more requirements to chain? (y/n) n
[19, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
```
Recall that previously the SBP method identified test cases ‘t03’ and ‘t08’ (each of which covered path 9 and one of the cycles 16 or 17) as redundant since test case ‘t01’ covered all simple basis paths that they covered. The assertion of chain and isolation requirements describing the McCabe basis paths caused these test cases to be selected.
The complete basis paths for McCabe's method were created by chaining the appropriate simple basis paths together. In order to ensure that these basis paths were tested independently, they were isolated from the other simple paths (in this case a single cycle) that were connected to the simple complete path in question. As seen above, this caused all test cases developed using McCabe's structured testing method to be selected for the strengthened SBP method.

3.7 Summary

The selection of test cases by the RTKBT for regression testing based on the Simple Basis Paths coverage criterion has been demonstrated. The SBP criterion is control flow based and provides a level of coverage based on the cyclomatic complexity number. The SBP criterion is subsumed by the McCabe criterion. The assertion of additional chain and isolation requirements provides a mechanism for strengthening the SBP method to equal that of McCabe's method.
Chapter 4

4. Data Flow Based Test Coverage Criteria

Data Flow testing has its origins in compiler optimization and may be viewed as an enhancement of the control flow methods in which the selection of structural test requirements also depends on the occurrences of variables in a program. There are usually more interactions observed with data flow techniques than with control flow techniques as it is usually the data behavior that either precludes or makes realizable the execution of any particular control path [McCa83]. As with the control flow methodology, a program is represented by a control flow graph in which it is assumed that each node can be reached from the entry node and each node can reach the exit node. In addition, associated with each node is information describing the occurrences of data variables. Using terminology from [Harr88], the occurrence of variables in basic blocks (nodes) are classified as either a definition (def), a compute use (c-use) or a predicate use (p-use). A c-use directly affects a computation being performed and may have an indirect effect on the flow of control in a program whereas a p-use directly affects the flow of control and may indirectly effect some computation.

The hierarchy of structural test coverage criteria that may be chosen using data flow methods that provide increasing degrees of test coverage as well as being increasingly difficult to achieve are [Harr88]:
a. all nodes;
b. all edges;
c. all definitions;
d. all p-uses;
e. all c-uses/some p-uses;
f. all p-uses/some c-uses;
g. all uses;
h. all definition-use paths;
i. all paths.

As mentioned earlier, the all paths criterion is often impossible to achieve since the presence of loops may provide for an infinite number of possible paths. While the "all definition-use paths" criterion provides a lesser degree of coverage, the number of du-paths will be finite [Rapp85] (du-paths are simple paths) making the criterion achievable.

One difficulty that the "all definition-use paths" criterion shares with the all paths criterion is the existence of infeasible paths. If a test case cannot be developed to cover a particular du-path, the user must determine that the path is infeasible before deciding that coverage is adequate.

The data flow method considered in this thesis is based on the 'all definition-use paths' criterion described in [Harr88] where the effect of software modifications in terms
of the insertion or deletion of definitions and uses of variables in the different types of basic blocks is discussed. Modifications may change the definition-use pairs and consequently the data flow paths that make up the test coverage requirements. A description of a process for identifying these requirements follows. Test cases that traverse modified definition-use paths will be classified as re-testable and will become candidates for regression testing.

4.1 The All Du-paths Criterion

The data flow coverage criterion implemented in the RTKB is the “all du-paths” criterion described in [Rapp85] and [Harr88]. As with the control flow method, this data flow method requires that the cdigraph fact represent the flow graph of the software unit with change value information showing modifications and testcase facts showing trajectories of test cases:

\[
\text{cdigraph( [ node1/Change, node2/Change, ... ], [ a( node1, node2)/Change, ... ] ).}
\]
\[
\text{testcase( t01, [node1, node2, ... ] ).}
\]
\[
. 
\]
\[
. 
\]

The set of definitions and uses of each variable within a basic block is represented as follows:

\[
\text{basicblock( Node, Uses).}
\]
where Uses is the set of uses of variables in the following format:

\[
\text{use(Variable, Usage).}
\]
and Usage is 'def', 'c-use' or 'p-use'.

55
Note that since the right hand side of assignment statements in programs are processed before the assignment is made to the variable on the left hand side, c-uses of variables in assignment statements are listed in the Uses set before the def of the assigned variable in the statement is listed.

This information is used in the data flow analysis to determine the test coverage requirements and the adequacy of the test cases in the test suite with respect to the all du-paths criterion.

The data flow graph will have a single entrance block and a single exit block with the following additional features:

the entrance block must provide definitions of the local variables representing passed parameters and global variables;
the exit block must provide uses of the return parameters and the global variables;
it is also required that each call site is represented by a separate block in the flow graph.

In [Harr88] the analysis is restricted to code written in which every call site has c-uses of the passed parameters. The RTKBT was developed with the assumption that a data flow static analysis tool is able to determine that input parameters of functions (call sites) are considered c-uses, output parameters are considered defs and in/out parameters are treated as a c-use followed by a def.
Given the above information provided in the domain base, the data flow rules contained in the knowledge base determine test requirements and coverage as described below. The data flow clauses (rules) and their implementation are contained in the technical report associated with this thesis [Midd97].

First the set of all variables is identified by examining all variable uses in the Uses set of all basicblock facts.

Basic blocks are analyzed to determine their type with respect to each variable as follows:

- a c-use block contains a c-use of the variable in the block but no definition of the variable in the block;
- a def block contains a definition of the variable in the block before any c-use of the variable in the block;
- a use-def block contains one or more c-uses of the variable in the block before a definition of the variable in the block;
- a pass-through block contains neither a c-use of the variable nor a definition of the variable
- a p-use block contains a p-use of the variable in the block.

Note that a block may be of type p-use and either of def, use-def, or c-use.
Chapter 4  

Data Flow Based Test Coverage Criteria

Next, for each variable, the set 'defset' is determined, which is the set of blocks that contain a definition of the variable (all def and use-def blocks). For each of the blocks in the defset for each variable, all blocks containing c-uses of the variable and the successors of all blocks containing p-uses of the variable are found. These sets of blocks are used to create def-use block pairs representing the start and end blocks used to determine all possible du-paths. These sets of blocks are determined as described below.

For each variable and for each node (basic block) in the defset for that variable, the set ‘dcu’ contains the nodes of all c-uses (c-use and use-def blocks) of the variable.

For each variable and for each node in the defset for that variable, the set ‘dpu’ contains the arcs following the nodes of all p-uses (p-use blocks) of the variable.

The sets dcu and dpu are both used to determine the du-paths. Testing from a definition to a c-use requires that all definition-clear simple paths (in which there are no defs between the definition and the use) from the block containing the definition to the block containing the c-use are traversed by test cases. If the definition block is a use-def type block, all definition clear simple cycles must be found that begin with the def in the use-def block and end with the c-use that precedes the def in the same block. Testing from a definition to a p-use requires that all definition-clear simple paths from the block containing the definition to both of the blocks that are successors of the block containing the p-use are traversed by test cases. The set of all du-paths joining all def-use pairs
associated with the defs in the defset and the uses in the dcu and dpu sets are determined using the automatic backtracking feature of Prolog.

The set of all the du-paths from all the def blocks for all the variables to all their c-uses and p-uses is built by combining the sets of du-paths that are returned from a hierarchy of smaller Prolog clauses. The set of all variables is passed to the main clause, which for each variable, will call clauses to find all the du-paths from all defs of a variable to all c-uses and to all p-uses, and will return the union of all these du-path sets as a complete set of du-paths. Calls to subordinate clauses are as shown in Figure 7 with recursion shown as loops.

![Figure 7. The all-du-paths clause hierarchy](image)

The algorithm used to determine the set of all du-paths as described above, differs slightly from the method used by [Rapp85] and [Harr88] in that:
Chapter 4  
Data Flow Based Test Coverage Criteria

- for each variable, the set defset (created by the defset clause) contains the set of all blocks containing a definition of the variable (all def or use-def blocks);
- the additional p-use block type is used by the dpu-set clause to build the dpu set;
- the dcu and dpu sets are not stored in the basicblock facts, they are determined each time a data flow structural requirements analysis is performed.

A du-path is considered to have been modified only if either the definition or use has been modified as indicated by the change value of the def (first) node, the c-use (last) node or the p-use (second last) node, or the p-use arc (connecting the last two nodes). A modified pass-through block in a du-path for a variable will be covered by a du-path for some other variable. Ideally, only those du-paths whose definition or use statement has been modified should be re-tested, but since the data flow representation used for the RTKBT only contains an indication of block modifications and no indication of the modification of individual uses, any modification in a def or use block will cause the du-path to be classified as re-testable.

4.1.1 Determining Testcase Coverage of du-paths

A test case can be said to cover a du-path if the du-path is traced completely and in the same sequence by the test case trajectory. It can be determined that a test case covers a du-path if the set of nodes that represents the du-path is a sub-list of the set of nodes
that represents the trajectory of the test case. Determining this association is not complicated by the presence of loops or cycles as all du-paths are simple paths.

4.1.2 Branch Coverage

If all du-paths are feasible, all-branch coverage is provided by du-path coverage and does not have to be explicitly checked for since the all-du-paths criterion strictly includes the all-edges (branches) criterion [Rapp85]. In other words, the all-branches criterion is subsumed by the all-du-paths criterion [Zhu96] in that a set of test cases that is adequate according to the all-du-paths criterion will also be adequate for the all-branches criterion. This relationship can be traced in the inclusion/subsume relation figures of [Rapp85] or [Zhu96]: all-du-paths includes all-uses which includes all p-uses, and by executing all p-uses of a program, you are assured of executing all branches in the program [Horg92].

If all du-paths are not feasible however, a weaker criterion, 'all feasible du-paths' will have to be used. The problem with this weaker criterion is that it may not provide coverage of all-edges [Fran88]. Since this is often the case, the coverage of branches is explicitly checked by the RTKBT with the intention of providing all-branches coverage as a minimum.
4.2 A Data Flow Example

The source code in Figure 8 and data flow graph in Figure 9 for the example are taken from [Harr88].

```
1. function SUB2(X,K: integer) : integer;
2.   var Z, Y, OUT: integer;
3.   begin
4.     if X <= 10 then begin
5.       X := X + 5;
6.       K := K + 1;
7.     end
8.     if K > X then Z := K + X;
9.   else Z := K - X end
10.   else M := SUB1(X,K);
11.   Y := X * 2;
12.   while Y <= 25 do begin
13.     X := X + 1;
14.     Y := X * 2;
15.     Z := X + Y;
16.     M := M + 2 end;
17.   if Z > 40 then OUT := X + Z
18.   else OUT := K + Y;
19.   SUB2 := OUT
20. end
```

Figure 8. Data flow example - source code.

In Figure 9, the 's' (start) block provides definitions of the local variables X and K that represent input parameters and the global variable M. The 'r' (return) block provides uses of the return parameter OUT and the global variable M. It is required that a call site such as that in block B5 is represented by a separate block in the flow graph. This will ensure that any def-use pairs related to the call site can produce sub-paths of two or more nodes, rather than having the pairs contained within the same block with the result that some du-path requirements may be hidden. Analysis in [Harr88] was restricted to code in
which every call site has c-uses of the actual parameters. The RTKBT only requires that the static analysis tool is able to determine the actual usage (c-use, def or c-use followed by def) associated with the parameter type (input, output or input/output).

The change digraph fact used to represent the flow graph of Figure 9 is as follows:

\[
\text{cdigraph}([s/0, b1/0, b2/0, b3/0, b4/0, b5/0, b6/0, b7/0, b8/0, b9/0, b10/0, b11/0, r/0] \ . \ % \ \text{Nodes} \\
[a(s,b1)/0, a(b1,b2)/0, a(b1,b3)/0, a(b1,b5)/0, a(b2,b3)/0, a(b2,b4)/0, a(b3,b6)/0, a(b4,b6)/0, a(b5,b6)/0, a(b6,b7)/0, a(b7,b8)/0, a(b7,b9)/0, a(b8,b7)/0, b(a,b,10)/0, a(b9,b10)/0, a(b9,b11)/0, a(b10,r)/0, a(b11,r)/0] \ % \ \text{Arcs} \).
\]

\[
X := 1^\text{st} \text{ actual} \\
K := 2^\text{nd} \text{ actual} \\
M := \text{global } M
\]

**Figure 9.** Data flow example - flow graph.
The basicblock facts for the example shown in Figure 9 follow. Recall that assignment statements are processed right hand side first, so c-use variables from the right side of the statement are listed before the def use of the assigned variable on the left side of the statement.

\[
\begin{align*}
\text{basicblock}(s, \text{use}(x, \text{def}), \text{use}(k, \text{def}), \text{use}(m, \text{def})). \\
\text{basicblock}(b1, \text{use}(x, \text{p-use})). \\
\text{basicblock}(b2, \text{use}(x, \text{c-use}), \text{use}(x, \text{def}), \text{use}(k, \text{c-use}), \text{use}(k, \text{def}), \text{use}(x, \text{p-use}), \text{use}(k, \text{p-use})). \\
\text{basicblock}(b3, \text{use}(x, \text{c-use}), \text{use}(k, \text{c-use}), \text{use}(z, \text{def})). \\
\text{basicblock}(b4, \text{use}(x, \text{c-use}), \text{use}(k, \text{c-use}), \text{use}(z, \text{def})). \\
\text{basicblock}(b5, \text{use}(k, \text{c-use}), \text{use}(x, \text{c-use}), \text{use}(m, \text{def})). \\
\text{basicblock}(b6, \text{use}(x, \text{c-use}), \text{use}(y, \text{def})). \\
\text{basicblock}(b7, \text{use}(y, \text{p-use})). \\
\text{basicblock}(b8, \text{use}(x, \text{c-use}), \text{use}(x, \text{def}), \text{use}(x, \text{c-use}), \text{use}(y, \text{def}), \\
\quad \text{use}(y, \text{c-use}), \text{use}(x, \text{c-use}), \text{use}(z, \text{def}), \text{use}(m, \text{c-use}), \text{use}(m, \text{def})). \\
\text{basicblock}(b9, \text{use}(z, \text{p-use})). \\
\text{basicblock}(b10, \text{use}(z, \text{c-use}), \text{use}(x, \text{c-use}), \text{use}(\text{out}, \text{def})). \\
\text{basicblock}(b11, \text{use}(y, \text{c-use}), \text{use}(k, \text{c-use}), \text{use}(\text{out}, \text{def})). \\
\text{basicblock}(r, \text{use}(\text{out}, \text{c-use}), \text{use}(m, \text{c-use})). \\
\text{% note the c-use of global variable M in basic block R.}
\end{align*}
\]

Initial test cases for this example have been developed using McCabe’s Structured Testing method. The cyclomatic complexity of the flow graph is 5, so there should be at least 5 test cases covering some set of the paths requiring testing. A baseline test case ‘t01’ is created using the True branches of the first two IF statements and avoiding the While loop. Test cases ‘t02’ to ‘t05’ were developed manually by flipping each decision along the baseline. The testcase facts that represent the trajectories of each test case are
shown below. The input values \( X \) and \( K \) of the function are indicated in comments above each test case:

\[
\% X = 10, \ K = 15 \\
\text{testcase(t01,[s,b1,b2,b3,b6,b7,b9,b11,r]).} \quad \% \text{baseline}
\]

\[
\% X = 13, \ K = 0 \\
\text{testcase(t02,[s,b1,b5,b6,b7,b9,b11,r]).}
\]

\[
\% X = 10, \ K = 14 \\
\text{testcase(t03,[s,b1,b2,b4,b6,b7,b9,b11,r]).}
\]

\[
\% X = 7, \ K = 12 \\
\text{testcase(t04,[s,b1,b2,b3,b6,b7,b8,b9,b11,r]).}
\]

\[
\% X = 10, \ K = 25 \\
\text{testcase(t05,[s,b1,b2,b3,b6,b7,b9,b10,r]).}
\]

An initial analysis (not shown) using the RTBKT revealed uncovered du-path requirements to test feasible paths from defs in blocks ‘b4’, ‘b5’ and ‘b8’ to c-uses in block ‘b8’. Test cases ‘t06’ to ‘t08’ were added to satisfy these requirements:

\[
\% X = 6, \ K = 12 \\
\text{testcase(t06,[s,b1,b2,b3,b6,[b7,b8]*2,b7,b9,b11,r]).} \quad \% \text{for sub-path b8 to b8}
\]

\[
\% X = 7, \ K = 11 \\
\text{testcase(t07,[s,b1,b2,b4,b6,b7,b8,b7,b9,b11,r]).} \quad \% \text{for sub-path b4 to b8}
\]

\[
\% X = 12, \ K = 0 \\
\text{testcase(t08,[s,b1,b5,b6,b7,b8,b7,b9,b11,r]).} \quad \% \text{for sub-path b5 to b8}
\]

The RTKBT was then used to perform the structural test requirements analysis and test planning before any program change so the results may be compared with a regression test analysis.

Since no program change has been made yet, all test cases will be classified as reusable. The structural test coverage requirements and coverage of these requirements by the test cases are shown on the following pages.
The classified test cases are: \{t01, t02, t03, t04, t05, t06, t07, t08\}

Identifying definition-use paths...
The definition-use paths are [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57]
### Data Flow Based Test Coverage Criteria

<table>
<thead>
<tr>
<th>Path</th>
<th>Coverage Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 1</td>
<td>covered by test cases [t02]</td>
</tr>
<tr>
<td>Path 2</td>
<td>covered by test cases [t02, t08]</td>
</tr>
<tr>
<td>Path 3</td>
<td>covered by test cases [t01, t03, t04, t05, t06, t07]</td>
</tr>
<tr>
<td>Path 4</td>
<td>covered by test cases [t01]</td>
</tr>
<tr>
<td>Path 5</td>
<td>covered by test cases [t03]</td>
</tr>
<tr>
<td>Path 6</td>
<td>covered by test cases [t01, t04, t05, t06]</td>
</tr>
<tr>
<td>Path 7</td>
<td>covered by test cases [t03, t07]</td>
</tr>
<tr>
<td>Path 8</td>
<td>covered by test cases [t01, t04, t05, t06]</td>
</tr>
<tr>
<td>Path 9</td>
<td>covered by test cases [t03, t07]</td>
</tr>
<tr>
<td>Path 10</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 11</td>
<td>covered by test cases [t02]</td>
</tr>
<tr>
<td>Path 12</td>
<td>covered by test cases [t08]</td>
</tr>
<tr>
<td>Path 13</td>
<td>covered by test cases [t05]</td>
</tr>
<tr>
<td>Path 14</td>
<td>covered by test cases [t01]</td>
</tr>
<tr>
<td>Path 15</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 16</td>
<td>covered by test cases [t03]</td>
</tr>
<tr>
<td>Path 17</td>
<td>covered by test cases [t04, t06]</td>
</tr>
<tr>
<td>Path 18</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 19</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 20</td>
<td>covered by test cases [t04, t06, t07, t08]</td>
</tr>
<tr>
<td>Path 21</td>
<td>covered by test cases [t06]</td>
</tr>
<tr>
<td>Path 22</td>
<td>covered by test cases [t05]</td>
</tr>
<tr>
<td>Path 23</td>
<td>covered by test cases [t01, t02, t03, t04, t06, t07, t08]</td>
</tr>
<tr>
<td>Path 24</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 25</td>
<td>covered by test cases [t02, t08]</td>
</tr>
<tr>
<td>Path 26</td>
<td>covered by test cases [t02, t08]</td>
</tr>
<tr>
<td>Path 27</td>
<td>covered by test cases [t01, t03, t04, t05, t06, t07]</td>
</tr>
<tr>
<td>Path 28</td>
<td>covered by test cases [t08]</td>
</tr>
<tr>
<td>Path 29</td>
<td>covered by test cases [t05]</td>
</tr>
<tr>
<td>Path 30</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 31</td>
<td>covered by test cases [t01, t04, t05, t06]</td>
</tr>
<tr>
<td>Path 32</td>
<td>covered by test cases [t03, t07]</td>
</tr>
<tr>
<td>Path 33</td>
<td>covered by test cases [t01, t04, t05, t06]</td>
</tr>
<tr>
<td>Path 34</td>
<td>covered by test cases [t03, t07]</td>
</tr>
<tr>
<td>Path 35</td>
<td>covered by test cases [t04, t06]</td>
</tr>
<tr>
<td>Path 36</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 37</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 38</td>
<td>covered by test cases [t06]</td>
</tr>
<tr>
<td>Path 39</td>
<td>covered by test cases [t01, t03, t04, t05, t06, t07]</td>
</tr>
<tr>
<td>Path 40</td>
<td>covered by test cases [t02, t08]</td>
</tr>
<tr>
<td>Path 41</td>
<td>covered by test cases [t01, t04, t05, t06]</td>
</tr>
<tr>
<td>Path 42</td>
<td>covered by test cases [t03, t07]</td>
</tr>
<tr>
<td>Path 43</td>
<td>covered by test cases [t01, t02, t03]</td>
</tr>
<tr>
<td>Path 44</td>
<td>covered by test cases [t04, t06, t07, t08]</td>
</tr>
<tr>
<td>Path 45</td>
<td>covered by test cases [t04, t06, t07, t08]</td>
</tr>
<tr>
<td>Path 46</td>
<td>covered by test cases [t01, t02, t03, t05]</td>
</tr>
<tr>
<td>Path 47</td>
<td>covered by test cases [t06]</td>
</tr>
<tr>
<td>Path 48</td>
<td>covered by test cases [t04, t06, t07, t08]</td>
</tr>
<tr>
<td>Path 49</td>
<td>covered by test cases [t05]</td>
</tr>
<tr>
<td>Path 50</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 51</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 52</td>
<td>covered by test cases [t05]</td>
</tr>
<tr>
<td>Path 53</td>
<td>covered by test cases [t01]</td>
</tr>
<tr>
<td>Path 54</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 55</td>
<td>covered by test cases [t03]</td>
</tr>
<tr>
<td>Path 56</td>
<td>covered by test cases [ ]</td>
</tr>
<tr>
<td>Path 57</td>
<td>covered by test cases [t04, t06, t07, t08]</td>
</tr>
</tbody>
</table>

The results of the test planning phase (shown below) indicate the following:

- test case ‘t04’ is redundant (test case ‘t06’ covers all requirements covered by test case ‘t04’ plus a few more);
Chapter 4  Data Flow Based Test Coverage Criteria

- as expected, the regression test set is empty since there has been no program change; and

- several def-use paths are not covered by any test case; these will be discussed later.

```
RTKBT> plan
Determining Structural Test Set
and Regression Test Set...

The Structural Test Set is:
[t01, t02, t03, t05, t06, t07, t08]

The hitting set algorithm has found
the following test cases to be redundant:
t04

The Regression Test Set is:
{}.
All arcs have been covered.
The following def-use paths have not been covered.
[10, 15, 19, 24, 30, 37, 50, 51, 54, 56]
Press <Enter> to continue.
```

The listed def-use paths that have not been covered contain an infeasible sub-path:

du-paths 10 and 24 contain sub-path [ b5,b6,b7,b9,b10 ];
du-paths 15, 30, 50 and 54 contain sub-path [ b4,b6,b7,b9,b10 ]; and
du-paths 19, 37, 51 and 56 contain sub-path [ b8,b7,b9,b10 ].

The infeasible sub-paths (there exist no combination of input values that will cause
the flow of control to traverse the sequence of nodes) as determined manually are
explained below:

[ b5,b6,b7,b9,b10 ] - there is no definition of Z along path [ s,b1,b5,b6,b7,b9 ] to
force the p-use (Z>40) to be True in order to branch to node b10 so the sub-path
from node b5 to b10 is not feasible (this type of data flow anomaly is normally detectable by data flow static analysis [Ntaf84]);

[ b4,b6,b7,b9,b10 ] - there can be no values of X and K such that K <= X and Z > 40, so the sub-path from node b4 to b10 is not feasible; and

[ b8,b7,b9,b10 ] - the while loop will always exit with x = 13, y = 26 and z = 39, so the branch from node b9 to b10 in which the predicate (Z>40) is True can never be reached from inside the while loop, therefore the sub-path from node b8 to b10 is not feasible.

Now that the example program flow graph analysis has identified the du-path coverage requirements and the set of test cases that will satisfy them, a program change is introduced to illustrate the effect on the du-path requirements and the set of test cases required in accordance with the all du-paths criterion to perform a regression test for the change.

The insertion/deletion of a c-use will create/remove def-use pairs and associated du-paths. The insertion of a def will create new def-use pairs and associated du-paths, but if the def appears along an existing du-path of the same variable, it will cause that du-path to be removed. Similarly, the deletion of a def will remove def-use pairs and associated du-paths, but may cause paths between other def-use pairs of that variable to become definition-clear, thereby creating new du-paths.
The program change described in [Harr88] consists of the two edits; the first edit causes the insertion of the statement $K := Z$ before line 12 in the code while edit 2 causes the insertion of the same statement after line 14 in the code.

![Data flow graph modification](image)

Both edits occur in block 'b8', so all test cases whose trajectories traverse that node are expected to be re-testable. The edits do not add, remove or modify any predicate (there are no new or deleted nodes or arcs) so the change is not expected to cause any test case to become unclassified.

```
RTKBT> classify
Classifying test cases...
The test cases are [t01, t02, t03, t04, t05, t06, t07, t08]
Test case t01 is Reusable.
Test case t02 is Reusable.
Test case t03 is Reusable.
Test case t04 is Re-testable.
Test case t05 is Reusable.
Test case t06 is Re-testable.
Test case t07 is Re-testable.
Test case t08 is Re-testable.
```
The du-paths and coverage by test cases follow:

```plaintext
Def-use path 1 is a c_use path for variable k : [b8, b7, b9, b11].
Def-use path 2 is a c_use path for variable k : [s, b1, b5, b6, b7, b9, b11].
Def-use path 3 is a c_use path for variable k : [s, b1, b5].
Def-use path 4 is a c_use path for variable k : [s, b1, b2].
Def-use path 5 is a c_use path for variable k : [b2, b3, b6, b7, b9, b11].
Def-use path 6 is a c_use path for variable k : [b2, b4, b6, b7, b9, b11].
Def-use path 7 is a c_use path for variable k : [b2, b3].
Def-use path 8 is a c_use path for variable k : [b2, b4].
Def-use path 9 is a c_use path for variable k : [b2, b3].
Def-use path 10 is a p_use path for variable k : [b2, b4].
Def-use path 11 is a c_use path for variable m : [b5, b6, b7, b9, b10, r].
Def-use path 12 is a c_use path for variable m : [b5, b6, b7, b9, b11, r].
Def-use path 13 is a c_use path for variable m : [b5, b6, b7, b8].
Def-use path 14 is a c_use path for variable m : [s, b1, b2, b3, b6, b7, b9, b10, r].
Def-use path 15 is a c_use path for variable m : [s, b1, b2, b3, b6, b7, b9, b11, r].
Def-use path 16 is a c_use path for variable m : [s, b1, b2, b4, b6, b7, b9, b10, r].
Def-use path 17 is a c_use path for variable m : [s, b1, b2, b4, b6, b7, b9, b11, r].
Def-use path 18 is a c_use path for variable m : [s, b1, b2, b3, b6, b7, b8].
Def-use path 19 is a c_use path for variable m : [s, b1, b2, b4, b6, b7, b8].
Def-use path 20 is a c_use path for variable m : [b8, b7, b9, b10, r].
Def-use path 21 is a c_use path for variable m : [b8, b7, b9, b11, r].
Def-use path 22 is a c_use path for variable m : [b9, b7, b8].
Def-use path 23 is a c_use path for variable out : [b10, r].
Def-use path 24 is a c_use path for variable out : [b11, r].
Def-use path 25 is a c_use path for variable x : [s, b1, b5, b6, b7, b9, b10].
Def-use path 26 is a c_use path for variable x : [s, b1, b5].
Def-use path 27 is a c_use path for variable x : [s, b1, b5, b6].
Def-use path 28 is a c_use path for variable x : [s, b1, b2].
Def-use path 29 is a c_use path for variable x : [s, b1, b5, b6, b7, b8].
Def-use path 30 is a c_use path for variable x : [b2, b3, b6, b7, b9, b10].
Def-use path 31 is a c_use path for variable x : [b2, b4, b6, b7, b9, b10].
Def-use path 32 is a c_use path for variable x : [b2, b3].
Def-use path 33 is a c_use path for variable x : [b2, b4].
Def-use path 34 is a c_use path for variable x : [b2, b3, b6].
Def-use path 35 is a c_use path for variable x : [b2, b4, b6].
Def-use path 36 is a c_use path for variable x : [b2, b3, b6, b7, b8].
Def-use path 37 is a c_use path for variable x : [b2, b4, b6, b7, b8].
Def-use path 38 is a c_use path for variable x : [b8, b7, b9, b10].
Def-use path 39 is a c_use path for variable x : [b8, b7, b8].
Def-use path 40 is a p_use path for variable x : [s, b1, b2].
Def-use path 41 is a p_use path for variable x : [s, b1, b5].
Def-use path 42 is a p_use path for variable x : [b2, b3].
Def-use path 43 is a p_use path for variable x : [b2, b4].
Def-use path 44 is a c_use path for variable y : [b6, b7, b9, b11].
Def-use path 45 is a c_use path for variable y : [b8, b7, b9, b11].
Def-use path 46 is a p_use path for variable y : [b6, b7, b8].
Def-use path 47 is a p_use path for variable y : [b6, b7, b9].
Def-use path 48 is a p_use path for variable y : [b8, b7, b8].
Def-use path 49 is a p_use path for variable y : [b8, b7, b9].
Def-use path 50 is a c_use path for variable z : [b3, b6, b7, b9, b10].
Def-use path 51 is a c_use path for variable z : [b3, b6, b7, b8].
Def-use path 52 is a c_use path for variable z : [b4, b6, b7, b9, b10].
Def-use path 53 is a c_use path for variable z : [b4, b6, b7, b8].
Def-use path 54 is a c_use path for variable z : [b8, b7, b9, b10].
Def-use path 55 is a c_use path for variable z : [b8, b7, b8].
Def-use path 56 is a p_use path for variable z : [b3, b6, b7, b9, b10].
Def-use path 57 is a p_use path for variable z : [b3, b6, b7, b9, b11].
Def-use path 58 is a p_use path for variable z : [b4, b6, b7, b9, b10].
Def-use path 59 is a p_use path for variable z : [b4, b6, b7, b9, b11].
Def-use path 60 is a p_use path for variable z : [b8, b7, b9, b10].
Def-use path 61 is a p_use path for variable z : [b8, b7, b9, b11].
```
Path 1 is covered by test cases [t04, t06, t08]
Path 2 is covered by test cases [t02]
Path 3 is covered by test cases [t02, t08]
Path 4 is covered by test cases [t01, t03, t04, t05, t06, t07]
Path 5 is covered by test cases [t01]
Path 6 is covered by test cases [t03]
Path 7 is covered by test cases [t01, t04, t05, t06]
Path 8 is covered by test cases [t03, t07]
Path 9 is covered by test cases [t01, t04, t05, t06]
Path 10 is covered by test cases [t03, t07]
Path 11 is covered by test cases []
Path 12 is covered by test cases [t02]
Path 13 is covered by test cases [t08]
Path 14 is covered by test cases [t05]
Path 15 is covered by test cases [t01]
Path 16 is covered by test cases []
Path 17 is covered by test cases [t03]
Path 18 is covered by test cases [t04, t06]
Path 19 is covered by test cases [t07]
Path 20 is covered by test cases []
Path 21 is covered by test cases [t04, t06, t07, t08]
Path 22 is covered by test cases [t06]
Path 23 is covered by test cases [t05]
Path 24 is covered by test cases [t01, t02, t03, t04, t06, t07, t08]
Path 25 is covered by test cases []
Path 26 is covered by test cases [t02, t08]
Path 27 is covered by test cases [t02, t08]
Path 28 is covered by test cases [t01, t03, t04, t05, t06, t07]
Path 29 is covered by test cases [t08]
Path 30 is covered by test cases [t05]
Path 31 is covered by test cases []
Path 32 is covered by test cases [t01, t04, t05, t06]
Path 33 is covered by test cases [t03, t07]
Path 34 is covered by test cases [t01, t04, t05, t06]
Path 35 is covered by test cases [t03, t07]
Path 36 is covered by test cases [t04, t06]
Path 37 is covered by test cases [t07]
Path 38 is covered by test cases []
Path 39 is covered by test cases [t06]
Path 40 is covered by test cases [t01, t03, t04, t05, t06, t07]
Path 41 is covered by test cases [t02, t08]
Path 42 is covered by test cases [t01, t04, t05, t06]
Path 43 is covered by test cases [t03, t07]
Path 44 is covered by test cases [t01, t02, t03]
Path 45 is covered by test cases [t04, t06, t07, t08]
Path 46 is covered by test cases [t04, t06, t07, t08]
Path 47 is covered by test cases [t01, t02, t03, t05]
Path 48 is covered by test cases [t06]
Path 49 is covered by test cases [t04, t06, t07, t08]
Path 50 is covered by test cases [t05]
Path 51 is covered by test cases [t04, t06]
Path 52 is covered by test cases []
Path 53 is covered by test cases [t07]
Path 54 is covered by test cases []
Path 55 is covered by test cases [t06]
Path 56 is covered by test cases [t05]
Path 57 is covered by test cases [t01]
Path 58 is covered by test cases []
Path 59 is covered by test cases [t03]
Path 60 is covered by test cases []
Path 61 is covered by test cases [t04, t06, t07, t08]

Du-paths 1, 51, 53 and 55 are new du-paths created as a result of the program change that require test coverage. Each edit is considered separately.
Before the program change, block 'b8' was a 'def' block with respect to variable Z. After edit 1, the insertion of the c-use of Z preceding the def of Z causes block 'b8' to become a 'use-def' block with respect to Z, making it eligible as a destination node for du-paths originating in the def blocks 'b3', 'b4' and 'b8'. The new du-paths 51, 53 and 55 are def to c-use paths for variable Z.

Before the program change, block 'b8' was a pass-through block with respect to variable K. Both edits would cause block 'b8' to become a 'def' block with respect to K, but since the second edit prevents any definition-clear paths originating from the first definition, only the second definition is considered. The only use of variable K that is reachable from 'b8' is located in block 'b11'. The new du-path 1 is the associated def to c-use path for variable K. Now the 'plan' step is executed.

RTKBT> plan
Determining Structural Test Set
and Regression Test Set...

The Structural Test Set is:
[t01, t02, t03, t05, t06, t07, t08]

The hitting set algorithm has found the following test cases to be redundant:
[t04]

The Regression Test Set is:
[t06, t07, t08]

All arcs have been covered.
The following def-use paths have not been covered:
[11, 16, 20, 25, 31, 38, 52, 54, 58, 60]

Press <Enter> to continue.
The results of the plan step shown on the previous page indicate the following:

- as before, test case ‘t04’ is redundant;
- the regression test set consists of the re-testable test cases ‘t06’, ‘t07’ and ‘t08’ and excludes the redundant re-testable test case ‘t04’;
- as before, several definition-use paths are not covered by any test case; these du-paths contain the same infeasible sub-paths discussed previously (du-paths 11 and 25 contain sub-path \([b5,b6,b7,b9,b10]\), du-paths 16, 31, 52 and 58 contain sub-path \([b4,b6,b7,b9,b10]\) and du-paths 20, 38, 54 and 60 contain sub-path \([b8,b7,b9,b10]\)); and
- since the existing test cases satisfy all the feasible du-paths and all branches after the program change, test coverage is considered adequate and therefore no additional test cases are required.

4.3 Strengthening the All Du-paths Criterion

The method of determining a test set for the all du-paths test coverage criterion does not take into consideration the possibility of interactions between du-paths. It simply attempts to ensure that all identified du-paths are covered by some test case in the test set. It is possible that a fault may exist that will not be observed if a series of du-paths are not exercised in conjunction with each other. Given that the data flow graph of a program is known and the du-paths to be covered by testing have been identified, I present a method of manually asserting that particular du-paths be tested together.
4.3.1 Chaining Du-path Requirements

The goal of the all du-paths criterion is to ensure that the paths from all variable definitions to their uses are executed by some test case. This however, will not ensure the execution of paths that originate at the node of one variable definition, pass through a node where the first variable is used in the computation of the value to be assigned in the definition of another variable, and subsequently to a node in which the second variable is used. Such a combination of du-paths is called a k-dr interaction (of k definition-references) [Ntaf84] or a du-chain [Zeil88]. The assertion of du-chains as test coverage requirements can be enabled by ‘chaining’ du-path requirements together to form a new ‘chain’ requirement as was done in section 3.6.1.

To demonstrate, the example from section 4.2 is used in which requirement 53 (a du-path from block ‘b4’ to block ‘b8’) and requirement 55 (a du-path from block ‘b8’ back to block ‘b8’) are ‘chained’. Recall that requirements 53 and 55 (see page 71) were covered by test cases ‘t07’ and ‘t06’ respectively (see page 72), but the new requirement 62 is not covered by any test case (see page 76).

```
RTKB> chain
Add a requirement chain:
Enter the first requirement to chain: 53
Enter the second requirement to chain: 55
Add more requirements to chain? (y/n) n
[62, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61]
Press <Enter> to continue.
```
RTKBT> list requirements
.
.
Def-use path 53 is a c-use path for variable z: [b4, b6, b7, b8].
Def-use path 54 is a c-use path for variable z: [b8, b7, b9, b10].
Def-use path 55 is a c-use path for variable z: [b8, b7, b8].
Def-use path 56 is a p-use path for variable z: [b3, b6, b7, b9, b10].
Def-use path 57 is a p-use path for variable z: [b3, b6, b7, b9, b11].
Def-use path 58 is a p-use path for variable z: [b4, b6, b7, b9, b10].
Def-use path 59 is a p-use path for variable z: [b4, b6, b7, b9, b11].
Def-use path 60 is a p-use path for variable z: [b8, b7, b9, b10].
Def-use path 61 is a p-use path for variable z: [b8, b7, b9, b11].
Chain 62 consists of requirements [55, 53].

Press <Enter> to continue.
.
.
RTKBT> plan
Determining Structural Test Set
and Regression Test Set...

The Structural Test Set is:
[t01, t02, t03, t05, t06, t07, t08]

The hitting set algorithm has found
the following test cases to be redundant:
[t04]

The Regression Test Set is:
[t06, t07, t08]

All arcs have been covered.
The following def-use paths have not been covered:
[11, 16, 20, 25, 31, 38, 52, 54, 58, 60]

The following chains have not been covered:
[62]

Press <Enter> to continue.

A new test case ‘t10’, which traverses the path [b4, b6, b7, b8] and traverses the
cycle [b7, b8, b7] more than once, can be added to satisfy the new chain requirement:

% X = 0, K = 0
testcase(t10,[s,b1,b2,b4,b6,[b7,b8]*8,b7,b9,b11,r]).

With the addition of the new test case, the structural and regression test sets now
cover the new chain requirement as shown on the following page.
Determining Structural Test Set and Regression Test Set...

The Structural Test Set is:
\{t01, t02, t03, t05, t08, t10, t06\}

The hitting set algorithm has found the following test cases to be redundant:
\{t04, t07\}

The Regression Test Set is:
\{t08, t10, t06\}

All arcs have been covered. The following def-use paths have not been covered:
\{11, 16, 20, 25, 31, 38, 52, 54, 58, 60\}

All chains have been covered.
Press <Enter> to continue.

Additional research is required to implement an automatic method of generating du-path chains of a given length. The representation of variable usage in basic blocks would have to include the association between defined variables and the variables used to compute the value assigned to the defined variable in order to permit the automatic generation of du-path chains.

4.4 Summary

The selection of test cases by the RTKBT for regression testing based on the all du-paths coverage criterion has been demonstrated. The all du-paths criterion is data flow based and provides a level of coverage based on the flow of data between blocks in which variables are defined and subsequently used. If not all du-paths are feasible (as determined by the user), the weaker 'feasible du-paths' criterion may be used. Since the feasible du-
paths criterion will not ensure the coverage of all branches, branch coverage is explicitly checked by the RTKBT. The assertion of additional chain requirements provides a mechanism for strengthening the du-paths criterion to that of du-path chains. The implementation of a test coverage criterion for du-chains of a specified length requires further research.
Chapter 5

5. Controlling the Size of the Test Suite

Regression testing deals with two problems: the test suite update problem and the test case selection problem [Harr90]. These problems give rise to a series of related regression testing problems beginning with the identification of testing requirements (e.g. basis path coverage for the control flow method or du-path coverage for the data flow method as described in chapters 3 and 4) and the determination of the requirement coverage provided by each test case, followed by the development of new test cases if required. A solution for these initial problems using control and data flow methods were described in the preceding chapters. The results of the structural test requirements analysis and the associations between the test cases and the requirements they satisfy are used for the regression test case selection and test suite update problems.

The test suite minimization and test case selection problem involves determining the minimum set of test cases containing at least one test case that satisfies each of the requirements. A linear approach involves the dynamic analysis of test cases from the test suite in whatever order they were cataloged until the desired coverage requirement is met. This linear approach of selecting test cases may not produce a minimized test suite [Harr90]. Chances are that some test cases would become redundant if they had been selected in a different order. So the problem becomes one of determining which test cases to select first.
A test case selection approach based on the number of requirements covered by each test case is presented. A second approach based on the number of test cases that satisfy each requirement is presented and compared with the first to demonstrate which method can produce a smaller set of test cases that satisfies the same set of requirements.

5.1 A Test Cases Approach

One approach is to select test cases based on which one covers the most requirements that have not been covered by a previously selected test case. This is based on the heuristic that a selected test case is more likely to cause another to become redundant if it covers more requirements. To demonstrate this approach, which will be referred to as the test cases approach, it is described more formally:

Given: a test suite TS consisting of test cases \([t_1, t_2, \ldots t_n]\) each of which may be used to test one or more of the testing requirements \([r_1, r_2, \ldots r_m]\). A set of test cases which satisfies all the requirements is called the Hitting Set, HS.

Algorithm: begin with a null hitting set (HS = \([\ ]\)), a null requirements covered set (RCS = \([\ ]\)) and the set of all available test cases TS = \([t_1, t_2, \ldots t_n]\). A test case is selected from TS which covers the greatest number of requirements that are not in the set RCS (i.e. not already covered by test cases in the HS). In the case of a tie, the test case that covers the greater number of requirements (including covered requirements) is
chosen. If this does not resolve the tie, a random choice is made. The selected test case is removed from the set TS, added to the set HS and the set of requirements covered by the selected test case is added to the set RCS. The selection of test cases continues until all requirements are covered.

Using the example from [Harr90], the relationships between test cases and the requirements each covers, the steps taken and the results are shown in Table 1. This method is simpler than that of [Harr90] and it may seem to produce the same results, but this is not always the case, as will be shown later.

<table>
<thead>
<tr>
<th>Test case</th>
<th>requirements covered</th>
<th>requirements remaining after step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c, e, f</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>a, c, h</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>c, d, g, h</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>e, g, h</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>a, b</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>d, f</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>g, h</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>step</th>
<th>hitting set HS</th>
<th>requirements covered RCS</th>
<th>available test cases TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[1, 2, 3, 4, 5, 6, 7]</td>
</tr>
<tr>
<td>1</td>
<td>[3]</td>
<td>[c, d, g, h]</td>
<td>[1, 2, 4, 5, 6, 7]</td>
</tr>
<tr>
<td>2</td>
<td>[3, 1]</td>
<td>[c, d, e, f, g, h]</td>
<td>[2, 4, 5, 6, 7]</td>
</tr>
<tr>
<td>3</td>
<td>[3, 1, 5]</td>
<td>[a, b, c, d, e, f, g, h]</td>
<td>[2, 4, 6, 7]</td>
</tr>
</tbody>
</table>

Table 1. Hitting set example 1 - a test cases approach

The method described in [Harr90] produces the hitting set [1, 3, 5]; the same test cases are selected but in a different order.
It is not clear from the above example that the simpler test case approach will not always produce a minimum hitting set. The problem occurs with the order in which test cases are selected and is most easily demonstrated for requirements that are covered by a single test case. In the test case approach demonstrated above, if a test case which is the only test case that satisfies a particular requirement is selected after another test case which would have been identified as 'redundant' had the test cases been selected in the reverse order, the 'redundant' test case will not have been eliminated. An example of this is shown in Table 2.

<table>
<thead>
<tr>
<th>Test case</th>
<th>requirements covered</th>
<th>requirements remaining after step</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>1</td>
<td>b, c, d</td>
<td>3 0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>c, d, f</td>
<td>3 1 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>a, b</td>
<td>2 1 1 0 0</td>
</tr>
<tr>
<td>4</td>
<td>d, e</td>
<td>2 1 1 1 0</td>
</tr>
</tbody>
</table>

Table 2. Hitting set example 2 - a test cases approach.

It may not be immediately obvious using the above representation and the simpler test case algorithm that test case 1 is redundant and yet was not eliminated from the hitting set. This becomes more apparent when the problem is approached from the point of view of the requirements.
5.2 A Requirements Approach

The methodology presented in [Harr90] approaches the problem from the point of view of the requirements and the size of the subsets of test cases that cover each requirement. This is based on the heuristic that requirements which are covered by only one test case should be satisfied first, since any hitting set solution must contain these test cases. The heuristic presented in [Harr90], which will be referred to as the requirements approach, is as follows:

Given: a test suite TS, a test requirements set RS = [r₁, r₂, ..., rₘ] that must be tested to provide the desired testing coverage of the program and a list of subsets of TS, [T₁, T₂, ..., Tₙ] associated with each of the rᵢ's such that any one of the test cases tᵢ belonging to Tᵢ can be used to test the requirement rᵢ. A representative set of test cases which satisfies the testing requirements must contain at least one test case from each Tᵢ and is called the Hitting Set, HS.

Algorithm: All test cases that occur in single element Tᵢ's (singleton subsets) are first included in the hitting set HS and all Tᵢ's containing any of these elements are marked. Then all unmarked Tᵢ's of cardinality two are considered. Test cases are selected based on the maximum number of unmarked Tᵢ's until no more Tᵢ's of cardinality two remain. This process is repeated for Tᵢ's of increasing cardinality until no more Tᵢ's remain. In the case of a tie when considering unmarked Tᵢ's of cardinality n, unmarked Tᵢ's of cardinality n+1 are examined for the tying test cases. If the tie cannot be broken, Tᵢ's of greater cardinality are examined and finally a random choice is made.
Using this algorithm, the hitting set for example 2 above is determined in Table 3 as follows:

<table>
<thead>
<tr>
<th>requirements</th>
<th>test cases covering requirement</th>
<th>marked (*) subsets after step</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>[3]</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>[1, 3]</td>
<td>*</td>
</tr>
<tr>
<td>c</td>
<td>[1, 2]</td>
<td>*</td>
</tr>
<tr>
<td>d</td>
<td>[1, 2]</td>
<td>*</td>
</tr>
<tr>
<td>e</td>
<td>[4]</td>
<td>*</td>
</tr>
<tr>
<td>f</td>
<td>[2]</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 3. Hitting set example 2 - a requirements approach.

Notice that after all the test cases belonging to singleton sets have been selected, all requirements have been covered, so there is no need for test case 1. The problem of ordering the selection of test cases occurs at all levels of cardinality (i.e. doubleton, tripleton subsets, etc.) but is most obvious in the case of the singleton subsets.

### 5.3 Knowledge Base Implementation

The test case selection method presented in [Harr90] has been implemented in the knowledge based tool where the information regarding the test coverage requirements and the coverage provided by existing test cases are represented as Prolog facts and the rules used to solve the problem are implemented as Prolog clauses.
The association between the test cases and the requirements they satisfy for a particular unit of software are represented in the domain base of the RTKBT as a series of assertions of the form:

\[ \text{covers}(T, R) \] where \( T \) and \( R \) are the unique identifiers of a test case and the requirement it covers (satisfies).

These testcase-requirement association facts are used to infer the facts that represent the subsets of test cases that cover each requirement in the form:

\[ \text{cov_by}(R, TS, N) \]

where \( R \) identifies the requirement (an alphabetic character in the examples), \( TS \) is the subset of test cases that cover the requirement (each test case is identified by a number in the examples) and \( N \) is the cardinality (size) of the set \( TS \). The cardinality \( N \) of the set \( TS \) could be determined during the execution of the hitting set algorithm, but is asserted as part of the \( \text{cov_by} \) fact to simplify programming.

The hierarchy of clauses that define the rules for determining the hitting set, as shown in Figure 11, are contained in the knowledge base of the RTKBT. These clauses have been implemented in Prolog and are detailed in the technical report associated with this thesis [Midd97].
5.3.1 Enhancement of the Tie Resolution Strategy

In [Harr90], if a tie between test cases cannot be resolved by checking subsets of higher cardinality, a random choice is made. The pick-test clause of the RTKBT implements an enhancement to this strategy. If a tie cannot be resolved at a higher level of cardinality, the test case which covers the most requirements (regardless of requirements already covered by test cases in the HS) is selected. If however, the test cases involved in the tie all cover the same number of requirements, then the random choice is made. The selection of the test case that covers more requirements has two advantages over a randomly chosen test case covering fewer requirements. It increases the possibility of producing a smaller hitting set after the next software modification, assuming that redundant test cases are deleted, as the selected test case will be a member of more of the
requirement subsets. It will also increase the breadth of coverage in that some requirements will be covered by more test cases in the updated test plan. This should increase the possibility of finding an error.

To demonstrate, another example from [Harr90] is shown in Table 4.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>test cases covering requirement</th>
<th>marked (*) subsets after step:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>[1, 2, 5]</td>
<td>*</td>
</tr>
<tr>
<td>b</td>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>[1, 2, 5, 6]</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>[3, 4, 6]</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>[4, 6]</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>[3, 4, 6]</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>[1, 2]</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>[6]</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>[3, 4]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>step</th>
<th>hitting set HS</th>
<th>requirements covered</th>
<th>available test cases TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[1, 2, 3, 4, 5, 6]</td>
</tr>
<tr>
<td>1</td>
<td>[2, 5, 6]</td>
<td>[a, b, c, d, e, f, g, h, i]</td>
<td>[1, 3, 4]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 5, 6, 4]</td>
<td>[a, b, c, d, e, f, g, h, i, j]</td>
<td>[1, 3]</td>
</tr>
</tbody>
</table>

Table 4. Hitting set example 3 - tie resolution enhancement

The tie resolution strategy above produces the HS = [2,5,6,4] compared with a HS = [2,5,6,3] produced in [Harr90]; test case 4 is chosen instead of test case 3 since test case 4 covers the same requirements as test case 3 plus an additional requirement.

5.3.2 Test Suite Update

The hitting set clause of the RTKB takes as input a set of requirements and produces as output the hitting set that will provide coverage of as many of the input requirements as possible. While the adequacy of the hitting set produced will have to be
determined, this is preferred over a failure of the hitting set goal, as a partial solution is considered to be better than no solution. This is especially true with respect to regression testing, as there are often structural elements after a modification that cannot be satisfied by the previous test suite.

To select test cases for an updated test suite, the list of all requirements is required as input for the hitting set clause. This list is provided by the list-all-reqs clause, so the updated test plan goal becomes the conjunction:

\[ \text{list-all-reqs}(RS), \text{hitset}(RS, HS). \]

For example 3 of Table 4 above, the response from the Prolog interpreter is:

\[ RS = [a, b, c, d, e, f, g, h, i, j] \]
\[ HS = [2, 5, 6, 4] ; \]

so all test cases that are not members of the hitting set HS above are redundant and may be eliminated from the test suite.

5.3.3 Regression Test Set

In the case of regression testing in which a hitting set is needed for only a subset of the total testing requirements, the hitting set HS for the set of regression-testable test cases RTS can be determined by posing the following question:

\[ \text{hitset}(RTS, HS). \]
For example, using the requirements and test case information from Table 4 above, if it was determined that after a software modification, requirements a, b, c, d, g and h had to be re-tested, the hitting set of test cases could be determined by asking the interpreter the following question:

\[
\text{hitset([a, b, c, d, g, h], HS).}
\]

and the Prolog interpreter would respond with:

\[
\text{HS = [2, 5, 6];}
\]

so only test cases 2, 5 and 6 would have to be rerun.

Note that after the singleton test cases 2 and 5 have been selected, the only remaining 'unmarked' regression testable requirement is ‘d’, which is covered by test cases 3, 4, and 6. Since this tie cannot be resolved by checking sub-sets of greater cardinality, test case 6 is chosen in favor of 3 or 4 as it covers the most requirements.

### 5.4 Summary

An algorithm to select a representative set of test cases of reduced size that will satisfy a given set of requirements has been presented. This algorithm has been adapted from the algorithm in [Harr90] which selects test cases based on the size of the subsets of test cases that cover each requirement. The algorithm has been enhanced by modifying the tie resolution strategy to select test cases that cover a greater number of requirements before making a random selection.
Chapter 6

6. A Regression Testing Model Using the RTKBT

6.1 Regression Test Process

Use of the Regression Testing Knowledge Based Tool depends on the availability of static and dynamic analysis information from some automated tool. Source code static analyzers examine source code without executing it. The digraph information required by the RTKBT could be produced by a type of static analyzer called a structure checker [Daic94]. Test execution tools dynamically analyze the software to be tested. These tools generally require instrumentation (the addition of special code) of the code to be able to monitor the execution paths. The test case trajectory information (executed paths) required by the RTKBT could be produced by a type of test execution tool called a coverage/frequency analyzer [Daic94]. Numerous exiting static and dynamic analysis tools are listed in [Daic94], such as C-Cover, cflow and C Source Analyzer for software written in the C programming language. The RTKBT was not developed with any particular analysis tools in mind and would have to be adapted to suit the analysis tools used in practice.

The steps for the model of regression testing for use with the RTKBT are outlined in Figure 12 (adapted from [Leun89]). Use of the RTKBT is indicated in the three boxes that mention ‘Apply’ or ‘Re-apply’ the RTKBT. This model is a theoretical model; its applicability in practice requires further research regarding the efficiency of the process.
The first step after a change is a static analysis of the modified software. A static analysis tool is needed which will produce the change digraph representing the control flow of the modified software, including an indication of new or modified nodes and arcs as compared with the previous version of the software. Next the RTKBT is used to classify test cases based on the new flow graph data and on the stored dynamic analysis information describing the trajectories of test cases from the last time they were executed (the problem that test cases may traverse the flow graph differently as a result of the...
modification is dealt with later in the process). The flow graph is analyzed by the RTKBT to determine the structural requirements for a chosen coverage criterion, the associations between these requirements and the test cases that satisfy them are inferred and test sets of reduced size that will provide coverage of all requirements as well as modified requirements are produced. Any structural requirement not covered by a test case is identified for the user.

The process continues as described in Figure 12 by dynamically analyzing test cases and re-applying the RTKBT as required. The RTKBT must be re-applied after the dynamic analysis (execution) of test cases in order to assess the effect of any change of a test case trajectory on the coverage of structural requirements and the subsequent selection of test cases for the updated structural and regression test sets. It should be noted that the user may have to decide that the coverage is adequate when the RTKBT identifies any structural requirements that represent infeasible paths which have not been (and cannot be) satisfied by any test case.

The infeasibility of paths has been determined manually for this thesis. It is known that in general there can be no algorithm to decide whether or not a given path is feasible [Rapp85]. Some existing automated methods of attempting to determine the feasibility of paths include the symbolic evaluation of path predicates (a path predicate is a predicate that describes the legal condition under which a particular sequence of logical branches will be executed [Test97]) and dynamic data flow analysis associated with test data.
generation techniques [Kore90]. The application of these techniques is beyond the scope of this thesis.

### 6.2 An Example

An example program used by both [Rapp85] and [Leun89] which calculates $x$ to the power of $y$, with both $x$ and $y$ integers, is also used here. Specification-based test cases for this example (taken from [Leun89]) are all possible combinations of:

- $x$ input: $x < -1$ (assume $x = -2$), $x = -1$, $x = 0$, $x > 1$ (assume $x = 2$); and
- $y$ input: $y = 0$, $y = 1$, $y < 0$ (assumes the test case used is $y = -1$).

An additional structural test case to iterate the while loop 2 or more times is included:

$x = 4$, $y = 2$.

The test cases are given identifiers using a prefix of ‘sp’ for specification-based test cases and ‘st’ for structural test cases (for discussion purposes only) as shown in Table 5.

<table>
<thead>
<tr>
<th>Test case id</th>
<th>$X$</th>
<th>$Y$</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp01</td>
<td>-2</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>sp02</td>
<td>-2</td>
<td>1</td>
<td>-2.0</td>
</tr>
<tr>
<td>sp03</td>
<td>-2</td>
<td>-1</td>
<td>-0.5</td>
</tr>
<tr>
<td>sp04</td>
<td>-1</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>sp05</td>
<td>-1</td>
<td>1</td>
<td>-1.0</td>
</tr>
<tr>
<td>sp06</td>
<td>-1</td>
<td>-1</td>
<td>-1.0</td>
</tr>
<tr>
<td>sp07</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
6.2.1 Control Flow

In order to provide a point of comparison prior to modification for regression testing, a control flow analysis will be performed for the unchanged example program using the simple basis paths method. The C source code and control flow graph are shown in Figure 13.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sp08</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sp09</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>sp10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sp11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sp12</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>sp13</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sp14</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>sp15</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>st01</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5. Power example test cases.
float power(int x, int y) {
    int power;
    float z, answer;
    if (y < 0) {
        power = -y;
    } else {
        power = y;
    }
    z = 1.0;
    while (power != 0) {
        z = z * x;
        power = power -1;
    }
    if (y < 0) {
        z = 1/z;
    }
    answer = z;
    return answer;
}

Figure 13. Power control flow example.

Since there has been no program change yet, all test cases will be classified as reusable. The test coverage requirements analysis and structural test set is shown below.

Simple paths 2 and 3 are infeasible as ‘y’ cannot be both less than 0 and greater than or equal to 0. Notice that the ‘feasible’ simple basis paths are covered by just two of the specification test cases - ‘sp02’ and ‘sp03’.
RTKBT> analyze

Purging old facts...

Select the methodology to be used:
Enter "control_flow" or "data_flow" : control_flow

The classified test cases are: [sp01, sp02, sp03, sp04, sp05, sp06, sp07, sp08, sp09, sp10, sp11, sp12, sp13, sp14, sp15, st01]

Identifying simple paths...

The simple paths are [1, 2, 3, 4, 5]

Simple path 1 is path : [start, yLT0, yLT0F, yLT0endif, powerNE0, yLT02, end].
Simple path 2 is path : [start, yLT0, yLT0F, yLT0endif, powerNE0, yLT02, yLT02T, end].
Simple path 3 is path : [start, yLT0, yLT0T, yLT0endif, powerNE0, yLT02, end].
Simple path 4 is path : [start, yLT0, yLT0T, yLT0endif, powerNE0, yLT02, yLT02T, end].
Simple path 5 is cycle : [powerNE0, powerNE0T, powerNE0].

Finding Classified test cases...

Checking simple path coverage...

Building structural requirement coverage sets...

Path 1 is covered by test cases [sp01, sp02, sp04, sp05, sp07, sp08, sp10, sp11, sp13, sp14, st01]
Path 2 is covered by test cases []
Path 3 is covered by test cases []
Path 4 is covered by test cases [sp03, sp06, sp09, sp12, sp15]
Path 5 is covered by test cases [sp02, sp03, sp05, sp06, sp08, sp09, sp11, sp12, sp14, sp15, st01]

Press <Enter> to continue.

RTKBT> plan

Determining Structural Test Set and Regression Test Set...

The Structural Test Set is:
[sp03, sp02]

The hitting set algorithm has found the following test cases to be redundant:
[sp01, sp04, sp05, sp06, sp07, sp08, sp09, sp10, sp11, sp12, sp13, sp14, sp15, st01]

The Regression Test Set is:
[]

The graph cyclomatic complexity is 4.
All arcs have been covered but 1 more simple path(s) must be covered in order to cover a basis set.
Up to 1 of the following simple paths should be covered: [2, 3]

Press <Enter> to continue.
The program is modified to take advantage of the binary representation of ‘y’ in order to perform the computation in time proportional to log y [Leun89], instead of proportional to y, as shown in Figure 14.

Figure 14. Modified Power example.

After modification, the test case classification using the new static analysis data and the old dynamic analysis data shows all test cases as unclassified except for reusable test cases in which \( y = 0 \). Control flow analysis produces a partial hitting set consisting of test case ‘sp01’. The user determines that coverage is inadequate as there are several branches requiring coverage.

```
RTXBT> classify
Classifying test cases...
The test cases are [sp01, sp02, sp03, sp04, sp05, sp06, sp07, sp08, sp09, sp10, sp11, sp12, sp13, sp14, sp15, st01]
Test case sp01 is Reusable.
Test case sp02 is Unclassified.
Test case sp03 is Unclassified.
```
Chapter 6 A Regression Testing Model Using the RTKBT

Test case sp04 is Reusable.
Test case sp05 is Unclassified.
Test case sp06 is Unclassified.
Test case sp07 is Reusable.
Test case sp08 is Unclassified.
Test case sp09 is Unclassified.
Test case sp10 is Reusable.
Test case sp11 is Unclassified.
Test case sp12 is Unclassified.
Test case sp13 is Reusable.
Test case sp14 is Unclassified.
Test case sp15 is Unclassified.
Test case sp16 is Unclassified.

Press <Enter> to continue.

RTKB> analyze

Purging old facts...

Select the methodology to be used;
Enter "control_flow" or "data_flow" : control_flow

The classified test cases are: [sp01, sp04, sp07, sp10, sp13]

Identifying simple paths...

The simple paths are [1, 2, 3, 4, 5, 6]

Simple path 1 is path : [start, yLT0, yLT0F, yLT0endif, powerNE0, yLT02, end].
Simple path 2 is path : [start, yLT0, yLT0F, yLT0endif, powerNE0, yLT02, yLT02T, end].
Simple path 3 is path : [start, yLT0, yLT0T, yLT0endif, powerNE0, yLT02, end].
Simple path 4 is path : [start, yLT0, yLT0T, yLT0endif, powerNE0, yLT02, yLT02T, end].
Simple path 5 is cycle : [powerNE0, pEVEN, pEvenF, powerNE0].
Simple path 6 is cycle : [powerNE0, pEVEN, pEvenT, powerNE0].

Finding Classified test cases...

Checking simple path coverage...

Building structural requirement coverage sets...

Path 1 is covered by test cases [sp01, sp04, sp07, sp10, sp13]
Path 2 is covered by test cases []
Path 3 is covered by test cases []
Path 4 is covered by test cases []
Path 5 is covered by test cases []
Path 6 is covered by test cases []

Press <Enter> to continue.

RTKB> plan

Determining Structural Test Set
and Regression Test Set...

The Structural Test Set is:
[sp01]

The hitting set algorithm has found
the following test cases to be redundant:
[sp04, sp07, sp10, sp13]

The Regression Test Set is:
[]

The graph cyclomatic complexity is 5.
The following arcs have not been covered:
[a(yLT0, yLT0T), a(yLT0T, yLT0endif), a(powerNE0, pEVEN), a(pEven, pEvenT), a(pEven, pEvenF), a(pEvenT, powerNE0), a(pEvenF, powerNE0), a(yLT02, yLT02T), a(yLT02T, end)]

Press <Enter> to continue.
Chapter 6  
A Regression Testing Model Using the RTKB

After the unclassified test cases have been dynamically analyzed (done manually for this thesis), the control flow analysis determines that an updated structural test set need only include test cases ‘st01’ and ‘sp03’. The regression test set only requires test case ‘st01’ (all modified basis paths are traversed by the single structural test case).

```rtkbt
RTKBT> plan

Determining Structural Test Set and Regression Test Set...

The Structural Test Set is:
[st01, sp03]

The hitting set algorithm has found the following test cases to be redundant:
[sp01, sp02, sp04, sp05, sp06, sp07, sp08, sp09, sp10, sp11, sp12, sp13, sp14, sp15]

The Regression Test Set is:
[st01]

The graph cyclomatic complexity is 5. All arcs have been covered but 1 more simple path(s) must be covered in order to cover a basis set. Up to 1 of the following simple paths should be covered:
[2, 3]

Press <Enter> to continue.
```

The second structural test case added in [Leun89] (x = 5, y = -2) intended to traverse both branches of the added code was not required for SBP coverage (since test case ‘st01’ traverses both branches in the added code) but if provided, will be chosen over ‘sp03’ as it covers more basis paths (test case ‘st02’ will be required for data flow coverage, as shall be shown later). The regression test plan would still be the single test case ‘st01’.
6.2.2 Data Flow

The data flow graph of the modified program is shown in Figure 15. Data flow analysis will be started from the point in the regression test process (Figure 12) after unclassified testcases have been dynamically analyzed (from the control flow example) and the RTKBT is reapplied using the same test cases provided after the control flow analysis. Notice that du-paths 4, 14 and 28 are only covered by the structural test case 'st02' in which 'y' is even and less than zero. It is determined that several du-paths are not
covered. Most of these paths are infeasible, but some are not so, new test cases will have
to be developed.

RTKBT> analyze
Purging old facts...
Select the methodology to be used.
Enter "control_flow" or "data_flow" : data_flow
The classified test cases are: [sp01, sp02, sp03, sp04, sp05, sp06, sp07, sp08, sp09, sp10, sp11, sp12, sp13, sp14, sp15, st01, st02]
Identifying definition-use paths...
The definition-use paths are [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]

Def-use path 1 is a c-use path for variable power : \([yLT0F, yLT0endif, powerNE0, pEVEN, pEvenF]\).
Def-use path 2 is a c-use path for variable power : \([yLT0F, yLT0endif, powerNE0, pEVEN, pEvent]\).
Def-use path 3 is a c-use path for variable power : \([yLT0T, yLT0endif, powerNE0, pEVEN, pEvenF]\).
Def-use path 4 is a c-use path for variable power : \([yLT0T, yLT0endif, powerNE0, pEVEN, pEvent]\).
Def-use path 5 is a c-use path for variable power : \([pEvenF, powerNE0, pEVEN, pEvenF]\).
Def-use path 6 is a c-use path for variable power : \([pEvenF, powerNE0, pEVEN, pEvent]\).
Def-use path 7 is a c-use path for variable power : \([pEvenT, powerNE0, pEVEN, pEvenF]\).
Def-use path 8 is a c-use path for variable power : \([pEvenT, powerNE0, pEVEN, pEvent]\).
Def-use path 9 is a p-use path for variable power : \([yLT0T, yLT0endif, powerNE0, pEVEN, pEvenF]\).
Def-use path 10 is a p-use path for variable power : \([yLT0F, yLT0endif, powerNE0, pEVEN, pEvent]\).
Def-use path 11 is a p-use path for variable power : \([yLT0F, yLT0endif, powerNE0, pEVEN]\).
Def-use path 12 is a p-use path for variable power : \([yLT0F, yLT0endif, powerNE0, yLT02]\).
Def-use path 13 is a p-use path for variable power : \([yLT0T, yLT0endif, powerNE0, pEVEN]\).
Def-use path 14 is a p-use path for variable power : \([yLT0T, yLT0endif, powerNE0, pEVEN, pEvenF]\).
Def-use path 15 is a p-use path for variable power : \([yLT0T, yLT0endif, powerNE0, pEVEN]\).
Def-use path 16 is a p-use path for variable power : \([yLT0T, yLT0endif, powerNE0, yLT02]\).
Def-use path 17 is a p-use path for variable power : \([pEvenF, powerNE0, pEVEN, pEvenF]\).
Def-use path 18 is a p-use path for variable power : \([pEvenF, powerNE0, pEVEN, pEvent]\).
Def-use path 19 is a p-use path for variable power : \([pEvenF, powerNE0, pEvenF]\).
Def-use path 20 is a p-use path for variable power : \([pEvenF, powerNE0, yLT02]\).
Def-use path 21 is a p-use path for variable power : \([pEvenT, powerNE0, pEvenF, pEvenF]\).
Def-use path 22 is a p-use path for variable power : \([pEvenT, powerNE0, pEven, pEvenF]\).
Def-use path 23 is a p-use path for variable power : \([pEvenT, powerNE0, pEven]\).
Def-use path 24 is a p-use path for variable power : \([pEvenT, powerNE0, yLT02]\).
Def-use path 25 is a c-use path for variable power : \([yLT0F, yLT0endif, powerNE0, pEvenF]\).
Def-use path 26 is a c-use path for variable x : \([\text{start}, yLT0, yLT0T, yLT0endif, powerNE0, pEVEN, pEvenF]\).
Def-use path 27 is a c-use path for variable x : \([\text{start}, yLT0, yLT0T, yLT0endif, powerNE0, pEVEN, pEvent]\).
Def-use path 28 is a c-use path for variable x : \([\text{start}, yLT0, yLT0T, yLT0endif, powerNE0, pEVEN, pEvenF]\).
Def-use path 29 is a c-use path for variable x : \([pEvenT, powerNE0, pEVEN, pEvenF]\).
Def-use path 30 is a c-use path for variable x : \([pEvenT, powerNE0, pEVEN, pEvenF]\).
Def-use path 31 is a c-use path for variable y : \([\text{start}, yLT0, yLT0F]\).
Def-use path 32 is a c-use path for variable y : \([\text{start}, yLT0, yLT0F]\).
Def-use path 33 is a c-use path for variable y : \([\text{start}, yLT0, yLT0F]\).
Def-use path 34 is a c-use path for variable y : \([\text{start}, yLT0, yLT0F]\).
Def-use path 35 is a c-use path for variable y : \([\text{start}, yLT0, yLT0F, yLT0endif, powerNE0, yLT02, \text{end}]\).
Def-use path 36 is a c-use path for variable y : \([\text{start}, yLT0, yLT0T, yLT0endif, powerNE0, yLT02, \text{end}]\).
Def-use path 37 is a p-use path for variable y : \([\text{start}, yLT0, yLT0F, yLT0endif, powerNE0, yLT02, yLT02T}]\).
Chapter 6  A Regression Testing Model Using the RTKBT

Def-use path 38 is a p_use path for variable y : [start, yLT0, yLT0T, yLT0endif, powerNEO, yLT02, yLT02T].
Def-use path 39 is a c_use path for variable z : [yLT0endif, powerNEO, yLT02, end].
Def-use path 40 is a c_use path for variable z : [yLT0endif, powerNEO, pEvenF].
Def-use path 41 is a c_use path for variable z : [yLT0endif, powerNEO, yLT02, yLT02T].
Def-use path 42 is a c_use path for variable z : [pEvenF, powerNEO, yLT02, end].
Def-use path 43 is a c_use path for variable z : [pEvenF, powerNEO, pEvenF].
Def-use path 44 is a c_use path for variable z : [pEvenF, powerNEO, yLT02, yLT02T].
Def-use path 45 is a c_use path for variable z : [yLT02T, end].
Finding Classified test cases...
Checking def-use path coverage...

Building structural requirement coverage sets...

Path 1 is covered by test cases [sp02, sp05, sp08, sp11, sp14]
Path 2 is covered by test cases [st01]
Path 3 is covered by test cases [sp03, sp06, sp09, sp12, sp15]
Path 4 is covered by test cases [st02]
Path 5 is covered by test cases []
Path 6 is covered by test cases []
Path 7 is covered by test cases [st01, st02]
Path 8 is covered by test cases []
Path 9 is covered by test cases [sp02, sp05, sp08, sp11, sp14]
Path 10 is covered by test cases [st01]
Path 11 is covered by test cases [sp02, sp05, sp08, sp11, sp14, st01]
Path 12 is covered by test cases [sp01, sp04, sp07, sp10, sp13]
Path 13 is covered by test cases [sp03, sp06, sp09, sp12, sp15]
Path 14 is covered by test cases [st02]
Path 15 is covered by test cases [sp03, sp06, sp09, sp12, sp15, st02]
Path 16 is covered by test cases []
Path 17 is covered by test cases []
Path 18 is covered by test cases []
Path 19 is covered by test cases []
Path 20 is covered by test cases [sp02, sp03, sp05, sp08, sp09, sp11, sp12, sp14, sp15, st01, st02]
Path 21 is covered by test cases [st01, st02]
Path 22 is covered by test cases []
Path 23 is covered by test cases [st01, st02]
Path 24 is covered by test cases []
Path 25 is covered by test cases [sp02, sp05, sp08, sp11, sp14]
Path 26 is covered by test cases [sp03, sp06, sp09, sp12, sp15]
Path 27 is covered by test cases [st01]
Path 28 is covered by test cases [st02]
Path 29 is covered by test cases [st01, st02]
Path 30 is covered by test cases []
Path 31 is covered by test cases [sp01, sp02, sp04, sp05, sp07, sp08, sp10, sp11, sp13, sp14, st01]
Path 32 is covered by test cases [sp03, sp06, sp09, sp12, sp15, st02]
Path 33 is covered by test cases [st01, sp04, sp05, sp08, sp10, sp11, sp13, sp14, st01]
Path 34 is covered by test cases [sp03, sp06, sp09, sp12, sp15, st02]
Path 35 is covered by test cases [sp01, sp04, sp07, sp10, sp13]
Path 36 is covered by test cases []
Path 37 is covered by test cases []
Path 38 is covered by test cases []
Path 39 is covered by test cases [sp01, sp04, sp07, sp10, sp13]
Path 40 is covered by test cases [sp02, sp03, sp05, sp06, sp08, sp09, sp11, sp12, sp14, sp15]
Path 41 is covered by test cases []
Path 42 is covered by test cases [sp02, sp05, sp08, sp11, sp14, st01]
Path 43 is covered by test cases []
Path 44 is covered by test cases [sp03, sp06, sp09, sp12, sp15, st02]
Path 45 is covered by test cases [sp03, sp06, sp09, sp12, sp15, st02]

Press <Enter> to continue.
First the infeasible paths are explained. Du-paths 5, 17, and 43 are infeasible since the variable 'power' will never be odd for two successive iterations of the while loop. Du-paths 16, 36, 38 and 41 are infeasible since 'power' cannot be equal to 0 on the initial traversal when 'y' is less than zero. Du-path 24 is infeasible since 'power' cannot be equal to zero immediately after an iteration of the while loop in which 'power' is even. Du-path 37 is infeasible since 'y' cannot be greater than or equal zero and simultaneously be less than zero.

The feasible du-paths for which new test cases will have to be provided are now explained. Du-paths 6 and 18 require a test case in which the while loop is executed for an iteration with an odd value of 'power' followed by an iteration with an even value of 'power'. Du-path 19 requires a test case in which the while loop is executed once for an odd value of 'power' followed an iteration in which 'power' is greater than zero. Du-paths 8, 22 and 30 require two successive iterations of the while loop in which the value of 'power' is even. All these du-paths can be satisfied by a test case in which y = 5, so a new structural test case 'st03' is added with input values x = -2 and y = 5.

Following the regression test process (Figure 12), the new test case would now be executed with a dynamic analysis tool to determine its trajectory (this is done manually for this thesis). The RTKBT is re-applied using the new test case data indicating that du-paths 5, 16, 17, 24, 36, 37, 38, 41 and 43 have not been covered. Since these paths have all been determined to be infeasible (as explained previously), coverage is considered
adequate and the regression testing process continues with the execution of the regression test plan consisting of test cases ‘st01’, ‘st02’, ‘st03’, and ‘sp03’.

```
RTKBT> plan
Determining Structural Test Set and Regression Test Set...

The Structural Test Set is:
[st01, st02, st03, sp03, sp01]

The hitting set algorithm has found the following test cases to be redundant:
[sp02, sp04, sp05, sp06, sp07, sp08, sp09, sp10, sp11, sp12, sp13, sp14, sp15]

The Regression Test Set is:
[st01, st02, st03, sp03]

All arcs have been covered.
The following def-use paths have not been covered:
[5, 16, 17, 24, 36, 37, 38, 41, 43]

Press <Enter> to continue.
```

After executing the regression test with a dynamic analyzer, the RTKBT is again re-applied to determine the coverage provided with the inclusion of new test cases. Since no program changes had been made after the previous dynamic analysis, the same result is obtained indicating that du-paths 5, 16, 17, 24, 36, 37, 38, 41 and 43 have not been covered. Again, since these paths have all been determined to be infeasible, coverage is considered adequate so, barring any errors in the execution of test cases in the regression test set, the regression testing process terminates. The results of the automated execution of test scripts (developed manually) for the regression test sets and the updated structural test sets showing statement coverage are given in Appendix 2 for comparison.
Chapter 7

7. A Potential Application of the RTKBT to OO Software Testing

7.1 State-Based Paradigm for Object-Oriented Software Testing

The control flow and data flow test coverage methods that have been developed for use with procedural programming languages are generally not adequate for object-oriented programming languages. [Bind95] explains that while the above method may be adapted for member function testing, they are not sufficient for object class testing since they are not consistent with the object-oriented paradigm. The object-oriented concepts of encapsulation, inheritance and polymorphism which promote modular and robust programming can actually be a hindrance to testing. The encapsulation of data members within classes or ‘data-hiding’ can make it hard to provide reporting on the internal state of an object. Inherited methods require re-testing due to the new context of usage within the derived class. Polymorphism will require that all possible bindings of polymorphic functions will have to be found (which may be difficult in the case of late binding where the class type is not known until runtime) and a separate test developed for each binding. These object oriented features present new challenges for the analysis of test coverage.

A state model can provide the conceptual basis for object-oriented testing as path models have provided a conceptual basis for procedural program testing [Bind95]. Flow graph techniques could be used instead to analyze a state diagram that represents the
intended behavior of an object. A potential application of the flow graph analysis functions implemented in the RTKBT to assess test coverage of an object using its state model is presented.

7.2 FSM Modeling of Dynamic Behavior

The dynamic behavior of objects can be represented through state-based modeling in which object classes (the generally accepted unit of test for object oriented software [Bind95]) are modeled as finite state machines (FSMs). The FSM relates events and states, where a state is an abstraction of the attribute values of an object and events are stimuli between objects (messages) occurring at points in time [Rumb91]. When an event is received, the next state depends on the current state as well as the event. A change of state caused by an event is called a transition. A state diagram can be represented by a digraph whose nodes are states and whose directed edges are transitions labeled by event names [Rumb91].

FSM diagrams can be used to represent ‘one-shot’ life cycles or continuous loops [Rumb91]. A one-shot diagram has initial and final state nodes and is used to represent objects with finite lives. A one-shot state diagram has a single entrance node corresponding to the construction of an object and a single exit node corresponding to the destruction of the object such that all state nodes are reachable from the entrance node and the exit node is reachable from all state nodes. The flow graph techniques
implemented in the RTKBT may be used to analyze FSM diagrams to support object oriented testing for objects whose behavior are modeled by one-shot diagrams.

If an object's FSM diagram is of the continuous loop type, the FSM digraph will have to be converted to a strongly connected digraph having a single entrance node and a single exit node to facilitate analysis using the flow graph techniques implemented in the RTKBT. Often a continuous loop state diagram contains an initial state node that is entered by default when the object is created and is returned to after the completion of activity sequences. Such a continuous loop stage diagram may be converted to a one-shot diagram by adding constructor and destructor nodes and a transition from the initial state node to the new destructor node such that the single exit may be reached from all nodes. Another method of converting the FSM graph to a one-shot diagram is to add an arc from every node (except the single entrance node) to the destructor node, representing the idea that it is possible for an object to be destroyed regardless of its state. This is not practical, as it would require many test cases simply to test the destruction of the object in each of its possible states. The addition of a single arc from the initial node to the destructor node is preferred.

7.3 FSM Diagram Test Coverage

If an object's behavior is represented by a graph having a single entrance node and a single exit node, the feasible paths through the FSM diagram from object creation (the single entrance node) through to object destruction (the single exit node) represent the
possible behavior patterns of the object. Test coverage adequacy may then be equated to the traversal of a particular set of paths in the state graph as determined by some coverage criterion. It must be determined whether or not a set of test cases associated with the object will cause the execution of sufficient number and types of paths of the FSM graph according to the test coverage adequacy criterion chosen.

There is a considerable body of literature regarding protocol conformance testing that deals with FSM test coverage. A hierarchy of some coverage adequacy criteria described in [Chow78] and [Mill91] that are appropriate for the testing of systems that can be modeled by a FSM are as follows (however the criteria described in d and e are often impractical due the existence of infeasible paths):

a. branch cover - For a FSM model, a branch cover is a set of test sequences in which every branch (transition) is traversed.

b. boundary-interior cover - Boundary test paths refer to test paths that enter a loop without causing it to be iterated. An interior test path or sequence causes a loop to be entered and iterated at least once. A boundary-interior cover consists of a set of test sequences in which at least one path from every class is covered.

c. switch cover - A more stringent test coverage is called a switch cover. A switch is a branch to branch pair. A switch cover is a set of test sequences in which every branch to branch pair is traversed.

d. n-switch set cover - An n-switch is defined as a sequence of consecutive branches of length n + 1 (e.g. a branch is a 0-switch and a branch to branch pair
is a 1-switch). For a branch ‘b’, an n-switch set is the set of all n-switches having b as its prefix. An n-switch set cover is a set of test sequences in which all n-switch sets are covered.

e. full cover - A full cover is an n - 1 switch set cover where n is the number of states in the FSM.

The first step in the FSM analysis involves extracting the graph information from the OO design tool for use in the RTKBT. To demonstrate, an example taken from the ObjecTime tutorial [OTTu95] is used. An object class (called an ‘actor’ in ObjecTime) called ‘device’ represents a component of a simple communications system. The FSM describing its intended behavior as specified using the ObjecTime graphic behavior editor has been extracted and is shown below.

Figure 16. Behavior model for the “device” actor class
This information can also be output in text format. The behavior portion of the device actor class cut from a text output file of the Real-time Object Oriented Model (ROOM) chart (FSM diagram) for the tutorial example is shown in Appendix 4. This behavior specification is read from a text file into the Prolog interpreter by a clause called getsentence and temporarily stored in a variable as a sentence in the form of a set of words (see Appendix 3).

A parser consisting of a clause for each production of the ObjecTime grammar for the behavior specification (specified in [OTTG95]) generates a parse tree from the behavior specification sentence. The grammar clauses exclude low level information such as the C++ source code from the parse tree to reduce the complexity of this prototype tool. The parse trees for comments are not required for FSM analysis but are included to make it easier to read the parse tree (and were used to aid the debugging of the parser). The parse tree generated by the behavior_spec clause is a set of smaller parse tree facts nested within the behavior specification parse tree fact (see Appendix 3).

The portion of the FSM graph information required for analysis is extracted using a technique similar to that used to create the parse tree and assembled in the form of a digraph fact:

```
digraph(['INITIALPOINT', 'Idle', 'WaitFirstAck', 'WaitSecondAck'], [a('INITIALPOINT', 'Idle'), a('Idle', 'WaitFirstAck'), a('WaitFirstAck', 'WaitSecondAck'), a('WaitSecondAck', 'Idle'), a('WaitFirstAck', 'WaitFirstAck'), a('WaitSecondAck', 'WaitSecondAck')])
```
Chapter 7  
A Potential Application of the RTKBT to OO Software Testing

The flow graph analysis performed by the RTKBT requires that the graph have a single entrance node and a single exit node. Since the ObjecTime behavior FSMs are continuous loop state diagrams that do not include an explicit destructor node, this node is added to the object’s FSM graph. A node called the INITIALPOINT in ObjecTime ROOM, which is the single entrance node of the FSM, is connected to the initial state of the graph by an existing initial transition arc. An additional arc is added to represent the transition from the initial state node to the destructor node. The digraph fact extracted from the behavior specification parse tree is modified by a clause called addexit to produce the following modified digraph:

digraph(['INITIALPOINT', 'Idle', 'WaitFirstAck', 'WaitSecondAck', 'Destructor'], [a('INITIALPOINT', 'Idle'), a('Idle', 'WaitFirstAck'), a('WaitFirstAck', 'WaitSecondAck'), a('WaitSecondAck', 'Idle'), a('WaitFirstAck', 'WaitFirstAck'), a('WaitSecondAck', 'WaitSecondAck'), a('Idle', 'Destructor')])

The above digraph fact is then converted to a cdigraph fact by adding change information (all change values are equal to zero for this example) and saved in a file in the domain directory ready to be loaded into the RTKBT:

cdigraph([  
'INITIALPOINT'/0,  
'Idle'/0,  
'WaitFirstAck'/0,  
'WaitSecondAck'/0,  
'Destructor'/0]  
,  
[a('INITIALPOINT', 'Idle')/0,  
a('Idle', 'WaitFirstAck')/0,  
a('WaitFirstAck', 'WaitSecondAck')/0,  
a('WaitSecondAck', 'Idle')/0,  
a('WaitFirstAck', 'WaitFirstAck')/0,  
a('WaitSecondAck', 'WaitSecondAck')/0,  
a('Idle', 'Destructor')/0]  
).
A top level goal called extract_FSM is implemented as a clause with the above steps as sub-clauses. It can convert a behavior specification file ‘behavior.txt’ into a Prolog data file ‘ot_model.pl’.

```prolog
extract_FSM :-
    see('/regtest/objtime/behavior.txt'),
    getsentence(S),
    seen,
    see(user),!,
    behavior_spec(DG,PT,S,[],!),
    addexit(DG,MDG),
    output_graph(MDG).
```

This completes the process that extracts a portion of the graph information representing the behavior of an object. The output is of the same format as that used for control flow analysis can be handled in the same way by the RTKBT.

### 7.4 Message Sequence Chart Analysis

Part of the requirements specification of an object oriented system is the expected behavior of the various aspects of the system, especially that of the interactions between the object classes of which it is composed. A portion of the interaction requirements may be captured in the form of Message Sequence Charts (MSCs) representing the exchange of messages between objects for scenarios describing the desired action sequences. In order to demonstrate that an object class implementation meets that part of its specified requirements, an object instantiation must be executed in an environment that will produce the message sequences specified in the associated MSCs. Ideally, objects should be tested
using some test mechanism in which a number of MSCs are treated as test cases belonging
to the test suite for that object.

A message sequence chart used to test the example device object is shown on the
following page. The tutorial example includes a TestSystem object which contains two
instantiations of the device object with appropriate interface connections. The MSC
shows a path traversed through the FSMs in terms of state transitions in response to
received messages, some of which are explicit (i.e. transition to the WaitSecondAck state
when an Ack message is received) and some of which are implicit (i.e. the self loop
transition back to the WaitFirstAck state when an Info message is received).

This information may also be output in text format, allowing it to be parsed and
modified as required (i.e. to add constructor and destructor nodes to the ends of the
trajectory if necessary) in a similar manner as was done for the behavior specification.
This has been done manually for this example to produce the following test case fact:

testcase('RTS_Trace', ['INITIALPOINT', 'Idle', 'WaitFirstAck', 'WaitFirstAck',
'WaitSecondAck', 'WaitSecondAck', 'Idle', 'Destructor'])
The test case information that could be obtained by parsing the portion of the linear form of the MSC related to the object under test consists of the MSC name (used as the test case identifier) and the path traversed through the FSM graph. It is of the same format as that generated by dynamic analysis of test cases for software written in procedural languages and is treated in the same way by the RTKBT. Now it must be
determined if the set of existing MCS test cases adequately test the object’s behavior represented by the FSM graph.

7.5 Determining Test Coverage Adequacy

Using the combined information from the behavioral specification and MSCs that have been converted into Prolog facts, the RTKBT is used to determine test coverage adequacy using the SBP method. A proper analysis would require the implementation of a FSM coverage adequacy criterion as mentioned in section 7.3. To demonstrate that such an analysis is feasible, the SBP criterion is used. This analysis shows that the single RTS_Trace test case is adequate to test the device actor in accordance with the SBP criterion.

RTKBT> plan

Purging old facts...

Select the methodology to be used;
Enter "control_flow" or "data_flow" : control_flow

Classifying test cases...
The test cases are [RTS_Trace]

Test case RTS_Trace is Reusable.

Identifying simple paths...
The simple paths are [1, 2, 3, 4]

Simple path 1 is path: [INITIALPOINT, Idle, Destructor].
Simple path 2 is loop: [WaitFirstAck, WaitFirstAck].
Simple path 3 is loop: [WaitSecondAck, WaitSecondAck].
Simple path 4 is cycle: [Idle, WaitFirstAck, WaitSecondAck, Idle].
Finding Classified test cases...

Checking simple path coverage...

Building structural requirement coverage sets...

Path 1 is covered by test cases [RTS_Trace]
Path 2 is covered by test cases [RTS_Trace]
Path 3 is covered by test cases [RTS_Trace]
Path 4 is covered by test cases [RTS_Trace]

Determining Structural Test Set and Regression Test Set...

The Structural Test Set is:
[RTS_Trace]

The hitting set algorithm has found the following test cases to be redundant:
[]

The Regression Test Set is:
[]

The graph cyclomatic complexity is 4. All arcs have been covered and simple path coverage is equal to the complexity number.

Press <Enter> to continue.

7.6 Test Automation

The latest release of the ObjecTime ROOM tool incorporates a feature which enables the automatic generation of test drivers from the MSC specifications [Seli96]. The feature is capable of creating test driver object classes, which when instantiated and executed will produce the message passing event sequences specified by the MSCs. The feature also provides a mechanism to automatically execute the generated test drivers. In order to take full advantage of such an automated testing mechanism, a method of executing a set of test drivers identified in the set of MSC identifiers produced by the RTKBT would have to be incorporated. Regression testing would depend on the implementation of a mechanism to transfer information regarding modifications available
Chapter 7  A Potential Application of the RTKB to OO Software Testing

from the ObjecTime difference tools into the linear text output in a form that could be parsed to identify changes to individual states and transitions.

7.7 Summary

A potential application of the RTKB to object oriented software testing has been demonstrated. The dynamic behavior of an object is modeled as a FSM diagram using an object oriented modeling tool called ObjecTime. A parser capable of recognizing the ObjecTime modeling grammar is used to extract the FSM graph information required for analysis by the RTKB. The feasibility of analyzing the FSM graph to determine coverage requirements has been demonstrated using the SBP coverage criterion. A practical application of this tool would require the implementation of proper FSM coverage adequacy criteria. The paths traversed through the FSM graph can be extracted from Message Sequence Charts that represent object oriented test cases. The RTKB can analyze the coverage of paths in the FSM graph by MSC test cases to determine test coverage adequacy.
Chapter 8

8. Conclusions and Future Work

8.1 Conclusions

The Regression Test Knowledge Based Tool depends upon flow graph and test case trajectory input data available from static and dynamic analysis tools. From this input data, the RTKBT is capable of classifying test cases, analyzing flow graph information to determine structural test coverage requirements, selecting test cases to produce an updated structural test set and a regression test set and identifying the requirement for additional test cases.

The RTKBT is capable of analyzing a directed graph that represents the control flow of a unit of software to identify simple paths as structural test coverage requirements. The RTKBT can then analyze the trajectories of test cases to determine which simple paths each covers. The SBP structural test coverage criterion requires that a basis set (of size equal to the cyclomatic complexity number) of simple paths must be covered. A set of test cases that will satisfy the SBP criterion, will provide all-branch coverage in the worst case (in which all splitting nodes are associated with loops or cycles) and will provide coverage equal to that of McCabe’s cyclomatic adequacy criterion if there are no loops or cycles in the program’s control flow.
The RTKBT is capable of analyzing a directed graph representing the flow of data within a unit of software to identify all definition-use paths as structural test coverage requirements. The RTKBT can then analyze test case trajectories to determine which du-paths each test case covers. Since not all du-paths may be feasible, and a set of test cases that will satisfy all feasible du-paths may not provide coverage of all-branches, the RTKBT also provides the tester with information regarding any branches that have not been covered by the set of test cases selected.

The RTKBT is particularly useful for assisting in the development of new test cases as it can identify the specific structural elements of a program that have not been covered by any test case. This information can be used to direct the tester to the point in the program where the determination of input values for new structural test cases should begin.

The RTKBT infers the associations between test cases and the structural test coverage requirements they satisfy, and uses the inferences to produce an updated structural test set and regression test set of reduced size in comparison with the total sets of reusable and re-testable test cases.

It has also been demonstrated that the RTKBT could be adapted to assist with testing of object oriented software that has been developed using a modeling tool capable of specifying the intended behavior of an object class in a FSM diagram. The directed graph extracted from the behavior specification of an object and the test case trajectory
extracted from a MSC specification have been used as input to the analysis of test
coverage adequacy demonstrated using the SBP coverage criterion.

8.2 Recommendations for Future Research

8.2.1 Integration with Automated Analysis and Test Tools

The use of the RTBKT in an actual regression testing scenario will not be practical
unless it is integrated with the tools necessary to provide the proper input data and utilize
the updated test suite and regression test plan produced as output.

A source code static analyzer capable of generating control flow and data flow
graph data in the syntax used by the RTKBT is required. The static analyzer must also be
capable of determining which blocks and branches in the flow graph are associated with
new or modified source code.

The RTKBT would also be more useful if it were integrated with an automated
testing tool having an automatic test script generation functionality. The testing tool
could use the sets of test cases provided by the RTKBT as input and automatically execute
test scripts representing the updated test suite or regression test set as needed. A dynamic
analysis tool would also be required to produce the test case trajectories in the syntax
required by the RTKBT and using the same node naming convention as the static analyzer.
8.2.2 Analysis for Different Test Coverage Adequacy Criteria

The selection of a test coverage adequacy criterion depends on the degree of risk associated with the failure of the software and the amount of time and money allotted for testing. It would be useful to provide alternatives with regard to how rigorously the software is to be tested depending on the circumstances at the time of testing.

The SBP control flow criterion can be relatively weak in comparison with McCabe cyclomatic adequacy as it does not require loops or cycles to be tested independently. The SBP criterion could be strengthened by providing an automated means of determining possible combinations (chains) of the simple cycles or self loops and their connecting simple complete paths and asserting that these chains be tested independently of other simple cycles or self loops dependent on the same simple complete path as isolation requirements. Alternatively, the McCabe cyclomatic adequacy criterion could be implemented by allowing the user to first identify a "baseline" function after which an implementation of the "flip each decision along the baseline" heuristic may be used to identify the other basis paths.

The all du-paths data flow coverage criterion is the strongest (second only to the impractical all paths criterion) of a spectrum of data flow coverage criteria, all of which could be implemented using the same data flow graph data. I have introduced a criterion stronger than that of all du-paths that could be implemented by providing an automated means of determining chains of related du-paths (the c-use of one du-path is used in the
def of the next du-path) and asserting that these du-path chains must be tested together by a single test case.

8.2.3 Automated Identification of Infeasible Paths

The achievement of an adequate test in accordance with a given test coverage criterion is often hindered by the existence of infeasible paths, as has been demonstrated several times in this thesis. As well, the identification of infeasible paths generally is not trivial. Further research is required to develop and implement automated means of determining the feasibility of paths so that the feasible paths or sub-paths associated with test coverage criteria can be determined more efficiently.

8.2.4 Inter-Module Analysis

The RTKBT only permits the analysis of a single digraph. The RTKBT would be more practical if it was capable of analyzing several graphs at once to allow for integration and system testing. This capability could facilitate more efficient control flow testing of modules containing calling and called functions and inter-module data flow analysis of variables passed as parameters and global variables. Issues of scale and the presentation of information in the user interface would have to be addressed however.

8.2.5 Combination with Functional Testing

Test cases can be considered to have a functional and a structural component in that a test case causes features specified in the structural component to be executed under
the conditions specified in the functional component [Ntaf84]. The RTKBT could be enhanced to differentiate between specification test cases and structural test cases so that the reduction of the test suite does not eliminate any of the required functional tests [Harr90]. Assuming that the set of required functional test cases is a subset of the required structural test cases and the associations between test cases and the functional requirements are known, a hitting set for the functional requirements could be determined. This hitting set would then be used as an initial hitting set for the structural test suite and the remainder of that test suite could be determined for the structural requirements by selecting test cases from the unmarked test sets.

8.2.6 Implementation of OO Test Coverage Analysis

The use of the RTKBT in support of object oriented testing would require the implementation of FSM specific coverage adequacy criteria mentioned in chapter 7. Also, the analysis of OO FSMs presented does not provide for the analysis of hierarchical FSMs (sub-FSMs contained within state nodes of a higher level FSM). A thorough test coverage analysis including hierarchical FSMs would also require the multiple graph analysis capability, in which sub-FSMs could modeled as separate graphs.
References


Criteria’, IEEE Transactions on Software Engineering, vol. SE-14 No. 10, pp. 1483-1498, 

Techniques in Software Maintenance Tools’, Conference on Software Maintenance - 


[Hoff95] D. M. Hoffman and P. A. Strooper, Software Design, Automated Testing and 
Massachusetts, 1995.


[Hump95] Watts S. Humphrey, A Discipline for Software Engineering, Addison-Wesley 

References


References


Appendix 1 FORTRAN Source Code for Blackjack [McCa83]

88. 0 C**
89. 0 C**
90. 0 C**
91. 0 SUBROUTINE HAND(WIN)
92. 0 C**
93. 0 INTEGER P,D,PACE,DACE,I
94. 0 CARDS(52),DEBUG,COUNT,WIN
95. 0 COMMON /DECK/CARDS,I,DEBUG
96. 0 P = 0
97. 3 D = 0
98. 6 PACE = 0
99. 9 DACE = 0
100. 12 WIN = 0
101. 15 C** WIN WILL BE 0 IF DEALER WINS, 1 IF PLAYER WINS, 2 IF A PUSH
102. 15 CALL HIT(P,PACE)
103. 21 CALL HIT(D,DACE)
104. 27 CALL HIT(P,PACE)
105. 33 CALL HIT(D,DACE)
106. 39 COUNT = 0
107. 42 C**
108. 42 WRITE(*,'(A,I2)')'PLAYER HAS BLACKJACK'
109. 108 960 FORMAT ('PLAYER = ',I2,' NO OF Aces = ',I1)
110. 108 WRITE(*,960)P,PACE
111. 137 IF(P .EQ. 21) THEN
112. 142 WRITE(*,'(A)')'PLAYER BUSTS-DEALER WINS'
113. 188 WIN = 1
114. 191 ELSE
115. 193 COUNT = 2
116. 196 11 WRITE(*,'(A)')'HIT?'
117. 226 READ(*,'(I1)') K
118. 258 IF( K.EQ. 1) THEN
119. 263 CALL HIT(P,PACE)
120. 269 COUNT = COUNT + 1
121. 274 WRITE(*,960),P,PACE
122. 303 IF(P .GT. 21) THEN
123. 308 WRITE(*,'(A)')'PLAYER BUSTS-DEALER WINS'
124. 363 GO TO 13
125. 365 ENDIF
126. 365 GO TO 11
127. 367 ENDIF
128. 367 ENDIF
129. 367 C** HANDLE THE BLACKJACK SITUATIONS, CASE WHEN DEALER HAS BLACKJACK:
Appendix 1

FORTRAN Source Code for Blackjack

130. 367  IF(D .EQ. 21) THEN
131. 372     WRITE(*, '(A)') 'DEALER HAS BJ'
132. 411  IF(WIN .EQ. 1) THEN
133. 417     WRITE(*, '(A)') 'PUSH'
134. 453     WIN = 2
135. 456    GO TO 13
136. 458  ELSE
137. 460     WRITE(*, '(A)') 'DEALER AUTOMATICALLY WINS'
138. 511    GOTO 13
139. 513  ENDIF
140. 513  ELSE
141. 515 CASE WHERE DEALER DOESN'T HAVE BLACKJACK:
142. 515 CHECK FOR PLAYER BLACKJACK OR FIVE CARD HAND:
143. 515     IF((P .EQ. 21) .OR. (COUNT .GE. 5)) THEN
144. 524     WRITE(*, '(A)') 'PLAYER AUTOMATICALLY WINS'
145. 576     WIN = 1
146. 579    GOTO 13
147. 581  ENDIF
148. 581  ENDIF
149. 581 CASE WHERE PLAYER HAS BLACKJACK:
150. 581 WRITE(*,970)D
151. 605 FORMAT('DEALER HAS ',I2)
152. 605 IF(D .LE. 16) THEN
153. 610     CALL HIT(D,DACE)
154. 616     WRITE(*,970)D
155. 641     IF(D .GT. 21) THEN
156. 646     WRITE(*, '(A)') 'DEALER BUSTS-PLAYER WINS'
157. 698     WIN = 1
158. 701    GO TO 13
159. 703  ENDIF
160. 703 GOTO 12
161. 705  ENDIF
162. 705 CASE WHERE DEALER HAS BLACKJACK:
163. 705 FORMAT(' PLAYER = ',I2, ' DEALER = ',I2)
164. 705 WRITE(*,980) P,D
165. 733     IF(P .GT. D) THEN
166. 738     WRITE(*, '(A)') 'PLAYER WINS'
167. 775     WIN = 1
168. 778  ELSE
169. 780     WRITE(*, '(A)') 'DEALER WINS'
170. 817  ENDIF
171. 817
172. 817 END
173. 817
174. 817
SUBROUTINE HIT(TOTAL, ACES)
INTEGER TOTAL, ACES
INTEGER I, CARDS(52), DEBUG
COMMON /DECK/CARDS, I, DEBUG

TOTAL = TOTAL + CARDS(I)
IF (CARDS(I) .EQ. 11) THEN
    ACES = ACES + 1
END IF
I = I + 1
IF ((TOTAL .GT. 21) .AND. (ACES .GT. 1)) THEN
    TOTAL = TOTAL - 10
    ACES = ACES - 1
ENDIF
Appendix 2 Power Example Source Code and Test Scripts

The power program used as an example in chapter 6 was tested using an automated program tester called PGMGEN as described in [Hoff95]. PGMGEN requires that the module under test have at least one exception handling routine, so a simple divide by zero exception routine was added and a test case to exercise the routine was included in each test script to ensure that coverage was provided for its structure. The test coverage criterion used with PGMGEN is statement coverage provided by a Unix utility called tcov. Since statement coverage is weaker than the criteria used with the RTKBT, it is expected and demonstrated that 100% statement coverage will be provided by the updated structural test sets generated by the RTKBT. The regression test sets however, may not provide 100% statement coverage since they are only intended to exercise the modified portion of the code.

Power Program

Source Code

File 1 : system.h

```c
#include <stdio.h>
#include <string.h>

#define SY_EXCFIL "POWER.excfil"

/* constants */

extern FILE *sy_excfilp; /* file for exception messages */
```
File 2 : power.h

/*
 * power.h
 */

#include "system.h"
#include "power.h"
#include <stdio.h>
#include <stdlib.h>

float pwr_g_power(int x, int y); 

/*
 * power.c
 */

File 3 : power.c

/*
 * power.c
 */

#include "system.h"
#include "power.h"
#include <stdio.h>
#include <stdlib.h>

float pwr_g_power(int x, int y) {
    int power;
    float z, answer;

    /* divide by zero exception */
    if (x == 0 && y < 0) {
        pwr_divzero_exc();
        return 0;
    }

    if (y < 0) {
        power = -y;
    }
    else {
        power = y;
    }
    z = 1.0;
    while (power != 0) {
        z = z * x;
        power = power - 1;
    }
    if (y < 0) {
        z = 1/z;
    }
    answer = z;
    return answer;
Exception Handling Routine

File 4: power_e.c

```c
/*
 * power_e.c
 */

#include "power.h"
#include <stdio.h>

FILE *sy_excfilp = stderr;

void pwr_divzero_exc()
{
    fprintf(sy_excfilp, "Exception pwr_divzero_exc occurred \n");
}
```
Test Script

module
    pwr_
accprogs
        <g_power>
extceptions
        <divzero_exc>
globcod
{%
#include "system.h"
#include "power.h"

int cmd;
%
}
cases

/***** exceptions *****/
<g_power(0,-1), divzero_exc, dc, dc, dc>

/***** specification test cases *****/
/* note: no initial trace required in first field */

/* negative base */
<, noexc, g_power(-2.0), 1.0, float> /* sp01 */
<, noexc, g_power(-2.1), -2.0, float> /* sp02 */
<, noexc, g_power(-2,-1), -0.5, float> /* sp03 */
<, noexc, g_power(-1.0), 1.0, float> /* sp04 */
<, noexc, g_power(-1.1), -1.0, float> /* sp05 */
<, noexc, g_power(-1,-1), -1.0, float> /* sp06 */

/* zero base */
<, noexc, g_power(0,0), 1.0, float> /* sp07 */
<, noexc, g_power(0,1), 0.0, float> /* sp08 */
<, noexc, g_power(0,-1), 0.0, float> /* sp09 */

/* positive base */
<, noexc, g_power(1,0), 1.0, float> /* sp10 */
<, noexc, g_power(1,1), 1.0, float> /* sp11 */
<, noexc, g_power(1,-1), 1.0, float> /* sp12 */
<, noexc, g_power(2,0), 1.0, float> /* sp13 */
<, noexc, g_power(2,1), 2.0, float> /* sp14 */
<, noexc, g_power(2,-1), 0.5, float> /* sp15 */

/***** structural test cases *****/
<, noexc, g_power(4,2), 16.0, float> /* st01 */
Updated Test Script for Control Flow

```c
module
   pwr_
acccprogs
   <g_power>
exceptions
   <divzero_exc>
globcod
{%
#include "system.h"
#include "power.h"
int cmd;
%
} cases
/***** exceptions *****/
<g_power(0,-1), divzero_exc, dc, dc, dc>
/***** updated test plan *****/
<, noexc, g_power(-2,1), -2.0, float> /* sp02 */
<, noexc, g_power(-2,-1), -0.5, float> /* sp03 */
```
Test Coverage by Updated Test Suite

```c
float pwr_g_power(int x, int y) {
    int power;
    float z, answer;

    /* divide by zero exception */
    if (x == 0 && y < 0) {
        powr_divzero_exc;
        return 0;
    }
    if (y < 0) {
        power = -y;
    } else {
        power = y;
    }
    z = 1.0;
    while (power != 0) {
        z = z * x;
        power = power - 1;
    }
    if (y < 0) {
        z = 1/z;
    }
    answer = z;
    return answer;
}
```

Top 10 Blocks

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>34</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
</tr>
</tbody>
</table>

10 Basic blocks in this file
10 Basic blocks executed
100.00 Percent of the file executed

17 Total basic block executions
1.70 Average executions per basic block
Modified Power Program

Modified Source Code

```c
/*
 * power.c
 */

#include "system.h"
#include "power.h"
#include <stdio.h>
#include <stdlib.h>

double pwr_g_power(int x, int y) {
    int power;
    float z, answer;

    /* divide by zero exception */
    if (x == 0 && y < 0) {
        pwr_divzero_exc();
        return 0;
    }

    if (y < 0) {
        power = -y;
    } else {
        power = y;
    }

    z = 1.0;
    while (power != 0) {
        if (power % 2 == 0) { /* even power */
            x = x * x;
            power = power / 2;
        } else {
            z = z * x;
            power = power - 1;
        }
    }

    if (y < 0) {
        z = 1/z;
    }

    answer = z;
    return answer;
}
```
Regression Test Set Script for Control Flow

```plaintext
module
  pwr_
accprogs
  <g_power>
exceptions
  <divzero_exc>
globcod
{%
#include "system.h"
#include "power.h"
%
int cmd;
%
}
cases

/*****exceptions *****/
<g_power(0,-1), divzero_exc, dc, dc, dc>

/***** regression test plan *****/
<, noexc, g_power(4,2), 16.0, float> /* st01 */
```
Test Coverage by Control Flow Regression Test Set

```c
/*
 * power.c
 */

#include "system.h"
#include "power.h"
#include <stdio.h>
#include <stdlib.h>

float pwr_g_power(int x, int y) {
    int power;
    float z, answer;
    /* divide by zero exception */
    2 -> if (x == 0 && y < 0) {
        1 -> pwr_divzero_exc();
        return 0;
    };
    1 -> if (y < 0) {
        1 -> power = -y;
        } else {
        1 -> power = y;
    };
    1 -> z = 1.0;
    2 -> while (power != 0) { /* even power */
        1 -> x = x * x;
        1 -> power = power / 2;
    } else {
        1 -> z = z * x;
        1 -> power = power - 1;
    };
    2 -> if (y < 0) {
        1 -> answer = z;
        return answer;
    }
}

Top 10 Blocks

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>36</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
</tr>
</tbody>
</table>

13 Basic blocks in this file
11 Basic blocks executed
84.62 Percent of the file executed

14 Total basic block executions
1.08 Average executions per basic block
```
Updated Test Suite Script for Control Flow

module
    pwr_
accprogs
    <g_power>
exceptions
    <divzero_exc>
globcod
{%
#include "system.h"
#include "power.h"

int cmd:
%
}
cases

******exceptions *****
<g_power(0,-1), divzero_exc, dc, dc, dc>

****** updated test plan *****
<, noexc, g_power(4,2), 16.0, float> /* st01 */
<, noexc, g_power(5,-2), 0.04, float> /* st02 */
Test Coverage by Control Flow Updated Test Suite

```c
/*
 * power.c
 */

#include "system.h"
#include "power.h"
#include <stdlib.h>
#include <stdio.h>

float pwr_g_power(int x, int y) {
    int power,
    float z,answer;
    /* divide by zero exception */
    if(x==0 && y<0) {
        pwr_divzero_exc();
        return 0;
    }
    if(y<0){
        power = -y;
    } else {
        power = y;
    }
    z = 1.0;
    while (power != 0) {
        if (power%2 == 0) { /* even power */
            x = x * x;
            power = power / 2;
        } else {
            z = z * x;
            power = power - 1;
        }
    }
    if(y<0){
        z = 1/z;
    }
    answer = z;
    return answer;
}
```

Top 10 Blocks

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

13 Basic blocks in this file
13 Basic blocks executed
100.00 Percent of the file executed

27 Total basic block executions
2.08 Average executions per basic block
Regression Test Set Script for Data Flow

```plaintext
module
  pwr_
accprogs
  <g_power>
exceptions
  <divzero_exc>
globcod
  {%
  #include "system.h"
  #include "power.h"
  int cmd:
  %}
cases

  /******** exceptions *******/
  <g_power(0,-1), divzero_exc, dc, dc, dc>

  /***** data flow regression test plan *****/
  <, noexc, g_power(-2,-1), -0.5, float> /* sp03 */
  <, noexc, g_power(4,2), 16.0, float> /* st01 */
  <, noexc, g_power(5,-2), 0.04, float> /* st02 */
  <, noexc, g_power(-2,5), -32.0, float> /* st03 */
```
Test Coverage by Data Flow Regression Test Set

```c
/*
 *  power.c
 */

#include "system.h"
#include "power.h"
#include <stdio.h>
#include <stdlib.h>

float pwr_g_power(int x, int y) {
    int power;
    float z,answer;
    /* divide by zero exception */
    if(x==0 && y<0) {
        pwr_divzero_exc();
    return 0;
    }
    if (y<0)
    {
        power = -y;
    }
    else {
        power = y;
    }
    z = 1.0;
    while (power != 0) {
        if (power%2 == 0) { /* even power */
            x = x * x;
            power = power / 2;
        }
        else {
            z = z * x;
            power = power - 1;
        }
    }
    if (y<0)
    {
        z = 1/z;
    }
    answer = z;
    return answer;
}
```

Top 10 Blocks

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>36</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>4</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
</tr>
</tbody>
</table>

13 Basic blocks in this file
13 Basic blocks executed
100.00 Percent of the file executed

55 Total basic block executions
4.23 Average executions per basic block
Updated Test Suite Script for Data Flow

```plaintext
module
  pwr_
accprogs
  <g_power>
exceptions
  <divzero_exc>
globcod
%
#include "system.h"
#include "power.h"

int cmd;
%
}
cases

****** exceptions *****
<g_power(-0.1), divzero_exc, dc, dc, dc>

****** data flow updated test plan *****
<, noexc, g_power(-2.0), 1.0, float> /* sp01 */
<, noexc, g_power(-2.1), -0.5, float> /* sp03 */
<, noexc, g_power(4.2), 16.0, float> /* st01 */
<, noexc, g_power(5,-2), 0.04, float> /* st02 */
<, noexc, g_power(-2.5), -32.0, float> /* st03 */
```
Appendix 2  
Power Example Source Code and Test Scripts

Test Coverage by Data Flow Updated Test Suite

```c
#include "system.h"
#include "power.h"
#include <stdio.h>
#include <stdlib.h>

float pwr_g_power(int x, int y) {
    int power;
    float x,answer;

    /* divide by zero exception */
    if(x==0 & y<0) {
    pwr_divzero_exc();
    return 0;
    }
    if(y<0){
    power = -y;
    }
    else {
    power = y;
    }
    z = 1.0;
    while (power != 0) {
    if(power%2 == 0) { /* even */
    x = x * x;
    power = power / 2;
    } else {
    z = z * x;
    power = power - 1;
    }
    }
    if(y<0){
    z = 1/x;
    }
    answer = z;
    return answer;
}
```

Top 10 Blocks

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>36</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>5</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
</tr>
<tr>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
</tr>
</tbody>
</table>

13 Basic blocks in this file
13 Basic blocks executed
100.00 Percent of the file executed

61 Total basic block executions
4.69 Average executions per basic block
Appendix 3 Parser for ObjecTime Linear Form

The ObjecTime ROOM charts (behavior FSM diagram) can be output in a text format called ObjecTime Linear form which is the language used to specify ROOM designs. The grammar for this language is defined in Bakus-Naur Form (BNF) in [OTTG95]. This grammar was used to develop a parser for the ObjecTime Linear form language that is detailed in [Midd97].

BNF Grammar

A grammar is a formal device for defining sets of sequences of symbols [Brat90] for a language. One popular grammar notation commonly used in the definition of programming languages is Bakus-Naur Form or BNF.

The notational conventions used to define the ObjecTime Linear form [OTTG95] are as follows:

- ::= - Syntactic categories are defined using the form: <syntactic-category> ::= definition
- [ ] - The item(s) contained within square brackets are optional.
- * - An asterisk after an item indicates a sequence of zero or more items.
- + - A plus sign after an item indicates a sequence of one or more items.
- | - A vertical line separates alternatives.
- ( ) - Parentheses indicate a grouping of terms.
- <non-terminal> - Words contained within angle brackets (possibly with dashes) represent non-terminal productions (i.e. syntactic categories).
- terminal - Bold words represent reserved literal words (keywords) or symbols (these are called terminals).

A grammar is used to generate a string of symbols called a sentence. To illustrate, a simple grammar to define command sequences for a robot arm that may move in steps up or down can be:

\[
\begin{align*}
<move> & ::= <step>* \\
<step> & ::= up \mid down
\end{align*}
\]

DCG Grammar
A grammar can also be used to recognize a sentence. A recognizer can determine if a sentence belongs to a language by checking to see if the sentence can be generated by the grammar of the language. The recognition process is basically the inverse of generation. The grammar rules are applied to a given sentence in the opposite direction as for generation. If the sentence contains a sub-string that matches the right hand side of a grammar rule, it is substituted with the left hand non-terminal production. This process continues and if the complete sentence can be reduced to the starting non-terminal symbol of the grammar, the recognition process terminates successfully. This process effectively disassembles the sentence into its components, so this process is often called parsing.

Normally, to implement a grammar, a parsing program must be written for the grammar. A special grammar rule notation called definite clause grammar, or DCG, can be recognized by many Prolog implementations. For such an implementation, a grammar written in DCG notation is already a parsing program for the language.

The differences between the BNF and DCG notations are as follows:

::= is replaced by - - >
non-terminals are not enclosed in angle brackets
terminals are enclosed in square brackets making them Prolog lists
symbols are separated by commas
each rule is terminated by a ‘full stop’ (period) making them Prolog clauses

The DCG grammar for the example robot arm command language is as follows:

move - - > step.
moves - - > step, move.
step - - > [up].
Step - - > [down].

Prolog Grammar
Appendix 3  Parser for ObjectTime Linear Form

The DCG grammar can be converted into Prolog clauses in which the sentences are represented by difference lists of terminal symbols. Each sentence or phrase is represented by two lists; the sentence represented is the difference between the two lists. The DCG grammar for the robot arm example is converted into Prolog clauses as follows:

\[
\begin{align*}
\text{move}(\text{List}, \text{Rest}) & : - \\
& \quad \text{step}(\text{List}, \text{Rest}). \\
\text{move}(\text{List}_1, \text{Rest}) & : - \\
& \quad \text{step}(\text{List}_1, \text{List}_2), \\
& \quad \text{move}(\text{List}_2, \text{Rest}). \\
\text{step}([\text{up} | \text{Rest}], \text{Rest}). \\
\text{step}([\text{down} | \text{Rest}], \text{Rest}).
\end{align*}
\]

**Generating a Parse Tree**

A sentence can be broken down into phrases that correspond to the non-terminal productions in the grammar. The parse tree of a sentence consists of sub-trees that correspond to the phrases in the sentence. The ends of the sub-trees, or leaves, are the terminal symbols of the grammar. The parse tree for terminal productions is returned as a fact in which the relation name is the production name and the argument is the terminal symbol. The parse tree for non-terminal productions is returned as a fact in which the relation name is the production name and the arguments are the parse trees returned by sub-tree productions.

The Prolog grammar for the robot arm example modified to return a parse tree as the first argument follows:

\[
\begin{align*}
\text{move}( \text{move}(\text{StepTree}), \text{List}, \text{Rest}) & : - \\
& \quad \text{step}(\text{StepTree}, \text{List}, \text{Rest}). \\
\text{move}( \text{move}(\text{StepTree}, \text{MoveTree}), \text{List}_1, \text{Rest}) & : - \\
& \quad \text{step}(\text{StepTree}, \text{List}_1, \text{List}_2), \\
& \quad \text{move}(\text{MoveTree}, \text{List}_2, \text{Rest}). \\
\text{step}(\text{step}(\text{up}), [\text{up} | \text{Rest}], \text{Rest}). \\
\text{step}(\text{step}(\text{down}), [\text{down} | \text{Rest}], \text{Rest}).
\end{align*}
\]
Generating a Parse Tree for the ObjectTime Linear Form Grammar

The behavior specification of the device actor class shown in chapter 7 is read into the Prolog interpreter as a set of words:

```
S = [BEHAVIOR, { LANGUAGE, 'C++', SAPS, { DEFINE, transcript, ISA, Log, ( ) }, /', end of, saps }, INCLUSIONS, { DEFINE, 'SpecialDefinitions.h', DESCRIPTION, { 12, September, 1996, 11:18:09, am }, }, /', end of, inclusions }, FUNCTIONS, { DEFINE, function1, ATTRIBUTES, { Inheritable, Polymorphic }, { DEFINE, SendAndlncre, ATTRIBUTES, { Inheritable, Polymorphic }, TYPE, 'void', ARGUMENTS, 'RTSignal, &signal', CODE, { infoTransfer.send(signal, dataElement), ( ), dataElement++, ( ) }, { DEFINE, dataElement, ISA, 'Counter', ( ) }, DEFINE, currentQueue, ISA, 'SimpleCircularQueue*', UNDEFINED, ( ) }, DEFINE, priorityQueues, [4], ISA, 'SimpleCircularQueue', UNDEFINED, ( ) }, /', end of, variables in, top }, TRANSITIONS, { DEFINE, InitialTransition, FROM, INITIALPOINT, TO, STATE, Idle, ACTION, { dataElement, = 0, ( ), currentQueue, = &priorityQueues[0, ( )], ( ) }, DEFINE, Begin, FROM, STATE, Idle, TO, STATE, WaitFirstAck, TRIGGERS, { DEFINE, SIGNALS, ( ), Begin, ( ) }, ON, { testControl, ( ) }, ( ) ), Action, { SendAndInc(Info), ( ), currentQueue->addElement(dataElement.copcy)(), char, *garb, = currentQueue->printString(), ( ), transcript.show(garb), ( ), delete, garb, ( ) }, ( ) }, DEFINE, Ack, FROM, STATE, WaitFirstAck, TO, STATE, WaitSecondAck, TRIGGERS, { DEFINE, SIGNALS, ( ), Ack, ( ) }, ON, { infoTransfer, ( ) }, ( ), ACTION, { SendAndInc(Info), ( ), ( ) }, DEFINE, Ack, FROM, STATE, WaitSecondAck, TO, STATE, Idle, TRIGGERS, { DEFINE, SIGNALS, ( ), Ack, ( ) }, ON, { infoTransfer, ( ) }, ( ), ACTION, { RTDataObject, *garb, ( ) }, testControl.send(End), ( ), if, (currentQueue->isFull()), ( ), transcript.show("Next, from, Queue", "") }, ( ) }, transcript.show(*gabcurrentQueue->nextElement()), ( ) }, transcript.cr(), ( ) ), delete, garb, ( ) }, ( ) }, ( ) }, DEFINE, Info, FROM, STATE, WaitFirstAck, TO, STATE, WaitFirstAck, TRIGGERS, { DEFINE, SIGNALS, ( ), Info, ( ) }, ON, { infoTransfer, ( ) }, ( ) ), ACTION, { infoTransfer.send(Ack, *msg->data), ( ) }, ( ) }, DEFINE, Info, FROM, STATE, WaitSecondAck, TO, STATE, WaitSecondAck, TRIGGERS, { DEFINE, SIGNALS, ( ), Info, ( ) }, ON, { infoTransfer, ( ) }, ( ) ), ACTION, { infoTransfer.send(Ack, *msg->data), ( ) }, ( ) }, ( ) }, DEFINE, Idle, from, STATE, WaitFirstAck, ( ) ), ( ), end of, state, Idle, ( ) }, DEFINE, WaitFirstAck, ( ) ), ( ), end of, state, WaitFirstAck, ( ) }, DEFINE, WaitSecondAck, ( ) ), ( ) }, ( ) }, end of, state, WaitSecondAck, ( ) ), ( ), ( ) }, end of, state, WaitSecondAck, ( ) ), ( ) }], SUBSTATES, { DEFINE, Idle, ( ) }, end of, state, Device, ( ) ]
```

The behavior specification sentence is parsed by the ObjectTime linear form parser implemented in Prolog (see [Midd97]) to generate a parse tree as follows:

```
PT = behavior_spec(comment(), lang_spec, saps_spec(comment(cstring(cword(end)), cstring(cword(fof), cstring(cword(inclusions))))), incl Spec(comment(cstring(cword(end)), cstring(fof), cstring(inclusions)))), func_spec(comment(cstring(cword(end)), cstring(fof), cstring(funcs))))}, fsm_spec(comment()), state_spec(transitions_spec(transition_list(transition_item(comment()))), transition_name(objecttime_identifier(InitialTransition)), trans_source_pt(tr_point_spec(initialpoint)), trans_dest_pt(tr_point_spec(state_name(objecttime_identifier(idle))))}, transition_list(transition_item(comment()), transition_name(objecttime_identifier(Begin)), trans_source_pt(tr_point_spec(state_name(objecttime_identifier(idle))))}, transition_dest_pt(tr_point_spec(state_name(objecttime_identifier(WaitFirstAck))))}, transition_list(transition_item(comment()), transition_name(objecttime_identifier(Ack)), trans_source_pt(tr_point_spec(state_name(objecttime_identifier(WaitFirstAck))))}, transition_dest_pt(tr_point_spec(state_name(objecttime_identifier(WaitSecondAck))))}, transition_list(transition_item(comment()), transition_name(objecttime_identifier(WaitSecondAck))))}, 148
trans_source_pt(tr_point_spec(state_name(objectime_identifier(WaitSecondAck)))),
trans_dest_pt(tr_point_spec(state_name(objectime_identifier(Idle)))),
transition_list(transition_item(comment()), transition_name(objectime_identifier(Info)),
trans_source_pt(tr_point_spec(state_name(objectime_identifier(WaitFirstAck)))),
trans_dest_pt(tr_point_spec(state_name(objectime_identifier(WaitFirstAck)))),
transition_list(transition_item(comment()), transition_name(objectime_identifier(Info)),
trans_source_pt(tr_point_spec(state_name(objectime_identifier(WaitSecondAck)))),
trans_dest_pt(tr_point_spec(state_name(objectime_identifier(WaitSecondAck)))),
comment()),
comment()),
comment()),
comment()),
comment()),
comment()),
comment(cstring(cword(end),
cstring(cword(of),
cstring(cword(transitions),
cstring(cword(in),
cstring(cword(top)))))),
substates_spec(states_list(state_item(comment(),
state_name(objectime_identifier(Idle)))),
states_list(state_item(comment(cstring(cword(end),
cstring(cword(state),
cstring(cword(Device))))),
state_name(objectime_identifier(WaitFirstAck))),
states_list(state_item(comment(cstring(cword(end),
cstring(cword(state),
cstring(cword(WaitFirstAck))))),
state_name(objectime_identifier(WaitSecondAck))),
comment(cstring(cword(end),
cstring(cword(state),
cstring(cword(WaitSecondAck))))),
comment()),
comment(cstring(cword(end),
cstring(cword(substates),
cstring(cword(end),
cstring(cword(behavior),
cstring(cword(behavior)))))))))

From the parse tree, the state and transition names are extracted, reassembled into a digraph fact and written to a file that can be loaded into the RTKBT for analysis.
Appendix 4 Device Class Behavior Specification

BEHAVIOR
| LANGUAGE 'C++'
| SAPS
|   { DEFINE transcript
|   | ISA Log;
|   } /* end of saps */
| INCLUSIONS
|   { DEFINE 'SpecialDefinitions.h’
|   | DESCRIPTION
|   | { 12 September 1996 11:18:09 am
|   | };
|   } /* end of inclusions */
| FUNCTIONS
|   { DEFINE function1
|   | ATTRIBUTES {Inheritable Polymorphic};
|   DEFINE SendAndIncr
|   | ATTRIBUTES {Inheritable Polymorphic}
|   | TYPE 'void' ARGUMENTS 'RTSignal &signal'
|   | CODE
|   |   { infoTransfer.send(signal, dataElement);
|   |   dataElement++;
|   } /* end of functions */
| FSM
| VARIABLES
|   { DEFINE dataElement
|   | ISA 'Counter';
|   DEFINE currentQueue
|   | ISA 'SimpleCircularQueue' UNDEFINED;
|   DEFINE priorityQueues [4]
|   | ISA 'SimpleCircularQueue' UNDEFINED;
|   } /* end of variables in top */
| TRANSITIONS
|   { DEFINE InitialTransition
|   | FROM INITIALPOINT
|   | TO STATE Idle
|   | ACTION
|   |   { dataElement = 0;
|   |   currentQueue = &priorityQueues[0];
|   |   }
|   DEFINE Begin
|   | FROM STATE Idle
|   | TO STATE WaitFirstAck
|   | TRIGGERS
|   |   { DEFINE SIGNALS {Begin} ON {testControl};
|   |   }
|   | ACTION
|   |   { SendAndIncr(Info);
|   |   currentQueue->addElement(dataElement.copy());
|   |   char *garb = currentQueue->printString();
|   |   transcript.show(garb);
|   |   delete garb;
|   | }
DEFINE Ack
FROM STATE WaitFirstAck
TO STATE WaitSecondAck
TRIGGERS
{ DEFINE SIGNALS {Ack} ON {infoTransfer};
}
ACTION
{ SendAndIncrement();
};

DEFINE Ack
FROM STATE WaitSecondAck
TO STATE Idle
TRIGGERS
{ DEFINE SIGNALS {Ack} ON {infoTransfer};
}
ACTION
{ RTDataObject *garb;
  testControl.send(End);
  if (currentQueue->isFull()) {
    transcript.show("Next from Queue: ");
    transcript.show(*tag->currentQueue->nextElement());
    transcript.crt();
    delete garb;
  }
};

DEFINE Info
FROM STATE WaitFirstAck
TO STATE WaitFirstAck
TRIGGERS
{ DEFINE SIGNALS {info} ON {infoTransfer};
}
ACTION
{ InfoTransfer.send(Ack, *msg->data);
};

DEFINE Info
FROM STATE WaitSecondAck
TO STATE WaitSecondAck
TRIGGERS
{ DEFINE SIGNALS {info} ON {infoTransfer};
}
ACTION
{ InfoTransfer.send(Ack, *msg->data);
};

} /* end of transitions in top */

SUBSTATES
{ Define Idle
  } /* end of state Idle */
DEFINE WaitFirstAck
  } /* end of state WaitFirstAck */
DEFINE WaitSecondAck
  } /* end of state WaitSecondAck */
} /* end of substates in top */

} /* end of behavior of Device */